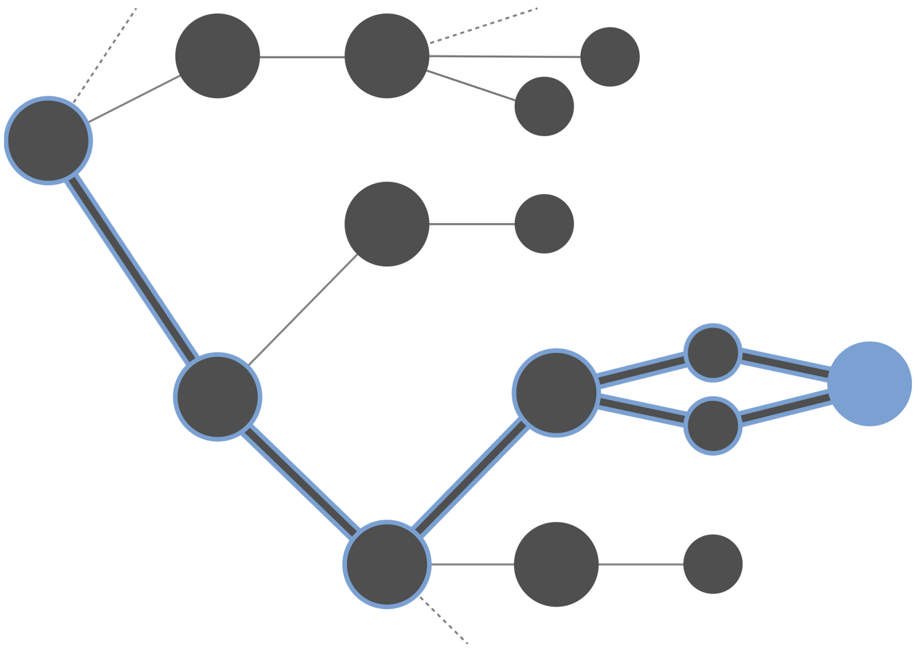
Tjøl Ravndal
Anders Solberg

State Management in Web Applications

Master's thesis in Computer Science
Supervisor: Trond Aalberg
June 2023

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science



NTNU
Norwegian University of
Science and Technology

Tjøl Ravndal
Anders Solberg

State Management in Web Applications

Master's thesis in Computer Science
Supervisor: Trond Aalberg
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

State management in JavaScript front-end frameworks is a critical yet complex aspect of web development, and is considered a major pain point for many developers. Despite its significance, there exists a considerable gap in the academic literature on the field of state management, which this thesis seeks to address. It is hypothesized that an implicit theory of state management exists within third-party libraries created to tackle the issue, and that through a detailed examination, this theory can be made explicit. To accomplish this, a qualitative literature analysis of selected state management libraries is conducted, split into the areas of client state management and server state management. The outcome of these analyses is the formulation of two theories consisting of key concepts and methodologies of client state management and server state management. These theories are then verified by being applied in three different scenarios, demonstrating their practical relevance. This process also shows how the theories facilitate an enhanced understanding of state management by providing a descriptive language that promotes discussion and reasoning. Ultimately, this thesis contributes to the body of knowledge in JavaScript front-end development, offering practical insights to assist developers in tackling the complexity of state management.

Sammendrag

Denne masteroppgaven tar sikte på å utforske og forstå et viktig, men komplekst, aspekt ved webutvikling: tilstandshåndtering i JavaScript front-end-rammeverk. Til tross for at mange utviklere anser tilstandshåndtering som en stor utfordring, finnes det lite forskning på området. Oppgaven søker å adressere dette hullet i litteraturen, med en hypotese om at det finnes en underliggende teori om tilstandshåndtering i tredjepartsbiblioteker som forsøker å løse problemet. Ved hjelp av en omfattende analyse av disse bibliotekene, er målet å avdekke denne teorien. Denne oppgave inneholder en kvalitativ litteraturanalyse av utvalgte biblioteker for tilstandshåndtering, inndelt i de to hovedområdene klient- og server-tilstandshåndtering. Analysene resulterer i utformingen av to teorier, som hver for seg inneholder sentrale begreper og metoder for henholdsvis klient- og server-tilstandshåndtering. For å verifisere teorienes praktiske relevans, blir disse deretter anvendt i tre ulike scenarioer. Disse anvendelsene viser også hvordan teoriene bidrar til en bedre forståelse av tilstandshåndtering ved å gi et beskrivende språk som fremmer diskusjon og resonnering rundt emnet. Denne oppgaven tilfører verdifull innsikt og kunnskap til feltet, og tilbyr praktiske verktøy til utviklere for å takle utfordringene knyttet til kompleksiteten i tilstandshåndtering.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

JavaScript front-end frameworks, like React [1], have empowered developers to build more dynamic and responsive web applications. An essential aspect of these frameworks is *state management*, dictating how data flows and changes over time in applications. However, the field of state management presents considerable complexities. According to Pete Hunt, a former member of the React core team, state management can be one of the hardest parts of building applications [2]. The *State of JavaScript 2021* survey, which had over 16,000 developer respondents, ranked state management as the third highest pain point for JavaScript web developers [3].

Although the field ranks high among pain points, there is not much scientific literature on the topic. In a review article from 2018, Sun and Ryu reviewed the research done in the last decade on client-side JavaScript programs, and state management was not mentioned [4]. When selecting metrics for measuring code quality in React-based applications, Lin et.al. included various metrics related to state, but no discussion on state management in and of itself [5]. State management was addressed as an important topic by Novac et.al. when comparing React to Vue, another JavaScript front-end framework, but it was only briefly considered and without details [6]. In a paper describing the decision-process and lessons learned when migrating an application from old technology to React, Roby et.al. mentioned the choice of state management solution and reflected upon the learning curve of using the solution they chose, but the decision-process of why they chose it, and how it relates to other state management solutions, was not discussed [7].

1.1 Research Objective

We can clearly see that there exists a gap in the literature, and in their paper regarding adoption of JavaScript front-end frameworks, Ferreira, Borges, and Valente even calls out for more research on JavaScript front-end frameworks [8]. The objective of this thesis is thus to heed this call and offer a thorough exploration of the topic of state management in JavaScript front-end frameworks. While the aim of this thesis is not to look at what makes state management frustrating for developers, the hope is that this theory can work to ease this frustration by giving a clearer understanding of the topic.

Due to the limited research on state management, there is currently no formalized and established general theory of state management. However, it is a practical problem that has been solved many times by different companies and individuals. The solutions are usually published as third-party libraries, which can be included in any suitable project. These libraries often tackle different aspects of state management, with some focusing on *client* state management and others on *server* state management. Client state management refers to handling the state information that is stored and managed on the client side, while server state management involves the caching and re-validating of state that is fetched from a server.

The hypothesis guiding this research is that a general theory of state management implicitly exists within the third-party libraries created to address this issue, and that by performing a thorough examination of these libraries, this implicit theory can be formulated and made explicit. By "theory" we mean the generalized concepts and terms used to describe state management as a whole. Thus, the research questions for this thesis are as follows:

- **RQ1**: What are the main concepts and methodologies in the field of state management, and how can they be formed into a generalized theory?
  - **RQ1.1**: What are the main concepts and methods in client state management
  - **RQ1.2**: What are the main concepts and methods in server state management
- **RQ2**: What are the potential scenarios where a generalized theory of state management can be employed, and how might these enhance understanding and practice in this field?

1.2 Methodology and Limitations

The methodology of this research involves a thorough literature analysis of some selected state management libraries. The analysis focuses on understanding the underlying principles and practises of the libraries, which is then used to construct a generalized theory of state management, encompassing the key concepts and terms that can describe state management as a whole.

To provide a focused perspective for this research, we look at state management primarily from the perspective of React [1]. The ecosystem of available front-end frameworks is diverse, and many of the state management libraries are made specifically for one framework. We base the thesis on React because of its standing as the most used front-end framework [9], making the resulting theoretical framework relevant for as many developers as possible. Although we are going to talk about state management from the perspective of React, several of the state management libraries we review are independent of any specific framework. Consequently, we believe that parts of the theory will be useful for developers utilizing other frameworks as well.

The included libraries are free, publicly available state management libraries, accessible via Node Package Manager (npm)[1], the largest open source package manager for the JavaScript ecosystem. The selection of which libraries to include in this thesis is primarily based on their popularity and usage within the developer community, and the potential insight they could offer towards constructing a generalized theory of state management.

1.3 Thesis Structure

The introductory chapter you are currently reading is the first of nine chapters in this thesis. This section presents a brief overview of the following eight chapters. The thesis is primarily focused on the two research sub-questions RQ1.1 and RQ1.2, respectively concerning client state management and server state management. Consequently, the research is divided into two parts, one for each research sub-question. This is reflected in the chapter structure, specifically chapters 4-7, where each pair covers a respective part of the research question.

Chapter 2: Preliminaries describes and presents key concepts and theories that are advantageous to understand when reading this thesis. Among other things, preliminaries includes a brief description of React, relevant concepts from functional programming, normalization from database theory, and an introduction to key concepts related to client-server interactions over a network.

Chapter 3: Method outlines our research methodology. It describes the selection process of which third-party state management libraries to include, and how they are analyzed. To provide clarity on our approach, we summarize the most important steps in our research process.

[1]https://www.npmjs.com/

Chapter 4/5: Client/Server State Management Libraries consists of summaries of the client and server state management libraries, as part of the literature analysis. All included libraries are presented, with a focus on the underlying principles and practices of the libraries.

Chapter 6/7: A Theory of Client/Server State Management are the main products of this thesis. These chapters present formulated theories of what client state management and server state management are, based on their respective literature analyses. The theories entail key concepts and terms of client state management and server state management.

Chapter 8: Verification examines three use-cases for the formulated theories. Whether or not the theories can be said to answer **RQ2** is the main area of investigation in this chapter.

Chapter 9: Conclusion gives a brief overview of our process and main findings, and we conclude whether the thesis answers the research questions. We also present our proposals for further work in the area of state management.

Chapter 2

Preliminaries

State is a term that encapsulates a multitude of definitions. A broad definition of state is "[...] data that is stored in memory" [10], but for our purposes it does not really help us any more than getting a gist of what we are talking about. Data - of some kind - that is stored - somehow and somewhere. It is hard to define state without explicitly specifying the *context* in which it is being discussed. Therefore, in this thesis, we focus on state management within the context of web applications.

To fully understand the issues related to state management in web applications, foundational concepts of the topic is presented. Grasping these core concepts will be beneficial in order to accept our reasoning regarding the theories presented in chapter 6 and chapter 7. The first sections in this chapter are universally applicable to state management, while the latter part specifically relates to server state.

The chapter begins with a description of the *React* framework, facilitating a solid understanding of its component structure and state. The core concepts of state management rely on methodologies such as *pure functions*, *imperative* and *declarative* programming, and *immutability*, hence they are covered. An examination of *normalization*, a concept from database theory, is included due to its relevance to how state should be stored. Additionally, the chapter briefly delves into *Finite State Machines*, owing to its valuable application in the realm of state management.

Following this, we go through *RESTful APIs* and *GraphQL APIs*, which are widely utilized for communication between client and server, and introduce *asynchronous* programming in JavaScript. As client-server communication often must consider *caching*, an introduction on the topic is provided, before discussing networking-related concepts like *retrying requests* and *offline features*. The concept of *streaming data* is introduced, a more advanced form of network communication often supported by server state management libraries, as well as *pagination*, another widely supported feature. To wrap up, we look at some of the research being done in web development, related to the field of state management.

Because we base the thesis on state management for React, which is a JavaScript framework, all code examples are written in JavaScript.

2.1 React

React is a JavaScript library, primarily used for building user interfaces [1]. It had its first public release back in 2013[1], and is now the most used JavaScript front-end framework[2] [9]. React can be used to build user interfaces for both web and mobile applications, and is maintained as an

[1]https://github.com/facebook/react/releases/tag/v0.3.0

[2]Strictly speaking, React is not a framework, but a library. However, we will not delve into the difference between a library and a framework in this thesis. One reason for the confusion, is that other, actual frameworks, such as NextJS (https://nextjs.org/) and Remix (https://remix.run/), utilize React as their foundational library.

open-source project by Facebook.

React embraces a declarative paradigm (further elaborated in section 2.4), enabling developers to define what the user interface should look like for various application states. When this state changes, React initiates a re-rendering process, updating and refreshing the user interface to accurately reflect the changes that have occurred in the state. To describe the user interface, React utilizes JSX, a syntax extension for JavaScript that looks like a blend of HTML and JavaScript.

The architecture of React is built around *components*, which are self-sustaining, independent entities that encapsulate the state and logic required to output a piece of the user interface. User interfaces built with React are a composition of components, assembled into a component tree. While each component manages its own state, it is possible to pass properties from one component down to its children, facilitating a unidirectional data flow, ensuring consistency and predictability.

Figure 2.1 illustrates a component tree that demonstrates a simplified version of Facebook. The top level component in this tree is the `<Facebook />` component. Its immediate child components are `<NewsFeed />` and `<Profile />`. Furthermore, each of these child components have their own child components. The dashed lines extending downwards from the leaf components indicate that these components may have additional children components. While the complete component tree is probably larger and more complex than shown, the figure adequately represents the hierarchical structure of a component tree.

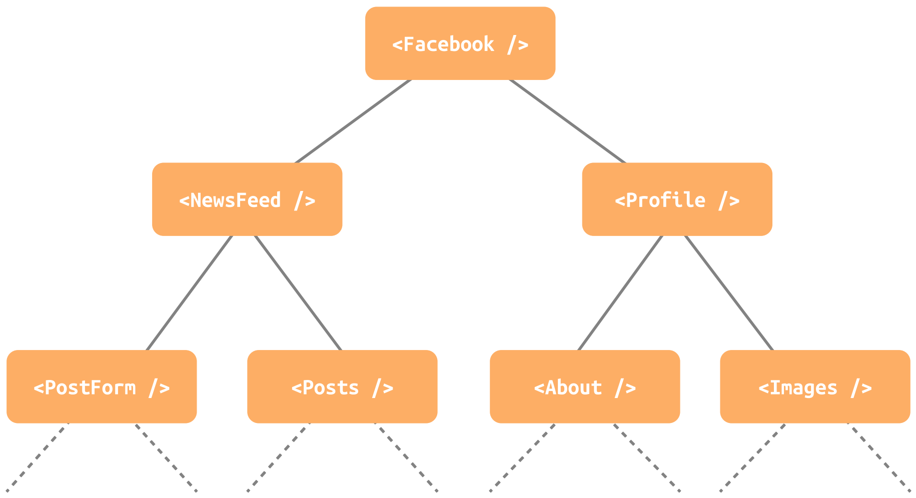


Figure 2.1: The component tree depicts a simplified version of Facebook, with `<Facebook />` as the top-level component. Immediate child components include `<NewsFeed />` and `<Profile />`, while additional nested child components exist within each of them. The dashed lines suggest that there could be more child components deeper in the tree.

2.2 React State

State in React acts as the component's memory, and it has the ability to hold any data type in JavaScript [11]. By leveraging state, components can dynamically update the displayed content within a user interface. This becomes particularly useful in various scenarios, such as tracking the state of a button click or capturing the current value in a search field. The utilization of state empowers React components to effectively manage and update their internal data in response to user actions.

State in React as described above is usually referred to as *local state*, meaning it is owned by a specific component. Nonetheless, components have the option to share their state directly with their children components if desired. Local state by itself does not have a simple or efficient way to share state with components that are far away from itself in the component tree. On the other hand, *global state* is state that is globally available to all components in the component tree. Global state can either be owned by single component, or be localized outside of the component tree. With global state, components can both read and update its values from anywhere within the application.

In figure 2.2, a component tree including both *local* and *global* state is shown. The `<NewsFeed />` component utilizes local state, visualized in the green font color. The grey circle including the global state indicates that the global state in this example is located outside of the component tree. The blue lines represent the utilization of the global state by the `<Profile />` and `<Posts />` components, highlighting the efficient sharing of state between components that are located at significant distances from each other within the component tree.

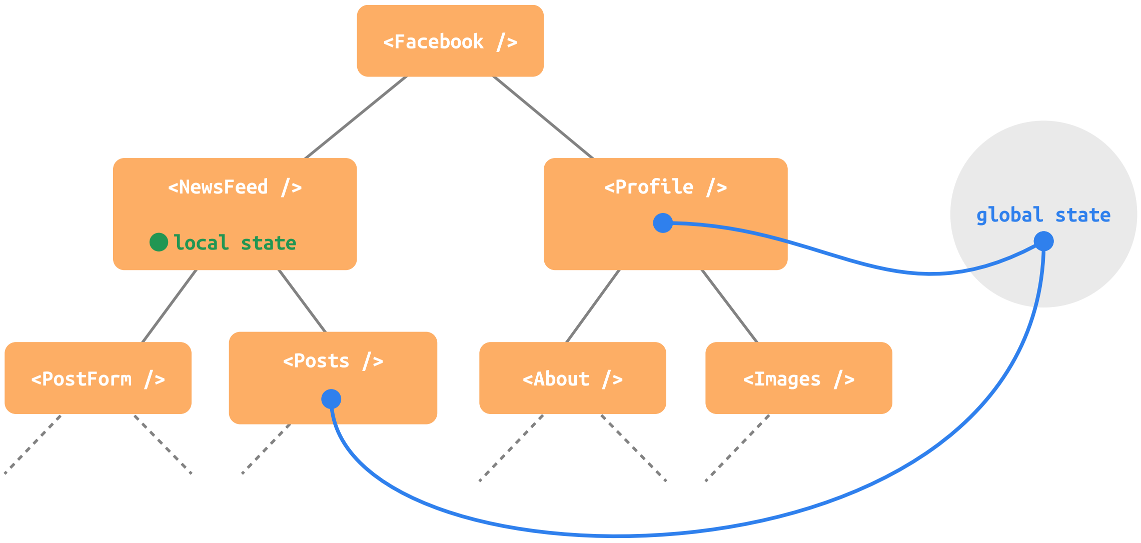


Figure 2.2: illustrates a component tree with both local and global state. The `<NewsFeed />` component utilizes local state (green font color), while the gray circle represents the global state located outside the component tree. Blue lines indicate the utilization of global state by the `<Profile />` and `<Posts />` components, showcasing efficient state sharing across distant components.

2.3 Pure Functions

To explain *pure functions* we will use the definition of *referential transparency* from functional programming. Referential transparency means that an expression can always be substituted with its evaluated value without changing the behavior of the program [12]. To further clarify this notion, let us consider a scenario where it is advantageous to isolate specific functionality into separate functions. For instance, in an application there might be a need to frequently concatenate the first and last name of a user, in order to display the user's full name. To avoid code repetition, one approach is to create a dedicated function specifically designed to handle this concatenation. The function might be implemented as follows:

```
const getFullName = (firstName, lastName) => firstName + " " + lastName;
```

Now you can replace all the parts in your code where you previously concatenated the names, with your new function `getFullName`. The reason that this works as expected is because of the property of referential transparency. The function `getFullName` is referentially transparent because substituting the function with its evaluated return value will not change the behaviour of the program.

Pure functions are essentially understood from the concept of *referential transparency*. These are functions that will consistently produce the same result when provided with the same inputs, and importantly, they do not trigger any side effects. A trivial example that illustrates the concept of *pure functions* is the square function:

```
const square = (number) => number ** 2;
```

If our program calls `square(3)`, it will return the number 9. If we repeat the call to the `square` function multiple times, passing it the argument of 3, we will get 9 as return value every time. Since the `square` function is referentially transparent we could replace the call to the `square` function with its evaluated value `number ** 2` (assuming the variable `number` is the same as the argument given to the `square` function) any place it was previously referenced, and the program would produce the same result as before. The `square` function does not modify or use any state variables outside its local scope, and therefore the function does not have any side effects. Hence, we classify the square function as a *pure function*.

2.4 Declarative and Imperative

To understand the difference between *imperative* and *declarative* programming, we can imagine giving instructions to someone. By telling them step-by-step what to do, we are giving imperative instructions. Similarly, in imperative programming, we tell the computer exactly what sequence of commands it needs to execute in order to achieve a specific goal [13]. This often involves giving explicit commands that modify variables over time. As an example of imperative programming in JavaScript, consider a program that uses the `square` function, discussed in section 2.3, to square a list of numbers:

```
// IMPERATIVE
const numbers = [1, 2, 3];
const squared = [];

for (let i = 0; i < numbers.length; i++) {
    squared.push(square(numbers[i]));
}
```

In this example, we tell the computer exactly how to square the numbers: by looping over them one by one and modifying the `squared` array in each step.

However, when simply describing what the end result should look like, we are giving declarative instructions. In declarative programming, instead of specifying each step, we describe what the outcome should be. We can recreate the example from above, but this time in a declarative style:

```
// DECLARATIVE
const numbers = [1, 2, 3];
const squared = numbers.map(n => square(n));
```

In the declarative version, we describe what we want: a new array where each number is the squared of the corresponding number in the original array. We do not specify how to get there, we just describe the outcome, and the `Array.prototype.map()` method takes care of the details.

To summarize, *imperative* programming involves giving the computer a sequence of steps to execute, whereas *declarative* programming involves describing the desired result, and the specific steps are abstracted away.

2.5 Immutability

In the context of computer science, *immutability* refers to the characteristic of a value or object that restricts it from being modified once it has been created [10]. Essentially, an object is said to be immutable, if any modification to it requires the creation of a new object, instead of altering the existing one. This is in contrast to *mutable* updates, where a value or object is modified directly.

In JavaScript, primitive data types, such as strings, numbers and booleans are inherently immutable [14]. When updating a primitive value, the assignment operator ("=") is utilized, which essentially discards the old value and assigns a *new* value to the variable. Every change in primitive types results in a new value, with no mechanism to alter the original value directly.

On the other hand, non-primitive data types such as objects and arrays are mutable. Mutable data types have an object reference, and even though the individual properties can change, the reference stays the same. To *immutably* modify a *mutable* data type in JavaScript, a new instance of the data type must be created, resulting in a new reference. The new instance is created with a copy of the original data, along with the intended modifications. Thus, the original variable is not actually modified, but rather replaced with a new instance containing the modifications.

Programs relying on immutable updates expect a new object reference every time a change occurs, whereas programs relying on mutable updates expect the object reference to be static, despite changes in value. The expected behaviour of a program can break if the type of updates does not align with the program's expectation of mutability or immutability. Because of this, a program that relies on immutable updates will need to be careful not to perform mutable updates, but ensure that updates are performed immutably.

We will show how mutable and immutable updates are performed on the non-primitive data types `Array` and `Object`. For example, the `Array.prototype.push()` method can be used to update an array mutably. This will maintain the same reference for the array, and mutate the array directly. Updating an object mutably is done by using the assignment operator on one of the properties of the object. This will also not change the object reference, but mutate the object directly.

On the other hand, changing an array immutably is done by defining a new array, optionally adding the previous items of the array, and optionally removing items. Let's look at some examples of how this can be done in JavaScript, using the spread syntax (`...`) to copy elements from the old array into the new array:

```
// Original array
const oldArr = ["apple", "banana", "mango"];

// Add item to end of array
const newArr = [...oldArr, "kiwi"];
```

Changing a mutable object immutably is similar to an array. First, we create a new object, copy the previous values as needed, and then add or remove values. Once again, we will use the spread syntax (`...`) to copy values from the old object into the newly created object:

```
const personObj = {
    name: "Charlie",
    age: 16,
};

// Increment age after birthday
const newPersonObj = {
    ...personObj,
    age: personObj.age + 1
};
```

Since the two objects `personObj` and `newPersonObj` have different object references, `newPersonObj` can be said to be an immutable modification of `personObj`. As shown, updating objects which are one level deep is pretty simple. However, dealing with nested objects is more complex as it requires the creation of a copy of each level of the object, and pasting it into the corresponding object level in the new object. Let us look at an example:

```
const nestedObj = {
    firstLevel: {
        secondLevel: {
            thirdLevel: {
                property: "I am a leaf node"
            }
        }
    }
};

// Change property
const newNestedObj = {
    ...nestedObj,
    firstLevel: {
        ...nestedObj.firstLevel,
        secondLevel: {
            ...nestedObj.firstLevel.secondLevel,
            thirdLevel: {
                ...nestedObj.firstLevel.secondLevel.thirdLevel,
                property: "I have changed"
            }
        }
    }
};
```

Updating nested objects can be error-prone, and the code can quickly become verbose and cumbersome, especially when making changes to deeply nested objects. There exist multiple third-party libraries that greatly simplify this process, a notable example being Immer[3].

2.6 Normalization

The concept of *normalization* is a technique from relational database theory. It can be described as a step-by-step process of organizing data in a database, with the primary goal of maintaining atomicity, eliminating data redundancy and inconsistency, and ensuring logical data dependency among attributes [15]. Atomicity implies that each value should be indivisible, not consisting of multiple smaller values. For example, the full name of a user should not be stored as a single entity, but rather be divided into first name, middle name, and last name. Data redundancy occurs if the same data is stored in more than one place. If data redundancy is present in a database, it can lead to data inconsistency, which means that the same piece of data is stored in different places with differing values, potentially due to being updated in one location but not in another.

The goal of normalizing a database is to optimize its structure to allow efficient execution of database operations such as insert, update and delete, without causing data anomalies, such as inconsistency. Through normalization, a database becomes more maintainable, often more efficient, and less prone to data anomalies. However, it is worth noting that the impacts on performance can vary depending on the specific database workload and use-case. Sometimes, de-normalized databases can result in more efficient reads, as all data needed to answer a query might be located in a single location. Because of this, a normalized database is not preferable in all scenarios, and has to be considered on a case by case basis.

[3]https://immerjs.github.io/immer/

2.7 Finite State Machines

Arthur Gill presented a model of the finite state machines in 1962 [16]. It is a mathematical model that serves as an abstract representation of a system, consisting of a finite set of states, and rules for transitions between these states. Gill remarks that the theory itself does not belong to any particular scientific area, but is practically applicable to a wide variety of fields, such as communication, psychology and business administration, to name a few. Additionally, finite state machines are highly relevant to computer science and networking, including the field of state management.

The basic model of a finite state machine presented by Gill consists of a finite input alphabet, a finite output alphabet, a finite set of possible states and characterizing functions [16]. To further clarify the concept, an example of a finite state machine representing a coffee machine is presented. The coffee machine consists of the finite set of possible states: *idle*, *brewing* and *done*. The finite input alphabet refers to the set of possible inputs or events that a finite state machine can receive. In our example, the coffee machine can be started by someone turning on the power. Additionally, the completion of brewing is signaled when there is no water remaining in the container. To prepare the machine for reuse, the used coffee beans must be removed. These events serve as inputs in our finite state machine, hence, our input alphabet is *start*, *complete* and *cleanup*.

The characterizing functions in a finite state machine determine the next state based on a given input. These transitions between states are visualized as directed edges. By utilizing these characterizing functions, a transition table can be constructed to provide an overview of each input and its corresponding next state. In the example of the coffee machine, the emphasis is solely on the states and their transitions, without considering any potential outputs that might be generated. Therefore, the example does not incorporate a finite output alphabet.

A finite state machine illustrating the example about the coffee machine is visualized in figure 2.3. The three states *idle*, *brewing* and *done* are represented as blue, orange and green nodes respectively. Furthermore, the edges *start*, *complete* and *cleanup* represents the possible state transitions. In addition, the figure includes a transition table, showing an overview of the next state based on a given input.

Input:	start	complete	cleanup
Next:	BREWING	DONE	IDLE

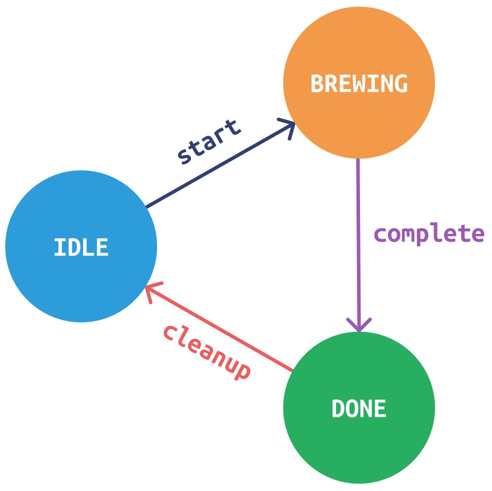


Figure 2.3: Overview of a finite state machine representing a coffee machine. The states, *idle*, *brewing*, and *done*, are depicted as blue, orange, and green nodes, respectively. The labeled edges indicate the possible state transitions: *start*, *complete*, and *cleanup*. The figure also includes a transition table summarizing the next state based on the given input.

2.8 Server APIs

All data that is not co-located with the client is seen as server data. The server may be a local server running in your basement, a server located on the other side of the world or it might be a cluster of servers in the cloud. Where and how data is stored does not have any significant impact on the content of this thesis, since the thesis is mostly concerned with how we manage the server data once it reaches the client. Thus, we can look at how the data is stored on the server as a "black box", indicating that we will not discuss it in detail, as illustrated in figure 2.4. This thesis primarily focuses on the client. The communication between the client and server is facilitated through an Application Programming Interface (API).

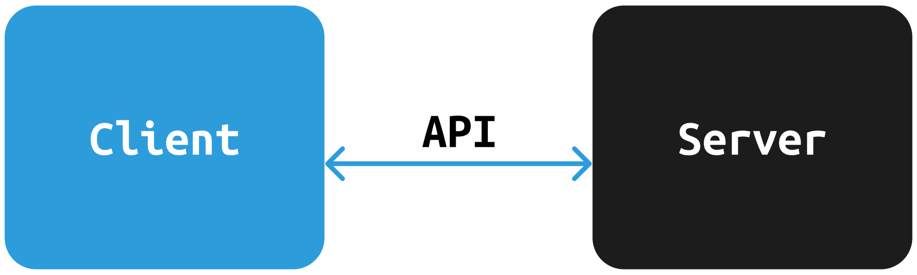


Figure 2.4: The figure illustrates the conceptual representation of the server as a "black box", indicating that its internal workings are not discussed in detail in this thesis. The primary focus of this thesis is on the client, with communication facilitated through an API.

Communication between a client and a server is done through an API, which is like a specification for the set of operations the client can perform on the server's resources. This API is what the client uses to interact with the server, making requests and updates in the specified manner. While this thesis is only focused on the client side of this interaction, it is necessary to know what type of API the server is using, as this affects how the client should request its data. We will cover two different architectural models for server API's, namely Representational State Transfer (REST) and GraphQL. These are two common types of server APIs [17], but they work in very different ways. In the examples of API provided, GitHub's API is used, as they have exposed both a REST API[4] and a GraphQL API[5].

2.8.1 REST

REST is an architectural style which is used to build RESTful APIs [18]. REST uses *resources* as its key abstraction of information, and a resource can be anything nameable, including server state. Resources can be static, all the time pointing to the same entity, or they can be dynamic, pointing to different entities depending on when it is requested. Let us say you want to find information about a release of a program that is on GitHub. You could request "the release with the given tag". This would be a static resource, always pointing to the specific release with that given release ID. On the other hand, you could request "the latest release" of the program, and this would be a dynamic resource, always pointing to the currently latest release.

Resources in REST are identified with *resource identifiers*, which for the examples above from GitHub's REST API is structured like this:

```
// the release with the given tag
/repos/{owner}/{repo}/releases/tags/{tag}

// the latest release
/repos/{owner}/{repo}/releases/latest
```

[4]Documented at https://docs.github.com/en/rest
[5]Documented at https://docs.github.com/en/graphql

The "{}" are used to represent variables, which would be substituted with the specific owner, repository and tag for a specific resource. For example, the resource identifiers for React version v18.2.0, or the latest React release would look like this:

```
// the react release with the v18.2.0 tag
/repos/facebook/react/releases/tags/v18.2.0

// the latest release of React
/repos/facebook/react/releases/latest
```

So far we have only talked about resources in a REST architecture and how to identify them, not how they are exposed to a client. RESTful APIs allow clients to access and manipulate resources on a server, using standard HTTP methods like GET, POST and DELETE. When creating a RESTful API you would define a set of *endpoints* to expose your resources, using the HTTP method and resource identifier. Such endpoints for the GitHub releases could for example be:

```
// GET the latest release
GET /repos/{owner}/{repo}/releases/latest

// DELETE the release with the given release ID
DELETE /repos/{owner}/{repo}/releases/{release_id}
```

The endpoints may return data in the form of *representations* of the resource they point to. In the case of GitHub's API (and often when it comes to JavaScript APIs) the representations are returned in the form of JSON objects. Not all endpoints return representations. For example, the `DELETE` endpoint above only returns a status code to indicate success or error. Some endpoints require that you send a representation of a resource with the request, for example if you want to add a new entity to a resource. If we wanted to create a new release of React, we would make a request to the corresponding endpoint, and pass along a JSON representation of the new release in the request body:

```
// POST (create) a release
POST /repos/facebook/react/releases
// JSON representation of the new release passed along in the request body
```

As every resource is exposed through its own endpoint, there will be a lot of endpoints. In cases where the data you need is not exposed through an existing endpoint, you could create a new resource on the server, which contains exactly the data you need, and then expose this through a new endpoint. However this is not always an option, as in many cases you do not have access to the server. This approach could also lead to the API having a plethora of different endpoints, each covering a very specific use-case. To get around this, you could combine the results from the different existing endpoints to gather the sufficient amount of data. This may affect efficiency, as each request to the server involves a round-trip, resulting in a certain amount of time and bandwidth cost. This potential inefficiency is a result of REST being a generalized interface, instead of tailored to the application's needs [18].

2.8.2 GraphQL

GraphQL is a query language used to define data requirements and interactions in user interfaces [19]. A GraphQL service offers a GraphQL API that client applications can leverage to precisely query the data they need. The GraphQL service endeavors to create a tailored API that matches the specific needs of the application, making GraphQL a product-centric query language. GraphQL was built by Facebook in 2012 and released as an open-source project in 2015. The GraphQL specification is currently maintained by the *GraphQL Foundation* which was formed in 2019.

Every GraphQL service defines a type schema [19]. The type schema ensures that types in a query are correct before code execution, and enable type validation in development time. The schema of a GraphQL API can be queried directly, enabled by GraphQL's introspection system. This enables the creation of powerful developer tools, and allows clients to dynamically discover and query the GraphQL service without any prior knowledge to the server's implementation. The type system and introspection capability of GraphQL also makes it self-documenting and viewable by a GraphQL schema viewer.

To send a GraphQL request, we write a GraphQL *query string*. First we define what type of request it is by specifying an execution operator, which can be one of `query`, `mutation` or `subscription`. Queries are for *fetching* data, mutations are for *modifying* data, and subscriptions are for *real-time* updates. After choosing execution operator, we specify the desired data, down to the individual field level. The client interacts with the GraphQL API by making requests to a single endpoint, and the GraphQL query string is added to the request body. An example query string for fetching the latest React release from GitHub is shown below:

```
query {
  repository(owner: "facebook", name: "react") {
    latestRelease {
      tagName
      publishedAt
    }
  }
}
```

The client is responsible for deciding the format returned from a request. The client can specify for *exactly* what fields it needs in a GraphQL query, resulting in only the data that were necessary. One of the benefits of using GraphQL is that the shape of the response data is the same as the shape of our query, as shown in the query response below.

```
{
  "data": {
    "repository": {
      "latestRelease": {
        "tagName": "v18.2.0",
        "publishedAt": "2022-06-14T19:54:21Z"
      }
    }
  }
}
```

GraphQL execution operators supports *Variables* as input to parameterize queries. Some use cases for *Variables* are to filter what data should be returned from a query, or to pass along data that should be stored on the server in a mutation. Furthermore, GraphQL *Directives* can be used to add conditional logic at runtime to alternate the query format, based on input *Variables*. A common use case of *Directives* is to determine whether a field should be included in the query or not.

An example mutation for creating a new repository is shown below. The repository data is passed in as *Variables* to the mutation function. We can also define what data should be returned from the mutation. In this example we want to receive the *id* of the newly created repository as response data.

```
// Mutation
mutation ($name: String!, $visibility: RepositoryVisibility!) {
  createRepository(input: {name: $name, visibility: $visibility}) {
    repository {
      id // returned from server after mutation
    }
  }
}

// Query variables
{
  "name": "learnGraphQL",
  "visibility": "PUBLIC",
}

// Response
{
  "data": {
    "createRepository": {
      "repository": {
        "id": "MDEwOlJlcG9zaXRvcnkxMjM4NTkxNjE="
      }
    }
  }
}
```

A GraphQL service exposes all of its data to the client, and the client can query exactly the data needed. RESTful APIs exposes resources through predefined endpoints determined by the server. If a client needs data from multiple endpoints, it can either fetch data from multiple endpoints and combine the data on the client, or create a new endpoint on the server that contains the desired data. The server is responsible for determining what data is exposed in a RESTful API, whereas a GraphQL API allows the client to determine what data it needs.

According to the 2021 State of the API report by Postman with over 28 000 respondents, 94% reported that they had used REST APIs, whereas 31% reported that they had used GraphQL APIs [17]. However, GraphQL has grown rapidly since it was released as open source in 2015, compared to REST that was formalized by Fielding in 2000 [18].

2.9 Asynchronous JavaScript

Making a request to a server over the network takes time. If we do it *synchronously*, meaning we perform one task at a time, it would lead to a lot of stale time waiting for the requests to complete. To avoid this, we use *asynchronous* programming, enabling us to write code that can do multiple tasks simultaneously, without blocking the main thread. In JavaScript, asynchronous programming would traditionally be done using *callbacks*. The callback pattern entails that you register a function (a callback function) that should run when an event occurs. For example, if you call a request to a server using the `http.get()` method in Node.js, you have to register a callback function if you want to perform logic with the result of that request.

```
http.get('/repos/facebook/react/releases/latest', (result) => {
    // this is the callback function that will
    // run when the request is complete
})
```

Callbacks are practical, but could in many cases result in messy, error-prone code, especially when needing to register callbacks inside other callbacks, creating a nested “callback hell” that is difficult to reason about [20]. As a response, the built-in JavaScript interface `Promise` emerged, a tool for managing asynchronous operations in a clean and efficient way. `Promise`s serve as an abstraction for the result of an asynchronous operation, and can be in one of three different states: *fulfilled*, *rejected*, or *pending* if it is neither fulfilled nor rejected [21]. The `Promise` constructor is given two functions: one to fulfill the `Promise` and another to reject it:

```
const myPromise = new Promise((fulfill, reject) => {
  fulfill("Hello");
  // Promise is already fulfilled,
  // but if we wanted to reject we would use this:
  reject("Oh no!");
})
```

To get the value that a `Promise` fulfills to, you attach reactions to the promise using the `.then()` method, which you pass an *onFulfilled* handler, and optionally an *onRejected* handler. If you return anything in the *onFulfilled* handler, it is automatically wrapped in a new `Promise`, making it possible to *chain* multiple `Promise`s.

```
myPromise
  .then((fulfilledValue) => {
    console.log(fulfilledValue); // "Hello" from myPromise defined above
    return fulfilledValue.toUpperCase();
  })
  .then((chainedValue) => {
    console.log(chainedValue); // "HELLO", chained result
  })
```

By using `Promise`s, you don't have to register callback functions next to functions performing asynchronous logic, but the asynchronous function can instead just return a `Promise`. Using this returned `Promise` you can chain reactions that handle fulfillment and rejection. The standard `fetch()` method, implemented by all modern browsers, utilize `Promise`s. Instead of registering callbacks, `fetch()` returns a `Promise` that you can chain reactions to by using `.then()`. If we take the example with the server request, it could look like this:

```
fetch('/repos/facebook/react/releases/latest')
  .then((serverResponse) => {
    // this function will run when the promise fulfilles
  })
  .catch((rejectedValue) => {
    // this function will run when the fetch promise rejects
  })
```

Nowadays it is even easier to work with asynchronous code, thanks to the *async/await* pattern, allowing you to *seemingly* synchronously wait on a `Promise` to complete. For our purposes however, the most important thing to know is that the `Promise` interface is what most server state management libraries use to implement its fetching mechanisms.

2.10 Cache

Cache is widely used in multiple contexts in computer science, and there exist a wide variety of cache types. Some of these are CPU cache, SSD cache, DNS cache, database cache or memory cache. The main feature of a cache is to store a copy of some data in a temporary storage, such that future requests can return the copy, instead of requesting the main storage. The relevance of cache in this thesis is targeted towards the web. The HTTP specification defines cache like the following: "A cache is a local store of previous response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cacheable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. Any client or server MAY employ a cache" [22]. From this definition, a cache is not only the data in the temporary storage, but the term also includes the system for retrieving, updating and deleting the temporary storage.

Figure 2.5 shows an overview of a possible combination of cache types in a software system. Proxy and CDN caches are examples of *shared caches* [23], which can store responses that can be shared among users on a network. Browser cache is a type of *private cache* which is not shared, and is useful when for example storing authentication credentials. The primary focus in this thesis is the cache type found within the application data of a web application. Software systems may use a multitude of cache types, and figure 2.5 is an example illustrating a software system utilizing multiple types of cache.

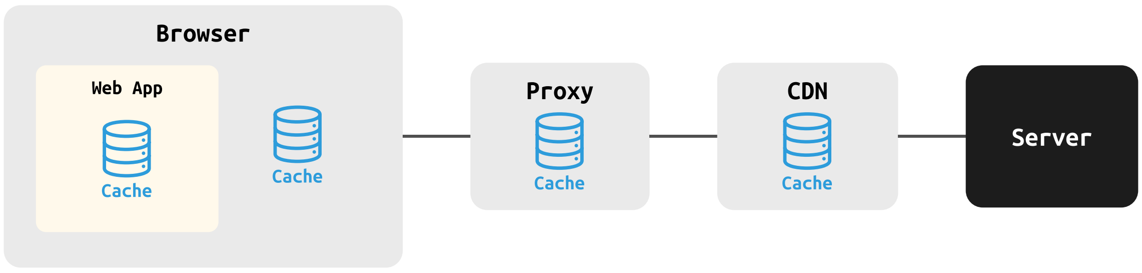


Figure 2.5: Overview of cache types in a software system, including shared caches and private cache. This thesis primarily focuses on the application data cache within a web application.

When referring to the term cache in the context of this thesis, we mean *server state cache*. Put simply, we refer to the process of storing a copy of server data on the client in hope of reusing this data in the future, instead of making a redundant network request. The location of *server state cache* can be seen in figure 2.5 inside the "Web App" container. This cache is stored *in-memory* as application data of a web application. Cache that is stored *in-memory* is physically located in Random Access Memory (RAM). RAM is by its nature a temporary storage, and it offers fast and direct access. An important note is that the term *server state cache* does not mean cache stored on the server, but rather a copy of the server data, stored on the client as *server state cache*.

When an application requests some data to present in the user interface, it can first ask the *server state cache*, and return the data if present. However, if the data is not present, the client must perform a full HTTP request targeting the server for data. The request might find the desired data in for example a proxy cache or in a CDN, thus returning the data without even touching the server. However, the different types of cache are built for different purposes and complements each other in order to build an efficient software system. We do not want requests to propagate farther than necessary, thus it is optimal to return data from the first available cache that match a requests' criteria.

Using data returned from any cache includes the risk of the data being outdated. Data that is outdated is often referred to as *stale*, in the context of cache. Nguyen et al. define stale cache as "resources which are out of date with respect to their master copies on the origin server" [24]. On the other hand, cached data that is not marked as out of date are referred to as *fresh*. How long cache is considered fresh, and the freshness of cache data are calculated can be determined in both the cache system utilized and developer configurations. The challenge of assessing whether data is fresh or stale is referred to as *Cache Coherence* [25]. This is important for server state management because we want to keep the server state that is cached on the client as fresh as possible, without making redundant network requests.

One of the main purposes of a cache system is to reduce superfluous request, while still presenting fresh data. However, a cache system can not guarantee that data *considered* as fresh, will always be up to date with the server, and neither that there will be no redundant requests. Thus, using a cache system can be seen as a compromise between having up to date data and system efficiency.

2.11 Networking

Utilizing cache effectively and communication with a server through an API require using network requests. Some of the advanced features implemented in server state management libraries are dependent on common patterns regarding networking. This section will describe the most common networking topics that are used in server state management libraries.

2.11.1 Retry

We are not always guaranteed to receive a successful response when sending network requests. There are many things that can go wrong when two parts are communicating over a network. However, retrying a failed request shortly after its failure does not necessarily ensure that the request will continue to fail. Retry functionality in networking have some commonalities with retrying in human interactions as well. Let us explain this with an analogy. You spontaneously visit a friends house and start knocking on the door, hoping that your friend is at home and able to hear you knocking. If your friend does not open, you would probably try to knock once again in case he hears you knocking this time. After a couple of failed tries you probably walk home, thinking to yourself that your friend is most likely not at home. Retrying network requests works much like retry in the real world.

When the client does not receive an answer from the server, we has to decide if it should try again, and how long to wait before retrying. The simplest option is to retry the request with a fixed time interval. As we would not want to retry forever if the request continues to fail, we usually specify a certain limit for the maximum number of retries. Alternatively, more sophisticated techniques can be employed to calculate the wait time, like utilizing an exponential backoff algorithm, where the interval between attempts is not constant but grows exponentially [26]. Fixed interval and exponential backoff are deterministic algorithms, but you could also use randomness to decide the time interval. A retry approach is not bound to use just one type of backoff algorithm deciding wait time. It is also a viable option to use a combination of backoff algorithms, like the Semi Distributed Backoff mechanism, which uses a combination of random backoff and deterministic backoff [27].

Retry functionality is very useful in server state management libraries, since the retry algorithm can be added as a configuration in the client responsible for fetching data. We will see later how different server state management libraries utilize retry functionality and what backoff configurations they provide.

2.11.2 Offline Features

Offline features in applications can enable additional features and use cases regardless of network connection. A common offline feature is to download media files onto a device enabling users to view the files later on without the need for a network connection. For example, music applications usually allows for downloading songs onto your device, which makes it possible to listen to your favorite music anywhere you go without requiring a network connection. Even though you have network connection, listening to a song that is already downloaded can possibly reduce network traffic by not needing to download the song again. However, downloading many songs that you do not listen to can result in excessive network traffic and unnecessary consumption of memory resources.

Advanced offline features can go beyond simply downloading files onto a device. They might allow users to perform actions that actually require network connection while still being offline. The actions might be perceived by the user as successful actions, but in reality the actions are saved on the client device, and will execute the actual network request once the client is back online. Let us look at some examples of such behaviour. Gmail enable users to send emails even though they are actually offline. The email is stored on the users device in the outbox, and sent automatically once the device reconnects to the internet. In a similar manner, Instagram allows users to post photos while offline.

Offline features are convenient for users, as it allows them to continue using the application even without a network connection, in contrast to applications that require an online status for all interactions. For example, people are empowered to spend time responding to emails while sitting on an airplane, or utilize applications in a parking basement with no or little network connection. Furthermore, offline capabilities can give very fast response times, since user triggered events does not require a network round trip. Mobile devices using a cellular network connection could possibly save money by downloading data when using a stationary network, like Wi-Fi, instead of their cellular network connection. There are many good things about an application supporting offline features, but it also requires careful planning ahead of implementation. An issue that quickly arises when discussing offline capabilities is "what should happen if the synchronization fails?". Before adding offline features, developers should consider whether they are worth the complexity and time investment required for implementation.

2.12 Streaming Data

Some applications has to present data that is changing frequently. These applications are dependent on showing up-to-date data, and will therefore not delay presenting the newest data. This type of data is usually labeled *real-time data*, and should reflect what the real value is as of *this* moment. Examples of applications depending on real-time data are chat, gaming, and IoT applications. What they have in common is that some of the value of their application lays in the freshness of the data. In multiplayer gaming, real-time data is crucial in order to obtain a sufficient gaming experience. Furthermore, there is not much value in a temperature sensor showing the temperature from hours ago, when the user is interested in knowing the temperature right now.

Streaming data refers to the processing of real-time data with low latency [28]. Streaming data has infinite length and is transmitted continuously in ordered sequences [29]. Due to the nature of streaming data being sent continuously, two communicating parts need a protocol supporting an ongoing connection. A notable example of such a communication protocol is WebSocket [30]. WebSockets enable real-time data exchange between client and server over a persistent connection, and is built into the Web API.

2.13 Pagination

A concept that quickly becomes relevant when dealing with server data is *pagination*, which simply means to manage large sets of data by dividing them into smaller portions. Pagination refers both to a UI design pattern and a server side pattern. Pagination as a UI design pattern involves building a user interface that does not show all the data at once. Instead one divides the data into smaller, more user friendly portions. A common trope is to have these portions be navigable by pages, hence the name pagination, but it is also possible to for example implement infinite scrolling. When using infinite scrolling, the "pages" are sowed together, perceived as a continuous list. Scrolling to the end of the list triggers the fetching of a new "page", just like an Instagram feed.

UI pagination is mostly concerned with the client user experience, and it can be implemented independent of where the data is stored. All the data can be on the client, with the pagination just taking care of showing it in a user friendly way. The implementation of UI pagination is not very relevant for this thesis. However, when the data is stored on a server, and fetched one “page” at a time, the server needs to implement server side pagination.

Server side pagination entails that the client only asks the server for the first portion of the data at first, and then for the next portion only if it becomes relevant. For example, a search result can show the 10 most relevant items, and maybe the user finds what it is looking for among these 10 first results. Only if the user clicks the next page will the client send a request for the next 10 items. This can help save bandwidth in cases where there is a lot of data on the server, but the client most likely just needs the first few items, or at least not all the data at once.

Implementing server side pagination demands collaborative effort from both the server and the client. The server has to implement a pagination pattern in the first place, enabling it to serve the data one portion at a time. The client need to ask for these portions in the correct way. There are different server side pagination patterns, two common ones being index-based and cursor-based. In index-based pagination, a request will contain the page number, and the server returns the data for this page. With cursor-based pagination, a pointer to the last item retrieved is given, indicating the current position in the data set. Subsequent queries can then use this position to serve the next portion of the data.

We will not delve deep into the different implementation details of the different patterns of server side pagination in this thesis. However, it is beneficial to have a grasp of what it is, as it is such a common data fetching pattern, and most server state management libraries will make a point of supporting it.

2.14 Research in Web Development

While the specific area of state management in JavaScript front-end frameworks has not been of much interest to researchers, as mentioned in chapter 1, there has been more investigations into the broader, related areas of web development, including JavaScript and its associated libraries and frameworks. In this section we will briefly touch upon some of this research. However, it should be noted that most of the research we will mention here is not directly relevant to our thesis, and this section will thus work to further point out the gap that exists in research regarding state management.

We can start by looking at some of the research that is being done in the field of JavaScript front-end frameworks. Lee and Busch examines how JavaScript front-end frameworks meets the requirements of modern web applications [31]. They found that the frameworks did not provide any performance boosts, but still managed to reduce the complexity of the code, easing their maintainability. On the other hand, sometimes frameworks may introduce complexity. Ferriera and Valente performed a study on the usage of React, and common bad practice code-patterns introduced in complex applications. Their research aims to assist in increasing the code quality of React applications [32]. Diniz et.al conducts a study to compare the performance of three JavaScript front-end frameworks: React, Vue and Angular, in an effort to aid the decision of choosing the right framework in web projects [33]. Still, as was pointed out in chapter 1, comparisons of state management libraries or examinations of the topic of state management is lacking.

Caching is a major area of computer science, and it is a reoccurring subject of research. As shown in section 2.10, caching encompasses multiple types of cache, each being the main subject of a field of research. Flores and Bedi explores the effectiveness of different caching policies in large-scale CDNs [34], and Ben-Ammar and Ghamri-Doudane investigate a new artificial intelligence-based caching scheme for CDNs to keep only the content that is expected to be of use in the near future [35]. Chhangte et.al. design a new method of caching on the edge, getting even closer to the user than CDN cache [36]. An attempt at introducing a new type of cache next to the browser is made by Heo et.al. to cache JavaScript bytecode, improving the performance of dynamic JavaScript web applications [37]. Given that caching is such an active topic in computer science research, we would think that caching in server state management libraries in JavaScript would also receive some attention, but alas, it does not.

This has only been a brief look at some of the research being done in the broader field of web development, and there is a wealth of research here. However, there is a noticeable gap when it comes to the specific area of state management, both directly, and indirectly via fields like caching, that otherwise is a thriving area of research. This is despite the fact that state management is a critical component of developing dynamic web applications, and that it is, as we have seen, cited as a major pain point for developers [2], [3]. This is the gap this thesis aims to address by providing a comprehensive exploration of state management in JavaScript front-end frameworks, with a particular focus on React.

Chapter 3

Method

The aim of this thesis is to explore the topic of state management in JavaScript front-end frameworks. Even though state management is a field that lacks a formalized general theory and scientific research, it is a practical problem developers face every day. There are many third-party libraries that attempt to offer practical methods for managing state effectively. These libraries might look different in syntax, architecture and functionalities, but they all converge towards a common goal: state management. The hypothesis this thesis is built upon is that a theory of state management exists *implicitly* within these libraries, and that it can be extracted, formulated, and made explicit through detailed analysis. Therefore, our research methodology consists of conducting a qualitative literature analysis of some selected state management libraries, before we try to generalize our findings to formulate a general theory of state management. This chapter presents and justifies the research design and methods employed. It details our process of library selection, our approach to the literature analysis of these libraries and the formulation of the general theory.

3.1 Research Design

Our methodology is inspired by Grounded Theory, a qualitative research methodology intended to generate new theories [38], [39]. Grounded Theory is relevant for this project, because our aim is to create a new theory of state management. You could say that we want to uncover the hidden theory that lies implicitly inside the different state management libraries, which makes Grounded Theory relevant, as it intends to *ground* its theories in the data.

However, Grounded Theory is a rigorous methodology with strict procedures and stages, leading to the generation of a theoretical framework. While we appreciate the guiding principles of Grounded Theory and its emphasis on letting theory emerge from the data, we found it not entirely practical or beneficial to adhere strictly to its methodology. Therefore, while our research methodology bears resemblance to Grounded Theory, it does not strictly follow it. A more detailed description of how Grounded Theory served as an inspiration for our methodology can be found in section 3.5.

3.2 Selection of Libraries

To ensure the theory we create is credible and relevant for as many developers as possible, we want to include the most used and popular state management libraries as part of our data set. Unfortunately, there does not exist any public record of all the available state management libraries. One reason for this is that most state management libraries are open source projects, published as npm[1] packages. Some libraries are created by companies, like Recoil [40] created by Meta, while others are created by individual developers, like Zustand [41] created by Daishi Kato. While this

[1] https://www.npmjs.com/

generates a lot of creative solutions for state management, it can also work to make the landscape of state management libraries unmanageable, and susceptible to rapid changes. Technically, there is no way to make sure we include every relevant library. Practically however, we can get a good indication by relying on weekly downloads, developer ratings and curated lists.

Some metrics that we used as a guidance to measure popularity of the libraries were GitHub Stars and npm downloads. GitHub Stars is a way for GitHub users to mark a project as something they like, and can give an indication to how popular a library is. Npm downloads represents how many times a library has been downloaded from npm[2].

3.2.1 Client State Management Libraries

Some client state management libraries has been around for a long time, and have been widely used for many years, often indicated by a large number of Stars and downloads. These libraries also often makes it into blog posts and curated lists of the "best" client state management libraries. We included three of these libraries: Redux [42], MobX [43] and XState [44]. However, recently there has been a surge of new state management libraries, that is often marketed as replacements for these "old" libraries. Jotai [45] and Zustand [41] are two popular examples, with Zustand actually acquiring more GitHub Stars than XState, and is still climbing. Other new libraries gain a lot of traction by being backed by big companies, like the aforementioned Recoil [40] by Meta.

While researching what libraries to include, we also decided to accept two libraries which did not mark themselves with a high number of Stars or downloads. Valtio [46] is another library by Daishi Kato, the author of both Jotai and Zustand. The reason we include Valtio as well is that it marks itself from the crowd by being one of the few state management libraries that handles state mutably. MobX is the only other library from our list that also does this, and we think that it is beneficial to include another library with mutable state. Being written by the same author as Jotai and Zustand also gives it some credibility. The other library was Elf [47]. Elf differentiates itself from the other libraries, as it bases itself on the principles of ReactiveX[3], elaborated further in section 4.9. Thus we think that Elf adds value, by making sure we include a certain amount of variety to the theory, even though it is rather new, and do not have the same amount of Stars or downloads as the rest[4]. The full list of the included libraries is shown in table 3.1.

Library	GitHub Stars (thousands)	Monthly downloads (millions)
Redux	58.9	36
MobX	25.9	4.5
XState	21.8	5.2
Zustand	23.8	3.2
Recoil	18.0	1.5
Jotai	10.9	0.79
Valtio	5.8	0.14
Elf	1.2	0.04

Table 3.1: Client state management libraries included in the project, along with their GitHub Stars[5] and monthly downloads[6]. Star count was recorded in November 2022, and monthly downloads were gathered in September 2022.

[2]This number includes automated build servers, downloads by mirrors and robots, and omit all the times a cached version was used instead of a fresh download. However, it still gives a good indicator of how many developers actually use the library. More details can be found here: https://blog.npmjs.org/post/92574016600/numeric-precision-matters-how-npm-download-counts-work.html

[3]https://reactivex.io/

[4]Akita is a library based on ReactiveX that has been around for a longer time and have more stars than Elf, but because they themselves recommends Elf in their documentation we decided to choose Elf: https://github.com/salesforce/akita/.

[5]Star count for the respective libraries were gathered from Github: https://github.com/

[6]Data on npm downloads were gathered from npm-stat: https://npm-stat.com/

3.2.2 Server State Management Libraries

One of the oldest and most used server state management libraries is Apollo Client [48], which had its initial major release back in 2017[7]. This was a couple of years after the open sourcing of GraphQL in 2015, and Apollo Client was primarily built to manage GraphQL APIs. During the same period, Meta developed their own library called Relay [49]. Relay is not included, as we wanted a newer GraphQL-based library in addition to Apollo. Therefore, URQL [50] is included, which was released later in 2019[8].

While Apollo, Relay and URQL are some of the earliest attempts at server state management libraries, they all focus on making GraphQL requests. They are possible to use without GraphQL, but this is their main area of focus. It was not until 2020 that we saw the rise of server state management libraries that were not primarily aimed at GraphQL. Tanstack Query [51] were one of the first[9], and it gained a lot of traction. It has the most GitHub Stars out of all the server state management libraries we have considered, hence it was natural to include it. Redux, the most used client state management library, released RTK Query [52] in 2021[10]. This is their answer to managing server state, and it was released as part of their Redux Toolkit library. It is difficult to get a realistic image of how many actually utilize RTK Query, since the number of downloads will include Redux users who does not use RTK Query as well, but we include it nonetheless. Lastly, we include SWR [53] on the basis of it being a very popular alternative, which also had its first major release in 2021[11]. The full list of the included server state management libraries is shown in table 3.2.

Library	GitHub Stars (thousands)	Monthly downloads (millions)
Tanstack Query	33.8	4.4
RTK Query	9.3	9.4
SWR	26.1	4.7
Apollo Client	18.6	10.9
URQL	7.9	0.9

Table 3.2: Server state management libraries included in the project, along with their GitHub Stars[12] and monthly downloads[13]. Star count was recorded in March 2023, and monthly downloads were gathered in February 2023.

3.3 Grey Literature in Library Analysis

In order to derive an implicit theory from the state management libraries, a systematic analysis of each selected library was conducted. Our purpose was to gain a comprehensive understanding of each library's features, principles and best practices. To best achieve this, the documentation of the libraries was used as the primary material for the review. These are "wiki-like" pages where library authors and contributors explain how to use the library and its underlying principles.

Library documentation falls into the category often referred to as "grey literature". As defined by Garousi et.al., grey literature encompasses materials "that are not formally peer-reviewed nor formally published" within the field of software engineering [54]. This type of literature is often overlooked in academic research, which traditionally favors peer-reviewed, published papers. Yet, it is precisely this category of grey literature that most practitioners in the field engage with, creating a distinct divide between academia and industry. Garousi et.al. encourage an increased use of grey literature in scientific research as a means to bridge this gap [54]. In computer science

[7] https://github.com/apollographql/apollo-client/pull/1519
[8] https://github.com/urql-graphql/urql/releases/tag/v1.0.0
[9] https://github.com/TanStack/query/pull/150
[10] https://github.com/reduxjs/redux-toolkit/releases/tag/v1.6.0
[11] https://github.com/vercel/swr/releases/tag/1.0.0
[12] Star count recorded from Github: https://github.com/
[13] Data on npm downloads were gathered from npm-stat: https://npm-stat.com/.

research, utilizing grey literature can be beneficial in gaining contemporary insights relevant to both practice and research, as highlighted by Williams and Reiner [55].

Given the nature of our subject - uncovering the implicit theory behind these libraries - we found it necessary to leverage grey literature for our analysis. While library documentation is not traditional academic literature, these documents are produced by experts in their field, offering invaluable insights into the current practices of each library. This approach provides us with an thorough understanding of the reasoning and principles guiding the library authors. Thus, the utilization of grey literature allows us to gain a realistic comprehension of the nuances of state management.

3.4 What is Documentation

Our main data source of the qualitative analysis is the official documentation of the libraries. However, there are no formal definition of what a library documentation should include or what structure should be used. This is one of the reasons that the format of the documentation of the different libraries can vary to a great extent. The topic of state management evolves rapidly, and there are new state management libraries being released almost every year. The format of the documentation is to some degree influenced by the other available libraries at the moment, and what their strengths and weaknesses are. Some libraries are created because of the lack of features in current libraries, or by a belief that the problem of state management could be solved in a better way. It is for this reason that the documentation format can differ based on the year they were released, what other libraries, concepts or principles they build upon, or what has changed in the technology itself. This section clarifies some of the different aspects of the documentation.

Redux and MobX are two of the oldest and most used state management libraries. They have in common that they have a comprehensive documentation. They provide thorough descriptions of the core principles and concepts of the library, and give clear answers to *why*, *what* and *how* the library solves the issue of state management. These libraries were some of the first to introduce their naming of the principles and share a wider understanding of the necessity of state management. One example is the concept of mutability. Redux uses an immutable data model, and MobX uses a mutable data model. Their documentation describes *what* mutability is, *why* they think it is advantageous to use and *how* it is implemented in practice.

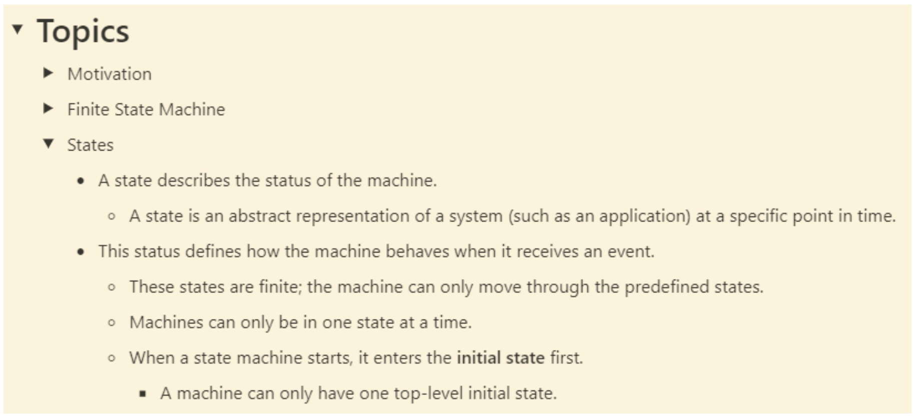
Libraries released in later years often builds upon the same principles and concepts of previously released libraries. A library may use some of the same principles popularized by Redux for example, add a new principle or two, and create its own implementation of it. If a library practices concepts that are already known in the developer community, there is less of a need to extensively describe the core principles in the documentation. A common pattern in documentations of newer libraries is therefore an increased focus on how the library is easier to implement, have a better developer experience, and solves the problem of state management in a better way. This shift places greater emphasis on implementation and reduces the focus on principles.

Some documentations have a larger part of implementation, and a small part about principles. Others have very much about principles and less about implementation. Our research tries to create a theory of state management which describes *what* state management is, *why* it is needed and to a lesser degree describe *how* it is solved. Therefore in our data analysis we have included data about *why*, *what* and something about *how*. Thus, code examples have been mostly omitted for our research, unless the code examples to a great extent shows or underpins the principles. We found that doing this abstraction was necessary to be able to create a generalized theory of state management, with informal data sources that have different documentation formats.

3.5 Library Analysis Approach

This section presents our analytical method, which we believe strikes a balance between thoroughness and efficiency. As mentioned in section 3.1, our method draws inspiration from Grounded Theory, particularly the process of *coding* - labeling segments of data into topics, which are then processed further to generate the theory.

Our approach unfolded in several steps. First, we briefly went through each library's entire documentation, to acquire an overview without delving too deeply into the details. Afterward, a more detailed examination was initiated, systematically coding segments of the text. A segment could range from a piece of a sentence to multiple sentences. The coding was done by copying the segments into a shared document, under a topic header. One segment could be copied into multiple topic headers, and sometimes the topics could be renamed during the process, to better describe the content. A screenshot of a few segments from the analysis of one of the client state libraries is shown in figure 3.1.



▾ Topics

- ▸ Motivation
- ▸ Finite State Machine
- ▾ States
  - A state describes the status of the machine.
    - A state is an abstract representation of a system (such as an application) at a specific point in time.
  - This status defines how the machine behaves when it receives an event.
    - These states are finite; the machine can only move through the predefined states.
    - Machines can only be in one state at a time.
    - When a state machine starts, it enters the **initial state** first.
      - A machine can only have one top-level initial state.

Figure 3.1: A screenshot of some of the segments that were copied under the topic of "states" in the XState client state management library.

This coding process was akin to the *open coding* phase in Grounded Theory. The essence of open coding is to remove preconceptions about the data, and extract concepts directly from the data itself. Given that our research consists of analyzing multiple libraries that all solve the problem of state management in different ways, we conducted the *open coding* phase for each library separately. This allowed us to capture the unique attributes and mechanisms of each library before drawing inter-library comparisons.

Following this, we read through the segments under each topic, checking for consistency, removing duplicates, and identifying whether additional data was needed to get a complete picture of the relevance of different topics. This phase deviated from the second phase of coding in Grounded Theory, *axial coding*, which seeks connections among codes and to identify common themes. Instead, we continued to treat the libraries separately, and summarized the theory of each specific library. By getting an overview of each library, we found it clearer to see which elements should be part of the general theory. These summaries will be listed in chapter 4 and chapter 5.

In brief, our library analysis approach are as follows:

1. Briefly review the library's entire documentation for a general understanding.
2. Perform a detailed reading, identifying key segments.
3. Transfer the identified segments into a shared document, categorizing them under appropriate topic headers.
4. Review the categorized segments, ensuring consistency, removing duplicates, and adding any necessary supplementary information.
5. Summarize the key points into an overview of the library.

3.6 Formulating the Theory

The final phase of our analytical process was the formulation of the theory. This drew inspiration from the *selective coding* phase of Grounded Theory, which aims to identify the codes that are essential to the theory being generated. Here we compared the different libraries while looking for connections to discern which topics were essential for the general theory of state management.

The result of this process is a pair of theories that formulates the core principles and methodologies employed in state management. These theories embody our response to the first research question, **RQ1** - the formulation of a general theory of state management. The theories are presented in chapter 6 and chapter 7, where we detail the main concepts and methodologies discovered through our analysis. We differentiate between those related to client state management, addressing **RQ1.1**, and those related to server state management, addressing **RQ1.2**.

Overall, this approach, inspired by Grounded Theory, facilitated a thorough understanding of the field of state management from the bottom-up, starting from the concrete implementations in each library and gradually building towards a generalized theory.

Chapter 4

Client State Management Libraries

In this chapter, we present summaries on our data on the selected client state management libraries for React. Our aim is to provide a comprehensive overview of each library's unique features, advantages, and limitations. As a starting point, we discuss state management with just React core. After this follows the client state management libraries, which include Redux, MobX, XState, Zustand, Recoil, Jotai, Valtio, and Elf. The methodology of our study, and the reasoning behind this selection, was covered in chapter 3.

We present a summary of each library, gathered by reading the respective documentations of the different state management libraries. These summaries primarily focus on how state is modelled, updated, and consumed by components. We hope to provide readers with a thorough understanding of the state management landscape for React applications. Our aim in presenting this information is to provide the reader with the basis of our data material, which will be integrated in chapter 6 to create an overall theory of client state management.

Unless otherwise stated, all information in the following sections are based upon the relevant library documentation. Additionally, the logo of each library is displayed in their respective sections.

4.1 React Core

Before entering the world of different state management libraries, it is necessary to understand what features React core offers regarding state management. Almost all of the state management solutions utilizes React state management in one way or another. The solutions that are closely coupled with React often utilize React's own features internally, and the solutions more loosely coupled with React do have to rely on React's hooks to access the state from within the components.

Managing State with Hooks

React's built in state management relies on hooks. Hooks are special functions that lets React function components access advanced React features. Whenever a React component re-renders, the whole component function is re-run. This means that any values that are defined inside the component function will be re-created for each render. This is a problem if you need some values, i.e. state, to be preserved between renders. This is where hooks come into play, and in the following sections, we look at some of the most important hooks provided by React to manage state.

The most basic hook is `useState`. `useState` takes an initial value as a argument, and returns an array with two elements. The first element is the current state, and the second element is a setter function that lets you change the state immutably. This state value is managed by React behind the scenes, instead of being stored inside the component. It is stored in a UI tree managed by React, which resembles the HTML Document Object Model. The state is connected to the components location in the UI tree. As long as the component is rendered at its location in the UI tree, React preserves the state. When the component unmounts, React discards the state.

One important thing to know about `useState` is that the current state value returned from the hook is not a dynamic value, but is captured as a closure. This means that the value is not connected to the "real" state, but is only a copy of the state value at the time of render. This is why you cannot modify the state directly, but have to use the setter function provided by `useState`, as this gives access to the real state and current value. It can also be confusing if you try to use the state value directly after setting it, since for the rest of that render, the value will be the same as it was at the start of the render. It is only after the component function is done executing, that the change of state will trigger the re-render, which will then collect the newest value of the state from the `useState` hook.

`useState` lets us preserve state between renders. For simple state management `useState` may be all you need, but in some situations you might want more control. The hook `useReducer` also returns an array with two elements, where the first element is the current state value. The second element however is not a state setter, but a dispatch function. In addition, `useReducer` requires two arguments, an initial value and a reducer function, which is in charge of managing the state updates. Instead of changing the state using a state setter, you modify the state by dispatching actions, which then gets handled by the reducer function. This approach is highly inspired by Redux which we cover in greater detail in section 4.2.

State Sharing

A problem that quickly reveals itself when you use these hooks for more than simple example projects is the need for sharing state between components. While some state only exist in a local scope, often the same piece of state is used in many different places. On the surface this is no problem for `useState` and `useReducer`, as both the state value and the setter, or dispatch, function can be passed to child components using props. This way you can just lift the state to the highest common ancestor component, and pass them down from there. Lifting state up to highest common ancestor often results in what we call prop drilling, as it can result in props being passed many layers of components because you need it at two different components low in the component hierarchy.

To address the aforementioned concerns, `useContext` and global state is introduced. With this you can define state values that you want multiple components to have access to, and then provide that value with a higher order component, which is a component wrapped around another component. Every descendant of the higher order component can access this state value using the `useContext` hook. If this higher order component is a top level component, the state becomes global to the entire application. You can also update this global state from the components, as the `useContext` hook returns two elements in an array, where the second element is a state updater function, just like with `useState`. Even though `useContext` gives you this state setter function, it is not intended to use with state that changes frequently. Whenever the value provided to the context is changed, React will automatically re-render all components that read this context. This may lead to a lot of re-renders, even though all components that listen to the context may not be affected by the change. This is why `useContext` is often recommended for values that are relatively stable.

4.2 Redux

Redux describes itself as "[. . .] a pattern and library for managing and updating application state, using events called 'actions'. It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion" [42]. The foundational building blocks of Redux are *state*, *actions* and *reducers*. Redux is a standalone library that can be used independently of which UI layer is used. This section will give deeper insights into how Redux stores data, and how data is updated and consumed.

The Redux pattern is supposed to enhance an understanding of when, where, why, and how the state in an application is being updated. The basic idea behind Redux is summarized as "[. . .] a single centralized place to contain the global state in your application, and specific patterns to follow when updating that state to make the code predictable" [42]. The centralized global app state is kept in a single store, and the store state is read-only for the rest of the application. The only way to update the global store is to `dispatch` an event called an *action*. Reducer functions are called in response to an action in order to update the global store.

State Updates

Actions can be thought of as events that occur in the app, which trigger an update in the state. Something happened, and we want the store to know about it. Dispatching an action is exclusively the only allowed way of updating state in Redux. Moreover, all components are allowed to `dispatch` actions. Actions are plain objects, containing the minimum amount of information needed to describe what happened. This is typically done with a type field - a unique name for the action - and an optional payload field that contains other values to be passed with the action.

Reducers are responsible for managing state updates in a Redux store, triggered by actions being dispatched. We can think of them as event listeners where the dispatched actions are the events. When an action is dispatched to the store, the store runs the reducer and checks to see if it cares about this action. If it does, it modifies the state accordingly. Reducers are pure functions that receive the current state and an action object as arguments. They determine how to modify the state, if necessary, and then return the new state.

Every Redux store has a single root reducer function that is responsible for handling all of the actions that are dispatched, and calculating the entire new state every time. Moreover, Redux recommends organizing your app folders and files based on features - code that relates to a specific feature or concept in your app should be co-located. This can be done by splitting the root-reducer into smaller reducer functions, based on the piece of the state they update. This makes the reducers easier to read and maintain, but they have to be combined into a single root-reducer at the creation of the store.

The reducer function can contain all necessary logic for calculating the next state, but since one of the goals of Redux is to make your code predictable, reducers cannot use asynchronous logic, have any side effects, or produce random values. This also makes the reducers simpler to test. If you need side effects you must introduce middlewares.

State Model

A *store* is a "container that holds your application's global state" [42], and it is the center of every Redux application. A Redux app always have a single store, and every store has a single *root reducer* function. For small apps, it may be sufficient with a single root reducer, but in larger apps it is common to split logic into separate reducers. The root reducer can be composed of multiple reducers through *reducer composition*. Composing reducers into one root reducer preserves the rule of only having one root reducer function in a store, and also enables the possibility of splitting responsibilities into separate modules.

State, *actions* and *reducers* are united through the store, and the store has several responsibilities:

- Hold the current application state inside
- Allow access to the current state
- Allow state to be updated via `store.dispatch`
- Register and un-register listener callbacks via `store.subscribe`

In a React application, any component are allowed to read from the Redux store. However, the only approved approach to cause an update to the state is through dispatching an *action*. Redux actions are defined declaratively - the action can describe and tell the store what event has happened in the application, and the store will update the state accordingly to the action. After a state update has happened, the store will notify all components that are listening for a specific part of the store.

State Consumption

Selectors are functions that extract specific pieces of information from the store state. A selector takes the root state as input argument, and returns the specific part of the store that is needed. Selectors can also return derived values based on the current state. If the derived selectors are doing processor-heavy computations, it can be an advantage to use a memoized selector, which will memorize previously calculated results. Redux Toolkit includes memoized selectors based on the library Reselect[1]. This accepts selector functions as inputs, and uses the result from the input selectors to calculate the output. If the result from the input selectors has not changed, it just returns the memoized value instead of recalculating the output selector.

Redux uses a uni-directional data flow. An example of Redux' uni-directional data flow would be like this: a component dispatches an action. The store runs the reducers, and the state is updated based on what occurred. The store notifies the subscribed components, causing the components to re-render based on the new state.

Middlewares and Enhancers

The functionality of the Redux store can be extended using *middlewares* or store *enhancers*. Middlewares form a kind of pipe around the `dispatch` method, which means when we call dispatch, we call the first middleware in the pipeline. This means you can extend the store to accept actions that are not plain objects. Middlewares can also introduce side effects, for example by dispatching actions through thunks. A Redux *thunk* is a piece of code that does some delayed work, which means it can contain asynchronous logic. A thunk function returns a `Promise`, which our components can wait for if they depend on the thunk result.

Compared to middlewares, enhancers are even more powerful as they wrap around the entire store instead of just the dispatch method. An example of a store enhancer is developer tools, for example allowing us to replay actions manually without the app being aware of it happening. The Redux ecosystem includes the browser addon Redux DevTools, which shows a history of what actions were dispatched, what payload were sent with the action and how the state changed after each dispatched action. Redux DevTools also contains the powerful feature of *time travel debugging*, which gives developers the possibility to restore the state to any given point in time and manually replay actions.

[1] https://github.com/reduxjs/reselect

Toolkit

The recommended way of writing and using Redux has evolved over the years. In 2019, Redux Toolkit was released, and it is described as “The official, opinionated, ‘batteries included’ toolset for efficient Redux development”. Redux Toolkit was created to specifically eliminate the “boilerplate” from hand-written Redux logic, simplify standard Redux tasks and includes recommended packages and functions that are essential for building Redux apps.

Immutability and Normalization

Redux reducers are not allowed to directly mutate the state, and has to perform immutable updates. Redux depends on immutable state updates for its features to work properly. There are several reasons for why mutable updates cannot be performed:

- It breaks the ability to use time travel debugging.
- It causes bugs such as the user interface not updating properly to show the latest values.
- It makes it harder to write tests.
- It is harder to understand why and how the state has been updated.
- It goes against the intended spirit and usage patterns for Redux.

When the size of a Redux application grows, it is normal to store data in a normalized form. This makes sure there is only one copy of each piece of data, and makes it easier to lookup specific items, as each item is stored in an object with an ID as key. This way you can also refer to other items by ID instead of copying the entire item. To help with this, Redux Toolkit has built-in utility functions to manage normalized data.

4.3 MobX

MobX goes against the grain by using mutable data as the basis of their state. They use a *reactivity system* to *track* and *react* to state changes. This system is designed to make sure calculations and data is updated automatically, only when the state it depends upon is updated. This enables you to have centralized data that you can use to derive everything you need. As is their philosophy: “Anything that can be derived from the application state, should be. Automatically” [43].

State Consumption

To enable MobX to track your state, it has to be marked as *observable*. However, MobX do not track *unused* properties. Consequently, you have to read the property inside one of MobX’s *tracked functions* to make it start tracking the property. MobX tracks value access, not values. An example of one of these tracked functions is `autorun`, which accepts a function as argument. `autorun` re-runs this function every time something it observes changes. These observed properties are tracked automatically by MobX just by reading one of the properties of the observable state inside this tracked function.

State Model

Since the state of MobX are just properties that in one way or another are marked as observable, it does not matter where your state lives. You can define local observable objects inside your components far down the component tree, or you can create big classes as stores at the top, and pass them down as props or using React Context. While the team behind MobX promotes an object-oriented approach to designing the state, MobX itself is unopinonated, and everything can be made observable.

Derived state can be achieved by utilizing what MobX calls `computed` values. These are values calculated from pure functions, based on the current state. The `computed` values are updated automatically and synchronously, which makes it close to impossible to observe values that are out of sync, as long as MobX is used as intended. You can even inspect values directly after modifying them.

State Updates

To modify the state you run an *action*. An action is a function that changes the state, and makes sure MobX propagates those changes to all the tracked functions, synchronously. MobX batches state updates that happen inside an action, so the observers are not notified until after the action has finished, and all state updates are complete.

Usage with React

MobX is independent of the UI layer, and you could theoretically implement much of the application logic before you start developing the user interface. When using MobX with React, you wrap the components that consume state in a higher order component, called an `observer`. This is almost the same as wrapping the component in `autorun`. Now MobX tracks which observable properties are accessed in the component, and re-renders when the accessed value changes. To share state globally you can pass observable objects or classes using React Context. Since you're not changing the object references, just the properties directly, this will not cause un-necessary re-renders, but instead lean on MobX to keep track of the changes.

To summarize, MobX uses mutable, observable state. MobX keeps track of who is reading the state properties, and makes sure to notify observers whenever the value changes. To update the state you execute pure functions marked as actions. React consumes the data by wrapping components in a higher order component, which makes sure MobX keeps track of the data that is read, and causes re-renders accordingly.

4.4 XState

When we talk about state in relation to state management, we often mean the values that are stored, accessed and manipulated. In XState, this is referred to as *extended* state, or *context*, and is not the main area of concern. When we talk about state in XState, we refer to the status of finite state machines that describe our data. Instead of having a state variable called "loading", and another called "data", XState could have a state node called "loading" and another state node called "loaded", with the data inside the context of the "loaded" state. This makes it impossible to access the data before the loading is completed and the machine is in the "loaded" state. Because of this, XState can be used to achieve a greater control of your application, making sure to avoid for example contradictory state, which could result in reading data that is not loaded.

XState adheres to the official W3C State Charts XML Specification [56], and David Harel's statecharts formalism [57]. This makes XState one of the few libraries that actually build upon formal definitions and standards.

Finite State Machines

By defining finite state machines for an application, explicit definition and specification of intended behavior is required. This can be accomplished visually using the XState visualizer, or by coding it manually. Instead of defining one large machine for the application, XState recommends defining multiple smaller machines to avoid complexity. It is also possible to nest machines, and have certain machines that activates only when in a specific state of another machine.

The states of the Finite State Machines are defined as state nodes. These can be *atomic* state nodes, *compound* states that have child states, or *parallel* states that have child states that can be active simultaneously. It is also possible to define *final* state nodes that represent a terminal state, and *history* state nodes that can help implement history features with the machine.

State Updates

The change from one state to another state is called a transition, and transitions are always caused by what XState calls *events*. Events are objects, containing a type property to signal what type of event it is. Events can also contain other properties with data relevant for the event. Events and transitions are deterministic, meaning that a specific event combined with a specific state will always trigger a specific transition, leading to a specific new state.

There are different types of transitions, depending on how you configure them. One of these types are *guarded* transitions. Guarded transitions contains guard conditions that have to be true for the transition to work. If the condition return false, then the machine will not transition to the new state. Another type of transition are *delayed* transitions. This transition will happen after a specified amount of time being in one state, without being triggered by an event.

Side Effects

To achieve side effects using XState you use *actions*, which are functions you can trigger on entering or exiting a state, or during a transition. Actions should not have any impact on the machine itself, and once the machine has fired an action it will just move on.

XState encourages the use of multiple state machines, but sometimes you may want to have these machines interact with each other. Machines can communicate with each other by sending messages. The messages are in the form of events, and can be sent using special actions provided by XState.

State Model

All this talk of transitions and state nodes and actions might almost make you forget about the state values themselves, which XState calls *extended* state. When describing the qualitative state of the system using XState, you still need to handle application data, and this can be stored inside the context property of the machine. To update the context data you use a special action from XState called *assign*. This way you should never update the context externally from the machine, and every context change happens in response to an event.

Usage with React

XState is independent of UI layer. To use a machine inside a React component, XState provides custom hooks. XState provides a hook to get a stable reference to a machine, which lets you pass it around using React Context without causing unnecessary re-renders. To use this machine in a component you could use the useActor hook, to listen to changes to the whole machine, or the useSelector hook to only listen to a part of the machine. It is also possible to start a machine directly inside a component using the useMachine hook, to manage local state.

4.5 Zustand

Zustand is supposed to be a state management solution that is small, fast and use the minimal amount of features to fulfill application needs: “A small, fast and scalable bearbones state-management solution” [41]. Zustand claims not to demand much boilerplate code, and is very unopinionated.

State Updates

An action is a function that is defined on the root level of a store. The actions change the state by using a custom setter function from Zustand. Actions are the only way to update the state in a Zustand store. Asynchronous actions works just the same way as synchronous actions. If an action is marked as asynchronous, the action will wait to execute the state update until all promises in the action are fulfilled. Actions also have a method to read from the current state in the store. This is useful when a state update is relying on the current state to determine the next state.

Zustand updates state immutably. The method for updating state in Zustand merges state, but the built-in `merge` method only works when merging state in one level. If the state is a deeply nested object, one have to merge the values explicitly. Zustand have made Immer[2] available as a middleware to simplify the process of updating deeply nested state.

State Model

Zustand stores data in a centralized store. The store supports any data type, like primitives, objects and functions. It is recommended to co-locate actions and states within the store, to create a self-contained store with data and actions together. Although co-location is recommended, Zustand is unopinionated, and another approach is to define actions external to the store. This makes actions callable without a hook, and it facilitates code splitting. An application can have a single or multiple stores. If you have multiple stores, the main store can be divided into smaller individual stores to acquire modularity.

State Consumption

State in a Zustand store is accessed with a hook, that takes a *selector* as input, and returns the selected piece of state. A selector in Zustand is a function that takes a store state as input, and returns a property of that state as output. Selectors can either be defined inline inside the store hook, or defined outside of the component. In addition, Zustand also provides a function that auto-generates selectors from a store, called `createSelectors`, removing the need to manually write selectors. When a property in the state is selected inside the store hook, the component will re-render when the selected property changes. By default, Zustand detects state changes with strict equality, which is efficient for atomic state picks. One can also provide the selector with any custom equality function to gain more control over re-rendering.

Middleware

A Zustand store can be functionally composed any way you like with middlewares, which can give custom functionality to setters and getters in a store. State and actions in a store can be wrapped inside a chain of middlewares. An example of a middleware is the persist middleware, which can store the data through any kind of storage, such as `localStorage` or `sessionStorage`.

[2]https://immerjs.github.io/immer/

4.6 Recoil

Recoil was created by Facebook, and it is today marketed as a project in Meta Open Source[3]. Recoil is simply described as a state management library for React, and its intention is that the API and semantics should be as closely related to React core as possible. React's built in state management has some limitations which Recoil tries to solve, such as unnecessary re-renders through prop drilling and that it is "[...] difficult to code-split the top of the tree (where the state has to live) from the leaves of the tree (where the state is used)" [40].

State Model

Atoms are units of state in Recoil, and they contain the source of truth for the application state. An atom has a default value which is used as initial state the first time the atom is used. A component can read the value of an atom with custom Recoil hooks, which implicitly subscribes the component to the atom. Any updates to that atom will result in a re-render of the component. The hook `useRecoilState` returns the state and also a method for updating the atom, just like `useState` in React. Each atom needs to be given a unique key among all atoms in the application, which is used for things like debugging and persistence.

State Consumption

Selectors are pure functions that accepts atoms or other selectors as input, and outputs a transformed state based on its input. Components can subscribe to selectors in just the same way as they subscribe to an atom. When the value of any input to a selector changes, the selector will be re-evaluated and update all components subscribing to the selector if the value changes. Derived state in Recoil is achieved through selectors. Since selectors accepts both atoms and selectors as input, they can be used to create nested derived state. A selector may do its state transformation both synchronously or asynchronously, and the latter option make selectors a great way to incorporate asynchronous data in the Recoil's *data flow* graph. The data flow in Recoil is described as a directed graph, where state changes flow from atoms through selectors into components.

Side Effects

An atom can be created with an array of effects, which is called *atom effects*. Initialization, synchronization and administering side effects are some features of atom effects. Each atom can define and compose their own policies of how the effects should behave, and some use cases for effects are logging, history, persistence or state synchronization.

[3]https://opensource.fb.com/projects/

4.7 Jotai

Jotai uses an atomic data model inspired by Recoil, and takes a bottom-up approach to React state management. State is made by combining atoms, and atom dependency is the way Jotai optimizes rendering in React. Jotai can work as a replacement for React `useState` and `useContext`. Its API is designed to be primitive for ease of learning, and is unopinionated and flexible

State Model

A piece of state is represented by an *atom* in Jotai. Atoms are considered to be the building blocks of your app state. To create a primitive atom, an initial value is provided to an *atom config*. Atoms in an application will be distinguished and identified by their object referential identity.

Jotai supports derived state with derived atoms. Derived atoms are divided into three different categories: read-only, write-only, and read-write. Since atoms are building blocks, we can implement complex and complicated logic by composing atoms as derived state. Derived atoms are reactive, and will be refreshed any time one of their tracked dependencies changes.

State Consumption

Atoms are shared globally through *providers*. A provider component contains a store, and it provides atom values for its child components, functioning similarly to React Context providers. A component can read and update state from an atom with the `useAtom` hook, which returns a tuple with the atom value and an update function. This looks very similar to React `useState` hook.

When an atom is created through an *atom config*, there is no value associated with the atom. The initial value only gets stored in a provider state once the atom is consumed in a mounted component through custom Jotai hooks. On the other hand, when all components consuming an atom are unmounted, the value in the state is garbage collected and removed from the provider.

One of the main goals of Jotai is optimize re-renders in React. Combining atoms and creating dependant atoms are a way to enforce that a component only have access to the least amount of data it needs: "Optimized renders are based on atom dependency" [45]. Unnecessary re-renders are reduced by having a component only being subscribed to the least amount of state it actually needs.

Asynchronous Data

Atoms in Jotai can be both asynchronous and synchronous. Atoms become asynchronous if either the read function or one of its dependants are asynchronous. Jotai leverages React Suspense to handle asynchronous data flows. However, one can also avoid using React suspense and manage asynchronous data flows by using Jotai's *loadable* utility function. Loadable wraps an atom inside a finite state machine, containing information about whether the atom is currently loading, if a request was successful and contains data, or if the request encountered an error response.

Jotai supports debugging through either React DevTools or Redux DevTools. React DevTools is suitable for simple debugging such as reading atom values. For complex debugging requirements such as time travel debugging and setting values, Redux DevTools would have to be used. Atoms can be persisted with a custom storage function, and it supports different storage types such as `sessionStorage`, `localStorage`, or the URL hash.

4.8 Valtio

Valtio is a state management library that aims to have a minimal API, and uses `Proxies` [58] to keep track of state. It is one of the newer libraries, and its documentation is pretty scarce. Because of this we supplemented with a couple of blog-posts by Daishi Kato, the creator of Valtio [59], [60]. Valtio uses mutable state, and when you create a proxy object with Valtio, the object knows when it has been updated. You can subscribe to pieces of this object, and Valtio takes care to notify listeners on changes. This works to make sure Valtio has render optimization out of the box.

State Model

In Valtio, state is stored as objects which are wrapped inside proxy objects. The proxy objects can mostly be treated as normal objects, but behind the scenes they will keep track of changes using a version number. You cannot store whatever you want inside these proxy objects, but "normal" data types and serializable objects are generally good. Since the state is just objects, you can choose whether to define it outside or inside a React component. When defined inside React, it can be shared among components using React Context.

State Updates

To modify the proxied object you modify the object directly. Nothing special is needed to make this work, as the proxies takes care to notify listeners on modification. The team behind Valtio does, however, recommend placing the code to modify state inside functions you can call actions, for code splitting. There is some form of batching to make sure the changes inside the same event loop are batched.

State Consumption

To read the state you can just read from the object directly. If you want to read it inside the render-function of React you use the hook `useSnapshot`. This returns a snapshot of the state, keeps track of which properties of the state you read, and triggers re-renders when those values changes. The fine-grained render optimalization is thus achieved by this hook, not by Valtio's proxies alone, as these only track version number of the object as a whole.

Valtio also supports derived data by custom derived proxies. Both in the form of computed values, that use the proxies own properties to compute new values, and in the form of derived proxies, that can get the values of other proxies, and re-run it's own computation when those values changes.

Properties in the proxy objects can have `Promise`s as values, which means Valtio comes with asynchronous support. These `Promise`s will resolve in snapshots, which makes it compatible with React Suspense.

4.9 Elf

Elf is a state management solution that builds upon RxJS[4], which is the official implementation of ReactiveX[5] for JavaScript. ReactiveX is an API for managing asynchronous streams of data or events using *observables*, which notifies subscribers of any new data or event. The store in Elf is a *subject*, which is a special kind of observable that can both publish changes, and listen to other observables. Elf provides custom RxJS operators to query and update the store. Elf also comes with customization options for the store to easily use design patterns like entity store, which simplifies the process of doing create, read, update and delete operations on a normalized store.

State Model

You can create multiple stores, and they exist outside of the React component tree. You initialize the store with an initial value, and then specify features you want to configure the store with. Here you can specify that you want it to be an *entity store* for example. Since an entity store uses a predictable structure for the data, it is possible to use pre-made queries and updater functions to select, update and delete entities in the store. You can also specify that you want the store to contain UI state. This makes Elf create UI entities in the same store, but separate from the normal entities, so you don't have to crowd the entity store with data that is not relevant for the business state. Elf also provide functions to further extend the store with related data for the entities.

State Updates

Elf uses an immutable data model, and to update the store you can use the update method of the store. This takes an updater function, which receives the state as a parameter, and returns the updated state immutably. Elf provides some pre-made updater functions, like the entity updaters already mentioned, and also a couple to simplify immutable updates to objects.

State Consumption

To listen to the whole store you can subscribe directly to the store, or you can create custom observables that select a part of the store to listen to. These observables will only fire when there is a new reference, and they can be customized with the full extent of RxJS operators. It is through these custom observables you create derived data. In React components, you listen to these observables by using the provided `useObservable` hook. When the observable receives a new value, the component will re-render, which means that by optimizing the observables you minimize re-renders of the component.

Side Effects

Elf does not enforce one style when it comes to side effects, but it does provide a separate package to manage this, called effects. With the effect package there are two different approaches you can take, one using actions and effects, and the other using effect functions. With actions and effects you define actions that can be dispatched, and register effect functions that gets triggered when you dispatch certain actions. The other approach utilizes effect functions, which can be triggered without using actions.

Elf provides pre-built solutions for common features such as persisting the store, for example to `localStorage`, and a history function with the ability of undoing and redoing store modifications.

[4]https://rxjs.dev/
[5]https://reactivex.io/

Chapter 5

Server State Management Libraries

Following our summaries on the client state management libraries, we now present summaries on our data on each of the server state management libraries. Again, our aim is to provide a comprehensive overview of each library's unique features, advantages, and limitations, as well as similarities. The libraries include Tanstack Query, RTK Query, SWR, Apollo Client, and URQL. The methodology of our study, and the reasoning behind this selection, was covered in chapter 3.

We present a summary of each library, gathered by reading the respective documentations of the different server state management libraries. Our aim in presenting this information is to provide the reader with the basis of our data material, which will be integrated in chapter 7 to create an overall theory of server state management.

All information in the following sections are based upon the relevant library documentation. Additionally, the logo of each library is displayed in their respective sections.

5.1 Tanstack Query

Tanstack Query markets itself as "the missing data-fetching library for React" [51]. It provides a simple and declarative approach to fetching, caching, synchronizing, and updating server state in an application. Queries and mutations can be monitored with detailed status information, including the ability to differentiate between loading, error, and success states, in addition to a separate *fetch status* to track currently ongoing fetch operations.

Querying Data

When it comes to querying data, Tanstack Query does not have a built in method for fetching your data, but instead lets you pass in a *query function* per query, or globally. The query function makes the request, using for example the browser `fetch`, and Tanstack Query takes care of "the rest". This includes caching, de-duplicating requests, re-fetching stale data, query retries, cancellations, and more. The query function can use any asynchronous data fetching client, as long as it returns a `Promise`. This means you can use Tanstack Query for both REST APIs and GraphQL APIs, or anything else, as long as the query function returns a `Promise`.

Every query in Tanstack Query has a `status`, which gives information about whether or not the query have any data. The `status` state can have one of three different values: *loading*, when we do not have any data yet, *error*, if an error has occurred, or *success*, when we have successfully received data from the request. `status` can be used to check if we can show the data in the user interface, or if we instead should show for example a loading spinner. After we have received data, and `status` is in the *success* state, it will stay in the *success* state going forward, as long as we do not remove the data from the cache, or change the query key.

While `status` gives information about whether or not we currently have any data, sometimes we want information about whether or not we currently are fetching. We can use the *loading* status to indicate that we are performing the initial fetch, but after we have received the initial data, `status` will just stay in *success*, not giving us any more information. Because of this, Tanstack Query exposes another status element, namely `fetchStatus`. `fetchStatus` tells us whether the query function is currently running or not. It too can be in one of three different states: *fetching*, *paused* or *idle*. Using the combination of `status` and `fetchStatus`, it is possible to get detailed information about the status of the query, and give appropriate feedback to the user.

In addition to a query function, every query needs a query key. Queries are uniquely tied to a query key, and you can configure queries per key or on a global level. The query key of a query have to be an array, and may consist of everything from a single string to complex nested objects. Query keys uniquely describe the data it is fetching, and the queries are cached by the query keys. By default, if you have a query that fetches a list, and another query that fetches one element of that list (using two different query keys), Tanstack Query stores them as separate entries in the cache. However, it is possible to to circumvent this by providing initial data to a query, and use the cache of another query to provide this initial data. You can access and edit the data in the cache using the query keys.

Mutating Data

Queries are generally used for requesting data, and if you want to edit data on the server, Tanstack Query offers *mutations*. Mutations are similar to queries, but they do not need a query key, as they are not cached automatically or connected to any queries. Where queries are triggered automatically on render, mutation gives you a function to call when you want to perform the mutation. The mutation also has a `status` state, which can be either *idle*, *loading*, *error* or *success*. Mutations also allow you to define lifecycle methods to configure additional behaviour. These are `onMutate`, `onError`, `onSuccess` and `onSettled`. With these you can implement features like optimistic updates or logging. Even though the mutation itself does not need a query key, it can use query keys to invalidate queries that are affected by the mutation to ensure that the user interface reflects the changes.

Tanstack Query aims to give pagination features that just works out of the box. By including page information in the query key, every page is cached separately, giving you working pagination. However, this would lead to the query `status` jumping between *loading* and *success*, since the key changes and it requests new data. Tanstack Query fixes this with an option called `keepPreviousData`, that will make the query use the data from the last page, until the data from the new page has arrived. Tanstack Query also enables you to define functions for calculating the parameters for the next page. You can populate the cache with data before any components are reading a query by using *prefetching*. Prefetching can for example be used together with pagination to request data for the next page before the user actually has requested the page.

When it comes to render optimization, Tanstack Query tracks which fields you access, and only re-render if those fields changes. When multiple components use the same query, the updates are batched, resulting in just one re-render. Also, Tanstack provides dedicated DevTools that enable developers to view queries, mutations and check what data is currently cached.

5.2 RTK Query

Redux Toolkit, as described in section 4.2, ships with an optional addon for data fetching and caching, called RTK Query. RTK Query is UI-agnostic, which means that similar to Redux Toolkit, you can use it with any UI layer. The fetching function used to perform network requests is flexible, enabling developers to use RTK Query with both RESTful APIs and GraphQL APIs.

Querying Data

As the name implies, querying data is the most common use case for RTK Query. It ships with a default `fetch`-wrapper called `fetchBaseQuery`, but this can be replaced by any data fetching library to support data fetching from any back-end service. The fetcher that is used can be defined either per API-slice, or per query endpoint if necessary, which empowers developers to seamlessly integrate data from different urls, fetcher methods and back-end services.

RTK Query allows for separation of concerns by defining the API ahead of time in a centralized API-slice outside of the components. This enable components to easily reuse data queries since they are not owned by any component. An RTK Query API slice is created with various configuration options and an `endpoints` object. For smaller applications it may be sufficient to have all query endpoints inside one API-slice. However, to maintain control and structure for larger applications it may be useful to split API requests into multiple different API-slices. RTK Query encourages to split endpoints into different API-slices based on features, similar to the slice-pattern from Redux Toolkit.

Each specific endpoint is defined inside the `endpoints` object in an API-slice, by using a builder function to return an endpoint object. This object must contain either a query callback to construct the URL from the given parameters, or a query function that can perform custom asynchronous logic. In addition, you can provide additional options to configure the endpoint behavior such as transforming request responses, handle errors, or tag the data from the endpoint to enable automatic invalidation of the request. You can also define custom lifecycle callbacks for when the query is started and when a cache entry is added.

In addition to being UI-agnostic, RTK Query provide additional hooks for React. These hooks are automatically generated with custom names based on the `endpoints` definitions in the API. For example, an endpoint field named `getPayment` will generate a hook named `useGetPaymentQuery`. Inside the component, using the generated hooks, you can pass additional query options, like skipping automatic execution of a query, setting a fixed polling interval, selecting only a subset of the query data or overwriting the re-fetching rules of the endpoint. The query hook return the values `data`, `error` and the status of the request. The request status is given by derived booleans such as `isUninitialized`, `isLoading`, `isFetching`, `isSuccess` and `isError`. These booleans are useful for informing the users about the status of the data, whether it is loading, re-fetching, or if an error occurred.

Mutating Data

Mutations are used to send data updates to the server and reflect the changes in the local cache. Similar to queries, mutations accept a query callback constructed of the mutation URL and query parameters. In contrast to queries, mutations are usually not automatically triggered when a component first mounts. Therefore, the mutation hook returns a tuple, consisting of a `triggerFunction` and the mutation `result`. When the `triggerFunction` is called, the mutation is executed. The `result` object contains the data returned from the last trigger response, errors, and derived booleans regarding the request lifecycle such as `isUninitialized`, `isLoading`, `isSuccess` and `isError`. As mentioned earlier, query endpoints are tagged with a `tagType`, which commonly represents an entity in the database, e.g payments. The mutation definition also accepts a list of `tagTypes` so that a successful mutation can invalidate queries with matching `tagTypes`.

Caching

One of the main features of RTK Query is its management of cached data. Data returned from a query will be stored as cache in a Redux store. The endpoint definitions and query parameters automatically derive the `queryCacheKey`, which is an identifier for the data in the cache.

The data will remain in the cache as long as there is at least one active subscriber to the data. When there are no active subscribers, the corresponding section of the cache will expire, and automatically be removed after a certain period of time. This period of time can be configured manually. Moreover, the cache can be manipulated manually by using the utility function `updateQueryData`. In combination with the lifecycle methods, this can be used to achieve optimistic or pessimistic updates

When a new component subscribes to an endpoint, a new cache entry is added based on the given `queryCacheKey`. If another component subscribes to the same `queryCacheKey`, it will return the data from the corresponding cache entry. RTK Query provides an `onCacheEntryAdded` lifecycle callback for individual endpoints. This contains information about whether the cache data is loaded or not, and information about when a cache entry is removed due to no components subscribing to that data. Using the information provided in the lifecycle callback, you can implement streaming updates, using for example WebSockets [30], that will subscribe to the data as long as there are components subscribing to the cache entry. When there are no components subscribing to the cache entry, the `onCacheEntryAdded` callback is notified and you may safely close the WebSocket connection.

The most common features in RTK Query are described above, but it also includes a variety of other features like prefetching, request deduplication and automatic code generation. RTK Query is built upon the core modules from Redux, making it very easy to integrate RTK Query with Redux or Redux Toolkit. For example, you can catch and handle errors from all requests in an RTK Query API by creating a traditional Redux middleware. However, it is important to note that RTK Query can also be used as a standalone service for managing server state, without using Redux for managing client state.

5.3 SWR

SWR is an abbreviation for Stale-While-Revalidate, which is a HTTP cache invalidation strategy [61]. When using the SWR-strategy, you first return the “stale” data from the cache, before you send a revalidation fetch request to the server, and then update the application with the “fresh” data. Using this strategy by default, the SWR library aims to give you a user interface that is always fast and reactive. By defining reusable data hooks with SWR, you can use these hooks in your components, making them declarative, and keeping the business logic outside of the components. Different components making the same requests will automatically be deduplicated, cached and shared by SWR.

Querying Data

SWR utilizes a simple API, and for the most part you will use one of two hooks: `useSWR` for queries and `useSWRMutation` for mutations. When querying data with `useSWR` you pass it a key and a fetcher function, as well as an optional *options* object. The key, serving as an identifier for the data, can take the form of an array encompassing potential parameters for the query. This key is then passed into the fetcher function, which can then use these parameters, and whatever asynchronous fetching library desired, and return some data. The `useSWR` hook return value also keeps things simple. It returns the data, the error if an error occured, and two status-booleans: `isLoading` and `isValidating`. `isLoading` is true when there is an ongoing request and there is no current data. `isValidating` is true whenever there is an ongoing request. This simple setup still enables you to show different UI indicators when there is no data, and when it is revalidating. `useSWR` also returns a fifth value, which is a `mutate` function.

Mutating Data

When you want to perform mutations you use the `useSWRMutation` hook. This hook accepts a key, which tells SWR to invalidate the cache of that key after the mutation. This way SWR will make sure to automatically revalidate queries with that key. Instead of a single key, you can pass in a predicate function to invalidate all keys that match the predicate. `useSWRMutation` also takes in a fetcher function, commonly implemented to make some mutation on the server, and return the modified data. The hook returns the data and error, same as `useSWR`, but just one status boolean, `isMutating`, which is true if there is an ongoing mutation. It also returns a trigger function, to actually trigger the mutation, as it does not fire automatically when the component mounts. You can pass options both to the hook and the trigger function.

As mentioned, the `useSWR` hook returns a `mutate` function, and SWR also includes a global `mutate` function. You can achieve much of the same using any of these, but one way to think about it is that `mutate` is used to manually update the cache. With the global `mutate` you can pass any key or predicate, but the one returned from `useSWR` is bound to its key. Furthermore, if you mutate one or more keys without any data, SWR will revalidate those keys. `useSWRMutation` on the other hand, is used to make a mutation request to a server. `useSWRMutation` will not automatically modify the cache, but instead invalidate the keys on a successful mutation. While this description fits with their default behaviour, both `mutate` and `useSWRMutation` can be configured to work much like each other. `mutate` and `useSWRMutation` share many of the same options, like whether to use some optimistic data while waiting for the async mutation, and if the cache should rollback on an error. In addition to some shared options, `useSWRMutation` has some lifecycle callbacks, where you can specify behaviour if the mutation is successful, or if the mutation results in an error.

Configuration and Advanced Usage

While you can configure SWR on a hook by hook basis, it is also possible to provide some global configurations, using an `SWRConfig` context. You can for example specify a default fetcher so you do not have to pass it for every hook. If you want to override some global settings in a hook you can just pass the specific setting to the hook. It is also possible to have multiple `SWRConfig` contexts in your application, and the lower configs in the tree will merge its settings with the parent configs. SWR uses a global cache, but also allows you to customize it by passing a different cache to the `SWRConfig`.

It is possible to achieve simple pagination using the normal `useSWR` hook, along with an option to `keepPreviousData`. If you want to implement infinite loading, SWR provides a custom hook for this called `useSWRInfinite`, which gives the ability to trigger a number of requests with one hook. It also accepts a key and a fetcher, and some pagination specific options. You must also pass it a function that will accept the current page and previous data, and return the next key. This enables `useSWRInfinite` to work nicely with both index based and cursor based pagination. Pagination is a typical place where prefetching data can be beneficial, and to do this SWR provides a prefetch-function, that accepts a key and a fetcher as arguments, and populates the cache with its result.

SWR allows for middlewares, and you can pass in an array of middlewares to either the global configuration, or in a specific hook. Middlewares facilitate the execution of custom logic before sending a request, and after receiving a response from the server.

As mentioned, SWR keeps its API minimal, but still allows you to achieve most of what you want to do with this minimal API. You can do conditional fetching by passing `null` as the key to `useSWR`, and update the key when you want to fetch. Or you can implement lazy loading by passing a fetcher instead of a mutator to `useSWRMutation`, which gives you a trigger function to call when you want to load the data. You can reset the cache by using the global mutate function, and just pass in true as the predicate, and null as the data, making it clear the whole cache.

5.4 Apollo Client

Apollo Client is a library that empower developers to handle remote and local data using GraphQL. It provides a normalized cache which supports features like optimistic updates, automatic re-fetching, subscriptions and pagination. In addition to remote data, the cache can be used to store local data, which can automatically update the user interface, or re-fetch dependent queries on changes. Advanced network configurations can be achieved by using a chain of middlewares called *links*. Apollo Client has many features, and we describe these in the following sections.

Querying Data

Querying a back-end service with Apollo Client can be accomplished using the `useQuery` hook. The `useQuery` hook accepts a GraphQL query string and additional query variables as arguments. The hook returns properties such as `loading`, `error` and `data`. Query results are automatically stored in the cache. To retain a cache that is up to date with the back-end, Apollo Client supports *polling* and *refetching*. The former execute the query periodically, while the latter give developers control to manually execute the query, for example after a user interaction or an event was performed. It is possible to execute queries lazily by using the `useLazyQuery` hook, which only executes the query when triggered manually.

Apollo Client needs to know whether it should look for data in the cache, query the network for data or use a combination of the two when deciding what data should be returned by the query. By default, Apollo Client will primarily try to check the cache for data, and perform a fetch request to the server if the data is not in the cache. However, you can also configure it to always ask the server, and bypass the cache. You can configure what strategy to use by setting it as default options in the client configuration, or per query by passing an argument to the `useQuery` hook.

Subscriptions and Links

Apollo Client also supports *subscriptions*, which can be achieved utilizing the `useSubscription` hook. This facilitates the implementation of streaming data. In addition, the `useQuery` hook returns a callback function, `subscribeToMore`, which can be used to subscribe to subsequent updates to the query result. Subscriptions and queries usually use different endpoints, and using *Links*, you can redirect a request to the correct endpoint based on the given GraphQL operation type. Links give developers fine grained control of HTTP requests. They are commonly organized into a chain of link-objects, executing in a sequential manner. Apollo provides a variety of links for special use cases, such as the `WebSocketLink` for configuring subscriptions, and you can also create your own custom links to achieve advanced networking behavior.

Modifying Data

Performing updates to a back-end service can be done using the `useMutation` hook. A GraphQL mutation string is passed as an argument to the hook, and it returns a tuple, containing a mutation trigger function, and an object representing the current status of the mutation's execution. Additional `options` can be passed to both the mutation hook and the mutation trigger function to customize mutation behavior. A successful mutation usually results in some parts of the cache becoming stale. The simplest way of achieving fresh data after a mutation is to re-fetch all affected queries by using the `refetchQueries` array. Here, the affected queries are manually passed in, and these queries will be re-fetched after a successful mutation.

If the response of the mutation contains all the data we need, we can remove unnecessary network requests by directly updating the cache instead of re-fetching affected queries. Using the identification field and the GraphQL typename we can use the `update` function from Apollo Client to update the cache. The `update` enable developers to perform GraphQL operations on the cache in a similar way as it would when interacting with a GraphQL server. All changes done inside the `update` function will be advertised to other queries listening to that data. To ensure that your direct cache modification is corresponding with your GraphQL server, the `onQueryUpdated` callback can be used to query the server to verify that the mutation result complies with the back-end service.

When we know the response of a successful mutation beforehand, we can perform an optimistic update by providing an *optimistic response* to a mutation. This will not be put directly into cache, but will be placed in its own *optimistic layer*. This will not affect the real cache, but will still be broadcasted to all subscribed queries. When real data is returned from the mutation, the optimistic layer is discarded. Managing optimistic updates this way makes it easy to rollback in case of errors, as it only affects the isolated optimistic layer, and not the rest of the cache.

Caching

Apollo Client employs a normalized cache, meaning that individual queries are not stored as is, but go through a normalization process before entering the cache. The cache uses a flat lookup table of objects, where the different objects can reference each other, using a *cache ID*. The cache ID is generated by Apollo using the GraphQL *typename* and ID of the object. This means that two different queries, asking for the same object, can share that object in the cache. Additionally, the queries themselves get stored in the cache, using the query name and its arguments to compose the cache ID. Every combination of arguments will normally result in a new cache ID. However, you can specify which exact arguments should be included or excluded in the derivation of the cache ID. The queries are cached with references to all the normalized objects that were the result of the query. The cache is configurable, and you can instruct it not to normalize certain objects. Non-normalized objects are stored directly inside their parent object.

The cache also lets you customize the behaviour of types and fields. As mentioned, Apollo Client automatically infers the cache ID from the typename and `ID` field. However, in certain cases an `ID` field is not applicable to an object type. For objects like these, the cache lets you configure which field or fields should be considered the identification of the type, enabling it to still correctly infer the cache ID. The cache also lets you configure how fields are read and written to the cache. By defining a *read* function, you can perform data transformations on the data before it is retrieved from the cache. Likewise, you can define a *write* function that runs whenever you are writing something to the cache, either from a server response or when writing directly to the cache.

Apollo Client lets you define a *merge* function, where you can decide how the cache should merge incoming data with existing data. For example, in a paginated field, maybe you want to append the new data to the existing data. Using different definitions and combinations of the read, write and merge functions, you can enable Apollo Client to support different pagination strategies. Apollo also provides helper functions for standard patterns like offset-based pagination, saving you the trouble of implementing it yourself.

The cache read function can be utilized to retrieve data from a source *outside* of the cache. This can be utilized to create local-only fields. When writing a query to the server, it is possible to include these local-only fields in the query, by applying the `@client` directive to these fields. This tells Apollo that it should not include these fields in the server request, but instead just fetch them directly from the cache. Because of this, local-only fields can go beyond the server schema definition. Apollo includes its own local state solution called *reactive variables*, which enable you to store, manage and share client state throughout your application, resembling an atomic state pattern. How local only fields are modified depends on the way they are stored. If reactive variables are used, they can be directly modified, because Apollo detects these changes and updates queries that depend on those variables automatically.

Error Handling and Developer Tools

A request executed in Apollo Client may result in a variety of errors. The errors are categorized as either GraphQL errors or network errors. A GraphQL error might be syntax errors caused by a malformed query, querying a field that does not exist on the schema or resolver errors which may occur while attempting to populate a field. Network Errors, usually resulting in 4xx or 5xx status codes, are put into its own object in the error response. An error policy can be defined on the `useQuery` hook. In Addition, links can be utilized to achieve advanced error handling like automatically retrying requests, or fetching a fresh access token when the current one is stale.

Apollo Client includes advanced DevTools, where you can watch active queries and mutations, inspect the cache, and explore the schema. If you have local-only fields in your application, you can create a local schema definition, and pass it to the Apollo Client to have them show up in the DevTools, alongside the server schema. This also makes it possible to mock schema fields, if the back-end has not implemented them yet. Apollo Client allows you to batch multiple requests together, or to split one query into multiple requests by deferring certain fields, allowing you to show a subset of the data before the whole dataset is loaded. Normalized cache and support for storing and manipulating both remote and local data makes Apollo Client a flexible library, covering many aspects of state management in complex applications.

5.5 URQL

URQL is a lightweight alternative for a GraphQL server state management library, that prioritizes usability and adaptability. It aims to give a simple starting point with the bare necessities to build an application, and then offer extensions of functionality through what they call *exchanges*. These exchanges offer additional features such as caching, error handling, and optimistic updates, enabling developers to tailor URQL to meet specific project requirements.

Querying and Mutating Data

To query the GraphQL API from a component you use the `useQuery` hook. The hook takes an options object, and returns a result object along with a function to re-execute the query. The result object contains the data, and also a status boolean called `fetching` to indicate whether the data is loading. `fetching` will only be true if there is no data, which means it will be false when you perform a re-validation request to the server, while showing possibly stale data from the cache. Because of this the result object also contains a `stale` boolean, to indicate that a re-validation request is ongoing. The result object also contains properties like `error` that is populated with potential errors.

The options object that is passed to the `useQuery` hook requires a GraphQL query string, as well as a few optional properties that can be used to configure the query. These options include possible variables and a `requestPolicy` that determines how results are retrieved from the cache, which defaults to performing a request to the server only if the data is not already in the cache. You can pass in a `pause` flag to the hook, to indicate that the query should not execute automatically once the component mounts. You may perform logic to dynamically flip this flag back to re-enable automatic execution, or use the returned re-execute function to trigger the query lazily.

For mutations, URQL exposes a `useMutation` hook that works similarly to the `useQuery` hook. However, `useMutation` does not need an options object, only a GraphQL mutation string. It returns the same result object, along with a function to execute the mutation. This execute function takes the GraphQL variables as arguments, and returns a `Promise` wrapped around the result.

Exchanges

URQL does not support subscriptions out-of-the-box, but this is one of the features that can be achieved using *exchanges*. When configured, you can use the `useSubscription` hook, which also is very similar to `useQuery`. It differs in that it takes an optional second argument: a reducer function that can be used to accumulate the data from the subscription over time.

The URQL Client is configured globally, and manages all requests and results. It accepts the API URL, among other configuration options as arguments. Some of these additional configuration options include a custom fetch function and a global `requestPolicy`. It is also possible to override these settings by using the `context` parameter in the options object of the `useQuery` hook.

The URQL Client treats all GraphQL requests as unique objects, identified by the query string and variables. Internally these objects are managed as *operations*, and each operation will result in an *operation result*. The client does not actually perform the request, but instead sends the operations through *exchanges*. Different exchanges are chained after each other, each sending the operation to the next exchange in the line. The last exchange is responsible for performing the network request, before returning the operation result, which then travels back through the exchanges in opposite order. Thus, exchanges can perform logic both on *operations* and *operation results*. The order of exchanges are important. For example, you would not want to fetch results from the cache *before* the request is deduplicated.

The default exchanges takes care of deduplicaction of requests, setting up the default cache, and actually performing the request using `fetch`. There are many pre-built exchanges provided by URQL, like the previously mentioned `subscriptionExchange` enabling you to perform subscriptions. There is an `authExchange` for setting up complex authentication flows, a `retryExchange` for retrying failed operations and a `refocusExchange` to re-fetch open queries when the window regains focus. In addition to the pre-built exchanges, you can also create your own.

Document cache

The default cache in URQL is called *Document Caching*. A key is generated for each request, using the query string and variables, and the results are cached on this key. When we perform a mutation, this may potentially invalidate results from previous queries. If we mutate an entity of a specific type and another query containing an entity of the same type exists, the cache entry for the latter query is removed. This happens regardless of whether the exact entity that was affected by the mutation was stored in it. You can also supply additional typenames to a mutation, to force it to invalidate queries that contains objects of those additional types.

Document Caching can work well for content-driven sites, however, as an app grows more complex and interdependencies between data increase, you could benefit from having a normalized cache. URQL provides their own normalized cache called Graphcache. The normalized cache is used to create an in-memory representation of a relational database, and it ensures that our user interface is fresh when queries, mutations and subscriptions are sharing the same relational data.

Graphcache

Graphcache is able to automatically normalize the data from the GraphQL server, as long as the data contains a `__typename` and a keyable field like `id`, which composes what URQL calls the *entity key*. The normalized Graphcache utilizes two tables: a *Link* table for storing relations to other entities, and a *Records* table which actually store the values. Graphcache use the *entity keys* to enable a GraphQL query to traverse these tables to return the queried data. If a given entity is not identified by an `id` field, we can provide a custom `keys` configuration to ensure the *entity key* will be generated correctly. When Graphcache encounters un-keyable types they will be stored alongside the parent entity.

By utilizing the `resolvers` configuration, you can leverage Graphcache to transform or modify the values of cache fields when queried. For example, when working with dates, `resolvers` can be used to transform a date string from a database into a JavaScript `Date` object. The resolver functions may be generalized and reused on multiple fields, but the returned fields have to match the fields defined in the schema. In addition, `resolver` functions may also obtain data from other records in the cache which can be used to achieve advanced resolver logic.

Manual updates of the cache can be achieved by utilizing `update` functions. These are usually used to modify the cache on specific events, or defining how mutations and subscriptions should modify the cache. For example, when a successful mutation create a new entity, an `update` function can be used to add this entity to the cache of other queries. `update` functions are flexible and useful to ensure the cache is reflecting changes that has happened on the server, without needing to re-fetch queries. The API of `resolver` and `update` functions looks similar, but they differ in use-cases: `resolver`s transforms data, and `update`s removes, updates or deletes data and relations.

You can configure optimistic updates with `optimistic` functions. Data from the mutation is received as arguments to the `optimistic` function, which can be used to create an optimistic result that is returned to the consumer immediately after a mutation has been fired. When the real result is successfully returned from the server, the optimistic result is safely deleted.

To obtain full functionality of the Graphcache, you can utilize *schema awareness*. The GraphQL schema can be passed as an argument to the Graphcache configuration, allowing for advanced features like returning partial results, deterministically matched fragments and checking existence of fields and options inside `resolver`, `optimistic` and `update` functions. In addition, URQL provides an enhanced cache named `offlineExchange` which can be used to achieve offline functionalities.

As we have seen, URQL may start off as a simple and lightweight GraphQL client, but offers enough extendability to get it to fit your needs. It provides a simple *Document Cache* out-of-the-box, but allows you to replace it with the more sophisticated, normalized Graphcache, if you want more advanced features for complex applications.

Chapter 6

A Theory of Client State Management

With the overview over the different client state management libraries from chapter 4, we are now ready to formulate a theory on client state management based on these libraries. We will start by categorizing *state* into different **State Types**, as it can be useful to specify precisely what we mean when we are talking about state. Furthermore, there are five main areas we have identified regarding client state management, the first being **State Reach**. This concerns where the state is located, and what components have access to it. The second area is **State Composition Model**, which considers whether the state management utilizes a single top-down store, many atomic state parts, or something in between. After we have a grip on the location and structure of the state we briefly describe **State Derivation**, before we consider how to get only the state we want from the store, with **State Selection**. Lastly will go through the different ways of handling **State Modification**.

6.1 State Types

The state we want to manage can be information whether the user has clicked a button or not. Or it can be data that the user has typed into a form. It can be information that is downloaded from a server, or it can be the result of complex calculations performed in the background. It may be information about the status of such a calculation, whether it is still calculating, or if it is finished. And more often than not it is a combination of all the above. It is clear that state is more than just state, and from how the different libraries talk about state we have categorized two important distinctions when it comes to state, which are *business and presentation* state and *qualitative and quantitative* state. We will also mention *server* state, but as it belongs to the domain of the server state management libraries, we will not cover it in depth in this chapter.

6.1.1 Business State and Presentation State

We start by creating a distinction between two kinds of state which we will refer to as *business* state and *presentation* state. This distinction comes from the desire to clearly separate the state that is relevant for the business itself, and state that is just relevant for one specific implementation of the application, i.e. the presentation. It is clear that information whether a user prefers dark mode or light mode, although important to the presentation of the application, is not relevant for the business logic of the application. One way to think about this, is that *business* state is state that *drives* the application. You might think of it as a kind of back-end. If you implement the application both as a mobile application and a web application, the user interface would differ, resulting in different *presentation* state in the two apps. However, the *business* state would be the same.

6.1.2 Qualitative State and Quantitative State

There is a difference between the state that stores *what* a user has typed into a form, and the state that stores *if* the user is currently typing into the form. In the same way there is a difference between the *status* of a request for data to a server, and the resulting *data* when it is loaded. One is finite, the user is either "typing" or "not typing". On the other hand we have the state that is infinite, in the sense that the user can write whatever he or she wants[1]. Another way to look at it, and the way we choose to distinguish between these types of states, is that one is *qualitative* and the other is *quantitative*. *Qualitative* state is almost like meta-state, state *about* the state, or about the status of a process or the entire application. *Quantitative* state is the raw data, the values we store. In one case the *qualitative* state can help us decide what to show to the user, and the *quantitative* state could be what we are showing to the user. The terms *qualitative* state and *quantitative* state are inspired by XState [44], which we covered in section 4.4.

6.1.3 Server State

One last term that is relevant to mention is *server state*. In applications today the state often resides on a server somewhere in the cloud [62]. This means that the state of the application is downloaded from the server, either all at once from the start, or whenever a piece of the state is required. This makes managing state more complex, as we now have to consider keeping the local, downloaded version of the state up to date with the server, and consider other concepts like caching, pagination, prefetching and more.

There are some client state management libraries that have integrated methods to deal with *server state*, such as Redux and Recoil. However, these approaches for managing *server state* are comparatively basic and limited in contrast to designated server state management libraries as described in chapter 5. Recall that **RQ1** is divided into two distinct aspects: **RQ1.1** explores client state management, while **RQ1.2** delves into server state management. Consequently, server state management is comprehensively addressed in chapter 7, and the discussion of server state is therefore omitted in this particular chapter.

To summarize, when we talk about *state* we can mean many different things. Therefore we introduce some distinctions and definitions to help us understand exactly what we are talking about:

- *business* state and *presentation* state
- *qualitative* state and *quantitative* state
- *server state*

These types of state are not mutually exclusive, and some piece of state that is *qualitative* may very well be *business* state, or some other piece that are *quantitative* may very well be *presentation* state. These definitions and distinctions can help us to talk about state in a more specific way, also helping us understand how to approach it. When we simply refer to the state as *state*, we refer to the overall combination of all the types of state, or simply "data that is stored in memory".

6.2 State Reach

State can exist in various locations, and the different locations has implications for which components can access or *reach* the state. Therefore, it is important to understand what we call **State Reach**. Some state is just relevant for a small part of the application, maybe a single component, and can thus be stored locally inside the component. We call this state *local*, and local state needs to be passed around as props if other components needs access to it. Other state is relevant for

[1]Strictly speaking it is not infinite, but it would be meaningless to enumerate all the different possible variations of input the user could choose to type, unless there is a really small set of input options.

the whole application or are used by many components, and thus needs to be *global.* Global state are accessible by all components, without using props.

There is also a difference in whether the state lives *inside* or *outside* React. All state that lives inside React is tied to a component. This is true for both local and global state; at the root it is always linked to one component. When it comes to global state, the component it is tied to is often a provider-component, which is a higher order component. Alternatively you can choose to store the state outside of any components, i.e. outside React, and reference them inside components when needed. This way your state can become entirely independent of React.

This gives us two distinctions for state reach: *local* or *global* state, and whether the state lives *inside* or *outside* React. Because state *outside* of React becomes *global* to the React application, we can define three main categories:

- Local
- Global inside React
- Global outside React

We will go through the different options in turn.

6.2.1 Local

Local state is just accessible by the component the state is defined inside, or the components you pass the state to, using props. Theoretically, this can be implemented directly in components as variables, but as discussed in section 4.1, `useState` is Reacts way of making sure this state persists between renders. Other libraries has to take advantage of `useState` under the hood to provide persistence between renders.

For *quantitative* state there is not much for the state management libraries to do, other than what React already has built into their hooks. MobX has a custom hook to create a local observable, but they even argue themselves that it is often preferable to use `useState` instead. *Qualitative* state on the other hand, can be useful as local state. For example, XState allows you to create a local finite state machine inside a component.

One thing to be aware of when it comes to local state is that while `useState` allows state to be preserved between renders, it does not preserve the state when the component unmounts. This is because `useState` ties the state to the component, so when the component disappears, the state disappears as well. For *presentation* state this will usually not be a problem, because when the component is unmounted, there is nothing to present. For *business* state however, it is rarely the case that you want the state to disappear just because a component is not mounted. To avoid this you would have to turn your state *global.*

6.2.2 Global Inside React

Global state is especially useful when there are multiple components that need the same state, and the components are located far away from each other in the component-tree. As mentioned in section 4.1, React Context is used to make state accessible by all child components, by putting it inside a higher order component. This component is called a provider, and by putting it at the top level of the application, the state can be available globally.

Two examples of state management libraries that does this is Recoil and Jotai. They are very similar in that they both use atoms, and that they are used very much like `useState` inside components. However, they both supports global state, because the actual value is not tied to the component where it is used, but stored inside a top-level provider. When different components access the same atom, they are actually collecting the value from that provider. Of course, if the provider component is unmounted, the state will disappear, but by placing the provider at the top-level of the application this is rarely the case.

6.2.3 Global Outside React

Lastly we have the state that is stored *outside* of React itself. The thing that distinguishes these libraries is that they can be used without React at all. It is possible to define the state and manage business logic in MobX, even before you create a single component. Same with Redux, XState, Zustand and Valtio. You could theoretically switch front-end framework and still use the same state management library, without changing the logic.

The problem when using React with these state management libraries is how to integrate the state with the components. For example, if you directly import the Redux store from a component, the component would get the right values from the store when rendering, but the component would not automatically re-render when the state changed. Redux has solved this by using a provider component to provide the store at the top-level of the application, and then use custom hooks to access the store from the components. This way the components are kept in sync with the store.

There are different approaches for achieving this synchronization between the components and the state. As we talked about in section 4.3, MobX uses a higher order component wrapper to make it react to changes in the MobX observables. This way MobX do not need to use a provider to pass the state, but it can also use React Context if desirable.

6.3 State Composition Models

The state of an application needs to be composed in a certain way in order to remain structured, and there are fundamentally different ways of composing state. For example, state can be divided into small, indivisible units of data, which acts like building blocks that can be combined to construct complex state. On the other hand, state can be composed of one large object that may be divided into smaller parts when needed. The following section presents different **State Composition Models** and discuss their characteristics and properties.

6.3.1 Single Tree

State that is organized in a tree structure, with one root node on top of the tree and edges to children nodes employs a *single tree* composition model. Accessing a particular part of the state is achieved by traversing the tree starting from the root node and traversing down to the particular node. There are no nodes in the *single tree* that are isolated or disconnected. Applications using the *single tree* composition model only have one *single* state tree. Figure 6.1 shows a visual example of a *single tree* composition model. The root node is on top of the tree, and the leaf nodes are on the bottom of the tree.

Redux is an example of a library using the *single tree* composition model. A Redux application has only one single store organized in a tree structure. Accessing state is done by traversing the state tree from the root node and further down the edges until the requested state is obtained. Zustand enables using a *single tree* composition model, but is highly unopinionated, thus allowing other state composition models as well.

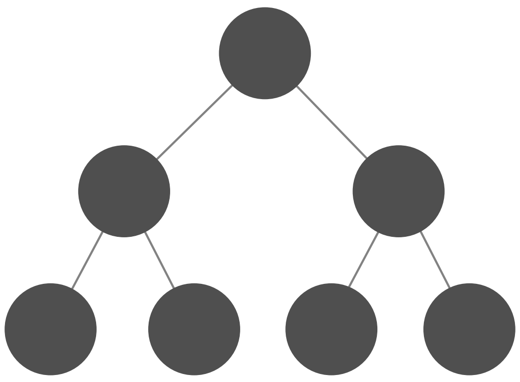


Figure 6.1: Illustration of the single tree composition model, where the state is organized as a tree data structure. Traversing the tree from the root node allows accessing specific parts of the state, and no isolated or disconnected nodes exist within the single tree.

6.3.2 Atomic

In an *atomic* composition model, state is supposed to be stored as indivisible units of data that are completely independent of each other. It is recommended to try to keep each unit of state as small as possible in an *atomic* composition, using these units to compose larger and more complex state. Figure 6.2 shows an overview of an *atomic* composition model. The nodes on the top row represents atomic state, which can be composed together with each other to create more complex state. The directed edges in the figure represent the one-way operation of composing atomic units, with the edges pointing downwards from the atomic nodes.

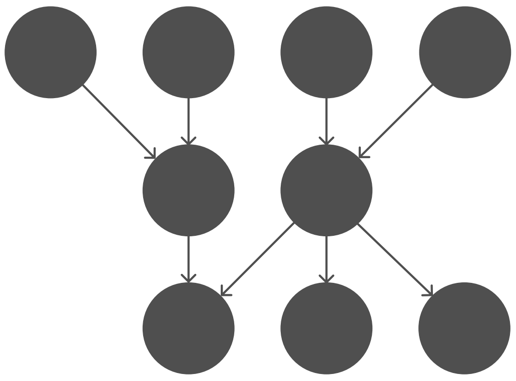


Figure 6.2: Overview of the atomic composition model, where state is stored as independent and indivisible units. Atomic states can be combined to create larger, more complex states using one-way operations, visualized by the directed edges.

Recoil use an *atomic* composition of state, where units of state are called *atoms*. Atoms in Recoil are supposed to be unique units of state that can be accessed directly, or used as building blocks in a collection of atoms to represent more complex state. Jotai is inspired by the *atomic* model from Recoil, and state is built by combining atoms.

6.3.3 Fragmented Store

The last composition model we will present is the *fragmented store* composition model. In a *fragmented store* composition, there are multiple stores that are globally available. There is no global state tree with a root node like in the *single tree* composition, and neither any demands for only storing as small units of state as possible, like in the *atomic* composition. The way the state inside the *fragmented store* is composed may vary. This can for example be a tree structure, atomic state or a combination of both.

Since the state of an application is fragmented into multiple stores, you will need to decide how and when to split different parts of the state into a new fragmented store. One possible fragmentation is to divide the state based on application features. Such a store may be responsible for all state about a certain feature in the application. For example, in a blog website, there could be a *fragmented store* that owned the state that has to do with all the posts, and another *fragmented store* containing all state about the user. Figure 6.3 illustrates an application state that is fragmented into three different stores. The tree structures, *single tree* composition model in this case, are representing the particular state of the *fragmented store.*

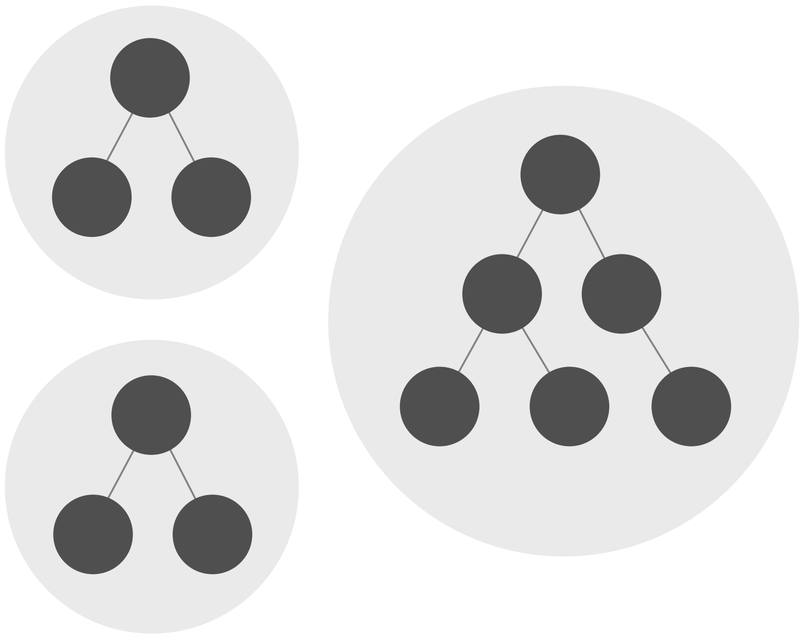


Figure 6.3: Illustration of the *fragmented store* composition model, where the application state is divided into three *fragmented stores.* The *fragmented store* can consist of tree structures, atomic states, or a combination of both.

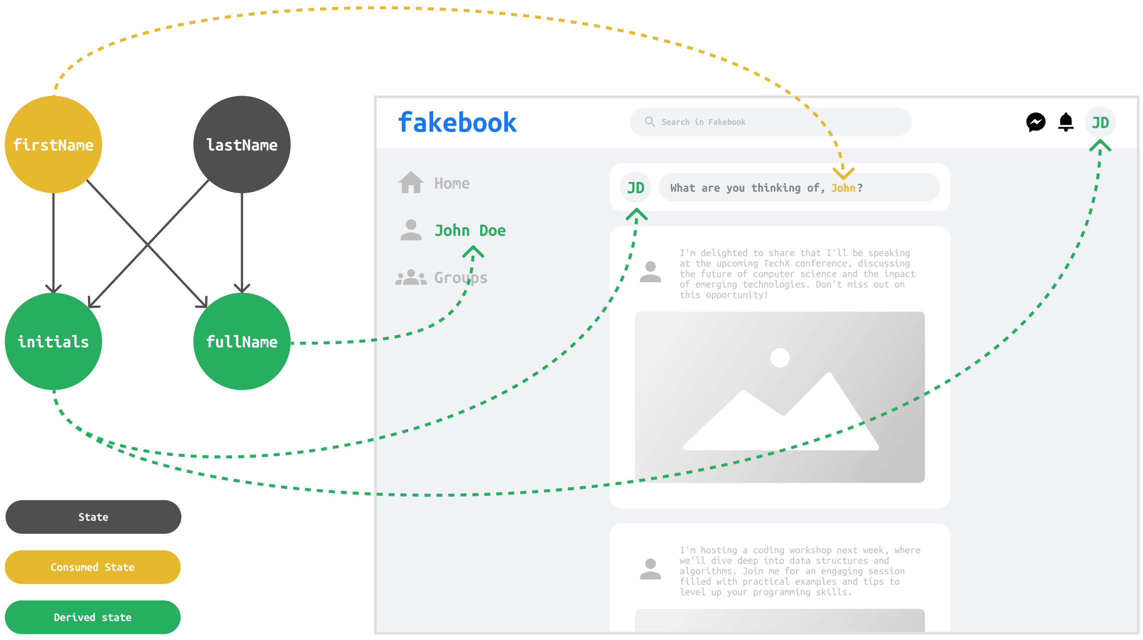
MobX, Elf and XState all utilizes the *fragmented store* composition model. For example a class in MobX can represent a *fragmented store*, because the class contains a part of the state that is independent of other classes containing state. Elf proposes to organize the state using either the *repository* or the *facade* design pattern. Using both of these patterns would result in each repository or facade representing a *fragmented store.* In XState, the machines represent the fragmented stores, which consists of an finite state machine containing the *qualitative* state, and a tree state holding the *quantitative* state. As previously mentioned, Zustand facilitates for a *single tree* composition model, but also supports a *fragmented store* model, where the state in the *fragmented store* can be stored for example as a tree structure.

6.4 Derived State

In chapter 2 we described that normalization is about maintaining atomicity and removing data redundancy. If our state is modelled to ensure atomicity, we might encounter the issue where the state we want to consume is too detailed or too specific. We might up end combining multiple atomic units of state to be able to obtain a sufficient amount of data. Let us explain this with an example about user information. We start by storing a users full name as one unit of state: `user: { name: "John Doe" }`. This might work as expected, but it will break the rule of atomicity, since the `name` property could be divided into the two units `firstName` and `lastName`. However, it feels kind of unnecessary to have to access two variables if our components need to display the users full name. This is where *derived state* is useful.

Derived state is state that is derived from existing state. Also, *derived state* is fully dependent on existing state as input to produce its output. Furthermore, *derived state* is always enabled through the usage of pure functions. Given our example in the paragraph above, we should store the user info within the variables *first name* and *last name* to maintain atomicity: `user: { firstName: "John", lastName: "Doe" }`, and use derived state to calculate the users full name when necessary. Recall the pure function `getFullName` shown in section 2.3. This can be used to create a derived state of the user's full name that extensively depends on the existing states `firstName` and `lastName`.

One advantage of utilizing derived state is that it can possibly enable future application requirements to be implemented without modifying the existing state. This is demonstrated in the example application Fakebook, shown in figure 6.4. Some of the requirements in Fakebook is that it must display the user's *first name*, *full name* and *initials*. Given that our *existing state* includes `firstName` and `lastName`, the aforementioned requirements can be fulfilled by deriving the rest of the state. Therefore, the *first name* can be consumed as is, while the *full name* and *initials* should be derived from the existing state. The resulting derived state for the user will be `initials: "JD"` and `fullName: "John Doe"`. The figure illustrates the relation between existing and derived state, and where these are displayed within the Fakebook user interface.



I'm delighted to share that I'll be speaking at the upcoming TechX conference, discussing the future of computer science and the impact of emerging technologies. Don't miss out on this opportunity!

I'm hosting a coding workshop next week, where we'll dive deep into data structures and algorithms. Join me for an engaging session filled with practical examples and tips to level up your programming skills.

Figure 6.4: Visualization of derived state in Fakebook. The first and last name of the user John Doe are used to derive his initials and full name. The location of where these are displayed within the Fakebook user interfase is shown.

6.5 State Selection

The problem of using React Context for global state is that whenever the Context value is changed, every component that depend on that Context will re-render, even if they do not depend upon the part of the state that changed. This is not optimal behavior, and what we want are components that re-render only when the piece of state they depend upon has been updated. The same is true for *derived state*; we want *derived state* to re-calculate only when the specific piece of the state it depend upon is changed.

This is one of the central problems of state management which we call **State Selection**, and the libraries solve it in a few different ways. *Selector functions* are pure functions that takes the whole state object or a fragment of the state and returns a specified piece of state. This enables the components, or the *derived state* to depend upon this specific piece of the state. Another approach available to the *atomic* state management libraries is to use the smallest amount of state possible by carefully *selecting atoms*. Lastly we have the mutable way. These libraries can not simply compare object reference, because the reference stays the same when they mutate the object. Because of this they have implemented automatic optimization based on *state access*, meaning they track which piece of state is depended upon in the background, and causes re-render, or re-calculation, only when necessary.

6.5.1 Selector Functions

Selector functions are based on the premise of pure functions that takes the whole state object as input, and returns a piece of state, as illustrated in figure 6.5. This is relevant if the composition model of your state is a *single tree* or a *fragmented store*, since the state object in these cases presumably is bigger than what you need in a single component. This way, when the state objects changes we do not have to assume that every component that utilizes a part of the state has to re-render. Instead all the selector functions are run, and only if the result is different from the last time will it trigger a re-render, or re-calculation in the case of *derived state*.

The different libraries can differ in how they compare the old and new result from the selector function. For primitive values it is enough with a simple equality check, and for objects and arrays it is often used a shallow equality check. This is one of the reasons that state management libraries often utilizes an immutable update model, because then they only have to check the object references to verify if the result from a selector function has changed. Most libraries also give the option of providing a *custom equality function* next to the selector function.

Redux, which uses *single tree composition*, is an example of a library that uses *selector functions* to optimize state access. Zustand, which utilizes a *fragmented store*, also use *selector functions*. XState uses what they call selectors, but with our classification it is actually what we will call a *derived selector function*.

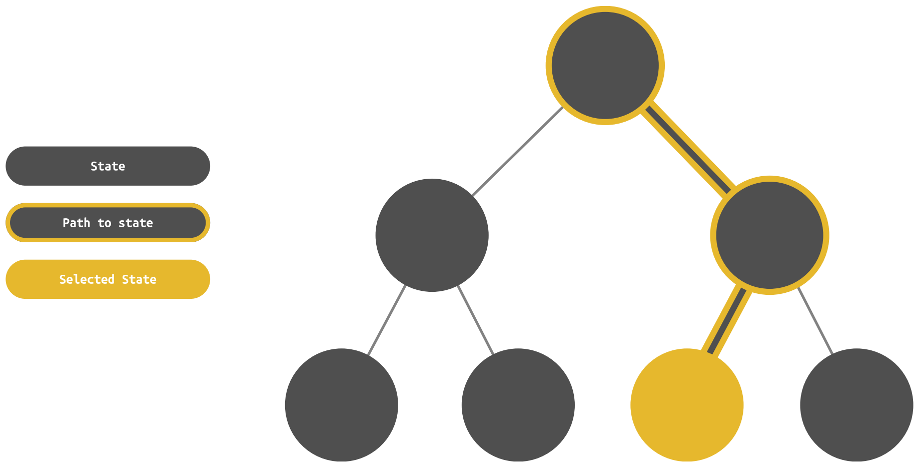


Figure 6.5: Selector functions accepts a state object, traverses the path to the selected state, and return the selected state. The nodes and edges that are outlined represents the traversed path.

6.5.2 Derived Selector Functions

In some cases it would be desirable to apply another level of optimization to the selection of state. Let us say we want to re-render a component whenever the length of a list changes. If we select the list from the state using a *selector function*, the component would re-render every time the list changed, regardless of whether the length actually changed. In this example it would be preferable to utilize what we call a *derived selector function*, which is a combination of **State Derivation** and *selector functions*. One can think of the derived selector function as a layer wrapped outside a selector function. First, the *selector function* gets a piece of the state, and then the *derived selector function* performs some logic with this state, returning this instead of the state. For example the *derived selector function* could take the *selector function* for the list as an input, and then return the length of the list. This way the component has only *selected* the length of the list, and will only re-render when the *derived selector function* returns a different length.

In the case of XState, what they call selectors are what we call *derived selector functions*, because the function takes the state of the machine and a specific state we want to check against as input, and returns a boolean indicating whether the machine is in the specific state. This way the component will only re-render when the machine enters or leaves the specified state, not every time the state changes. Redux also has support for advanced *derived selector functions*, using Reselect[2].

Figure 6.6 shows an illustration of a *derived selector function* in a *single tree* composition model. Two different *selector functions* are selecting one leaf node each in this example, which are used as inputs to the *derived selector function*. The *derived selector function* outputs a new *derived state*, shown in a green color, that is fully dependent on the two *selector functions* used as its inputs. Additionally, *derived selector functions* are consumed the same way as *selector functions* are consumed within components.

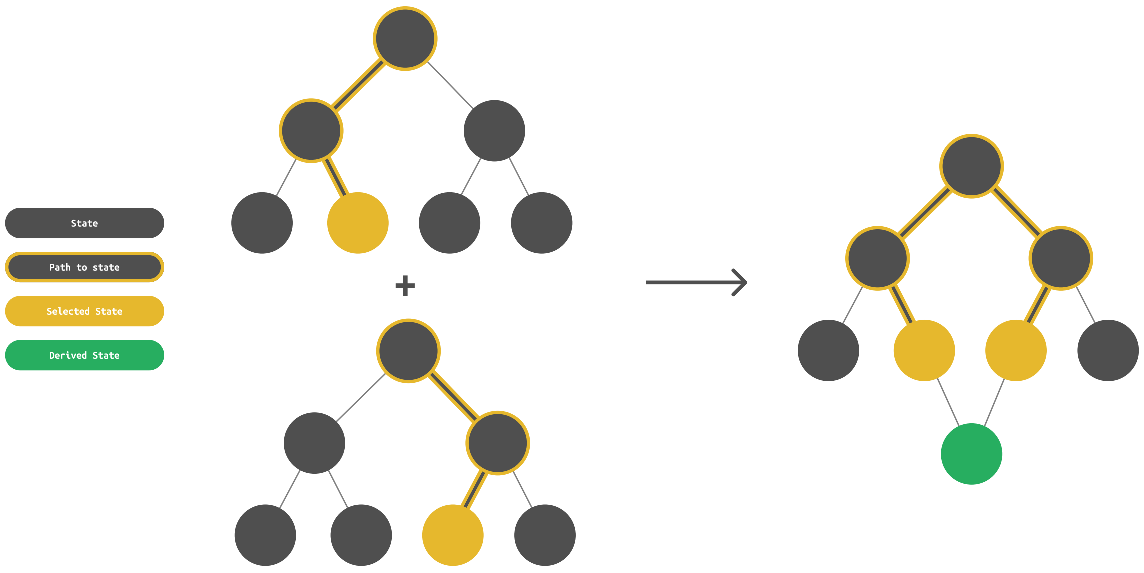


Figure 6.6: The figure illustrates a derived selector function that receives two selector functions as inputs and utilizes their results to produce its derived state.

6.5.3 Atomic

One of the benefits of an *atomic* **State Composition Model** is that the state is already nicely packaged for state optimization. You do not need to use any special tricks to depend upon the smallest amount of state possible, as you can just select the smallest atom you need. If you need data from multiple atoms combined, create and use a derived atom, similar in fashion to how you would use *derived selector functions*.

It may be noted that in the case of *atomic* libraries like Jotai and Recoil, as we mentioned in section 6.2 the state is actually stored inside a provider. This means that under the hood, using an atom is actually the same as using a *selector function*, and using a derived atom is the same as using a *derived selector function*. However, for the developer, working with the two models are

[2]https://github.com/reduxjs/reselect

very distinct experiences, requiring different mental models, so we still think it makes sense to have an own category for *atomic*.

Illustration of *derived state* used in *atomic* composition model are shown in figure 6.7. The atomic units of state are shown in grey color. These can be composed together in order to create *derived state*, which is shown as green circles in the figure.

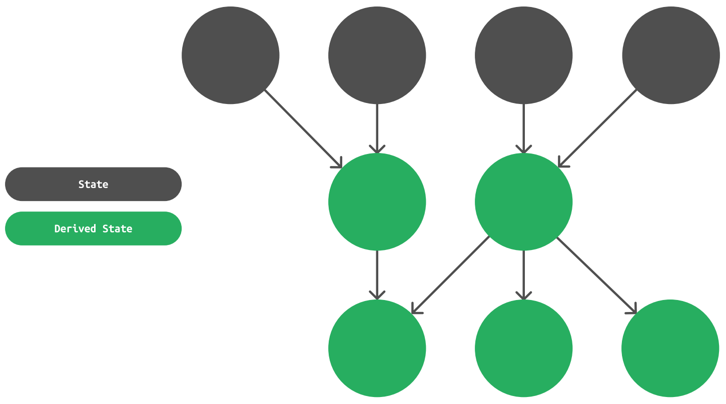


Figure 6.7: Illustration of atomic derived state. The grey nodes are atomic units of state, and the green nodes represents derived state.

6.5.4 Automatic Based on Access

A completely different approach to the problem of depending upon as little state as possible, is to do it automatically behind the scenes. MobX and Valtio automatically tracks dependency based on state access. This way, if you have a big state object you can pass the whole object into the relevant component, as long as you access only the values you depend upon inside the component. Then the libraries will notice this state access, and keep track of the dependency, and automatically trigger re-render when the accessed value changes. This approach fits nicely with the mutable state types, which is what MobX and Valtio uses.

Figure 6.8 illustrates access based selectors, which represents a *fragmented store* with a *single tree* composition inside. The whole *fragmented store* is sent to the component, and the component itself accesses part of the store that it should depend upon. In the illustration only one leaf node is accessed, thus making the component only depend upon that leaf node even though the whole *fragmented store* is passed to the component.

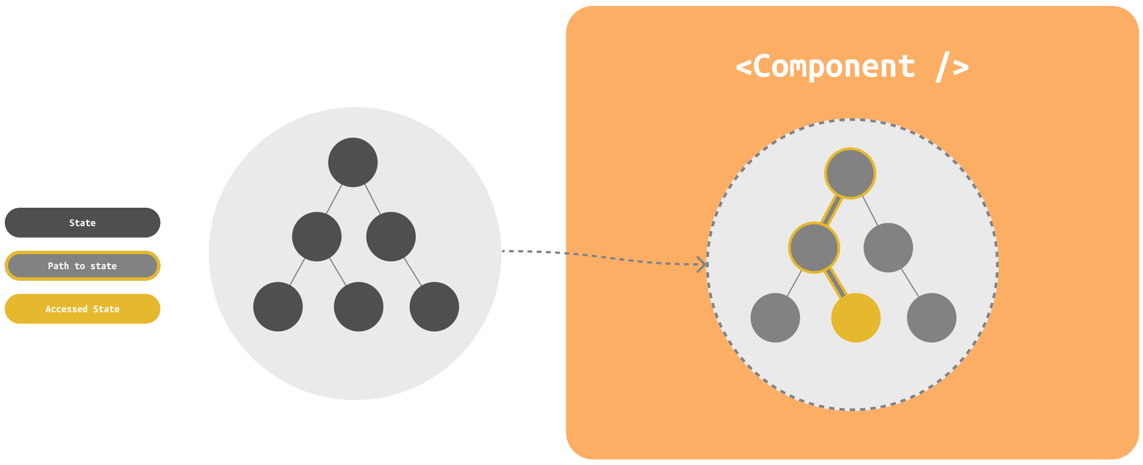


Figure 6.8: Overview of an access based selector. A fragmented store is passed into a component, but the component only depends upon the part of the fragmented store that is accessed.

6.6 State Modification

So far we have described what state is, how state is composed and how to select part of the state that a component need. In this section, we will explore the concept of **State Modification** and examine the various approaches and techniques used to achieve it. To begin with, we introduce the characteristics and properties of mutability in the context of state modification. Following that, we discus the location of the state modification logic itself.

6.6.1 Mutable and Immutable

There are two ways to modify state - *mutably* or *immutably*. As we have seen in chapter 2, *mutable* updates in non-primitive data types are done by directly manipulating a variable, while an *immutable* update require that you create a new variable and copy the previous values to the new variable. Mixing these methods might lead to unintended behaviour. Modifying state in a library that are using *immutable* state updates with a *mutable* update will not show visual change in the application because the shallow equality check will say that the reference is the same and therefore no change is made. Moreover, if a we perform an *immutable* update to a state management library that expects *mutable* updates, the code still works, however, it can result in unnecessary computations and performance drop.

When state is changed *mutably*, the object reference holding the state will never change no matter how many times the state values is changed. When a consumer of the state wants to know if the state has changed, the current value of the state is used to compare with the new state value. Of the state management libraries we have looked at, MobX and Valtio are the only ones that updates the state *mutably*. In MobX, functions that modifies the state are called *actions*, which directly *mutates* the state. State in Valtio is stored inside an enhanced version of the JavaScript `Proxy` [58] object, that enables batched updates, ignores duplicate changes, and allows the mutation of the proxy state.

On the other hand, when using *immutable* state modification, the object reference should never remain the same after a state change has occurred. The main reason for doing *immutable* state updates is because this allows for consumers of state to check if the state has changed by only comparing object references, and not by its values like in *mutable* updates. If the state contains deeply nested object or a large array, it is much faster to compare states by its reference, than recursively traversing all values in a deeply nested object.

All libraries from our data set, except the two mentioned above, use *immutable* state modifications. Many of them use an external library as a dependency, or recommends using a library to simplify the issue of updating deeply nested objects *immutably*. For example, Redux, Zustand, Jotai, XState and Elf explicitly recommends Immer[3] to solve the issue of *immutably* updating nested objects.

Immutable state updates have the benefits of making it easier to implement undo-redo functionality or time travel debugging, since every state change results in a completely new object, hence you can go back or forth between previous versions of the state. Checking if a deeply nested object has changed by comparing object references is faster than recursively traversing the object and checking all its values. However, *mutable* updates are usually simpler to perform because you can directly *mutate* the state.

6.6.2 Local and Global State Modification Logic

When talking about global state as described in section 6.2, we refer to state that is located outside of components, and is globally available. All components should be able to read and update global state. However, there are differing practices for where the logic for updating state should be

[3]https://immerjs.github.io/immer/

located. One approach situates this logic within the components themselves, while an alternative method positions the logic for updating global state externally, detached from the components.

Logic for how the state is modified or changed is hereby labeled as *modification logic*. The *modification logic* contains instructions about how the state is changed. Examples of *modification logic* might be to increment the value of a counter state, or update user-related state about a newly authenticated user with first name, last name and email address. *Modification logic* also include rules for how state is modified based on potential previous values. For example, *modification logic* for incrementing a counter needs to know the previous counter value to be able to increment it by one. *Modification logic* varies from very simple to highly complex state modifications.

When the *modification logic* is placed inside a component, we say that it uses a *local modification logic* pattern to modify state. This gives the component responsibility for how the global state is changing. Some libraries that use *local modification logic* are Recoil and Jotai. In Recoil, the component has full control over how a globally available atom is updated.

On the contrary, *global modification logic* is defined as *modification logic* that is placed outside of the components, and is globally available to all components. This enables the reuse of *modification logic* because multiple components can request state changes from the *global modification logic* when an event occurres. Examples of libraries that are practicing the *global modification logic* pattern are Redux, MobX, Zustand, XState and Elf. In Redux, *global modification logic* are placed inside reducers which are located outside of the components. MobX uses a *fragmented store* composition model, and actions containing *modification logic* that changes the state are globally available and decoupled from components. Elf proposes two different design patterns which both apply the *global modification logic* pattern. Zustand recommends to store *modification logic* inside actions that are siblings to the global state, thus it uses the *global modification logic* pattern as well.

6.7 Summary

In this chapter we have presented a general theory of client state management, based upon the client state management libraries described in chapter 4. We have distinguished between different **State Types**, namely *business* and *presentation* state and *qualitative* and *quantitative* state. The **State Reach** describes where the state is located, and as a result, who can reach it. *Local* state is only reachable by the component it is defined in, and the components it is passed to as props. *Global* state can be available to the whole application, but we also have to distinguish between the global state that is tied to a higher order component *inside* React, and the global state that resides *outside* React, and thus is independent of front-end framework.

Further on we saw that the **State Composition Model** can differ, either being stored in a *single tree*, a combination of *fragmented stores* or as individual *atoms*. This greatly influences how you do **State Selection** to optimize performance in the components. When working with state that is bigger than single units, namely *single tree* or *fragmented store* composition, one would normally have to use a *selector function* to get the piece of state you want to depend upon. It is also possible to use in combination with **Derived State**, and use *derived selector functions*. If you have composed the state in an *atomic* manner, you can just choose the minimal *atom*. It is also possible to go a completely different route using automatic tracking based on *state access*, to make the library handle the optimization for you behind the scenes. Lastly we saw that **State Modification** can happen *mutably* or *immutably*, and that the modification logic happens either *locally* inside the components or *globally* inside the library.

Chapter 7

A Theory of Server State Management

Based on the server state management libraries included in our literature analysis from chapter 5, a theory of server state management is presented. To begin with, the concept of **Server API Compatibility** involves examining whether server state management libraries are tailored towards a specific server API, or enable the integration with various APIs. The **Operations** section outlines the main query operations that are performed against a server: *queries*, *mutations* and *subscriptions*. Moving forward, the topic of **Cache** entails a thorough discussion of the two primary cache strategies in server state management: the *query cache* and the *normalized cache*. Following that, **Organization of Queries** discuss two approaches for structuring queries in an application, and **Asynchronous Metadata** provides an overview of metadata that can be used to monitor network requests. Many applications have a need for utilizing both server state and client state., and for this reason, **Integrated Client State** discuss how client state can be integrated into server state management libraries. Lastly, **Server State Features** delve into some advanced features that can be efficiently enabled by server state management libraries, such as *prefetching*, *pagination* and *offline mode*.

7.1 Server API Compatibility

In section 2.8 we looked at two different types of server APIs, namely RESTful APIs and GraphQL APIs. One of the first distinctions we could make in a theory of server state management is thus what type of server API the libraries support. However, while libraries such as Apollo Client and URQL markets themselves as GraphQL clients, libraries like Tanstack Query, RTK Query and SWR do not mention anything about being directed towards RESTful APIs. In some cases they even go so far as to call themselves back-end *agnostic*. Differentiating on what server API is supported is thus not a precise differentiation, although not so far off the mark. While there are some libraries without any constraints on the type of server API, two of the libraries we have studied *are* aimed at GraphQL APIs. Although you *can* configure Apollo Client to use a RESTful API endpoint, this is not the intended use-case. And herein lies the key, the *intended* use-case. Some libraries are *specialized* for one specific type of server API, while other libraries are *general*, usable with a larger variety of server APIs.

7.1.1 General

General server state management libraries are flexible and adaptable when it comes to the type of server API you use to interact with your server. These libraries serve to manage the server state effectively, but do not really care where that server state originates. Because they do not know what type of API you want to use to fetch your data, *general* server state management libraries typically do not include pre-existing features for data fetching. They delegate this implementation to you. This enables you to implement a custom fetcher that sends a GET request to a RESTful API endpoint, a GraphQL query string to a GraphQL server API, or even return some local data. Although RTK Query does include a default fetcher suitable for RESTful APIs, we would not call it a *specialized* library, given that the fetcher is easily replaced, and the library does not include any other RESTful-specific features.

7.1.2 Specialized

Specialized server state management libraries are intended to use in combination with a specific server API. We have looked at two libraries specialized for GraphQL APIs, but there is nothing in the way of having *specialized* server state management libraries intended for other types of server APIs, like RESTful APIs. Specializing at one server API means you loose flexibility, but you gain a streamlined experience in working with this type of server API. For example, since the GraphQL *specialized* libraries can assume that we are going to talk to a GraphQL server API, they can provide a built-in fetcher, saving us the trouble of implementing a fetcher ourselves.

Depending on the nature of the specific type of server API, this streamlined experience may give you various technical advantages and features that would not be possible with *general* libraries. One of the advantages of using a GraphQL-specific library is knowledge of the structure of the data, allowing more advanced caching features, which is covered in more detail in section 7.3.

7.2 Operations

A crucial aspect of server state management libraries is the ability to interact with a server. However, there are different *types* of interactions that can be performed, different **Operations**. An analogy can be drawn to database theory, where there are four fundamental database-operations that can be performed on a database. These four database-operations are encompassed in the acronym *CRUD*, which stands for *Create*, *Read*, *Update* and *Delete*. We can use the fundamental database-operations to explain the three fundamental **Operations** a server state management library can perform.

First we have *queries*, which corresponds to *read* for databases, where a client requests some data, and receives the result of the requested data from the server. For databases, *create*, *update* and *delete* all make changes to the database. In server state management, the *mutation* operation encompasses all the interactions that modifies data on the server. The last operation is called *subscriptions*, which is comparable to the *read* database-operation, like *queries*. However, *subscriptions* differs from *queries* in that instead of sending a request and receiving a response *once*, it subscribes to some data and gets notified whenever there are any changes to that data. *Subscriptions* are especially useful when presenting rapidly changing real-time data. This section presents the three **Operations** *queries*, *mutations* and *subscriptions* and how they are applied in server state management libraries.

7.2.1 Queries

A *query* operation refers to the process of requesting information from a data source. *Queries* are used to perform the equivalent of a *read* database-operation on a server, and will not make alterations to the server data. A *query* is a one-time request for data, although most implementations of *queries* include a method for re-fetching the query in order to send the same request again manually. Figure 7.1 illustrates a query operation, where the client sends a query request to the server, and the server responds with the requested data.

In React, server state management libraries usually lets you execute *queries* using hooks. By default, the *query* hook normally trigger the *query* automatically on mount. You can however disable the automatic execution of a query, and rather control the query execution by an external

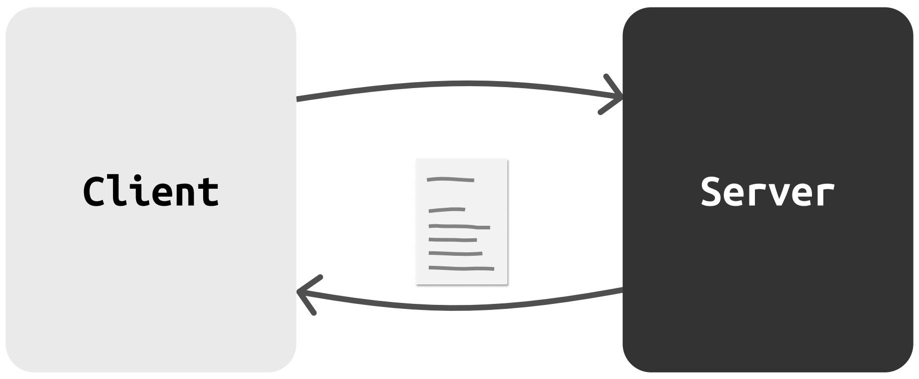


Figure 7.1: Overview of a query operation. The client sends a query to the server requesting specific data, and the server responds with the requested data.

state. This can be useful when a query depends on result data from a former query, or when data should not be fetched until a certain event has happened.

7.2.2 Mutations

Mutations are employed to make changes to the data on a server, just like *create*, *update* or *delete* for databases. In contrast to *queries*, *mutations* are not automatically executed when the component mounts. Usually, a mutation is triggered by a user event, such as clicking a button after filling in a form. For this reason, *mutation* hooks normally return a trigger function that is used to execute the mutation. The manual triggering of *mutations* give developers control of when, and whether a mutation should be executed or not.

Every *mutation* require some type of data to be sent along with the mutation. For example when creating a new entity, the data describing the entity needs to be passed along with the mutation. When deleting or updating an entity, at a minimum, an identification needs to be included in order to know what entity should be deleted or updated. A properly designed application should ensure that all the required data for the given mutation are present before executing it. Figure 7.2 shows a client sending a mutation request including mutation data. The modification is illustrated by the orange line. In response, the server potentially responds with the same data, as it is after being modified on the server, indicated by the green line.

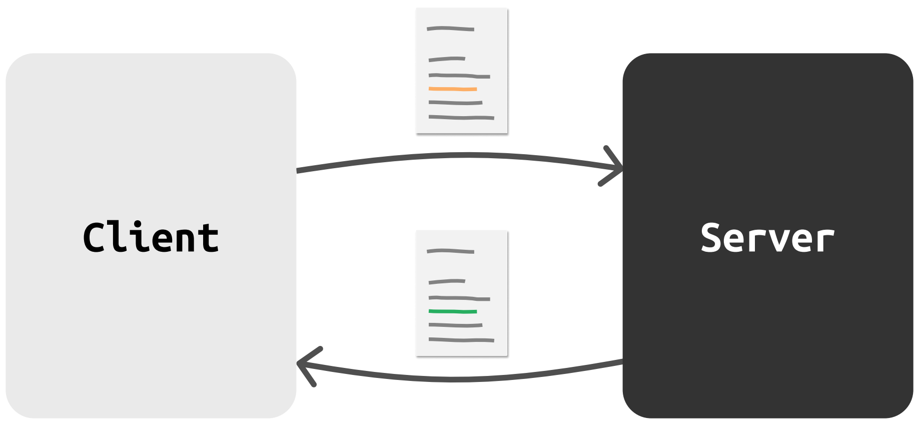


Figure 7.2: Overview of a mutation operation. The client initiates a mutation request, denoted by the orange line, indicating mutated data. The server, in response, potentially provides the modified version of the same data, represented by the green line.

7.2.3 Subscriptions

Like *queries*, *subscriptions* performs the equivalent of a *read* database-operation. In contrast to *queries*, that reads the data *once*, the *subscribe* operation initiate a subscription for a set of data, and gets notified whenever there has been a change to that data. *Subscriptions* can be applied in order to implement *streaming data*, as described in section 2.12, and a successful *subscription* between a client and a server facilitates the exchange of data streams between the two parties. Figure 7.3 shows an overview of a *subscription*, where the client initiates a subscription to some data on the server, and any occurring changes to that data on the server is pushed to the client. Compared to *queries* and *mutations*, the *subscription* operation is more complex because it includes multiple events over a longer period of time.

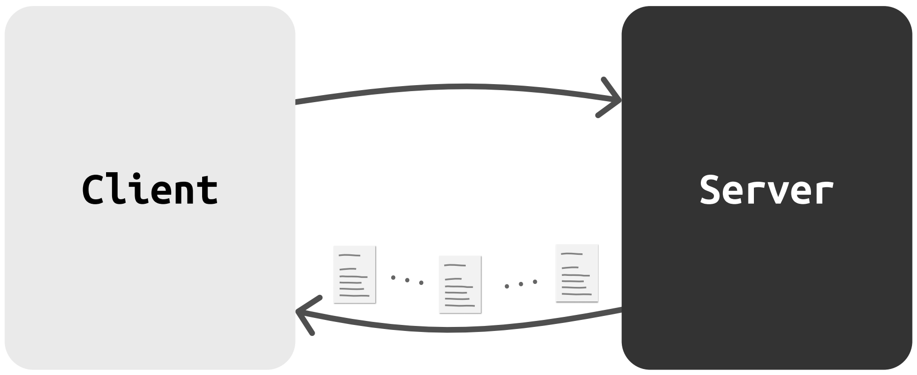


Figure 7.3: Overview of a subscription operation. The client initiates a subscription to the server, and the server pushes changes to the client as they occur on the server.

A common approach when using *subscriptions* is to start subscribing on initial mount. The data returned when a subscription starts is usually the whole set of data. Eventually, when the data changes on the server, the client may receive *only* the part of the data that changed. Let us explain how subscriptions can be used to achieve real-time updates in a chat application. When you first enter the chat with your friend, your device will initiate a subscription to your messages. The initial data received when subscribing is most likely a set of recent messages between you and your friend. After some time, your friend may send you a new message, which you will receive as a payload containing *only* the new message. This message will then be added to the message thread and included in the user interface. If you respond to the message, your friend that is actively subscribing to the same message thread will receive the message you just sent. After chatting with your friend for a while, you exit the chat. In order to stop listening for future messages, it is important to unsubscribe properly. Forgetting to unsubscribe from the data might cause superfluous network traffic and unnecessary memory consumption.

A successful implementation of *subscriptions* not only depends on the client accepting it, but also on the server's ability to enable it, since it is the one actually pushing new changes to all the currently subscribed clients. For this reason, server state management libraries that are *specialized*, implements *subscriptions* different than the ones who are *generalized*.

Specialized server state management libraries can assume detailed information about the communication protocol utilized because it is target towards a single server API. For example, specialized server state management libraries target towards GraphQL APIs can apply the GraphQL operation type *subscription*, defined by the query language GraphQL [19]. This in turn enables *specialized* server state management libraries to configure *subscriptions* globally for all queries. Once the *subscription* is configured, subscriptions can be employed anywhere in the application which all implements the globally configured subscription settings.

Generalized server state management libraries can make no assumptions on communication protocols, since it allows for multiple server APIs. In comparison to *specialized* server state management libraries, the *generalized* ones leaves more room for developers to decide how the implementation of *subscriptions* are carried out. Hence, implementation require more work by developers, but it also offers greater flexibility. In addition, applications can use different communication protocols for *subscriptions* per individual endpoint or query. For example, an application can use WebSocket [30] for one query, and Socket IO[1] for another query.

7.3 Cache

One of the most central concepts when it comes to server state management is **Caching**. Caching plays a critical role in optimizing the performance of web applications, and server state management libraries use caching to minimize the amount of requests made to the server. Due to the essential role of caching in server state management, many of the features provided by server state management libraries interact with the cache, either directly or indirectly.

Cached data are structured in what we call *cache entries*, which can be thought of as a unit of cache. A cache entry consists of an identifier we call *cache key*, and the cached data. The cache key enables direct access to its corresponding cache entry, resulting in fast retrieval of cached data.

There are two main approaches for organizing cached data in server state management: *query caching* and *normalized caching. Query caching* stores the result of one specific query in the cache as a cache entry, using *query keys* as *cache keys. Normalized caching*, on the other hand, transforms the returned data into a normalized form before storing it in the cache as multiple cache entries, using both *query keys* and *entity keys* as *cache keys.* This section looks at both approaches in more detail, including how the different types of *cache keys* are generated, and how the approaches differ in managing stale data.

7.3.1 Query Cache

Query caching is the most basic form of caching in server state management, but it is not necessarily less efficient or useful than more complex approaches. When executing a query, the server state management library stores the response to that query in the cache as a *cache entry.* The server state management libraries use query identification mechanisms that allows them to differentiate between queries and create separate cache entries for each. The query identification, which we call *query key*, is used as the *cache key* for the *cache entry.* The response data from a query is stored in the cache using its *query key* as *cache key.* If the same query is executed multiple times, it will be identified with the same *query key*, and can thus use the data from the same *cache entry* instead of performing a new network request.

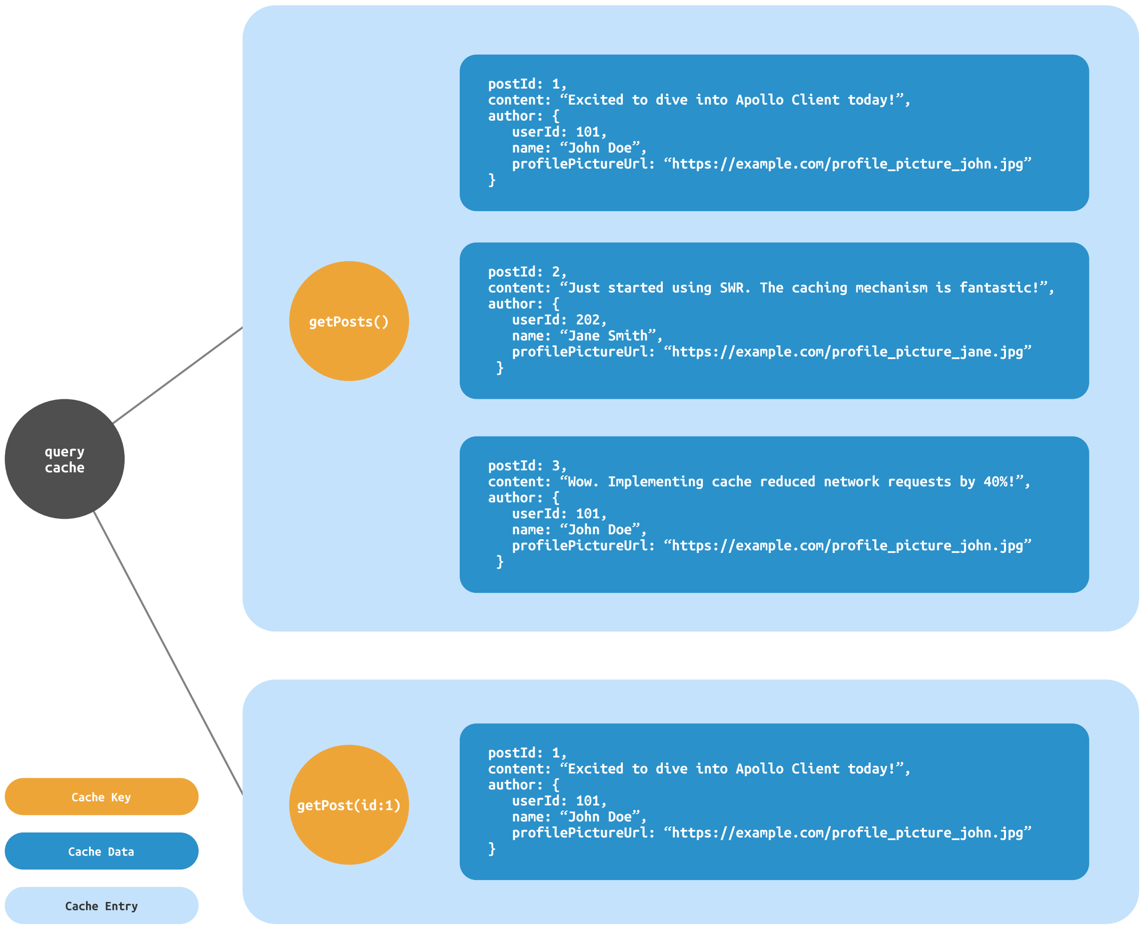
Query Keys

There are two important elements to consider when creating a *query key.* The first is *what* is being queried. For example, we would probably want different cache entries for fetching a Facebook feed, and fetching a user profile. The second important element is the *parameters* of the query, if any. We want to be able to cache a query for one specific user profile separately from a different user profile. Using these two elements, *what* is being queried and the *parameters* of the query, you can create unique *query keys* for each distinct query. For example, in the *query key* `getPost(id:1)`, `getPost()` refers to the *what*, and `id:1` is a *parameter.*

Figure 7.4 shows the structure of a cache for querying posts in a Facebook feed. The cache has two *cache entries*, one for each query. The *query keys* of the two queries, `getPosts()` and `getPost(id:1)`, are used as *cache keys* for the two cache entries. The *cache data* is placed alongside the *cache keys* in the respective *cache entries*, including both `post`-information and `author`-information.

Some libraries lets you define *query keys* yourself. Others will infer it based on the query configuration. *General* server state management libraries will often give you more manual control over

[1] https://socket.io/



```
postId: 1,
content: “Excited to dive into Apollo Client today!”,
author: {
   userId: 101,
   name: “John Doe”,
   profilePictureUrl: “https://example.com/profile_picture_john.jpg”
}
```

```
postId: 2,
content: “Just started using SWR. The caching mechanism is fantastic!”,
author: {
   userId: 202,
   name: “Jane Smith”,
   profilePictureUrl: “https://example.com/profile_picture_jane.jpg”
 }
```

```
postId: 3,
content: “Wow. Implementing cache reduced network requests by 40%!”,
author: {
   userId: 101,
   name: “John Doe”,
   profilePictureUrl: “https://example.com/profile_picture_john.jpg”
 }
```

```
postId: 1,
content: “Excited to dive into Apollo Client today!”,
author: {
   userId: 101,
   name: “John Doe”,
   profilePictureUrl: “https://example.com/profile_picture_john.jpg”
}
```

Figure 7.4: Overview of a query cache. The cache consists of two *cache entries*, each corresponding to a specific query. The *cache keys* used are `getPosts()` and `getPost(id:1)` for their respective queries. *Cache data*, including post-information and author-information, is stored inside the *cache entries*.

the query key, as they assume less about how you retrieve your data. This is true for Tanstack Query and SWR, which both require you to define the query key manually. On the other hand, *specialized* libraries that know the structure of the query can infer the query key to a greater extent. For example, Apollo and URQL use the GraphQL query name and query string to identify *what*, and then combine it with the *parameters* to infer the query key. Although RTK Query is a *general* library, it also generates the query keys automatically. It can do this, because it requires you to name all queries, enabling RTK Query to use this name, alongside the *parameters*, to generate the query key.

Query Invalidation

Let us say you *query* a list of all your Facebook friends. Then you make a new friend, and this new friend sends a friend request, so you perform a *mutation* accepting the request. The list of friends in the cache is now outdated, as it does not contain your newest friend. *You* know that the cache entry of the Facebook friends *query* will be outdated after you perform a *mutation* adding a friend, but the server state management library does not. Hence, you need a way to tell it that the *query* has become stale, and this is what *query invalidations* are for.

Query invalidation is simply the process of telling the server state management library that one or more *queries* has become stale, and needs to be re-fetched. There are different approaches to accomplish this, but they all aim to establish a relationship between a *mutation* and one or more *queries*. In the case of the libraries that utilize a manual *cache key*, you invalidate *queries* by using the *cache key* directly. RTK Query lets you attach tags to *queries* and *mutations*, and when a *mutation* is triggered, all *queries* containing the same tags are invalidated. URQL uses an automatic approach, where all *queries* containing an entity of the same type as an entity in the *mutation* are invalidated.

7.3.2 Normalized Cache

As discussed in section 2.6, properties of a normalized database are atomicity and no data redundancy or inconsistency. Even though the cache is not a database, it shares some of the properties of a database such as data storage, modification and retrieval. If the cache is structured in a normalized form, it can be classified as a *normalized cache*. However, data that we request from a server is *not* automatically stored in a way that enable the cache itself to be normalized. Hence, employing a normalized cache require some manual work and assumptions about the data returned from the server.

The non-normalized cache illustrated in figure 7.4 shows examples of data redundancy. The post with `postId: 1` is stored twice, both within the `getPosts()` and `getPost(id:1)` cache entries. Furthermore, the data about the author John Doe exist three times. If the author John Doe changes `profilePictureUrl`, all queries that contains the affected url needs to be re-fetched. On the other hand, if the same data were stored in a normalized cache, the author John Doe would only be stored once. In order to store entities only once in our cache, each entity needs a unique *cache key*. In the same way that unique *query keys* are used as *cache keys* for queries, we need unique *entity keys* to use as *cache keys* for entities.

Entity Keys

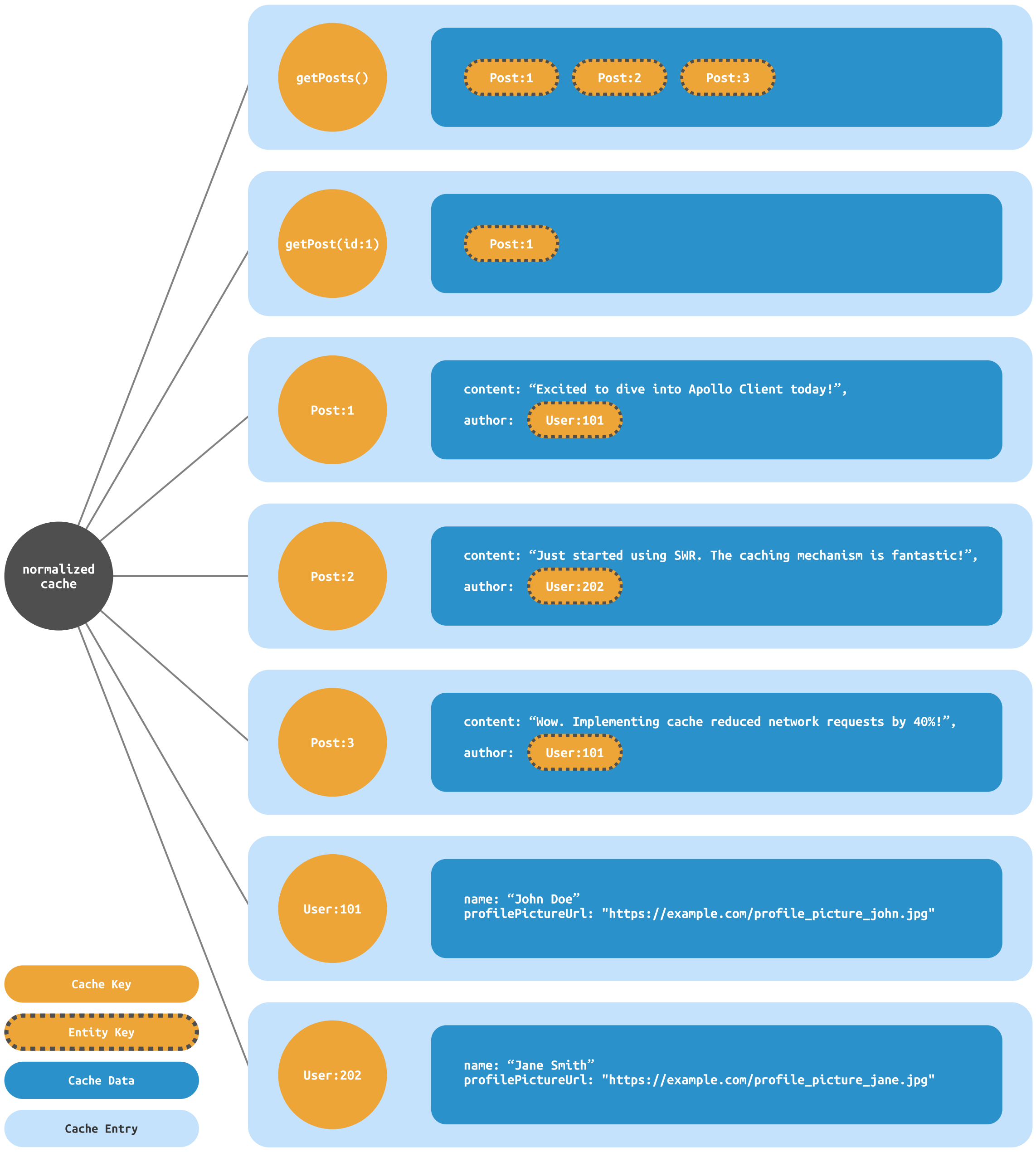
Entity keys are used to uniquely distinguish entities from each other, and they are used as *cache keys* in the cache. An *entity key* consists of an entity *type* and *id*. For this reason, a requirement for creating a normalized cache is that the server API provide an entity *type* and *id*. Because the general server state management libraries does not assume anything about the server API, they cannot assume that this information is provided. Consequently, to enable a normalized cache, it becomes necessary to utilize a server API that offers this information, along with a server state management library *specialized* for this type of server API.

A *type* is metadata about an entity, and in our Facebook feed example, the types could be `Post` and `User`. The entity *id* need to uniquely distinguish the entity from other entities of the same *type*. An *entity key* for the author John Doe could be `User:101`, consisting of the *type* `User` and *id* `101`.

Once the *entity keys* are established, we can start structuring our cache in a normalized form. For queries, the *query key* is used as *cache key*, and for entities, the *entity key* is used as *cache key*. If an entity contains another entity, this other entity is stored in the cache as its own *cache entry*, and its *entity key* is used as a reference in the consuming entity. This way, each entity is only stored once in our cache, and redundant data is removed.

Figure 7.5 shows an overview of a normalized cache. Each query and entity has its own *cache entry*, consisting of a *cache key* and associated *cache data*. The *cache data* in the queries `getPosts()` and `getPost(id:1)` contains *entity keys* which are references to other entites, such as `Post:1`. The *cache entries* with *entity keys* as *cache keys* store the actual data of the entity. These *cache entries* may also include references to other related entities, such as `Post:1`, authored by the user with *entity key* `User:101`, representing John Doe in this example.

A normalized cache only stores one object for every entity received from the server. As shown in figure 7.5, the author John Doe with `userId: 101` is only stored once, compared to the non-normalized cache shown in figure 7.4 where the author John Doe was stored three times. This example illustrates how data duplication is removed in a normalized cache.



getPosts()
Post:1
Post:2
Post:3
getPost(id:1)
Post:1
Post:1
content: “Excited to dive into Apollo Client today!”,
author: User:101
normalized cache
Post:2
content: “Just started using SWR. The caching mechanism is fantastic!”,
author: User:202
Post:3
content: “Wow. Implementing cache reduced network requests by 40%!”,
author: User:101
User:101
name: “John Doe”
profilePictureUrl: "https://example.com/profile_picture_john.jpg"
User:202
name: “Jane Smith”
profilePictureUrl: "https://example.com/profile_picture_jane.jpg"
Cache Key
Entity Key
Cache Data
Cache Entry

Figure 7.5: Overview of a normalized cache. Each query and entity is represented by a dedicated *cache entry* consisting of a *cache key* and associated *cache data.* The *cache data* within the queries `getPosts()` and `getPost(id:1)` contains *entity keys*, serving as references to other entities, such as `Post:1`. The *cache entries* that has *entity keys* as *cache keys*, store the actual entity data. Additionally, these *cache entries* can include references to related entities, such as `Post:1` authored by the user with *entity key* `User:101`, John Doe in this example.

Automatic Cache Updates

Having a *normalized* cache enables new options for managing outdated data caused by mutations. As we have seen, a simple way to handle outdated data is to re-fetch all queries affected by the mutation, using *query invalidation.* In the case of a *query cache*, this is often the preferred approach, as the modified entity may be stored at multiple locations in the cache, making it complicated to edit all these places manually after a mutation. However, with a *normalized* cache, all entities are only stored once. This means that after a mutation we only have to modify this *single* entity in this *single* location, which is a much simpler operation.

Let us say that John Doe changes his profile picture, making the previous `profilePictureUrl` on his `User` entity outdated. Because the entity is stored only once, we only need to modify the single John Doe entity in the cache to correct the outdated data. `Post:1` includes a reference to John Doe's entity, with the *entity key* `User:101`. When John Doe's entity is modified, the *entity key* stays the same, as *entity keys* are stable. `Post:1` does not need to change its reference to John Doe's entity, as it is the entity itself that has been modified, not the reference.

Furthermore, if the server returns the modified entity from the mutation, including its *type*, *id* and the modified fields, the library can construct an *entity key* using this information, and use it to modify the entity in the cache *automatically*. Due to the cache being normalized, no queries may need to be re-fetched. However, this approach would only work for updating entities that already exist in the cache. If the mutation for example creates a new entity, some manual work may be required.

Manual Cache Updates

In the example above with John Doe changing his profile picture, the cache was updated automatically. This is because the mutation on the server only modified a field in a single entity. However, updating a normalized cache is not as straightforward when *creating* or *deleting* entities. In figure 7.5, we can see that the cache stores the individual `Post` entities using their corresponding *entity keys*. Additionally, it also stores the query `getPosts()` associated with its respective *query key*, which contains a list of posts. If we execute a mutation that creates a new `Post`, the new `Post` entity will be stored automatically in the cache by the server state management library, but it will not be added automatically to the list in the `getPosts()` query. A newly created entity will itself be added automatically to the cache, but developers need to manually add or remove a reference to the entity in every cached query that contains a list that *should* include the newly created entity.

The aforementioned problem of updating references in cached queries is demonstrated in figure 7.6. The figure shows a representation of a normalized cache*after* a new post with the *entity key* `Post:4` is created. The entity data is stored automatically as a *cache entry* alongside the corresponding *cache key*. However, the `getPosts()` query does not contain a reference to `Post:4`. Furthermore, there is no way for the server state management library to know that `getPosts()` *should* contain `Post:4`. Thus, we need a way to reflect the modifications done on the server in our cache as well. There are two possible solutions for this: *query invalidation* or *manual cache update*. *Query invalidation* was covered in section 7.3.1 and denotes the process of marking the affected queries as invalid, triggering a re-fetch.

Manual cache updates is employed with the single purpose of updating all affected queries when an entity is added or removed. Assuming that the successful mutation returns, at a minimum, the entity *type* and *id*, we can construct the *entity key*, which can be added as a reference to all the affected queries. Nevertheless, the cache can *not* infer *where*, *if*, or *whether* an entity reference should be added or removed from a cached query. Thus, it is required by the developers to know when and where to add a reference from an entity to a cached query. Figure 7.6 shows that the query `getPosts()` is missing the post `Post:4`, hence a *manual cache update* would add a reference for `Post:4` to the list of posts. Once a *manual cache update* has been executed successfully, the *cache entry* for the query should appear as follows: `getPosts(): [Post:1, Post:2, Post:3, Post:4]`.

To summarize, *updating* an entity requires neither *manual cache updates* or *invalidation* because the reference to the current entity is already present in the cached queries. If the author of `Post:1` edits the content of the post, only the entity itself is updated in the cache, and the query `getPosts()` will not need to be updated because the *entity key* `Post:1` is stable and does not change even when the entity data changes. However, when you *create* or *delete* entities, there may be queries and entities that should contain references to these new entities. Server state management libraries are unable to infer these modifications automatically, thus, *query invalidation* or *manual cache updates* are required in order to keep data updated.

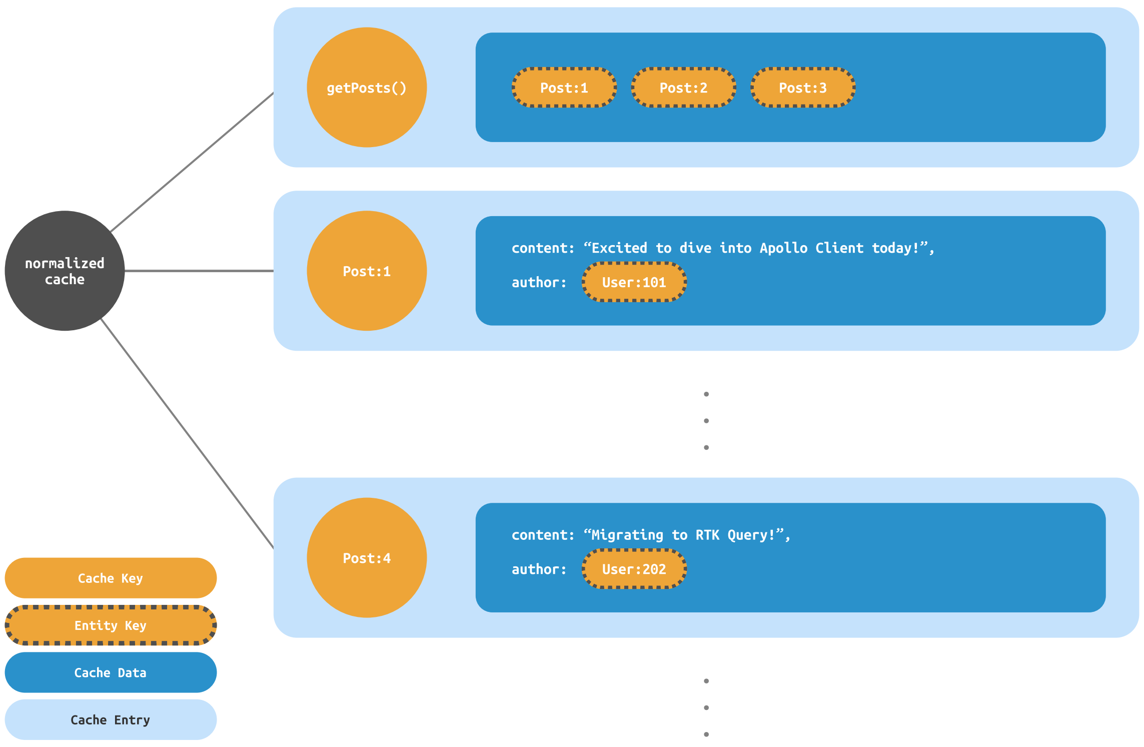


Figure 7.6: Representation of a normalized cache *after* the creation of a new post. The entity data for the new `Post:4` is automatically stored in a *cache entry* alongside its corresponding *cache key*. However, the `getPosts()` query does not include a reference to `Post:4`. The server state management library lacks the knowledge that `getPosts()` *should* include a reference to the entity `Post:4`. *Manual cache updates* are necessary in order to add `Post:4` to the *cached data* of `getPosts()`.

7.4 Organization of queries

An important consideration when working with server state management is how to organize and configure queries. The organization of queries refers to where queries are defined within the codebase, while query configuration is about defining the specifics for a given query, such as endpoint and parameters. There are two main approaches to the configuration and organization: *centralized* or *decentralized*. In *centralized* organization, queries are defined in one single location and then imported and used wherever needed. In contrast, *decentralized* organization involves defining queries directly in the components or hooks that use them.

7.4.1 Decentralized

In the *decentralized* approach, queries are configured directly in the hook that execute them. There is no centralized entity where you would pre-define all the different queries, just the query hooks being used in multiple locations. For example, in a library that has a query hook called `useQuery`, the query could be configured like this:

```
const Component = () => {
    const { data } = useQuery(
        // query configuration
    );

    // ...
}
```

The hook can be encapsulated inside your own custom hooks, enabling the reuse of queries and locating the hook outside the component. For example, a query to fetch your friends on Facebook could look like this:

```
const useFriendsQuery = () =>
    useQuery(
        // query configuration
    )

const Component = () => {
    const { data } = useFriendsQuery();

    // ...
}
```

Using this approach, it is possible to create custom hooks for all different queries. For example, you could have another custom hook called `useFeedQuery`, that executes a query to fetch your Facebook news feed. In that case, `useFriendsQuery` and `useFeedQuery` would be unrelated, possibly defined in different files, and not have anything in common other than being managed and cached by the same server state management library. Figure 7.7 shows an illustration of how the aforementioned hooks may be organized. In the figure, the hooks are not located at the same place, indicating a *decentralized* organization of queries. The reuse of the `useFeedQuery` in both the `<Profile />` and `<Messenger />` components highlights the possibility of reusing queries when defined outside components. An important characteristic of decentralized queries is their independence from each other, enabling their *decentralized* placement.

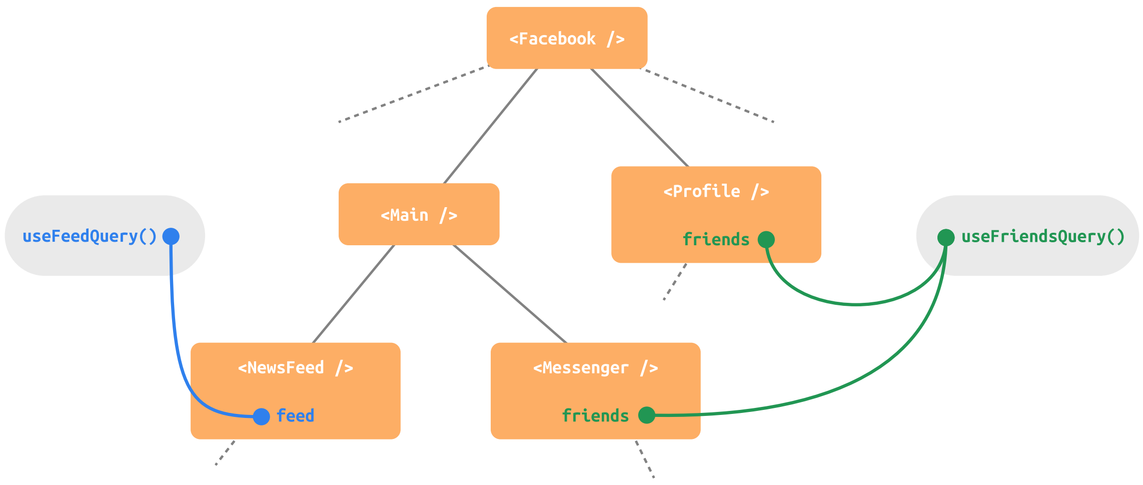


Figure 7.7: The figure illustrates a decentralized organization of queries, with the hooks `useFriendsQuery` and `useFeedQuery` located separately. This highlights an important characteristic of a *decentralized* organization, which is the queries' independence from each other, allowing for a decentralized placement.

Due to the absence of constraints on where the custom hooks are located, the *decentralized* libraries allows the co-location of custom hooks with the components that use them, or extracting them somewhere else. While it is possible to mimic a *centralized* organization by grouping all queries into the same file, each query would still need to be configured individually.

Decentralized organization offers flexibility and modularity, but it can also present challenges when it comes to managing your queries. Although the server state management library would manage and cache all queries being executed, it would not know anything about your custom hooks you use to encapsulate the queries. For example, the server state management library cannot know the names of these custom hooks. It is for this reason that the *decentralized* libraries require manual *query keys* to identify queries, as discussed in section 7.3.1.

While *decentralized* organization requires configuring each query individually, it also allows defining global default configurations. For example, fetcher functions are configured per query, but developers can provide a global default fetcher function that will be used if no fetcher is provided to a given query.

7.4.2 Centralized

In contrast, the *centralized* approach configures all queries in the same location. Using the example of a Facebook friends query and a feed query, these query configurations would be co-located, for example in the same configuration object that might look something like this:

```
const queries = {
    friendsQuery: {
        // query configuration
    },
    feedQuery: {
        // query configuration
    },
    // ...
}
```

Due to the queries being configured in a *centralized* location, they have to be uniquely identified to enable the server state management library to keep them apart. When you want to execute the queries from a component, you would use the query identifier to identify the query to be executed. For example, a *centralized* server state management library may provide a query hook that takes the query identifier as argument and executes the associated query. RTK Query solves this by providing custom generated hooks for each query that you can use in the components.

Figure 7.8 shows how queries are co-located using the *centralized* approach. The `<Profile />` and `<Messenger />` components use the same query for fetching friends. The `<NewsFeed />` component use the feed query, and because of the nature of the *centralized* approach, it is retrieved from the *same* location as the friends query.

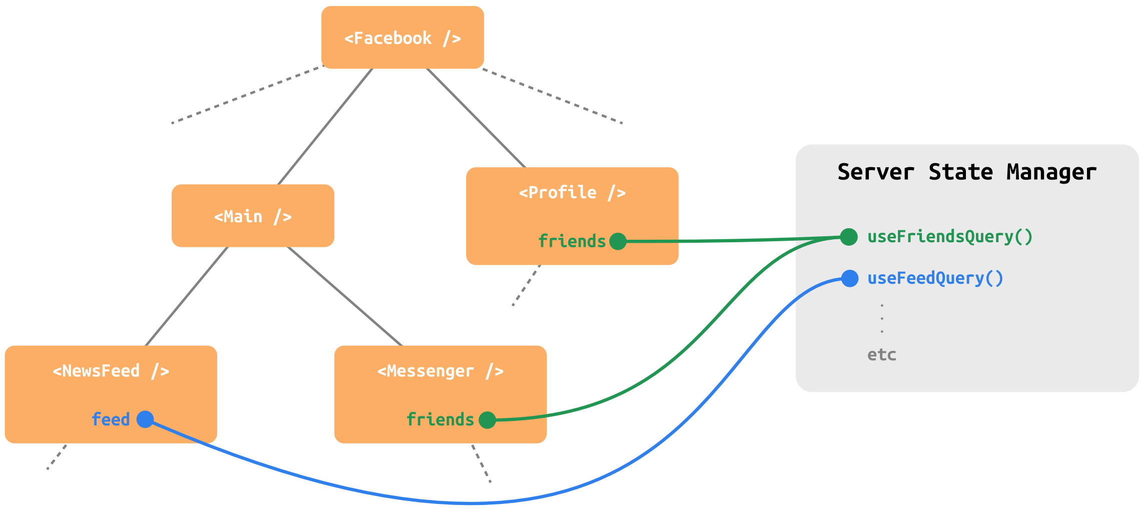


Figure 7.8: Overview of the *centralized* approach, where queries are co-located. The `<Profile />` and `<Messenger />` components both consume the `useFriendsQuery`, and the `<NewsFeed />` component retrieves the `useFeedQuery` from the same, *centralized* location.

Because a *centralized* approach requires identifying the queries to keep track of them, they enable things like automatic generation of *query keys* using the specified query identifier. This is the reason that RTK Query, which is an example of a *centralized* server state management library, can automatically generate query keys, even though it is a *general* library, as discussed in section 7.3.1.

While the *centralized* approach to query organization may be less modular and flexible than a *decentralized* approach, it provides a clear overview of all queries, helping to maintain consistency across an application. The libraries can provide options of splitting the configuration into separate sections, enabling you to provide different default values for different types of queries. For example, while the fetcher function could be centralized for all queries, it is possible to allow the creation of separate sections that utilizes other fetcher functions. Often it is also possible to provide configuration options when you execute the query in the component, giving further individual customization options.

In summary, the *centralized* and *decentralized* approaches to query organization are a more nuanced distinction than they may appear. The *decentralized* approach starts as individual query configurations, defined in the components that use them. However, they may be extracted into custom hooks, co-located with other queries, and utilize defined global default values. Conversely, the *centralized* approach initially involves all queries being configured in a single location, but can still offer modularity by the separation of queries into different sections with different default values, and individual configuration on execution. *Decentralized* and *Centralized* can be viewed as two ends of a continuum. While individual configuration gives flexibility, default settings ensure consistency across queries. Server state management libraries may start at one end of the continuum, but they often allow you to take steps in the other direction, enabling you to find the balance between structure and customization that fits your needs.

7.5 Asynchronous Metadata

Performing a network request from client to server usually takes some time. For this reason, network requests are executed using asynchronous JavaScript, as discussed in section 2.9. However, the client requesting the data from the server needs a way to monitor the process of the asynchronous request. Consequently, server state management libraries provide *asynchronous metadata* to let the consuming components know the current status of a network request.

Asynchronous metadata can be utilized to determine what interactions should be enabled or disabled, or what is shown as a response to the user while there is an ongoing asynchronous request. Furthermore, an asynchronous request over a network might fail due to a network fault, server error or other various reasons. Hence, asynchronous metadata should also specify whether the request was a success or a failure.

For example, when a user submits a form for signing up as a new user on a website, it would be appropriate to indicate to the user that the system *is* currently processing the request. The button label could transition from “Sign up” to “Signing up”, and potentially show a spinner animation. In addition, the button could be *disabled*, in order to prevent subsequent form submits. After some time, when the request is fulfilled, the system should clearly display whether the request resulted in a success or an error. An error could show a notification on the top of the screen, containing a description of the error, while a successful result should clearly indicate that the user has been registered successfully.

Determining *how* the user interface changes during an asynchronous request is a topic that belongs in the field of user experience, and it is out of scope for this thesis. Nevertheless, an essential aspect of *asynchronous metadata* is that it can *enable* the user interface to show these changes. To retrieve and obtain information about an asynchronous request, two complementary statuses are introduced: *data status* and *fetch status*.

7.5.1 Data Status

Data status indicates whether an asynchronous request is in progress, was successful, or encountered an error. The fundamental states of *data status* are *loading*, *success* and *error*, but more states could be added if needed. The states in a *data status* are mutually exclusive, meaning they cannot exist concurrently. For example, a system should not allow the states *success* and *error* to exist at the same time. Consequently, *data status* is *qualitative*, as described in section 6.1.2. Finite state machines can be used to represent qualitative states such as *data status*, and figure 7.9 presents a finite state machine with the three states *loading*, *success* and *error*.

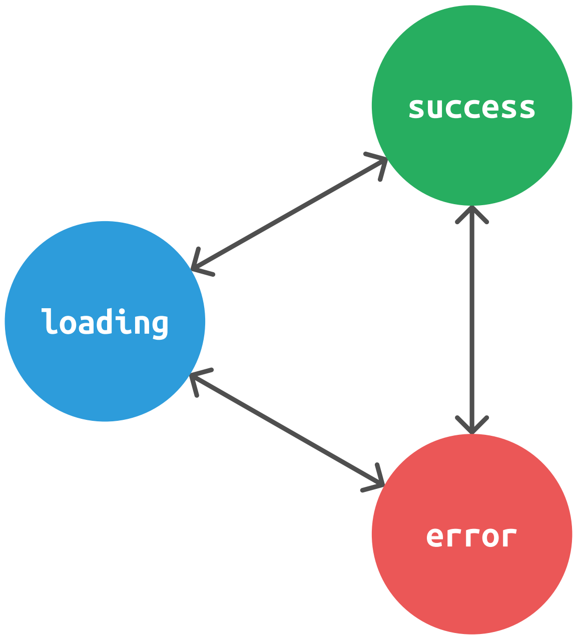


Figure 7.9: Finite state machine representing data status with the states *loading*, *success* and *error*. Transitions between all states are possible.

The *data status* is frequently utilized to determine, based on the current state, what information should be displayed in a user interface. Often, booleans like *isLoading*, *isSuccess* and *isError* are *derived* from *data status* to provide convenient access to the status information. An example employing *data status* to conditionally render elements to a user interface is shown below. The component queries a list of Facebook friends, and manages loading and error states for the corresponding query.

```
const Component = () => {
    const { data, isLoading, isError } = useFriendsQuery()

    if (isLoading) return <Loading />

    if (isError) return <Error />

    return <div>{data.map((friend) => <Friend friend={friend} />)}</div>
}
```

In this code example, the component starts by checking whether *isLoading* or *isError* are true, and shows an appropriate feedback. Due to the nature of qualitative state being mutually exclusive, conditionally checking and excluding two out of the three possible states will imply that the current state is the last one. Hence, we know that *data status* is *success* whenever the two former are false, and a list of friends is displayed.

As discussed in section 7.2, queries are usually executed automatically when a component mounts, whereas mutations are triggered manually by an event or a user input. Accordingly, *data status* of mutations are a bit different from *data status* of queries, and should include a state for when it is uninitialized as well. Thus, the state *uninitialized* is employed in mutations, and it is true as long as the mutation has not yet been executed. Some server state management libraries also include the *uninitialized* state on queries as well, which can be useful if the query is configured not to execute automatically on component mount.

7.5.2 Fetch Status

Recall the HTTP cache invalidation strategy Stale-While-Revalidate [61]. The re-validation of data in this strategy implies that there is data already present in the cache. If not, there would be no data to actually *re-validate*. This shows that when utilizing cached data, there is a need to distinguish between querying data to an empty cache entry, or querying data when we already have data present in the cache entry. For this reason, among others, *fetch status* is introduced. *Fetch status* contains information about the most recent query operation, and its fundamental states are *fetching*, *idle* and *paused*. Figure 7.10 illustrates *fetch status* as a finite state machine containing the aforementioned states. The following sections will describe the characteristics of *fetch status*, why it is useful and how it complements *data status* in various ways.

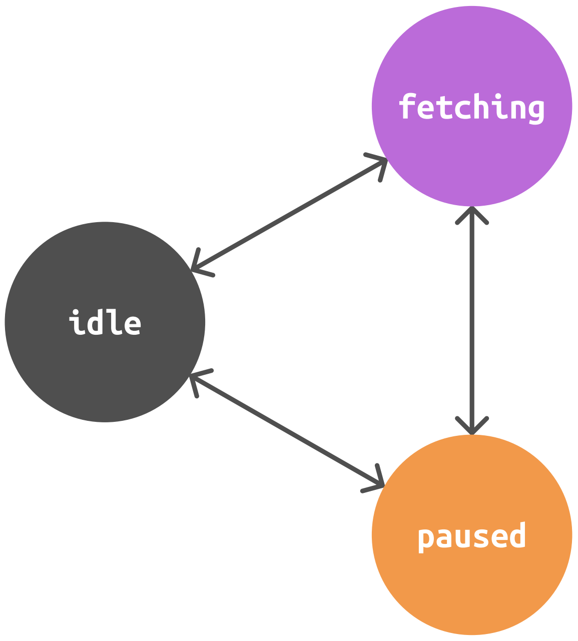


Figure 7.10: Finite state machine representing fetch status with the states *idle*, *fetching* and *paused*. Transitions between all states are possible.

Data status and *fetch status* are qualitative states that exist in parallel and change independently. *Fetch status* differs from *data status* because it does not take into account whether there is existing data in the cache or not. *Fetch status* can be employed in order to implement an *indication* of a background re-fetch in an application, such as a Facebook feed. On initial load of the Facebook feed, *data status* will be *loading*, and *fetch status* will be *fetching*. When the feed is loaded, *data status* will be *success*, and *fetch status* will be *idle*. After this, a user can force a refresh of the feed, which will show a spinner animation while still showing the current feed data. In this case, *data status* will still be in the *success* state, because the initial load was successful, but *fetch status* will be in the *fetching* state once again. This refresh of the feed triggered by the user is an example of a re-validation of *existing* data in the cache. If there were no new posts to show in the feed, the data already present in the cache were valid, but if the latest fetch returned new posts, these are added to the feed, and the re-validation resulted in an update of the Facebook feed.

The *idle* state indicates that there is currently no ongoing requests, and the query is ready to be executed. By default, *fetch status* of a query is set to the *idle* state when initialized. The *idle* state can be described as a state where the query is passive and not doing anything.

The last state of *fetch status* is the *paused* state. It indicates that a request is *prevented* from execution, or that an *ongoing* request has been stopped. A handy utilization of the *paused* state is related to network connectivity. Let us say that a user is uploading some files to a web service, but during the upload, network connectivity is lost. Consequently, the network request for uploading files is aborted, resulting in *fetch status* being set to *paused.* The *paused* state can for example be utilized to show an appropriate message to the user regarding the paused file uploads, and it might encourage the user to take actions to restore network connectivity. Whenever network connectivity is restored, *fetch status* transitions from *paused* to *fetching*, and the user can be informed that the upload has been resumed.

Leveraging both *data status* and *fetch status* can provide valuable information to applications regarding network and data requests in various scenarios. *Data status* provide information about whether there are any data available at the moment, and whether a request was successful or resulted in an error. *Fetch status*, on the other hand, complements *data status* because it contains details about whether or not there are any active requests. This can be utilized to provide indications of background re-fetches or a loss of network connectivity. *Data status* and *fetch status* work together to keep users informed about ongoing requests in an application, enhancing overall user experience.

7.6 Integrated Client State

The main focus of this chapter is *server state.* It is important to note, however, that many applications need to manage both *server state* and *client state. Server state* originates from a server, and modifications to the state are usually related to an asynchronous network request. On the other hand, *client state* originates on the client itself, and modifications are performed synchronously. Consequently, the approach for managing *client state* and *server state* differ due to their distinct natures and characteristics. However, some server state management libraries choose to integrate *client state* management as part of their solution, and this section will explore this aspect in more detail.

If the application requires little or no *client state*, utilizing the internal state management provided by React might be satisfactory. Otherwise, if the internal state management provided by React is not suitable, an external library can be employed, resulting in separate libraries for *client state* and *server state.* However, utilizing this approach require cautiousness regarding the consolidation between *client state* and *server state.* For example, it becomes easier to *unintentionally* duplicate state by storing the *same* data as both *client state* and *server state.* Also, *client state* reacting to changes in *server state*, and conversely, might be sources of unnecessary re-renders if not managed properly.

In response to these challenges, among others, some *server state* management libraries provide internal methods for managing *client state*, hereby referred to as an *integrated* approach. With this approach, it is possible to manage both *client state* and *server state* with a *single* library. The *integrated* approach also allows the sharing of concepts and potential custom features between *client state* and *server state*, as the management strategies reside within the same library. The *integrated* approach facilitates the co-location of state, enabling *client state* to react directly to modifications on the *server state.* While the possibility of data duplication still exists, it is greatly reduced, as the *server state* and *client state* can directly depend on each other, eliminating the need to store a duplicate when a combination of the two is required.

Managing both *client state* and *server state* is crucial in many applications. For minimal *client state* requirements, React's internal state management suffices, but for more complex requirements, external libraries may be necessary. In the *integrated* approach, a server state management library handles *client state* internally. This enables co-location of state, allowing *client state* to react to *server state* modifications. Additionally, it facilitates the sharing of concepts and potential custom features between the two.

7.7 Server State Features

Now we have a basic understanding of the cache and its main functionality. However, as mentioned when introducing the cache, caching is a central concept when it comes to server state management, and basic caching mechanics will not cut it. What follows are some of the configurations and features of server state management libraries, enabled by the cache.

7.7.1 Prefetching

The idea behind *prefetching* is to anticipate what data will be accessed next and to load it into the cache before it is actually needed. This can significantly reduce the amount of time that would otherwise be spent waiting for data to be loaded from the server. For example when displaying a paginated list to the user, we might assume that the user will navigate to the next page. Without prefetching, a user navigating to the next page would need to wait some amount of time for the page data to load. However, if the data for the next page was prefetched, a navigation to the next page would show its content instantly, providing a better user experience.

One might be tempted to prefetch everything, but it is not without its costs. Prefetching uses network bandwidth and memory resources for data that the user potentially might not need. If the user decides *not* to navigate to the next page for example, the prefetched data would just have resulted in an unnecessary network request. A lot of unnecessary network request can put the server under an unnecessary heavy load, slowing down the requests even more. Other times, it might not even be relevant to prefetch data, simply because the data may change quickly, and already be stale by the time it is needed. These are some of the reasons it is a good idea to think through where and when prefetching actually makes sense to use.

Prefetching data is not very different from running a normal query, except that it can be performed in the background, as nothing on the current page should depend on what we prefetch for the next page. While Tanstack, SWR and RTK Query all provide a custom method for prefetching data, Apollo and URQL simply has you performing a manual request through the client directly.

7.7.2 Pagination

As discussed in section 2.13, pagination is a common data fetching pattern. For this reason, most server state management libraries aim to simplify and optimize pagination by providing built-in support for it. In this section, we will discuss the general concepts and strategies used by these libraries to handle pagination.

When performing a request for paginated data, it is necessary to specify what part of the data to retrieve, or, in some cases, what *page*. This is achieved by including special *pagination parameters*, such as *page number*, *limit*, *offset*, or *cursor*, that specify exactly what data you want. The *pagination parameters* are included in the query parameters along with other relevant parameters. Server state management libraries provide mechanisms to update the *pagination parameters*, such as the page number or cursor, for example when a user clicks on the next page in the user interface. Upon updating the *pagination parameters*, the library will trigger a new fetch request, fetch the next set of data, and update the user interface accordingly. This ensures a seamless user experience while navigating through the paginated data.

Different combinations of *pagination parameters* will result in different *cache entries.* In some cases, you may not want to have different *cache entries* for different pages, but keep all the data together. This may be relevant when showing more than one page at a time, for example when implementing an infinitely scrolling list, like a Facebook feed. For such cases, regular query hooks will not suffice, as they will create new *cache entries* for all different *pagination parameters.* Instead of loading a *new page*, you want to *load more* data. The server state management libraries often provides custom hooks that handle this kind of pagination, or enable the specification of how results from requests with different *pagination parameters* should be merged together.

It is worth noting that pagination is a scenario where both *data status* and *fetch status* can provide valuable information. When navigating to a new page, you might want to still show the previous data, while also showing an indication that the new page is loading.

In conclusion, server state management libraries effectively handle pagination by utilizing *pagination parameters*, cache management, and data and fetch status. When working with pages that are shown separately, it is often enough for the libraries to provide a seamless way to update the *pagination parameters.* However, when working with paginated data that should be loaded into the same list, things are not that simple. To manage this, the libraries can implement custom hooks, or enable ways of merging the results. By offering support for various pagination scenarios, the server state management libraries enable smooth navigation through large datasets, resulting in an enhanced user experience.

7.7.3 Fetch Policy

When you execute a *query*, it populates the cache with the response. When you execute the same query again later, it does not need to perform a new network request, as it can find the data in the cache. This illustrates a basic usage of the cache, but sometimes this is not the behaviour you want. Maybe you may want to show the data from the cache, but perform a query in the background, to verify that it has not become stale. Or the query may be of such a nature that you do *not* want to use potentially stale data from the cache at all. This could for example be the case when querying rapidly changing, real-time data.

This is where *fetch policy* comes in. We use the term *fetch policy* to talk about the strategy in use when fetching data. The last paragraph outlined the most common strategies, and using Apollo and URQL's terminology, we call these strategies *cache-first*, *cache-and-network* and *network-only.* While Apollo and URQL has abstracted these options into predefined categories, the other libraries lets you configure the fetch policy using various timers for when cached data is to be considered stale.

7.7.4 Middleware

Middleware is a powerful mechanism used to add custom behaviour to network requests sent from a server state management library. It enables the interception and modification of these requests before they reach the server, and the processing of the responses before they are returned to the library. In addition to modifying requests, middlewares can be used to perform side effects, such as logging. Typically, middlewares are composed in chains of middlewares that are executed in a sequential manner. Therefore, a singular middleware in this chain often has a singular focus or purpose, and a collection of middlewares can be chained in order to achieve advanced behaviour.

Similar to organization of queries as discussed in section 7.4.2, *middlewares* are *centralized.* Therefore, middlewares are especially useful in situations where there is a need to incorporate custom behaviour for potentially *all* requests. Some examples of such features are user authentication and retrying failed requests. When it comes to user authentication, a middleware can be utilized to initiate the process of refreshing an access token if the server returns an unauthorized message. Moreover, *retry* behaviour as described in section 2.11.1 can be implemented using middlewares. In the event of a request failure, a retry middleware will initiate a retry of the same request, given that the maximum number of retries has not been reached. Retry is an example of a feature that can be useful to apply to many requests in an application, making it an ideal use-case for a middleware.

7.7.5 Offline Mode

Most applications require an active network connection in order to communicate with various servers. When an application has an active network connection, we refer to the application as being in *online mode*. On the other hand, a client without an active network connection is referred to as being *offline*. If an application supports *offline mode*, users are empowered to perform certain operations even though there is no active network connection, as discussed in section 2.11.2.

In *online mode*, results from a successful query is stored in the cache. The result from the last query can still be present in the cache when the client transitions from *online mode* to *offline mode*, enabling the user to still view data that was fetched the last time it was online. Furthermore, some mutations may also be performed while using *offline mode*. For example, in a Facebook chat app, a user is able to send a message even though there is no active network connection. Whenever the user's network connection is restored, the message is sent as a network request to the server. As shown, both queries and mutations can employed while in *offline mode*. However, it is important to consider that not all operations are suitable to implement in *offline mode*. For example, when placing a bid on a property, it is crucial to ensure that the mutation has been successfully processed on the server.

Server state in the cache is stored in a *temporary storage*. To elaborate further, server state management libraries utilize an *in-memory* cache, which is stored in a device's Random Access Memory (RAM). As discussed in section 2.10, one feature of an *in-memory* cache is its fast retrieval speed. Because of its nature as a *temporary storage*, *in-memory* cache is deleted whenever the corresponding application stops consuming the data.

A *temporary storage* stores data only for a limited amount of time. For this reason, an application fully utilizing *offline mode* may also store cached server state in a *permanent* storage, in order to keep data across sessions or reboots. For example, web applications may persist cached data between sessions by utilizing browser-based WebStorage [63].

7.8 Summary

In this chapter we have presented a general theory of server state management, based upon the server state management libraries described in chapter 5. We started by examining the type of **Server API** the server state management libraries were aimed at, making a distinction between *general* and *specialized* libraries. *General* libraries are flexible enough to be used with most APIs, while *specialized* libraries tailor their experience toward a specific type of API. They may be less flexible, but often comes with more specific features. We moved on to categorize the different **Operations** a server state management library could perform when communicating with a server. Briefly stated, *queries* ask for data, *mutations* make modifications to data, and *subscriptions* listens to real-time-changes.

Cache is one of the major areas of server state management, and a substantial portion of the theory is dedicated to exploring it. An important concept is *cache keys*, being the identifier of a *cache entry*, containing the data in the cache. To cache queries, a query is identified by a *query key*, which is then used as its *cache key*. This is the basic mechanism of the most basic type of cache, which we call *query cache*. A more advanced form of caching can be achieved by utilizing a *normalized cache*, involving the automatic normalization of the data. Here, the queries are still cached using their *query keys*, but the individual entities of a query are cached separately. Entities are identified with *entity keys*, which are used as their corresponding *cache key*. The *normalization* approach is mostly used in *specialized* libraries, as its complexities can be more easily addressed when the data structure is known.

The caching strategy also has consequences for the invalidation of data in the cache. Using *query cache* the developer is responsible for invalidating certain queries when appropriate, to perform a re-fetch. However, using *normalized cache*, it is possible to avoid much of the invalidation, with the library automatically modifying relevant entities in the cache upon mutations. However, in some cases a *normalized cache* demands more advanced forms of invalidation, specifically regarding relations in lists, which requires manual cache modifications.

Moving on from the cache, we looked at the **Organization of Queries**, specifically whether the queries are defined in one, *centralized* location, or rather *decentralized*, in multiple different locations. **Asynchronous Metadata** is metadata on the status of a request. We divide it in two, one part being *data status*, giving information whether the data is loading or present, and *fetch status*, indicating whether or not a request is currently underway. We briefly mention how some server state management libraries offers **Integrated Client State**, which allows the management of client state within the same library as server state. We finish of the theory be going through some central concepts to server state management, that are considered to be standard as a part of a server state management library. This include the option of *prefetching* data, ways of efficiently handling *pagination*, configuring the *fetch policy* strategy to use when fetching data, how *middleware* can be used to enhance the features of a library, and lastly, how *offline mode* can enable using the application without a network connection.

Chapter 8

Verification

Chapter 6 and 7 presented our generated theories of client and server state management, making those chapters our attempt at answering **RQ1**. In this chapter, we turn our attention to the *verification* of these theories. The theories are applied to different scenarios to evaluate their practical usefulness, thereby addressing our second research question, **RQ2**: "What are the potential scenarios where a generalized theory of state management can be employed, and how might these enhance understanding and practice in this field?"

Firstly, the theories are utilized to categorize and generate an overview of the different state management libraries used in their generation, as well as two libraries that were not present in the original set. This demonstrates a practical scenario utilizing our theories, and verifying their extendibility to other libraries. Next, the theories are applied to a typical real-world scenario, modelling state management in a project management tool. This illustrates how our theories can give developers a shared understanding of concepts and methods, enabling collaborative reasoning about state modelling that remains applicable across various state management libraries. Lastly, we explore how the theory of *client* state management can be utilized to gain an enhanced understanding on the field of *server* state management. In other words, the theory of client state management is used to discuss server state management. This demonstrates the interplay between the two theories of state management, and shows their versatility.

Through these verification exercises, we hope to demonstrate the value and effectiveness of our theories as tools for enhancing understanding, discussing, and reasoning about topics related to state management, effectively answering **RQ2**.

8.1 Categorizing State Management Libraries

The process of generating our theories involved a comprehensive review of various state management libraries. In other words: the *generalized* theories were synthesized from *specific* libraries. Therefore, a logical first step in validating the theories is to verify that they accurately describe the libraries used as basis for their construction. This section will leverage our theories to categorize and catalog the different state management libraries that contributed to their formulation. This approach also demonstrates the theories' practical utility in understanding diverse state management libraries.

As an extra step of verification, the categorization include two libraries that were not part of the original list of libraries: a client state management library called Effector [64], and a server state management library called Relay [49]. This is to evaluate the applicability and extendibility of our theories to libraries *beyond* those from which it was initially derived. These new library additions will appear at the bottom of their respective tables.

8.1.1 Client State Management Libraries

Table 8.1 shows all client state management libraries used when generating our theory of client state management, in addition to the Effector library, categorized according to this theory. Not all categories of our theory are relevant when categorizing the libraries, and as such, *state type* and *derived state* are omitted from the table. *State type* is used to categorize *state* itself, and thus does not significantly differentiate between the libraries. Even though *server state* could have been included, we chose not to, as it is extensively covered in its own theory. *Derived state* is not included as its own category, because the usage of *derived atoms* are implied from an *atomic* state composition model, and *derived selector function* are included as a form of *state selection*.

Library	State Reach	State Composition Model	State Selection	State Modification
React Core	LOCAL GLOBAL INSIDE REACT	ATOMIC		IMMUTABLE LOCAL MODIFICATION LOGIC
Redux Toolkit	GLOBAL OUTSIDE REACT	SINGLE TREE	SELECTOR FUNCTION DERIVED SELECTOR FUNCTION	IMMUTABLE GLOBAL MODIFICATION LOGIC
MobX	LOCAL GLOBAL OUTSIDE REACT	FRAGMENTED STORE	ACCESS BASED	MUTABLE GLOBAL MODIFICATION LOGIC
XState	LOCAL GLOBAL OUTSIDE REACT	FRAGMENTED STORE	DERIVED SELECTOR FUNCTION	IMMUTABLE GLOBAL MODIFICATION LOGIC
Zustand	GLOBAL OUTSIDE REACT	SINGLE TREE FRAGMENTED STORE	SELECTOR FUNCTION DERIVED SELECTOR FUNCTION	IMMUTABLE GLOBAL MODIFICATION LOGIC
Recoil	GLOBAL INSIDE REACT	ATOMIC	ATOMIC	IMMUTABLE LOCAL MODIFICATION LOGIC
Jotai	GLOBAL INSIDE REACT	ATOMIC	ATOMIC	IMMUTABLE LOCAL MODIFICATION LOGIC
Elf	GLOBAL OUTSIDE REACT	FRAGMENTED STORE	SELECTOR FUNCTION DERIVED SELECTOR FUNCTION	IMMUTABLE GLOBAL MODIFICATION LOGIC
Valtio	GLOBAL OUTSIDE REACT	ATOMIC	ACCESS BASED	MUTABLE LOCAL MODIFICATION LOGIC
Effector	GLOBAL OUTSIDE REACT	FRAGMENTED STORE	SELECTOR FUNCTION DERIVED SELECTOR FUNCTION	IMMUTABLE GLOBAL MODIFICATION LOGIC

Table 8.1: Overview of the client state management libraries utilized to generate our theory of client state management, in addition to the Effector library. The libraries are categorized using our theory of client state management.

Categorizing the libraries can be challenging, as many libraries are designed to be highly flexible and adaptable, making them hard to pin down into rigid categories. For example, when it comes to **State Composition Model**, Zustand gives first class support for a *fragmented store* model, while still facilitating for a *single tree* model. Consequently, our categories are not mutually exclusive, and Zustand is listed as both *single tree* and *fragmented store*.

It is also crucial to mention that our categorizations are primarily based on the intended use of the libraries, not on all possible ways of using it. Most libraries could technically be manipulated to fit most categories. For example, Jotai, traditionally an *atomic* library, theoretically allows you to put all state inside one giant atom, making it utilize a *single tree* model. However, categorizing based on every potential usage would lead to a classification that is less meaningful and practical. Therefore, our categorization represents the most fitting categories when using the libraries as they are primarily intended to be used.

The final entry in table 8.1 introduces the Effector client state management library. This library was not included in the original collection used to develop the theory, thereby serving as a test of our theory's extendibility. It was categorized by conducting a brief review of its documentation. Several of Effector's characteristics easily align with the categories of our theory: it has a *global* **State Reach** *outside* React, and employs *selector functions* that can also function as *derived selector functions*. In terms of **State Modification** it leverages *immutable* updates and *global modification logic*.

Effector's **State Composition Model** proved to be less straightforward. As mentioned previously, most libraries can be adapted to use most of the composition models. This can make the model difficult to pinpoint, as an effort has to be made to discover the *intended* usage from the library documentation. For example, Effector allows you to define state at a granular level, which could potentially be employed to implement an *atomic* model. Nonetheless, as depicted in table 8.1, we opted to categorize it as a *fragmented store*. This classification is due to its other library features that facilitate building state beyond just *atoms*, such as the aforementioned *selector functions*.

8.1.2 Server State Management Libraries

Table 8.2 presents all the server state management libraries used in generating our theory of server state management, in addition to the Relay library, categorized using this theory. However, as was the case with the client state management library categorization, we have excluded certain categories from this classification. **Operations** is a category describing the different types of requests a server state management library can execute, and is not relevant to the library categorization. Similarly, we have excluded **Asynchronous Metadata**, along with the various server state features covered in section 7.7. These are all features supported in various ways by the libraries. Including them in our categorization would necessitate a detailed comparison of their respective implementations, a level of granularity beyond the scope of our theories. Hence, we have chosen to omit these categories as well.

Library	Server API	Cache	Organization of Queries	Integrated Client State
Tanstack Query	GENERAL	QUERY CACHE	DECENTRALIZED	NO
RTK Query	GENERAL	QUERY CACHE	CENTRALIZED	YES
SWR	GENERAL	QUERY CACHE	DECENTRALIZED	NO
Apollo Client	SPECIALIZED	NORMALIZED CACHE	DECENTRALIZED	YES
URQL	SPECIALIZED	QUERY CACHE NORMALIZED CACHE	DECENTRALIZED	NO
Relay	SPECIALIZED	NORMALIZED CACHE	DECENTRALIZED	YES

Table 8.2: Overview of the server state management libraries utilized to generate our theory of server state management, in addition to the Relay library. The libraries are categorized using our theory of server state management.

The last library listed in the table is Relay, a server state management library not included in the original libraries used to generate the theory. Its categorization was derived by a brief review of its documentation. In contrast to the categorization of Effector in the client state libraries, Relay was relatively straightforward to categorize using our server state management theory. It is a *specialized* library, specifically tailored for a GraphQL API, just like Apollo Client and URQL. It utilizes a *normalized* cache, employs a *decentralized* organization of queries, and offers an *integrated* approach to client state.

It should be observed that Relay is categorized identically to Apollo Client, just like Tanstack Query is to SWR, which raises interesting considerations. It might indicate that our theory should be expanded to include categories that manage to differentiate between such libraries more effectively. Alternatively, the identical categorization might just validate the similarity of their fundamental models. It is worth noting, however, that despite identical categorization, each library's specific implementation details and functionalities still vary significantly. Rather than negating the individuality of each library, our theory provides a framework to understand the patterns and strategies used across the various state management libraries.

8.1.3 Evaluation

This section applied our theories in order to categorize the state management libraries from which they were derived. Even though not all concepts from the theories are relevant for such a categorization, we successfully managed to classify all libraries, which can be seen in table 8.1 and table 8.2. In addition, we also extended the categorization by including two libraries that were not part of the original set, demonstrating the extendibility of our theories.

The categorization of the client state management libraries presented some challenges when classifying the **State Composition Model**, particularly in the case of Effector. The exercise highlighted a possible weakness in our theory: this category might be too vague. The conceptual distinction between an *atomic* and a *fragmented store* composition model is relatively clear, yet, in practice these differences become blurred. In theory, atoms are indivisible units, the smallest unit of state. However, *atomic* libraries often lets you store large objects. Questions arise concerning how large atoms must be before they qualify as fragmented stores, and whether it is the size of the stored object alone that determines this categorization. To enhance the precision of this category, it could be beneficial to identify distinct features or characteristics that must be present before classifying something as a *fragmented store* as opposed to an *atom.* Although we managed to categorize Effector by considering the overall features of the library, our current theory does not sufficiently address the distinction between an atomic and a fragmented store.

One observation from the categorization of the server state management libraries was that the theory might lack sufficient categories to differentiate between certain libraries. Relay and Apollo Client, as well as Tanstack Query and SWR, were identically categorized. This similarity might indicate a need for additional categories within our theory to better distinguish between such libraries. However, it is important to remember that our theory strives for broad generalizations, and does not negate the individual strengths and nuances of each library. Rather, it attempts to provide a framework to understand common patterns and strategies across the various state management libraries.

In conclusion, applying the theories as a framework for categorizing different state management libraries proved to be an insightful scenario, where our theories were effectively employed. This not only enabled us to summarize the libraries in two concise tables, as shown in table 8.1 and table 8.2, but also to highlight areas of our theories that could benefit from further refinement. By providing an enhanced understanding of the different libraries, we have addressed **RQ2**. Simultaneously, the exercise helped identify potential weak-spots within our theories, paving the way for further refinement and improvement.

8.2 State Management in a Project Management Tool

This section models state management in a typical real-world web application: a project management tool. This demonstrates how a generalized theory of state management can be employed in order to model and reason about state management in real-world applications. Our example is a project management tool that incorporates task management similar to a Kanban board. A prototype of this tool, aptly named **CannyBan** is shown in figure 8.1. In the menu to the right, a list of boards are shown, including the selected board named *WebApp.* In the main content of the screen, a board for managing issues are shown. The board includes the stages: *backlog*, *in progress* and *done.* Each stage comprises a list of issues, including a title, sprint and relevant domains such as front-end or back-end. Core features of this tool is to *create*, *edit* and *delete* issues, and to *move* them between the different stages. At the bottom, a line chart visualizes the distribution of issues in the various stages. Furthermore, CannyBan offers functionalities such as the ability to select an application theme, collapse stages (visually minimize to save space), and filter issues based on selected domains.

Given the distinct division between client state management and server state management throughout this thesis, the state modelling of CannyBan is no exception. Hence, section 8.2.1 contains state modelling of CannyBan using *only* the theory of client state management, as defined in

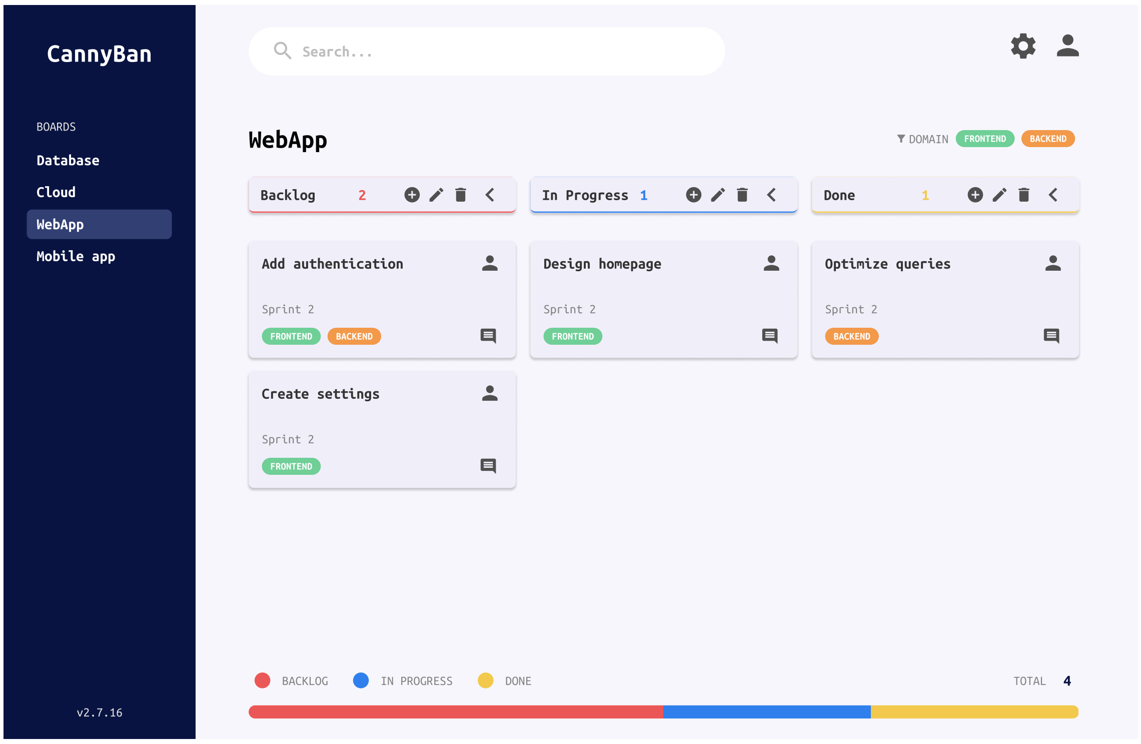


Figure 8.1: Prototype of a project management tool called *CannyBan*, which incorporates task management similar to a Kanban board. The tool demonstrates core features such as issue creation, editing, deletion, and movement between stages. Additionally, it offers functionalities like stage collapsing, and issue filtering based on domains.

chapter 6. Subsequently, in section 8.2.2 we expand our model to incorporate elements of server state management as defined in chapter 7. However, all state will not be moved to the server, making this a demonstration of a more complex scenario where state is maintained both on the client and the server. This illustrates the usefulness of utilizing a combination of our theories, and how these can adapt and apply to different state management setups within the same application.

8.2.1 Client State Only

We begin by employing the theory of client state management to model the state management of CannyBan. The categories defined in the theory are examined one by one in the context of CannyBan. Before we delve into each category, it is worth noting that the choice of *state composition model* will have a significant impact on following categories, such as *state selection*, *derivation* and *modification*. Our thesis does not cover the question of when to use which model, and while we could try to initiate a discussion on the matter, it is a complex question that would rather warrant its own research and thesis to be answered. Consequently, while we utilize a *single tree* composition model for this exercise, this is not an attempt at marking this as superior to the other models for this application.

Another note is that while the focus of this exercise is on client state, and thus does not include any server state, data can still be persisted to various storages. The ability to retrieve and populate the client state with data from a persisted store is something that is often offered by the client state management libraries, or you could alternatively implement it yourself. Examples of such persisted stores include WebStorage [63], local databases or even remote servers. However, the storage location of the state is irrelevant for this modelling exercise, which primarily focuses on articulating state management within the client. Consequently, the client state modeling approach for CannyBan remains consistent regardless of whether or where the client state is persisted.

State Types

Figure 8.2 illustrates the states in the CannyBan application. The figure distinguishes between the different state types *qualitative*, *quantitative*, *business* and *presentation*. In this example, *qualitative* state always appears in pair with *presentation* state, and *quantitative* state pairs with *business* state. Nonetheless, it should be noted that this pattern is not given in all cases.

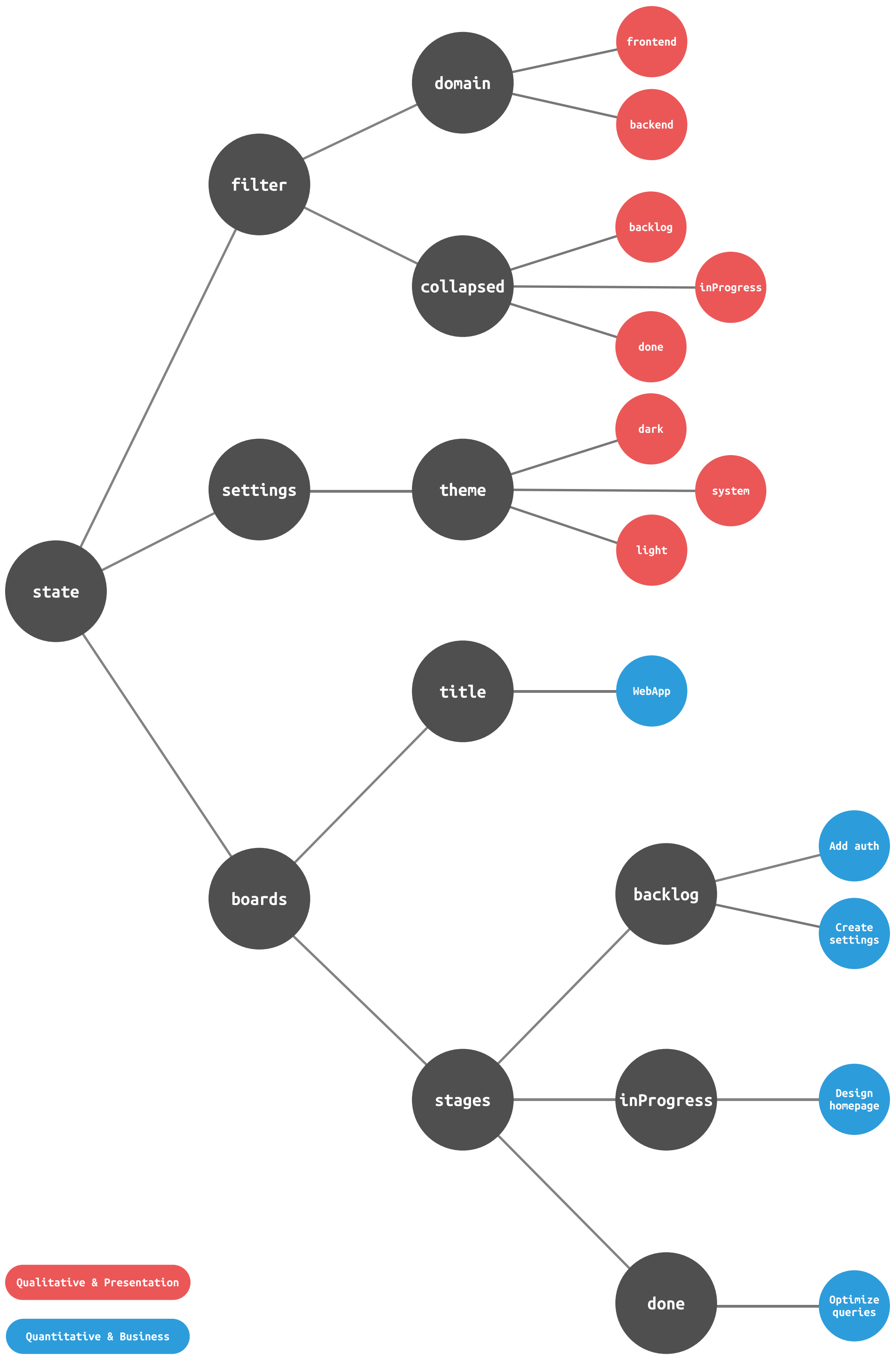


Figure 8.2: Overview of the state in CannyBan, composed using the *single tree* composition model. The *filter* and *settings* nodes contain only *qualitative* & *presentation* state, represented by the red nodes. The *board* node consist of only *quantitative* & *business* state, recognized as blue nodes.

The list of issues serves as an example of *business* state because it is an essential part of the data driving the application. This is relevant for all versions of the application, be it mobile or web. On the other hand, an example of *presentation* state is whether any of the stages are *collapsed*. When viewing CannyBan on a desktop there might be enough space to display all stages, but when using CannyBan on a mobile, it might be useful to collapse some columns due to a reduced screen size. The *collapsed* state, although not part of the driving data in the application, enhances the view layer in a specific implementation of CannyBan, making it an example of a *presentation* state.

The *title* of an issue is preferably a non-empty, arbitrary string. Due to the *title* being infinite in the sense that it can be any combination of characters, *title* is an example of a *quantitative* state. The *theme* state do not have an infinite number of values, as CannyBan only supports a limited number of themes. For this reason, the *theme* state is categorized as a *qualitative* state since it supports a finite number of options, including *dark*, *light*, and *system*.

State Reach

CannyBan will most likely consist of numerous components organized in a hierarchical structure within a component tree. State within the application might need to be shared with components far away from each other in the component tree. This makes it advantageous to utilize state with a *global* reach, facilitating the sharing of state between multiple components. For example, the state containing the list of issues in a particular stage is required in at least three components: the component responsible for displaying the actual issues, the component displaying the stage header including the total number of issues within the stage, and the line chart showing distribution of issues in stages. Moreover, expanding CannyBan with new features might also increase the distance between the components sharing the same state. This further emphasizes the benefits of employing a *global* state. However, the distinction between whether the state should be *global inside React* or *global outside React* is not significant for this modelling exercise.

Employing *global* state does not imply that there there is no need for *local* state. There are scenarios where *local* state is useful in CannyBan, such as a state for storing whether the settings dialog is open or not. However, managing *local* state is *trivial* compared to managing *global* state with its complexities and possibilities. For this reason, a discussion of *local* state in CannyBan is omitted.

State Composition Model

In CannyBan, the state is modelled using the *single tree* composition model. This involves organizing the state in a tree structure with a root node on top of the tree, where every child node is accessible from the root node by traversing its edges. As shown in figure 8.2, *state* is the root node, and its direct children are *filter*, *settings*, and *boards*. By continuing a tree traversal, subsequent children nodes can be accessed. Figure 8.3 shows a less detailed version of the state tree in figure 8.2, rotated to highlight its resemblance to a tree structure.

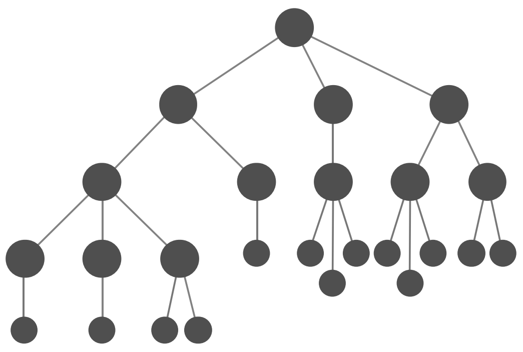


Figure 8.3: Illustrating the state in CannyBan resembling a tree structure.

State Selection

State selection enable components to listen to only the state that they require. Due to the CannyBan application employing a *single tree* composition model, this is achieved using *selector functions*. Figure 8.4 illustrates how a *selector function* can be utilized to select all the issues currently present in the *backlog* stage. Such a *selector function* traverses the path from the *state* node until it reaches the list of issues in the *backlog*, and outputs the issues in the backlog stage. Additional examples of state selection that could utilize *selector functions* include selection of the board's *title*, the available *stages*, and the current *theme*.

In addition to employing regular state, CannyBan also benefits from utilizing derived state, which can be achieved through the use of *derived selector functions*. Figure 8.1 shows that each stage header includes a numerical representation indicating the number of issues within that particular stage. This number can be derived from the state that contains the list of issues specific to each respective stage. Assuming we have a *selector function* that selects the list of issues in a stage, this can be used as input to a *derived selector function* that produce an output representing the number of issues in each state. Figure 8.4 illustrates this, where the *number of issues* state is derived from the list of issues in the *backlog*. Furthermore, derived state could be employed in the line chart located at the bottom of figure 8.1, representing the distribution of issues between the stages. The line chart could use corresponding derived states for each particular stage to create a visual representation of the distribution.

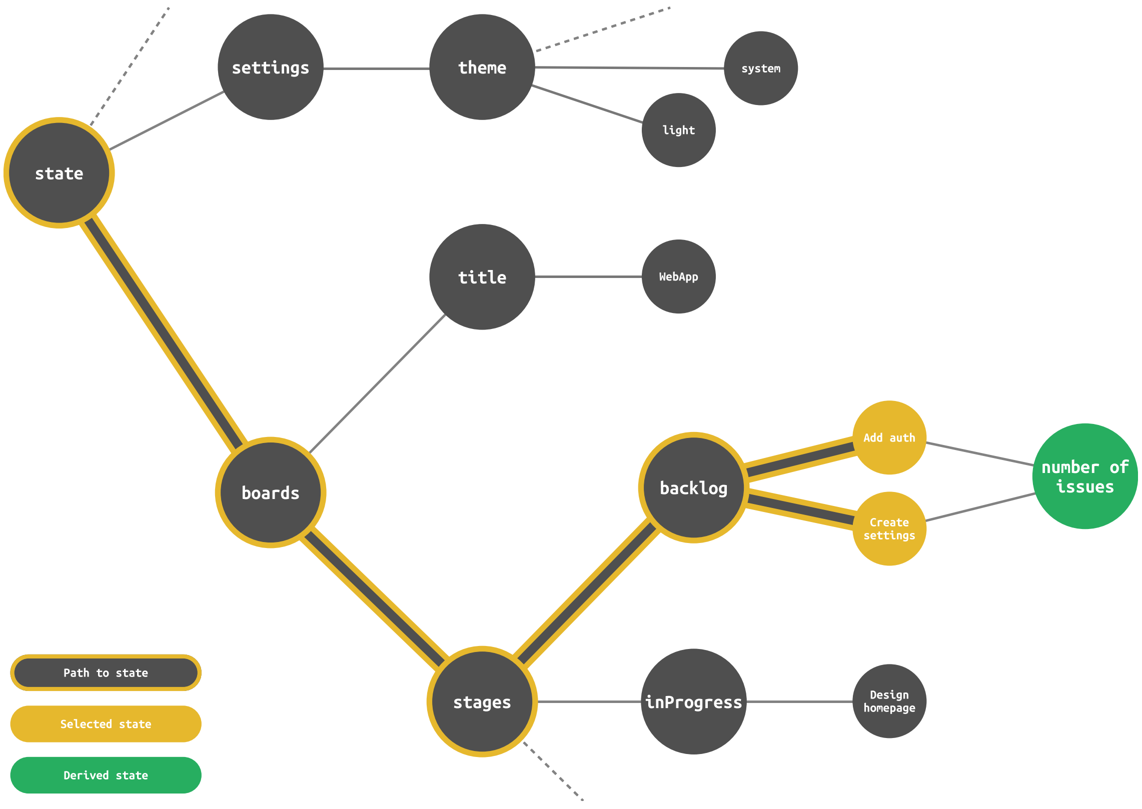


Figure 8.4: A derived selector function is utilized to obtain the number of issues currently in the backlog stage. A selector function traverses from the root node *state* to the list of issues in the backlog. The selector function is inputted to the derived selector function, producing the derived *number of issues*.

State Modification

As discussed in section 8.2.1, CannyBan employs state with a *global* reach. Because of this, an appropriate approach for modifying state is to use *global modification logic*. Consequently, the *modification logic* for modifying the *global* state is co-located with the *global* state itself. The components are not responsible for *how* the *global* state is modified, but they are empowered to *initiate* certain modifications on the *global* state.

Figure 8.5 illustrates the data flow of a *global modification logic* which occurs when an issue is *moved* from one stage to another in CannyBan. If the issue with the title "Add authentication" is moved from the *backlog* stage to the *in progress* stage, the respective component initiates a modification on the *global* state for moving the issue across those particular stages. It should be emphasized that the component itself does not know *how* the *global* state is modified, it only *initiates* a modification. Additionally, the *global modification logic* will be performed on the *global* state. This modification may result in a change of the state, and if this is the case, all components that has selected that particular state will be notified.

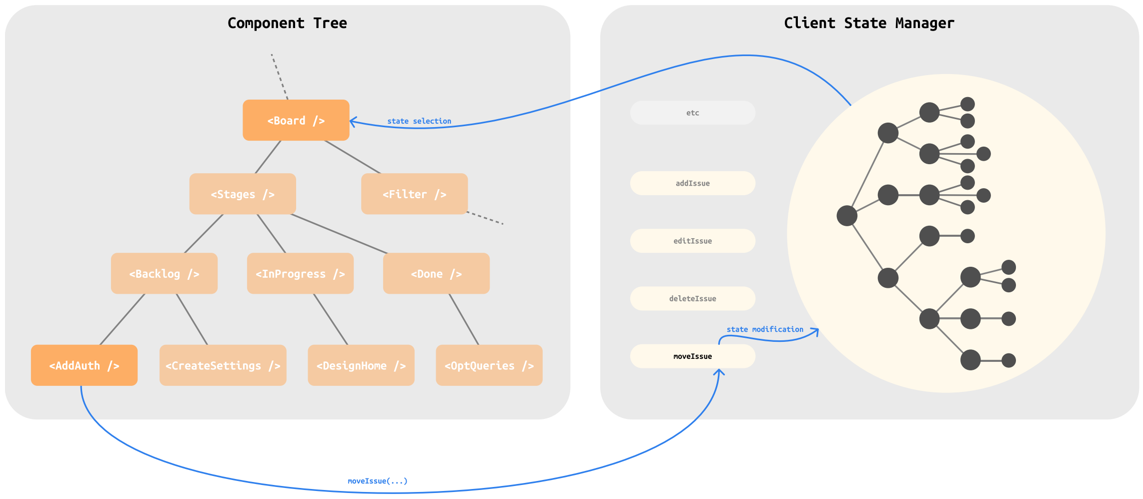


Figure 8.5: The figure illustrates a scenario where the issue titled "Add authentication" is moved from the *backlog* stage into the *in progress* stage. The respective component initiates a *moveIssue* modification on the *global* state. The modification is performed on the *global* state and the client state manager notifies all components that have selected the affected state.

A quick note should be made on some of the simplifications of figure 8.5. In the component tree, all stages use the JSX syntax for React components, such as `<Backlog />`, `<InProgress />` and `<Done />`. In an actual implementation, these would not be static components, but rather dynamically rendered with a generic `<Stage />` component. For example, the number of stages and their titles may change, making static components like this a fragile approach. The reason for this simplification in the figure is to save space, make the figure more readable, and enable the reader to easily differentiate between the various components. A similar simplification is applied to the issues components, such as `<AddAuth />` and `<CreateSettings />`.

Summary

The modelling of the client state management in CannyBan is explored in this section, facilitated by the principles of the generalized theory of client state management, including state *types*, *reach*, *composition model*, *selection*, *derivation* and *modification*. CannyBan adopts a *single tree* composition model, which impacts subsequent state management aspects due to the way the state is composed. In our modelling exercise of CannyBan, we demonstrated the advantages of incorporating both *qualitative* & *presentation* state, such as application *theme*, as well as *quantitative* & *presentation* state such as the list of *issues*. The use of *selector functions* and *derived selector* *functions* is presented as approaches for selecting a subset of the *global* state and utilizing derived state, such as displaying the number of issues in the stage headers. *Global modification logic* enable modifications to the *global* state, with components initiating these modifications, without directly controlling them.

After exploring the client-only state model of CannyBan, we now turn our attention to an expanded scenario that integrates *server state*, bringing an additional layer of complexity and capability to our model.

8.2.2 Client and Server State

It is rather limited what we can do in an application such as CannyBan without introducing server state. For example, in its current form, it would be difficult to use CannyBan cooperatively by more than one user at a time, because different users would be using different, local copies of the data. In this section, CannyBan is upgraded to **CannyBan 2**, incorporating server state. Thus, the theory of server state management will be employed, and each category of the theory is explored in the context of CannyBan 2.

However, it is important to note that CannyBan 2 will still preserve some of its client state. This is mainly its presentation state, as seen in figure 8.2. Figure 8.6 shows the updated state tree, containing only the client state of CannyBan 2. All business state will be moved to the server, making sure all users work with the same driving data.

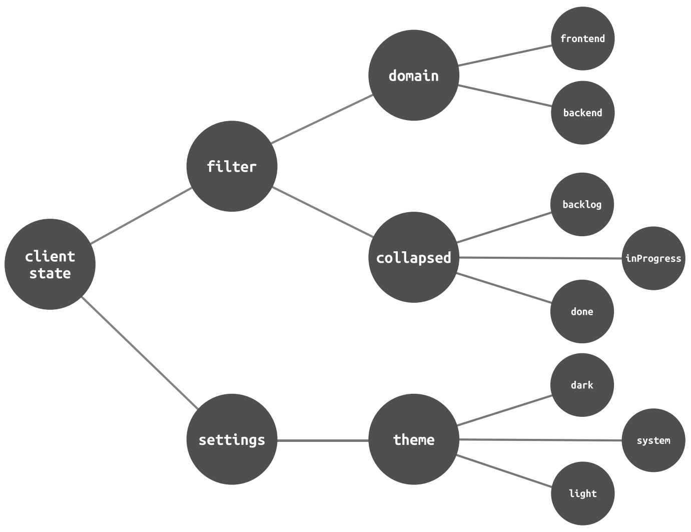


Figure 8.6: Overview of the client state in CannyBan 2. This encompasses the application theme, filtered domains and collapsed stages.

Operations

The *operations* in use in CannyBan 2 are *queries* and *mutations*. *Queries* are essential to the application, as they enable it to read data from the server. Examples of queries in CannyBan 2 could be `getBoards`, returning a list of all the boards the user has access to, and `getBoard(WebApp)`, returning the data for the *WebApp*-board. Modification of state, for example moving an issue from one stage to the next, is enabled by *mutations*. Additional examples of mutations might be to create, edit or delete issues. CannyBan 2 does not incorporate *subscriptions* at the moment. *Subscriptions* enable you to listen to real-time data, which is not yet essential for the functionality of CannyBan. In a potential future version, *subscriptions* can facilitate the introduction of features like a live feed or displaying real-time activity. For this exercise though, CannyBan 2 will do with just *queries* and *mutations*.

Cache

CannyBan 2 is rather limited in scope and complexity, and for this example we model it using *query cache*, rather than a *normalized cache*. This means that each query will be stored with all its data in the cache, under its own *cache entry*. An example of this is shown in figure 8.7, where the *cache entries* for the two queries `getBoards` and `getBoard(WebApp)` are illustrated. In this example, `getBoards` and `getBoard(WebApp)` are the *query keys* of the queries, and they are used as *cache keys* to identify their respective *cache entries*. The response data of a query is stored as cache data, inside its corresponding *cache entry*.

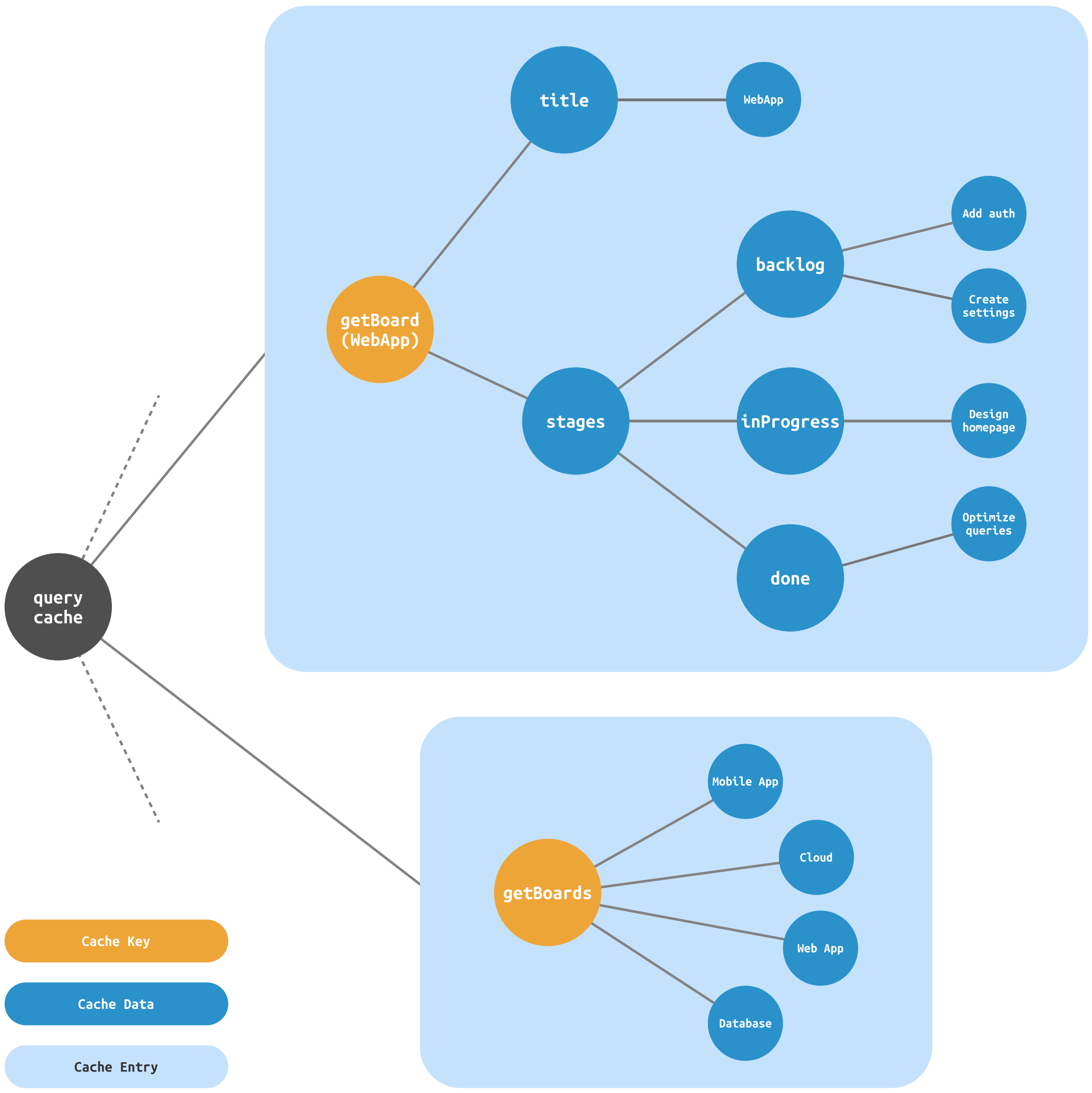


Figure 8.7: Demonstrates the use of a *query cache* in CannyBan 2. The figure showcases the *cache entries* for the `getBoards` and `getBoard(WebApp)` queries, alongside its corresponding *cache data*.

When CannyBan 2 performs a mutation, for example moving an issue from a stage to another, the data inside the cache entry of `getBoard(WebApp)` will be stale. To update the stale data, mutations in CannyBan 2 will mark affected queries as invalidated. Thus, the mutation that moves an issue, will trigger a re-fetch of the `getBoard(WebApp)`-query.

It is important to note that the *queries* and resulting *cache entries* of CannyBan 2 does not *need* to resemble figure 8.7. For example, the `getBoard(WebApp)` contains a big chunk of the state in this model, and it might be reasonable to split it into multiple smaller queries. One query could fetch the associated stages of a board, while another fetch the associated issues. This could reduce the probability that a mutation invalidates a specific query, as the more data each query contains, the higher the chances are that some of the data will be invalidated. Even though you utilize a *query cache*, you are still free to decide whether you want big queries, containing lots of data, or multiple smaller, more specific queries.

Query Organization

We use a *decentralized* approach to the **query organization** of CannyBan 2. This entails that the different queries are defined separately, and not in a *centralized* location. Figure 8.8 shows how this might look in a component tree. The `useGetBoard(WebApp)` and `useGetBoards()` query hooks are defined right next to the components that uses them. Because there are no constraints on where to define the queries in a *decentralized* approach, we could also have chosen to define all the query hooks in a centralized location if we wanted.

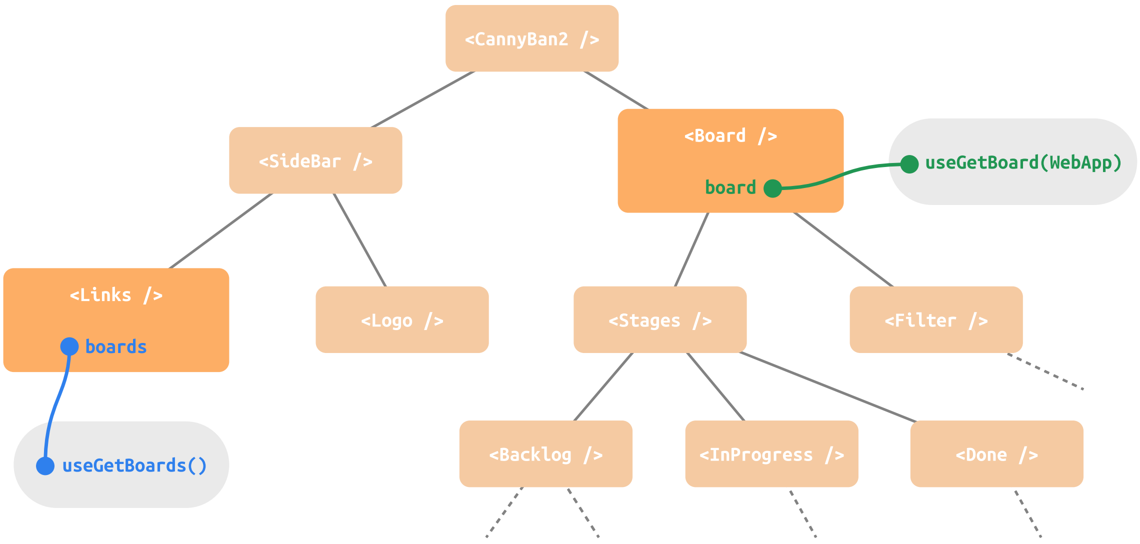


Figure 8.8: Illustration of the *decentralized* query organization approach in CannyBan 2. The figure showcases the location of query hooks, such as `useGetBoard(WebApp)` and `useGetBoards()`, adjacent to the components that utilize them within the component tree.

Asynchronous Metadata

The asynchronous metadata of the queries in CannyBan 2 is used to show appropriate feedback to the user when data is loading. For this purpose, it is primarily *data status* that is utilized. An example of how the sidebar component of CannyBan 2 utilizes *data status* to determine what to show to the user is shown in figure 8.9. When loading, a so-called "skeleton" of the data is shown, indicating that something will appear here when it is done loading. If the loading completes successfully, the sidebar is shown with the fetched links to corresponding boards. However, it might happen that an error occurs while loading, and in that case, CannyBan 2 shows an appropriate message to notify the user.

Another appropriate use case for *data status* might be in the `<Board />` component, which uses the `getBoard(WebApp)` query. During the *loading* state, a skeleton could be shown, if an error occurs, an error message should be presented, and in the case of a successful query, the board is displayed. Any other components using server state could also benefit from similar utilization of *data status*. Currently, CannyBan 2 does not have a need for indicating that a background re-fetch is happening, but if it did, the *fetch status* could be employed to achieve this.

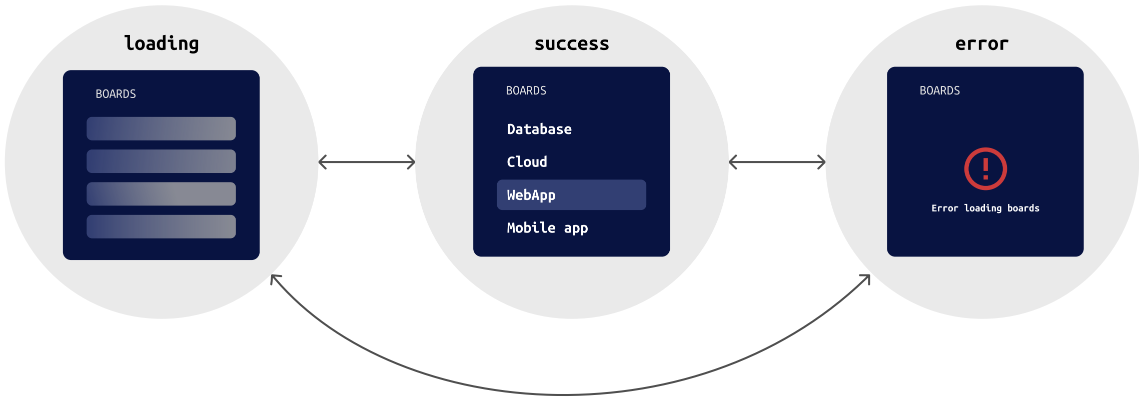


Figure 8.9: Demonstrates how CannyBan 2 utilizes *data status* in its sidebar component to provide appropriate user feedback during the asynchronous operation of fetching boards. During the *loading* state, a loading skeleton is shown, in *success* state, the boards are displayed, and in the occurrence of an error, an error symbol alongside an appropriate feedback message is presented.

Client State Integration

As mentioned when introducing CannyBan 2, the application still has some client state along with its server state. Because CannyBan 2 is a direct upgrade of the original CannyBan, which only used client state, we imagine that it can just continue using the original client state manager to manage its client state. Consequently, we do *not* have to transition the client state into an *integrated* client state solutions of one of the server state libraries.

The consequences of not using an *integrated* client state approach, is that whenever a component needs a combination of server and client state, it has to take care of combining them itself. Where an *integrated* solution might allow you to create derived state combining client state and server state, the approach CannyBan 2 adopts is illustrated in figure 8.10. The `<Board />` component needs to combine the server state from `useGetBoard(WebApp)` and the client state from `selectCollapsedStages()` to decide which of its stages should or should not be collapsed. To achieve this, the component gets both of these states, before using their values to derive stage data, including information on whether or not it should be collapsed. This can then be passed to the appropriate `<Stage />`-component.

This example might be a bit convoluted, and in practice the `selectCollapsedStages()` would probably just be used in the `<Stage />` component itself. However, it was included to make a point of how the server and client state would be combined in a scenario that does not utilize an *integrated* approach to client state.

Summary

CannyBan has been upgraded to version 2, and this section explored the server state modelling of the application, using the generalized theory of server state. CannyBan 2 uses *queries* to read data from a server, and *mutations* ta make modifications to its data. At its current version it was decided that it did not need any real-time data through the use of *subscriptions*. CannyBan 2 uses a *query cache*, and all data from a query is stored inside its own *cache entry*. The *query keys* are used as *cache keys* for the *cache entries*. When a mutation causes the data inside a cache entry to become stale, the mutation marks the affected queries as invalidated, triggering a re-fetch of the relevant query.

The organization of queries is done in a *decentralized* approach, allowing queries to be defined separately, for example along with the components that uses them. The application uses asynchronous metadata to provide user feedback during data loading, utilizing *data status*. For example, in the sidebar, a “skeleton” of the data is displayed while loading, a notification of an error in the case

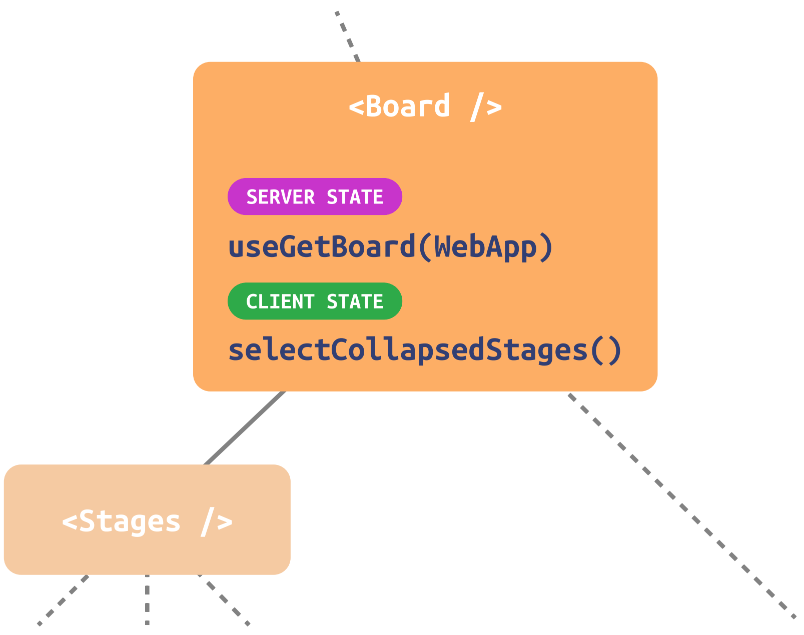


Figure 8.10: Illustrates how CannyBan 2 combines server and client state in the `<Board />` component. By utilizing both the server state obtained from `useGetBoard(WebApp)` and the client state retrieved from `selectCollapsedStages()`, the component determines the collapsed status of each stage.

of a loading failure, and the links to the boards are displayed when it has successfully received its data. Despite the inclusion of server state, CannyBan 2 continues to utilize some client state. This is not transitioned into an *integrated* solution, but is still stored in the client state manager of the original CannyBan. Consequently, components that need to combine server state and client state must do the combination themselves inside the component.

8.2.3 Evaluation

The aim of this chapter is to explore the practical applicability of our theories of state management in real-world scenarios, thereby addressing **RQ2**. The theories have in this section been used to discuss the modelling of CannyBan, an example of a project management tool. The first modelling exercise, involving only client state, demonstrated a clear and practical utilization of our theory. CannyBan's state was modelled using the key concepts of *state types*, *reach*, *composition model*, *selection*, *derivation*, and *modification*. The modeling exercise facilitated an understanding of the application state, without specifying a specific library to use for the implementation. It thus highlights how the theory can be utilized to reason about client state management, independent of a specific state management library.

In the second modelling exercise, CannyBan was extended to utilize server state, demonstrating how our theories can adapt and be applied to more complex scenarios, including both client and server state. Key concepts from our theory was discussed, like *operations*, *cache*, *query organization*, *asynchronous metadata*, and *client state integration*. The theory simplified the process of gaining a mental model of the server state management, and could facilitate discussions of topics like how large the queries should be, or the consequences of dividing client and server state.

A significant limitation of our theories, however, is that they do not currently offer any guidance with regards to the ideal modelling of an application. While our theory can be used to claim that CannyBan 2 uses a *single tree* composition model and a *query cache*, it cannot be used to claim that this was the optimal choice for this specific application. Thus, our theories can be said to function more as a tool for *describing* rather than *designing*. That being said, possessing a descriptive language is an essential component of designing, providing the vocabulary to articulate and evaluate. Hence, despite the limitation, our theories still hold considerable value for modelling and designing applications.

Overall, these modelling exercises have demonstrated the utility of our theories in enhancing understanding and facilitating informed discussions about state management in real-world applications. The versatility of our theories has been demonstrated as they addressed different scenarios: first, a client state-only application, and then an application leveraging both client and server state. Despite not providing explicit guidance on the *ideal* state model for CannyBan, our theories facilitated discussions and informed reasoning, which in turn can lead to more optimal decision-making. Thus, application modelling can be a fitting use-case for our theories, facilitating enhanced understanding by providing a descriptive language that promotes discussion and reasoning, potentially resulting in more optimal design choices.

8.3 Understanding Server State Management through Client State Management

Server state management is about managing *server state*. While server state *originates* from the server, it is still stored and managed *on the client*. Therefore, all *server state* is also *client state*. This may be confusing, but remember that the scope of this thesis is the *client's* side, and that *server state* refers to the state on the client that originates from, and should be kept up to date with, the *server*. Thus, *server state* is state *on the client* that *originates* from a server.

We have spent chapter 7 describing what server state management entails. But because server state *is* client state, as explained above, managing our server state *is* a sort of client state management. This means that we can use our theory of client state management, postulated in chapter 6, to reason about server state management as well. For example, let us take the topic of **State Reach**. Server state is stored in the cache, and the cache should be reachable from the whole application, as all components that use the same query should reuse the same *cache entry*. Thus, server state has a *global* reach. Whether it is *inside* or *outside* React depends on the specific library implementation.

Having established that server state has a *global* reach, we turn our attention to the **State Composition Model**. When using *query caching*, we can say that an *atomic* model is used, where the cache stores the entire result of a query as a single, indivisible unit. However, when considering that each of these queries may contain a lot of nested data, it could be considered a *fragmented store*, where each query represents one fragmented store. On the other hand, *normalized* caching involves breaking the data from the server down into smaller entities before storing it in the cache, which resembles an *atomic* composition model. Thus, we have three different ways of managing our server state: *atomic query cache*, *query cache as fragmented stores* and *atomic normalized cache*. Each of these three models of server state management will dictate how it is analyzed further, using the categories of **State Selection**, **Derived State** and **State Modification**.

8.3.1 Atomic Query Cache

The *atomic query cache* model applies when using *query cache*, and each query is considered as an indivisible unit, thus employing the *atomic* **State Composition Model**. In many cases it might be a stretch to think of queries as atoms, as they may contain a lot of nested data. However, embracing this perspective can simplify how we manage the server state, such as **State Selection**. Recall from section 6.5 that **State Selection** is about selecting only a subset of the state in a store. However, when utilizing an *atomic* model, there is no need to perform such selections, because the atoms themselves already are the smallest piece of state. Consequently, employing an *atomic query cache* model, **State Selection** simply entails using the queries directly.

It is possible to implement **Derived State** in the form of *derived atoms*, however, it is worth noting that none of the libraries utilizing *query cache* included in this thesis implemented any advanced form of derived data. A transformation of the data from one cache entry was employed by Tanstack

Query and RTK Query, but none offered the functionality of merging different cache entries into more complex derived state. This could be attributed to the fact that it is not a common use-case for server state, and could be implemented manually, albeit without render optimization.

Conceptualizing queries as atoms also simplifies the topic of **State Modification**. Since queries are considered indivisible units, there is no need to partially modify them. Whenever we perform a re-fetch, or manually modify state in the cache, the entire *cache entry* can just be replaced. This makes it natural to use an *immutable* approach, where changes are detected by a new object reference, instead of comparing individual values.

The *atomic query cache* is a simple, but efficient, approach to server state management. Tanstack Query, RTK Query, SWR and URQL (with document cache) utilize *query cache*, and can be used with the *atomic query cache* model. Unlike SWR and URQL, Tanstack and RTK Query allow you to derive data from a query. This can also be used to implement more advanced *selector functions*, but for that to be relevant we have to move away from thinking of the queries as atoms, and rather as their own *fragmented stores.*

8.3.2 Query Cache as Fragmented Stores

As noted, in many cases it might be a stretch to think of the result of one query as one atom. Query results may contain much data, and sometimes it would be preferable to think of the data from a query as a store, rather than an indivisible unit. Thus, the *fragmented store* **State Composition Model** can be utilized.

Using the model of *query cache as fragmented stores*, we can implement *selector functions* and **Derived State** as *derived selector functions*, that retrieve and transforms lesser parts of the state from *cache entries.* It is important, however, to remember that a key aspect of **State Selection** is render optimization. To justify using a *selector function* instead of just selecting the whole query, there should be situations where only certain parts of the query are updated, and you want to re-render only if the part you have selected is updated. This can be the case if the cache selectively updates only the modified parts of a *cache entry* during a re-fetch, making the optimizations of *selector functions* effective.

It should be noted, here as well as with the *atomic query cache* model, that no *query cache* libraries included in this thesis implement any advanced state derivation. Tanstack Query and RTK Query allow you to implement the equivalent of *derived selector functions* for individual queries, but do not offer the functionality of merging different queries into more complex derivations.

Thinking of queries as *fragmented stores* rather than indivisible atomic units enables more fine-grained **State Modification**. Consequently, it becomes relevant to consider both *mutable* and *immutable* approaches to modification. A possible *mutable* implementation for the cache is to use a tracking method similar to Valtio or MobX to monitor accessed fields, while another option is to leverage *immutable* state modification and track references. All of the server state management libraries included in this thesis use an *immutable* approach to **State Modification**.

Using the *query cache as fragmented stores* model is a bit more complicated than *atomic query cache*, but also offers more flexibility. You can choose to implement *selector functions* and *derived selector functions*, or choose to utilize a *mutable* state modification approach, and use an access-based selection like Valtio or MobX. Tanstack Query and RTK Query can be employed to utilize this model.

8.3.3 Atomic Normalized Cache

The *atomic normalized cache* model applies when utilizing a *normalized* cache. As mentioned, *normalizing* the cache entails breaking the data from the server down into entities before storing them. These entities can be conceptualized as atoms, consequently resembling the *atomic* **State Composition Model**. In contrast to the *atomic query cache* model, it is not the queries themselves that are considered atoms, but the entities that contrive them.

Queries are stored with references to the atomic entities. In other words, they are compositions of different atoms, making them a form of **State Derivation**. It is also possible to implement other *derived atoms* that selects atomic entities and performs derivations on these. For example, Apollo Client allows you to create the equivalent of *derived atoms* by creating custom local-only fields. These can read data from other entities in the cache, and perform derivations before returning the result. Because we utilize the *atomic* composition model, **State Selection** just entails selecting the right atom.

The entities, conceptualized as atoms, can actually contain multiple data fields, and the queries that select atoms also has to select which of these fields to read. This means that strictly speaking entities are not *indivisible* units, and you could argue that it is these fields that should be called atoms. However, just like it sometimes is reasonable to store an object as an atom in libraries like Jotai and Recoil, it is reasonable to conceptualize the entity as an atom, as it is the smallest *identifiable* unit of a normalized cache. This separates the *atomic normalized cache* model from the *atomic query cache* model, where we treated the atoms as indivisible units.

The possibility of individual fields inside the atoms makes **State Modification** more nuanced than the straightforward approach of replacing the entire *cache entry* of the *atomic query cache* model. It must be possible to modify individual fields of an atom. Just like the *query cache as fragmented stores* model, we are free to choose an *mutable* or *immutable* **State Modification** approach. It is feasible to imagine a *mutable* approach making sense in this case, as the queries that select which fields to read can work as a dependency array for the fields. When a field is modified, it can notify the queries that listen to it, which can trigger a re-render. However, both Apollo Client and URQL opts for the *immutable* approach.

8.3.4 Evaluation

In this section, the theory of client state management from chapter 6 have been used to analyze different approaches to server state management as described in chapter 7. By combining the concept of **State Composition Model** from client state management and *cache* type from server state management, three different models of server state management was identified: *atomic query cache*, *query cache as fragmented stores*, and *atomic normalized cache*. Each of these server state management models were analyzed further using the client state management concepts of **State Selection**, **Derived State**, and **State Modification**.

The analysis allowed us to get a comprehensive understanding of the different models of server state management, including their strengths and limitations. For example, while the *atomic query cache* model is simple and efficient, *query cache as fragmented stores* offers greater flexibility in terms of state selection and modification. We were also able to identify areas where some server state management libraries lacked in features. For example, SWR employs a model resembling that of *atomic query cache*, but they do not offer any way of deriving state from queries, which is an essential feature of an *atomic* **State Composition Model**. Thus, our theory enabled us to identify a potential area of improvement in a respected server state management library.

It is important to acknowledge that while state derivation is a part of our theory of client state management, it does not necessitate that it should be included in server state management libraries. Client state management and server state management aims to solve different problems, and even though server state management must include some client state management, it does not necessarily need all the advanced features of libraries dedicated to client state management. Such features should be justified by satisfactory use-cases. In the case of SWR, it might very well be that the team behind the library has discussed this feature, and concluded with them not seeing enough significant use-cases to incorporate it into their library. While this may be a good decision that may be right for SWR, it should not undermine the significance of using our theory to enable a valuable discussion on the matter.

Chapter 9

Conclusion and Further Work

This thesis embarked on an exploration of the field of state management in JavaScript front-end frameworks, with a specific focus on React. Although state management is a source of difficulties for developers, it has largely been overlooked in scientific research. There exist a large number of JavaScript third-party libraries attempting to address this problem, yet a formalized theory of state management remains missing. Guided by the hypothesis that an implicit general theory of state management exists within these third-party libraries, this thesis sought to extract and formulate this theory. Thus, the following research questions were posed:

- **RQ1**: What are the main concepts and methodologies in the field of state management, and how can they be formed into a generalized theory?
  - **RQ1.1**: What are the main concepts and methods in client state management
  - **RQ1.2**: What are the main concepts and methods in server state management
- **RQ2**: What are the potential scenarios where a generalized theory of state management can be employed, and how might these enhance understanding and practice in this field?

9.1 Conclusion

To answer our research questions, we opted to do a qualitative literature analysis of a set of server and client state management libraries. From these analyses, we extracted and generalized key concepts and methodologies, resulting in two theories: a theory of client state management and a theory of server state management. The formulation of these theories, including the proposed terms and categories to differentiate and classify state management patterns, directly answers **RQ1**.

After defining these theories, we applied them to different scenarios, each constructed to demonstrate practical use-cases. There were three scenarios: categorizing state management libraries using the theories, modelling state management in a sample application, and using the theory of client state to gain an enhanced understanding of server state. These applications not only showcased the practicality of the theories, but also facilitated more informed discussion on the various models and libraries, thus effectively answering **RQ2**.

The aforementioned applications of our theories also served to highlight potential areas for improvement in our theories, thus challenging the completeness of our answer to **RQ1**. However, given the demonstrated usefulness of our theories across the different scenarios we conclude that both **RQ1** and **RQ2** are sufficiently answered, while the theories still have room for further refinement and improvement.

9.2 Further Work

There are multiple directions of future work that could build upon the work of this thesis. As mentioned, potential areas of improvement of the theories themselves have been identified through applying them in practical scenarios. Specifically, work could be done to further remove the ambiguity from the concept of **State Composition Model**. As addressed in section 8.1, this category can be too vague to effectively classify libraries. A more thorough examination of this part of the theory could result in a more precise category, enhancing the usefulness of the theory.

Another direction for future research could be to expand upon the practicality of our theories by examining how the different categories can assist in decision-making during application design. In section 8.2 we discussed and modelled state management in CannyBan, a project management tool. However, while our theories offered insightful discussion and enhanced understanding of the state management in CannyBan, it did not offer direct guidance with regards to the design decisions made. Therefore, a potential area of research is to explore and define the circumstances in which certain composition models is to prefer over others, such as when to use a *single tree* model instead of an *atomic* model. This type of research could potentially make the area of state management even more approachable for developers.

Given that our thesis primarily focuses on the JavaScript front-end framework of React, future work could potentially examine if and how our theories apply to other front-end frameworks, like Vue, Angular and Svelte. Such explorations might work to further enhance the applicability and universality of our theories. It is also worth investigating how newer advances and trends in front-end development, like server-side rendering and React Server Components, might have an impact on our theories.

While our research was motivated by an overarching frustration among web developers with the field of state management, this thesis has not made any effort to investigate whether or not this work actually eases this frustration. Do our theories of state management make the field of state management more understandable for developers? This could be an exciting question to pursue, potentially generating more specific feedback on how the theories could be improved.

A last compelling direction for further exploration we would like to propose, is to introduce this theoretical framework into education and academia. Throughout the course of this thesis, our understanding of state management in React has been substantially enriched, and we believe many developers could similarly benefit from a wider understanding on the field of state management. In particular, newcomers to the field of front-end web development, like students and self-learners, could find such a structured presentation of state management very useful. This knowledge could potentially reduce some of the initial frustrations associated with understanding state management.

Library Documentations

[1] Meta Platforms, Inc. 'React - A JavaScript library for building user interfaces'. reactjs.org. (n.d.), [Online]. Available: https://web.archive.org/web/20221114083752/https://reactjs.org/ (visited on 14th Nov. 2022).

[11] Meta Platforms, Inc. 'React Docs Beta'. beta.reactjs.org. (n.d.), [Online]. Available: https://web.archive.org/web/20221115152433/https://beta.reactjs.org/ (visited on 15th Nov. 2022).

[40] Meta Platforms, Inc. 'Recoil'. recoiljs.org. (n.d.), [Online]. Available: https://web.archive.org/web/20221121215125/https://recoiljs.org/ (visited on 21st Nov. 2022).

[41] Poimandres. 'Zustand Documentation'. docs.pmnd.rs. (n.d.), [Online]. Available: https://web.archive.org/web/20221202010759/https://docs.pmnd.rs/zustand/getting-started/introduction (visited on 2nd Dec. 2022).

[42] D. Abramov and the Redux documentation authors. 'Redux'. redux.js.org. (n.d.), [Online]. Available: https://web.archive.org/web/20221119041846/https://redux.js.org/ (visited on 19th Nov. 2022).

[43] M. Weststrate and the MobX team. 'MobX'. mobx.js.org. (n.d.), [Online]. Available: https://web.archive.org/web/20221121172353/https://mobx.js.org/README.html (visited on 21st Nov. 2022).

[44] Stately. 'XState'. xstate.js.org. (n.d.), [Online]. Available: https://web.archive.org/web/20221121211233/https://xstate.js.org/ (visited on 21st Nov. 2022).

[45] Poimandres. 'Jotai'. jotai.org. (n.d.), [Online]. Available: https://web.archive.org/web/20221121215301/https://jotai.org/ (visited on 21st Nov. 2022).

[46] Poimandres. 'Valtio'. valtio.pmnd.rs. (n.d.), [Online]. Available: https://web.archive.org/web/20221103232506/https://valtio.pmnd.rs/ (visited on 3rd Nov. 2022).

[47] Elf, Inc. 'Elf'. ngneat.github.io. (n.d.), [Online]. Available: https://web.archive.org/web/20221208123051/https://ngneat.github.io/elf/ (visited on 8th Dec. 2022).

[48] Apollo Graph Inc. 'Apollo Client'. apollographql.com. (n.d.), [Online]. Available: https://web.archive.org/web/20230327200133/https://www.apollographql.com/docs/react/ (visited on 27th Mar. 2023).

[49] Meta Platforms, Inc. 'Relay'. relay.dev. (n.d.), [Online]. Available: https://web.archive.org/web/20230406013852/https://relay.dev/ (visited on 6th Apr. 2023).

[50] Formidable Labs. 'URQL documentation'. formidable.com. (n.d.), [Online]. Available: https://formidable.com/open-source/urql/ (visited on 8th Mar. 2023).

[51] T. Linsley. 'Tanstack Query documentation'. (n.d.), [Online]. Available: https://tanstack.com/query/latest (visited on 1st Feb. 2023).

[52] D. Abramov and the Redux documentation authors. 'RTK Query Overview'. redux-toolkit.js.org. (n.d.), [Online]. Available: https://web.archive.org/web/20230213212832/https://redux-toolkit.js.org/rtk-query/overview (visited on 13th Feb. 2023).

[53] Vercel Inc. 'SWR'. swr.vercel.app. (n.d.), [Online]. Available: https://web.archive.org/web/20230213170753/https://swr.vercel.app/ (visited on 23rd Feb. 2023).

[59] D. Kato. 'How Valtio Proxy State Works (Vanilla Part)'. blog.axlight.com. (Aug. 2021), [Online]. Available: https://web.archive.org/web/20221103232754/https://blog.axlight.com/posts/how-valtio-proxy-state-works-vanilla-part/ (visited on 3rd Nov. 2022).

[60] D. Kato. ‘How Valtio Proxy State Works (React Part)’. blog.axlight.com. (Dec. 2021), [Online]. Available: https://web.archive.org/web/20221127014744/https://blog.axlight.com/posts/how-valtio-proxy-state-works-react-part/ (visited on 27th Nov. 2022).

[64] zerobias & Effector Core team. ‘Effector’. effector.dev. (n.d.), [Online]. Available: https://web.archive.org/web/20230522011336/https://effector.dev/ (visited on 22nd May 2023).

Bibliography

[2] CACM Staff, ‘React: Facebook’s functional turn on writing javascript’, *Commun. ACM*, vol. 59, no. 12, pp. 56–62, Dec. 2016, ISSN: 0001-0782. DOI: 10.1145/2980991. [Online]. Available: https://doi.org/10.1145/2980991.

[3] S. Greif. ‘The state of js 2021 - opinions’. 2021.stateofjs.com. (n.d.), [Online]. Available: https://2021.stateofjs.com/en-US/opinions/ (visited on 27th Nov. 2022).

[4] K. Sun and S. Ryu, ‘Analysis of javascript programs: Challenges and research trends’, *ACM Comput. Surv.*, vol. 50, no. 4, Aug. 2017, ISSN: 0360-0300. DOI: 10.1145/3106741. [Online]. Available: https://doi.org/10.1145/3106741.

[5] Y. Lin, M. Li, C. Yang and C. Yin, ‘A code quality metrics model for react-based web applications’, in *Intelligent Computing Methodologies*, D.-S. Huang, A. Hussain, K. Han and M. M. Gromiha, Eds., Cham: Springer International Publishing, 2017, pp. 215–226, ISBN: 978-3-319-63315-2.

[6] C. M. Novac, O. C. Novac, R. M. Sferle, M. I. Gordan, G. BUJDOSó and C. M. Dindelegan, ‘Comparative study of some applications made in the vue.js and react.js frameworks’, in *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*, IEEE, 2021, pp. 1–4. DOI: 10.1109/EMES52337.2021.9484149.

[7] W. Roby, X. Wu, T. Goldina *et al.*, ‘Firefly: embracing future web technologies’, in *Software and Cyberinfrastructure for Astronomy IV*, G. Chiozzi and J. C. Guzman, Eds., International Society for Optics and Photonics, vol. 9913, SPIE, 2016, 99130Y. DOI: 10.1117/12.2233042. [Online]. Available: https://doi.org/10.1117/12.2233042.

[8] F. Ferreira, H. S. Borges and M. T. Valente, ‘On the (un-)adoption of javascript front-end frameworks’, *Software: Practice and Experience*, vol. 52, no. 4, pp. 947–966, 2022. DOI: https://doi.org/10.1002/spe.3044. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3044. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3044.

[9] S. Greif. ‘The state of js 2021 - front-end frameworks’. 2021.stateofjs.com. (n.d.), [Online]. Available: https://2021.stateofjs.com/en-US/libraries/front-end-frameworks/ (visited on 27th Nov. 2022).

[10] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber and F. Shull, ‘Exploring language support for immutability’, in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16, Austin, Texas: Association for Computing Machinery, 2016, pp. 736–747, ISBN: 9781450339001. DOI: 10.1145/2884781.2884798. [Online]. Available: https://doi.org/10.1145/2884781.2884798.

[12] H. Søndergaard and P. Sestoft, ‘Referential transparency, definiteness and unfoldability’, *Acta Informatica*, vol. 27, no. 6, pp. 505–517, 1990. DOI: 10.1007/BF00277387. [Online]. Available: https://doi.org/10.1007/BF00277387.

[13] P. Hudak, ‘Conception, evolution, and application of functional programming languages’, *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, Sep. 1989, ISSN: 0360-0300. DOI: 10.1145/72551.72554. [Online]. Available: https://doi.org/10.1145/72551.72554.

[14] Mozilla Corporation. ‘Mutable - MDN Web Docs Glossary: Definitions of Web-related terms’. developer.mozilla.org. (Sep. 2022), [Online]. Available: https://web.archive.org/web/20221107104640/https://developer.mozilla.org/en-US/docs/Glossary/Mutable (visited on 7th Nov. 2022).

[15] K. Kumar and S. K. Azad, ‘Database normalization design pattern’, in *2017 4th IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics (UPCON)*, 2017, pp. 318–322. DOI: 10.1109/UPCON.2017.8251067.

[16] A. Gill, *Introduction to the Theory of Finite-State Machines.* NY, USA: McGraw-Hill Book Company, 1962.

[17] Postman, Inc. ‘2021 State of the API Report - API Technologies’. postman.com. (2021), [Online]. Available: https://www.postman.com/state-of-api/2021/api-technologies/ (visited on 24th Mar. 2023).

[18] R. T. Fielding, ‘Architectural styles and the design of network-based software architectures’, Ph.D. dissertation, University of California, Irvine, 2000.

[19] The GraphQL Foundation. ‘GraphQL Specification’. spec.graphql.org. (Oct. 2021), [Online]. Available: https://spec.graphql.org/October2021/ (visited on 21st Mar. 2023).

[20] S. Alimadadi, D. Zhong, M. Madsen and F. Tip, ‘Finding broken promises in asynchronous javascript programs’, *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. DOI: 10.1145/3276532. [Online]. Available: https://doi.org/10.1145/3276532.

[21] Ecma International. ‘ECMAScript® 2022 Language Specification - Promise Objects’. tc39.es. (Jun. 2022), [Online]. Available: https://tc39.es/ecma262/2022/#sec-promise-objects (visited on 24th Mar. 2023).

[22] Fielding, Roy and Nottingham, Mark and Reschke, Julian. ‘RFC 9110 HTTP Semantics - Caches’. rfc-editor.org. (Jun. 2022), [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110#name-caches (visited on 24th Mar. 2023).

[23] Mozilla Corporation. ‘HTTP caching’. developer.mozilla.org. (Mar. 2023), [Online]. Available: https://web.archive.org/web/20230327131310/https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching (visited on 27th Mar. 2023).

[24] H. V. Nguyen, L. Lo Iacono and H. Federrath, ‘Systematic analysis of web browser caches’, in *Proceedings of the 2nd International Conference on Web Studies*, ser. WS.2 2018, Paris, France: Association for Computing Machinery, 2018, pp. 64–71, ISBN: 9781450364386. DOI: 10.1145/3240431.3240443. [Online]. Available: https://doi.org/10.1145/3240431.3240443.

[25] A. Dingle and T. Pártl, ‘Web cache coherence’, *Computer Networks and ISDN Systems*, vol. 28, no. 7, pp. 907–920, 1996, Proceedings of the Fifth International World Wide Web Conference 6-10 May 1996, ISSN: 0169-7552. DOI: https://doi.org/10.1016/0169-7552(96)00020-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0169755296000207.

[26] B.-J. Kwak, N.-O. Song and L. Miller, ‘Performance analysis of exponential backoff’, *IEEE/ACM Transactions on Networking*, vol. 13, no. 2, pp. 343–355, 2005. DOI: 10.1109/TNET.2005.845533.

[27] S. Misra and M. Khatua, ‘Semi-distributed backoff: Collision-aware migration from random to deterministic backoff’, *IEEE Transactions on Mobile Computing*, vol. 14, no. 5, pp. 1071–1084, 2015. DOI: 10.1109/TMC.2014.2341613.

[28] T. Kolajo, O. Daramola and A. Adebiyi, ‘Big data stream analysis: A systematic literature review’, *Journal of Big Data*, vol. 6, no. 1, p. 47, Jun. 2019, ISSN: 2196-1115. DOI: 10.1186/s40537-019-0210-7. [Online]. Available: https://doi.org/10.1186/s40537-019-0210-7.

[29] X. Zheng, P. Li and X. Wu, ‘Data stream classification based on extreme learning machine: A review’, *Big Data Research*, vol. 30, p. 100 356, 2022, ISSN: 2214-5796. DOI: https://doi.org/10.1016/j.bdr.2022.100356. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214579622000508.

[30] Mozilla Corporation. ‘WebSocket’. developer.mozilla.org. (Jun. 2023), [Online]. Available: https://web.archive.org/web/20230407011110/https://developer.mozilla.org/en-US/docs/Web/API/WebSocket (visited on 7th Apr. 2023).

[31] N. Lee and P. Busch, ‘User interface advances through the ajax javascript framework’, English, in *Proceedings of the 35th International Business Information Management Association Conference*, K. Soliman, Ed., International Business Information Management Association Conference (35th : 2020), IBIMA 35 ; Conference date: 01-04-2020 Through 02-04-2020, International Business Information Management Association (IBIMA), 2020, pp. 236–247, ISBN: 9780999855141.

[32] F. Ferreira and M. T. Valente, 'Detecting code smells in react-based web apps', *Information and Software Technology*, vol. 155, p. 107 111, 2023, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2022.107111. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922002208.

[33] R. N. Diniz-Junior, C. C. L. Figueiredo, G. De S.Russo *et al.*, 'Evaluating the performance of web rendering technologies based on javascript: Angular, react, and vue', in *2022 XVLIII Latin American Computer Conference (CLEI)*, 2022, pp. 1–9. DOI: 10.1109/CLEI56649.2022.9959901.

[34] M. Flores and H. Bedi, 'Caching the internet: A view from a global multi-tenant cdn', in *Passive and Active Measurement*, D. Choffnes and M. Barcellos, Eds., Cham: Springer International Publishing, 2019, pp. 68–81, ISBN: 978-3-030-15986-3.

[35] H. Ben-Ammar and Y. Ghamri-Doudane, 'Revopt: An lstm-based efficient caching strategy for cdn', in *2021 IEEE Global Communications Conference (GLOBECOM)*, 2021, pp. 1–6. DOI: 10.1109/GLOBECOM46510.2021.9685051.

[36] L. Chhangte, N. Karamchandani, D. Manjunath and E. Viterbo, 'Towards a distributed caching service at the wifi edge using wi-cache', *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4489–4502, 2021. DOI: 10.1109/TNSM.2021.3105496.

[37] J. Heo, S. Woo, H. Jang, K. Yang and J. W. Lee, 'Improving javascript performance via efficient in-memory bytecode caching', in *2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, 2016, pp. 1–4. DOI: 10.1109/ICCE-Asia.2016.7804810.

[38] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine Publishing Company, 1967.

[39] B. J. Oates, *Researching Information Systems and Computing.* SAGE Publications Ltd., 2006, ISBN: 1412902231.

[54] V. Garousi, M. Felderer, M. V. Mäntylä and A. Rainer, 'Benefitting from the grey literature in software engineering research', in *Contemporary Empirical Methods in Software Engineering*, M. Felderer and G. H. Travassos, Eds. Cham: Springer International Publishing, 2020, pp. 385–413, ISBN: 978-3-030-32489-6. DOI: 10.1007/978-3-030-32489-6_14. [Online]. Available: https://doi.org/10.1007/978-3-030-32489-6_14.

[55] A. Rainer and A. Williams, 'Using blog-like documents to investigate software practice: Benefits, challenges, and research directions', *Journal of Software: Evolution and Process*, vol. 31, no. 11, e2197, 2019, e2197 smr.2197. DOI: https://doi.org/10.1002/smr.2197. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2197. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2197.

[56] World Wide Web Consortium. 'State Chart XML (SCXML): State Machine Notation for Control Abstraction'. w3.org. (Sep. 2015), [Online]. Available: https://www.w3.org/TR/2015/REC-scxml-20150901/ (visited on 14th Nov. 2022).

[57] D. Harel, 'Statecharts: A visual formalism for complex systems', *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987, ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(87)90035-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167642387900359.

[58] Ecma International. 'ECMAScript® 2022 Language Specification - Proxy Objects'. tc39.es. (Jun. 2022), [Online]. Available: https://tc39.es/ecma262/2022/#sec-proxy-objects (visited on 6th Dec. 2022).

[61] Nottingham, Mark. 'RFC 5861 HTTP Cache-Control Extensions for Stale Content'. rfc-editor.org. (May 2010), [Online]. Available: https://www.rfc-editor.org/rfc/rfc5861.html (visited on 18th Apr. 2023).

[62] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, 3rd. Addison-Wesley Professional, 2012, ISBN: 0321815734.

[63] Mozilla Corporation. 'Web Storage API'. developer.mozilla.org. (Mar. 2023), [Online]. Available: https://web.archive.org/web/20230421003755/https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API (visited on 21st Apr. 2023).