Tampere University

Prosper Evergreen

SELECTING A STATE MANAGEMENT STRATEGY FOR MODERN WEB FRONTEND APPLICATIONS

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
April 2023

Abstract

Prosper Evergreen: Selecting a State Management Strategy for Modern Web Frontend Applications
Master's thesis
Tampere University
Master's Degree Programme in Computing Sciences
April 2023

State management system plays an essential role in any application. Just as the circulatory system in a living organism aids in transporting nutrients and other vital materials from where they are produced to where they are used to keep the body alive and healthy, the state management system of an application defines how data is controlled, distributed and moves around within the application. However, managing the state of an application could quickly get complex and if not properly addressed, could lead to various issues with the application.

The purpose of this research work is to provide a strategy for managing the state of a modern frontend web application to avoid common state-related pitfalls that could arise in the lifetime of an application.

To achieve this goal, the thesis work provides a guideline that answers the questions on what should be considered when choosing a state management system and how the states of a frontend application should be handled. To show the effectiveness of this guideline, a case study was conducted on an application called Nokia Test Automation Platform (NTAP) which is an automated test-line solution developed at Nokia. The case study involved re-implementing the NTAP state management system based on the proposed strategy and then comparing the results of similar tests taken before and after the re-implementation.

The result of this work includes the identification of features of frontend state and frontend state management, a strategy for modern frontend web state management and the re-implementation of the state management system of NTAP.

Keywords: NTAP, frontend, state management.

Acknowledgments

I would like to begin by thanking God for granting me the strength, wisdom, and comprehension to successfully complete this thesis.

Special recognition to my supervisors Timo Nummenmaa and Pia Niemelä. I am grateful for the time and effort invested in this thesis work, and also for the encouragement which boosted my confidence to pursue my goals.

An extra-special thank you goes to Daniel Girmay my line manager at Nokia. Your suggestion and encouragement were key to the realization of the thesis.

This section would be incomplete without mentioning the input of my brother, Uchenna Ezeobi, whose experience as a PhD student and continual assistance contributed immensely towards the success of this thesis.

Lastly, I would like to thank Titus Eze, Promise Emeh, Benjamin Iwuchukwu and Elina Pesonen for the support and encouragement you provided in various ways throughout the duration of my work on this thesis. This enhanced the quality of the work and simplified the process.

Contents

List of Symbols and Abbreviations

API Application Programming Interface. 6, 41, 44

AR Augmented Reality. 1

DOM Document Object Model. 23

GUI Graphical User Interface. 20, 39–41, 44

HTML HyperText Markup Language. 1, 5, 23

NTAP Nokia Test Automation Platform. 4, 39–50, 52, 53

PWA Progressive Web Applications. 7

SPA Single Page Applications. 7

SWR State While Revalidate. 36

UI User Interface. 1, 6–9, 23, 45

VR Virtual Reality. 1

Glossary

GraphQL GraphQL is a programming language designed for Application Programming Interfaces (APIs) that also acts as a runtime environment to execute these queries with the existing data. This language offers a comprehensive and easily comprehensible description of the data required by the API, empowering clients to request only the data they require, simplifying the process of updating APIs over time and allowing for advanced developer tools [1]. 14, 36, 37

REST Representational State Transfer (REST) is a style of software architecture that outlines a consistent, decoupled, stateless, cachable, and layered system communication interface between separate components, usually across the internet within a client-server framework [2]. 14, 41, 44

1 Introduction

Nowadays, most applications are developed to be interactive and not just read-only HTML pages. An interactive application is one that enables users to engage with audiovisual data via methods such as gamification, visualization, and even VR/AR, with the ultimate goal of achieving a specific objective. These interactions could be in the form of clicking a button, liking a post, etc., which could originate from the users and are usually accompanied by feedback from the application. An interactive web app could be a game, social media, blog, etc. For example, by clicking a login button of a social media app, the user is sent to the login page, by liking a post the number of likes on the post increases and the user sees a like indication, by deleting a comment the comment is no longer visible, etc. For this to be possible the data of the application needs to change to be in sync with the actions of the user.

The state of an application can be seen as a representation of the data variables that make up the application at any given time. Consider a blog website where users can make posts or comment on posts, the posts and comments at any given time before interaction with the application can be regarded as part of the state of the application. Any application having a state needs to have a means of managing the data affecting the state of the application [3]. In the blog website, for example, there should be a way to add, delete or even edit existing posts and comments. The process of updating the state of the application to reflect the actions of the user i.e., from the moment the user interacts with the application to the point the feedback is given back to the user, can be quite complicated and if not done properly will lead to poor user experience as well as loss of data. Managing this process is essential for any interactive application to work properly and can be regarded as state management.

Frontend web application which is an application or part of an application that runs on a browser also requires a state management system like most applications do to be interactive, responsive and more interesting for the users. The state of a frontend web application is usually reflected by the User Interface (UI) or view of the application that is the browser web page.

The importance of state management should not be overlooked as it is not only a core requirement for building complex applications but also crucial for applications that require some sort of communication with other independent components such as third-party applications [4] or within the same application.

A state management system is a core feature of any application. It serves as the building block for defining what an application can do. It controls how data changes within each component of the application. The same can be said for any

web frontend application.

Understanding front-end state management techniques is necessary for the efficient handling of data and memory usage, optimizing performance, avoiding unnecessary operations, improving predictability, providing a better user experience, and supporting the scalability and maintenance of an application.

In order to tackle some of the issues with frontend state management, Song, Jeong, Hutto, *et al.* [5] proposed three models in the article "State management in Web services": state server model, database model, and proxy model as solutions to improve the performance, fault-tolerance, and data persistence problems for web services state management. Song, Jeong, Hutto, *et al.* [5] state that although the models are fault-tolerant to some certain level, they still have additional overheads. Furthermore, the authors point out that the models come with some advantages as well as disadvantages that would be important to consider. For example, the database model supports the persistence of data at the expense of performance, the proxy model is simple but has security, coordination, and reliability issues and the server state model comes with complexity issues.

Erdanto and Sujarwo [6] in their article "Improving Effectiveness of State Management Using Prop Drilling Pattern on Jala Tech's Financial Feature" pointed out that the states of an application get more complex as the application size increases and as such would require a state management system. The authors seek to improve the state management architecture of their product. Finding it important but difficult to track how state changes, the authors present a prop drilling approach as a solution to the problem. The authors also highlighted how poor state management could lead to state redundancy or duplication which harms the performance of the application as a whole.

Szymanek and Pańczyk [7] compare four (4) state management libraries (NgRx, Ngxs, Redux, and VueX) in the article "Comparison of web application state management tools", using React.js, Vue.js, and Angular which are front-end development frameworks (i.e., a set of tools for developing an application). The authors tested the state management libraries on similar applications, comparing the code size, architecture, ready-made solutions, support, and performance. The result found VueX as the best solution. However, the research did not consider the development framework compatibility with the state management libraries. Furthermore, the authors did not consider how suitable the state management libraries were for the application they were tested on.

In summary of the review conducted on related works, there was not much research done on choosing tools for frontend state management from an application's point of view. State management tools are designed to meet the specific state needs of an application. Therefore, there should be a justification for using a state

management solution based on the application's state requirements. Furthermore, the guidelines to be considered when choosing a state management tool were found to be limited to certain development frameworks and applications rather than generic ones. These constraints make the guideline mostly useful for people working in such a development environment and could even be harmful to other development environments. However, the research conducted gives clear importance to proper state management which builds a solid foundation for this thesis work.

The purpose of this thesis work is to answer the following questions with regard to modern web frontend applications:

- What are the requirements for choosing state management tools?
- How should the states of a modern web frontend application be managed?

Choosing the right methods for managing the states of an application is fundamental in determining how the application reacts to changes occurring during its life span, as well as controlling and providing the expected responses, keeping the application from crashing. Although this could be ambiguous and confusing, this thesis work simplifies the process of selecting the most suitable solution for a given application. This includes picking the right tools to do the job without which it will be more demanding to lay a good foundation for implementing a suitable design for the application. The various methods, architectures, and approaches used to manage the states of a modern front-end application are identified. However, to better understand when and where the techniques are best suited, this thesis work also aims to study, identify, and categorise the various types of states of an application and provide recommendations on which management approach should be considered.

For the purpose of this thesis, descriptive and analytic research methods were used in developing the state management strategy in Chapter 3.

To identify and analyse factors that affect the state management of a frontend application, a study of publications which addressed web design, architecture and development was conducted with a focus on highlighting the requirements that play certain roles in the state and state management of a web application. Furthermore, the analytical method was used to study several state management technologies and tools based on their popularity, uniqueness, etc. This was done to understand common problems faced by modern web applications which the technologies or tools try to solve. The study also identified the key factors that could lead to such problems. Having a good knowledge of factors that affect the state of an application as well as issues related to state management, research was conducted on relevant works which attempt to solve similar problems. These solutions were then compiled to form a strategy with a justification for the importance of each part of the strategy.

As part of the research methods, a descriptive and quantitative case study was conducted in Chapter 4. This was to study, analyse, compare and ascertain the usefulness and effectiveness of the proposed state management strategy. A software application called Nokia Test Automation Platform (NTAP) that is been developed at Nokia was used for the case study. The state management of the application had previously been implemented without the use of proposed guidelines making it a perfect fit for the study. Some state-related statistics of the application were collected as a means to determine and measure the impact of the state management system on the application. To study the effectiveness of the proposed strategy, the state management system of NTAP was re-implemented using the proposed strategy. After the re-implementation was completed, the same state-related statistics were gathered as before the implementation but this time with the new implementation. This was done to get an insight into the impact the newly implemented state management system plays on the application. The results being comparable show the effectiveness of the proposed state management strategy on an application as a whole.

2 Web State Management

In the early days of web application development, most web applications were either basic static HTML documents or server-side generated web pages [8]. This was largely related to the fact that most devices had little memory space, low computational capabilities as well as slow internet speed. Furthermore, browsers were not mature enough to process complex tasks but focused on rendering HTML documents. To meet the performance and computational demands of web applications, an environment separate and independent of the browser were utilised called a server. Servers usually have more computing power than the browser and handle most and sometimes all computational tasks of the application such as web page generation, data handling as well as response to user actions on the web application. Traditional web applications usually follow this approach of using a server to handle most or all of the applications' tasks.

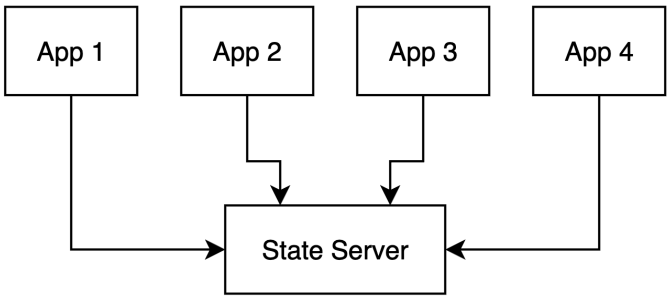
With the advancement in technology, devices became more powerful to quickly handle complex computational tasks and browsers became more mature to better utilize the resources of the device on which they are running. These features paved the way for the realization of a true interactive front-end application, i.e., an application that can respond to users' actions with little or no interaction with a server or external control. Examples of these types of applications can be seen in some web games which sometimes do not require any communication with a server after they are loaded on the web page.

When compared to traditional applications, front-end web applications try to separate responsibilities between what the server handles and what the client application controls [9]. The separation of responsibilities has resulted in many advantages which include:

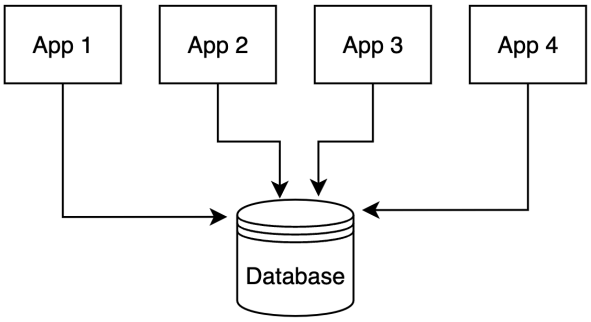
1. Avoiding fetching unnecessary data for rebuilding the same page
2. Fewer data needed to run the application
3. Optimized re-rendering, i.e., only the part of the page requiring changes is updated rather than full-page reloading.
4. Faster response rate on the client-side
5. Reduce server-side code complexity by focusing only on server state
6. Better user experience with offline support

For an interactive application to work properly, there should be a means to respond to interaction with the application. This interaction results in changes in the application's state. The process of controlling the way by which the application transitions from one state to another can be regarded as state management.

Before the introduction of front-end web frameworks, web states were mostly managed from an external environment using different models such as:



(a) State Server Model.



(b) Database Model.

Figure 2.1 *Front-end State Management Models [5]*

- **State Server Model**: In this model, the application's states are managed on a dedicated server and communicated to the client on-demand as shown in Figure 2.1(a) [5].
- **Database Model**: The state values are stored and updated by a database located in a separate environment and queried when needed as shown in Figure 2.1(b) [5].

However, Song, Jeong, Hutto, *et al.* [5] pointed out that the models have proven to have both performance and complexity issues as the state management system is relatively distanced from the client or front-end application. The approach where the application's state is directly managed from an external environment would result in calling different APIs each time to sync the state change with the UI. The downside is that it will need multiple network calls and requires more time to store the state in

one component and provide it to other components of the application. This overhead is not only inefficient for sharing the state but also prevents the application from working in an offline mode.

The introduction of modern front-end frameworks and libraries for developing front-end applications such as React, Vue.js, Angular, etc., revolutionized the way applications are built. These frameworks made the development process easier and the applications developed with them are designed to efficiently utilize the client's resources. Web applications often developed with these frameworks are usually in the form of:

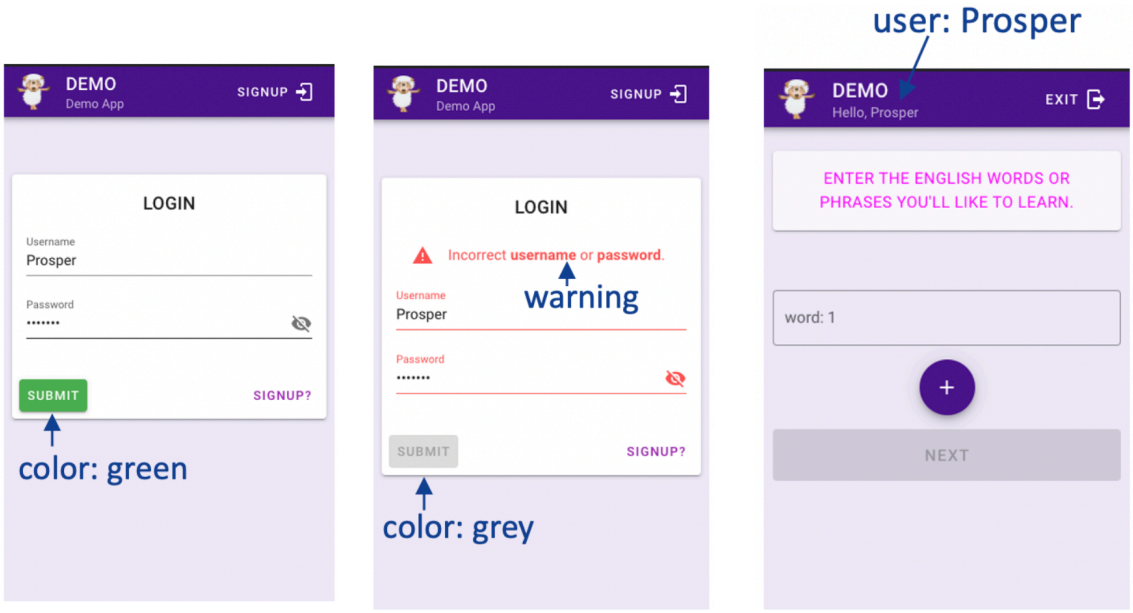
- **Single Page Applications (SPA)**: web applications that are located on a single page and operate without re-rendering the page. The application works by dynamically rewriting the current web page with new data, instead of loading new pages.
- **Progressive Web Applications (PWA)**: web applications that give a look and feel of a native or hybrid application.

As the popularity of front-end applications grew, so did front-end frameworks becoming more involved in building large and complex applications. Managing the states of such applications from an external environment could be very expensive and inefficient which resulted in the need for a specialized client-side state management system for better handling of data used by the client. This approach also reduced the exposure of the server-side thereby reducing security concerns when building resilient applications [3].

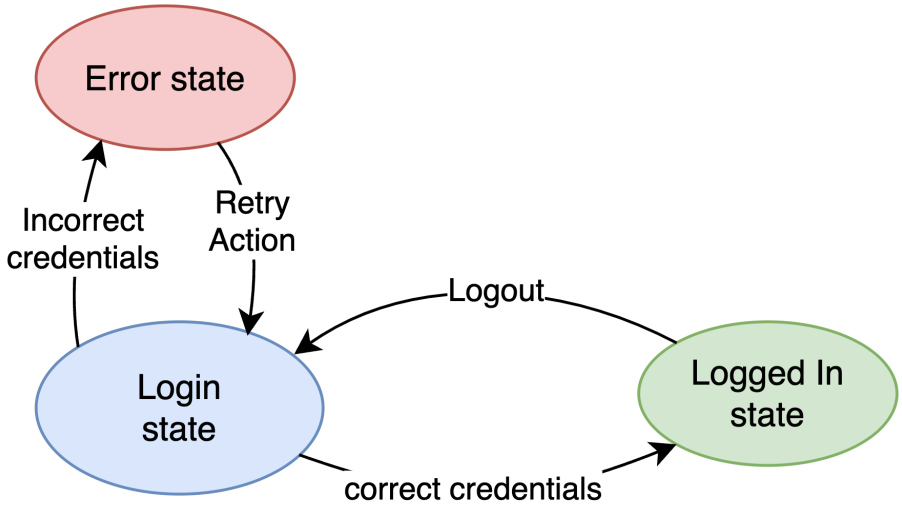
2.1 Front-End State

As front-end frameworks became more frequently used in building complex and large applications, so did it become essential to understand what are considered to be front-end states. A proper understanding of the components of the front-end state helps determine what the front-end framework should handle.

Frontend state, also known as client state can be defined as a representation of the data that is used by an application on the client side, e.g., a browser, at a given point in time. In other words, the state of a web application is dependent on the properties of the application that can change over the cause of the use of the application [6] or with time. These properties are usually in the form of data. The data could be in the form of strings, arrays, objects, etc. The state of an application affects how the application works, what can be done with the application, and in some cases, how the application looks, i.e., the UI and sometimes, what the user can see.



(a) Authentication state UI.



(b) Authentication state flow diagram.

Figure 2.2 *Authentication states.*

For example, the colour of the submit button can be regarded as a property of the front-end state as shown in Figure 2.2(a). The button's colour changes from green to grey when the front-end state goes from **login state** to **error state** as shown in Figure 2.2(b). In the error state, the submit button is disabled preventing the user from submitting a request (functionality restriction).

2.1.1 Front-end State Management

The responsibility of a front-end state management system is to handle every action or interaction with the application that requires a change to the application's state.

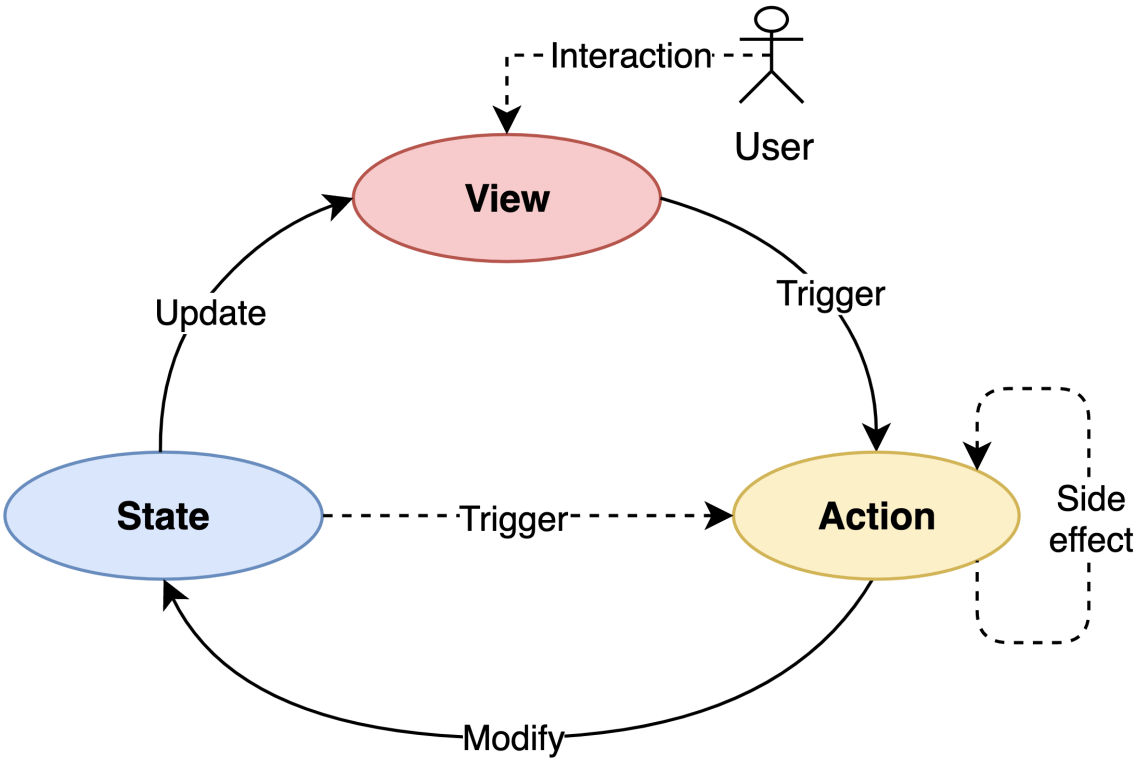


Figure 2.3 *Basic front-end states management system.*

Frontend state management can be seen as the process of defining the structure of data or state of a web application and controlling operations or actions that modify such data [10].

A basic process of state management can be described in three stages as shown in Figure 2.3. The UI or View shows a representation of the state with which users interact. These interactions could be a button click (Fig. 2.2(a)), page change, timer, login, etc. User's interactions with the application trigger certain events which tell the application what happened. Based on the event, the application can then decide what to do by taking an action. The action could be to mutate the state or cause other side effects like fetching data from an external source. When changes are made to the state of the application, it is then reflected in the view or UI for the user to understand what just happened. State change could be changing theme colour, login, logout, etc.

When developing an application, enough effort should be put into choosing the appropriate state management architecture and framework. An efficient and well-fitting design will lay the foundation for building a scalable and maintainable application [11]. A good front-end state management system should not only be beneficial in handling the state of the application but also ease the management and development process of the application. The features expected of a good front-end state

management system as highlighted by Song, Jeong, Hutto, *et al.* [5], Erdanto and Sujarwo [6], Lee, Wei, and Mukhiya [12], Kudiabor and Travis [13], and McFarlane [14] include:

1. **Accessible**: State data should be available and accessible when needed. Communication among web-based application components such as the frontend and server is usually stateless. This means that for every request, the client should be able to provide enough information for the accurate action to be carried out [5]. Modern front-end frameworks like React.js, Vue.js, etc. could be used to develop applications as a composition of small components that communicate and interact with one another. This decomposition would be advantageous in terms of re-usability and maintenance of the application but will require a good state management system to function together as a whole. Having access to the right data on demand will facilitate not only client-server communication but also inter-component collaboration.

2. **Readable**: The state management system should be understandable at a glance. It is very important for things to be organized and well structured, especially for large applications [12]. This will go a long way in making the system maintainable. A poor structure can lead to doing things inefficiently, for example storing the same data multiple times or unnecessarily modifying the state. A well-structured and readable system would make the development process easier.

3. **Persistence**: Even though volatile data is common in pervasive computing settings, it is crucial to occasionally persist data for further processing and analysis [5]. Data should survive the period of its requirement such as across pages, components or, even reload [5], [12]. It is important that the state management can persist data as long as it is needed to prevent data loss or unnecessary network requests. Data should also not outlive its scope which could lead to memory issues. Different caching techniques can be adapted for optimal performance and memory usage.

4. **Predictable**: The state management setup should reflect what the application state looks like or how it changes, i.e., being the source of truth [13]. It should tell the expected outcome of a given action. It would be a reliable source of truth at any given point in the application. Being predictable also comes with the advantage of easy debugging. This makes the process of finding errors a lot easier, such as the cases of unhandled state change.

5. **Performance**: The state management system should improve the performance of the application. This could be in the form of serving as a data

recycle station or handling state change in an optimized manner to minimize side effects like page re-rendering. Promoting re-usability such as providing the same data to multiple components prevents the processing of data multiple times or makes multiple network requests for the same data. Such improves the performance of the application.

6. **Scalable**: The ease to extend or expand an application or vice versa can be used to determine its scalability. Scalability is tidied to data as expanding an application results in the use of more data and most likely involves handling more states [6]. The state management system is among the fundamental building blocks for supporting the scalability of any application therefore, must be able to manage such scenarios with ease rather than limit it [14].

3 Frontend State Management Strategy

Managing the state of frontend applications easily gets complex if not well planned thereby making the application difficult to maintain or scale. To tackle this problem, this chapter proposes steps to take to avoid falling into such issues when developing a frontend application.

One of the most important aspects of frontend state management is tool selection. Choosing the right tools in software development is very important because they serve as a foundation on which the design is implemented.

Every technology is designed to solve certain problems. Just because a tool is used by others for state management does not justify using it in any application. If a tool is used without proper consideration, it could introduce further problems to the development process. In the case of state management, there could be some limitations on how the states can be managed as well as memory, performance and other issues. Therefore, choosing the tools with the design best fitting for an application's state management is essential.

When it comes to strategising the approach to state management, being able to provide answers to the following question puts the developer or development team on a pedestal for understanding how to properly manage the state of their application.

- What are the software requirements for the application?
- What is the architectural design of the application?
- What are the state types required by the application?
- Which development framework is used?
- What is the preferred state management approach?
- What are the available state management libraries and frameworks?
- How can the selected tools be used correctly?

Justifications as to the importance of these questions are presented below.

3.1 Software requirements

Before writing software code, it is necessary for the developers to specify the software requirements through analysis [11]. A solid understanding of a product's requirements ensures that developers tackle the given problem with the best solution [15].

In the same way, it is important to understand and analyze the software requirements when it comes to selecting the necessary tools for software development. State management is more concerned with the business, functional and non-functional requirements of the application.

3.1.1 Business requirements

Business requirements give insight as to why an organization is developing a system and usually contain the objectives the organization hopes to achieve [15]. Some applications are developed for experimental purposes whereas others are for business reasons. The motive behind the development of an application should influence the choice of tools used for developing the application. Enterprise-level applications must use enterprise-level tools and solutions for the development process. This means using solutions that have been tested and trusted. There are so many solutions for state management but it would be very important to consider the community behind the chosen solution, the number of downloads, the support team as well as the support life of the library or framework. For example, when developing enterprise-level applications, developers should consider choosing long-term supported tools with a wider community over tools that might offer more benefits but have the risk of being deprecated quickly. When developing a business-level application, it is important to use tools that the developers can fully manage. This becomes important when some tools run out of support and would need full maintenance by the development team to avoid putting the business at serious security risks.

3.1.2 Functional requirements

The functional or behavioural requirements of the software include the functionalities the application must have to enable end-users successfully perform the tasks expected of the application [15]. The functional requirements affect state management because it determines the size and complexity of an application. Being able to estimate the size and complexity of a software application is critical to the management process [16]. From a state management point of view, understanding the size and complexity of the application will help the developer in selecting tools that are designed to efficiently handle the needs of the application. A lot of metrics can be used to measure and determine the size of software, this includes metrics of function points, function blocks, lines of code, object points, or Bang metrics [16]. In general, the functional requirements of an application are directly proportional to the size and complexity of the application. Although, most frontend development frameworks offer some sort of state management solution. These solutions might not

be well suited for managing large or complex applications. When choosing frontend state management solutions, it is very important to ensure that the demand of the application is met by the state management solution. A good state management tool should simplify the process of managing the state of an application and not introduce more complexity.

3.1.3 Non-functional requirements

Non-functional requirements describe the quality attributes of a system [15]. It contains the requirements or benchmarks as to how the system should operate and not what the system should do. Performance, scalability, capacity and portability are some of the non-functional requirements that directly influence the selection of state management tools. Using an inappropriate state management tool could severely harm the performance of an application. For example, when the state management tool cannot cache data that do not change often, this will lead to unnecessary re-fetching or computation of such data and will affect the performance of the application. For applications that expect to grow with time must have scalability in mind when choosing the tools for state management. The state management tool used can help make the process of scaling an application easier. As technology changes quickly, it is recommended to use tools that can be decoupled and replaced by other tools easily. Having this in mind when choosing a state management tool will reduce the dependency of the application on a specific state management tool and make it easier to switch to other alternatives.

3.2 Application architecture and design

The system architecture of a web application can be simply defined as a blueprint which specifies how the various components such as databases, user interfaces, middleware systems, and servers of a web application interact with each other as a whole system [17]. The architecture does not only provide information as to how the components communicate but also how the components depend on each other. When choosing a state management tool, the developer should be able to extract from the architecture some useful information for the frontend state management such as:

1. Means or protocol of communication between the components of the application e.g. REST, GraphQL, etc. Some state management tools are designed with RESTful or other communication protocols in mind and are more suitable to support such architectures.

2. Paradigm such as declarative, imperative, etc. Using state management tools with similar paradigms will be easier for the developers to work with.

3. Design pattern e.g., Model-View-Controller(MVC), Model-View-Presenter(MVP), Model-View-ViewModel(MVVM), etc.

It is also important to understand how often the components communicate because caching techniques could be used to save a lot of performance for the application. The chosen state management tool should satisfy not only the architecture but also handle all the properties and components of the architecture which affect the frontend state. Choosing the right state management tool which supports the architecture of the application will not only ease the process of handling the states of the application but also ensure full benefits of the architecture are achieved.

3.3 Front-end state classification

The state of an application is dependent on the architecture of the application as well as the functional requirements of the software which are the functionalities that the developers have to build into the application to enable end-users successfully perform the tasks expected of the application [15].

It is essential to understand what kind of states will be required by the application because most state management tools are designed to handle certain types of states more efficiently than others.

Having a clear understanding of the types of states that exists in frontend applications will help developers identify and categorize the states necessary for the development of the application as well as choose the appropriate state management tools. In frontend applications, the states could be classified as server state, client state, global state, local state, derived state, page state, etc.

3.3.1 Server state

The server state also referred to as the domain state is the state in the application which are independent of the client-side of an application. Server states are persistent and accessible by any client application. Server states are usually fetched from an external source (server) and are always in sync with the server state. This means that when a server state value is modified, the server should be updated accordingly. For example, the username of the logged-in user in Figure 3.1 can be regarded as a server state.

3.3.2 Client state

Some states are only needed on the client-side. These states are usually related to the user interface of an application. When the state does not need to be saved to a database nor required to persist across client applications, such states can be

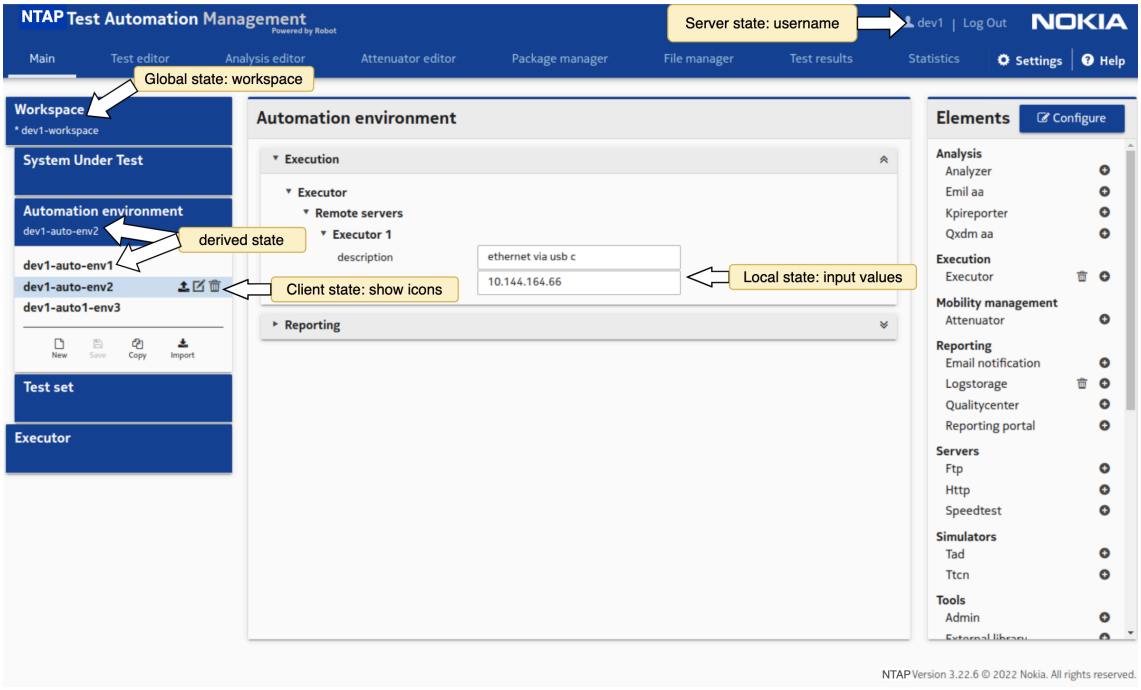


Figure 3.1 *Types of frontend state.*

regarded as client states or UI states. In Figure 3.1, when the mouse hovers over the items in the Automation environment section, the item is highlighted, and then the edit and delete icon is shown to the user. The highlighting and displaying of more actions on an item are regarded as a client state.

3.3.3 Global state

States that are needed across multiple parts of the application can be regarded as global states based on the scope of their usage. Global states are usually of a generic form and contain vital information used by multiple parts or components of the application. The states are expected to be designed to be managed from a central place, reusable, and accessible by the needed components of the application. In the example shown in Figure 3.1, the workspace is regarded as a global state as it provides data used by other sub-components such as the System under test, Automation environment, and Test set. Global states can also be client or server states.

3.3.4 Local state

Local states are states that are only needed in a specific part of an application. That is to say that they do not influence other parts of the application. Local states are not commonly required to be persistent. They are usually needed for a short time

and change very often. A typical example of a local state could be the value of an input field as shown in Figure 3.1. Local states are often client states but can also be server states.

3.3.5 Derived state

Some states are subsets or computed forms of other states. When a state is dependent on other states, it can be regarded as a derived state. Derived states can be modified through the states they are dependent on or vice versa. In the example, as shown in Figure 3.1, the Automation environment is a subset of the workspace and so is dependent on the workspace. When a workspace is deleted, all the dependent state data, including the Automation environment, is also deleted. A derived state can be a global state, local state, or client state.

3.3.6 Persistent state

Persistent states are states that are designed to persist beyond the lifespan of the components they are used in. In other words, when certain components are no longer in use or have been replaced, the persistent state remains intact and available for future use. Some of such states even need to survive browser refreshes. Caching is commonly used when managing such states. A persistent state can be a client or server state. Persistent client states are mostly used to provide a better user experience by remembering the users' settings or for offline functionalities of the application.

3.3.7 Non-persistent state

States that change very frequently or states that need to be updated all the time are called non-persistent states. This state needs to be re-evaluated most of the time. Managing such states close to where they are used will improve the performance of the application. An example of a non-persistent state would be the state of an input field.

3.3.8 Page state

The state stored in the URL of the application is regarded as the page state. This state usually identifies the page. It can also provide the initial data for setting up the page or tells the current state of the page. This state is usually represented as the path or query parameters of the page.

A simple guide to better understand how to classify a state would be to answer the following questions:

- Users: Where is the state used?
- Source: Where is the state coming from?
- Life span: How long should the state persist?

Table 3.1 *Classification of frontend state.*

Types of states	**Users**	**Source**	**Life span**
Client	multiple, single	internal	usually short
Server	multiple, single	external	usually long
Global	multiple	internal, external	long
Local	single	internal, external	short
Derived	multiple, single	internal, external, both	long, short
Persistent	multiple, single	internal, external	long
Non-persistent	multiple, single	internal, external	short
Page	multiple, single	internal	page life

3.4 Development framework and libraries

State management starts from the development framework. The development framework serves as the foundation on which every other component of the application is built. Using the right development framework will not only give you an extensive option of external state management libraries that could be integrated into the application but can also completely handle the state management needs of the application.

While it is possible to write a frontend application with pure JavaScript, the emergence of several libraries and frameworks has made it unnecessary to reinvent the wheel. In 2022, React.js, Vue.js, and Angular accounts as the most popular frameworks for developing frontend applications [18]. These technologies are discussed in detail to highlight the state management solutions they offer.

3.4.1 React.js

React is a JavaScript library for building a user interface (UI) in a tree form. This means that the UI is created by integrating smaller units called components [19]. React follows a declarative approach for defining the view of its components. React encourages breaking down an application into small components. This approach brings the benefits of component reusability, easy debugging as well as reduced complexity in terms of management.

The components in React can display data that may change with time or as users interact with the application and also control how the data changes. To control how the states in a react application change, react comes with some built-in function for state management. Components that contain data that may change are called **stateful components** whereas those that do not change the state of the application are called **pure components** [19]. React components can be function-based or class-based components depending if they are defined as a JavaScript function or class [20].

Similar to most modern frontend frameworks, React comes with tools for state management out of the box. This includes *setState*, *useState*, *useReducer*, *context* and *useContext* functions. In addition, react also provides means for improving efficiency when it comes to state management. This includes the use of *useMemo* to cache values that could be expensive to compute or *useEffect* which could trigger a function such as a network calls when the state changes. Furthermore, React can easily integrate with other state management tools which gives developers a wide variety of options to choose from.

3.4.2 Angular

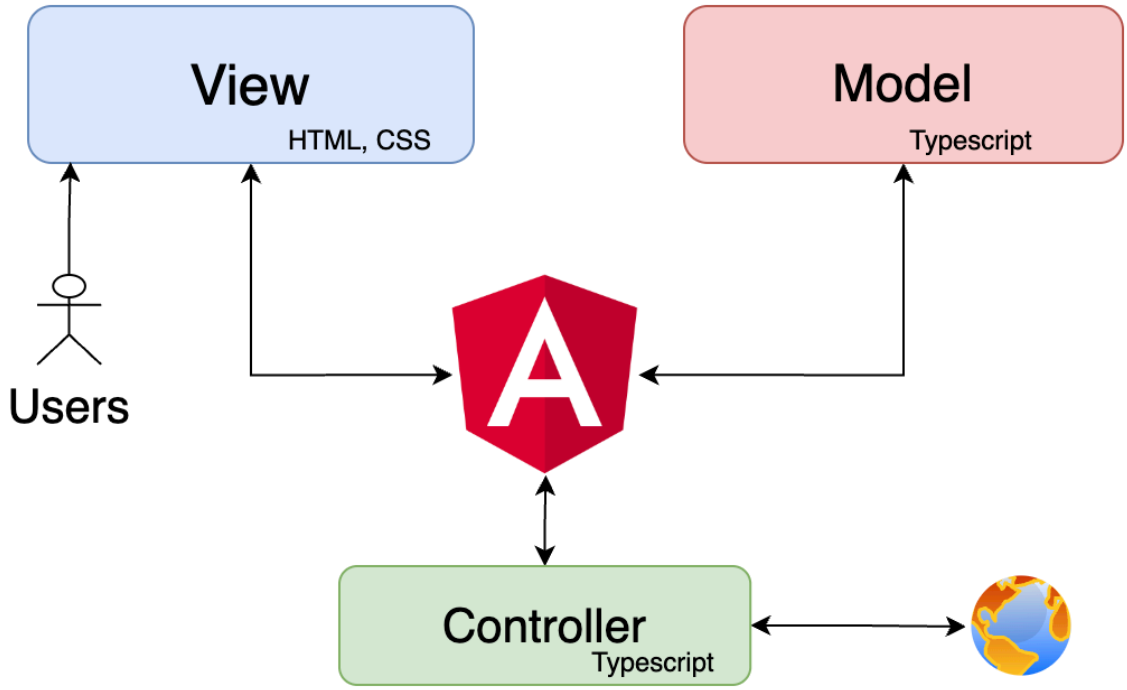


Figure 3.2 *Angular Model–View–Controller architecture [21]*

Angular is a client-side development framework for building a frontend web application developed by Google. It provides a modular and component-based way of combining HTML, CSS and Typescript to develop a web application [21], [22]. Angular provides a solution for handling the necessary requirements of a frontend application. This includes handling user interactions such as input, manipulating data on the client side which involves state change, and displaying or updating the view to reflect the state of the application. Angular is written using Typescript which is a superset of JavaScript. Using Typescript to enforce type safety, in this case, provides further benefits over JavaScript such as reducing type errors, improving consistency, providing better error messages, and supporting IntelliSense suggestions. Angular makes the development of SPAs easy and simplified by utilizing the MVC architecture (Figure 3.2).

Features of Angular include:

1. Modularity: The modular structure of Angular makes management of the application components easier and well organised.

2. Data binding: Angular provides a method of automatically reflecting state changes of the application on the UI and vice versa.

3. Extensibility: The framework architecture supports customisation by allowing the developer to extend most aspects of the language.

4. Clean: The framework enforces writing neat and well-structured code.

5. Reusable code: As a result of the code structure, code reusability can be easily achieved.

6. Support: Having a backing and huge investment from Google, Angular has very strong support.

3.4.3 Vue.js

Vue.js is an open-source Model-View–ViewModel(MVVM) JavaScript framework for building user interfaces and single–page applications [24] as shown in Figure 3.3.

The MVVM design pattern is a software architecture that enables the segregation of the GUI development, which may involve the use of either GUI code or a markup language, from the development of business logic or internal functionality (model). This ensures that the GUI does not rely on a specific model platform, thereby improving its portability and flexibility [25].

The MVVM representation model is a value converter [24], which means that the representation model is responsible for displaying (transforming) data objects from the model in such a way that objects are easily managed and demonstrated.

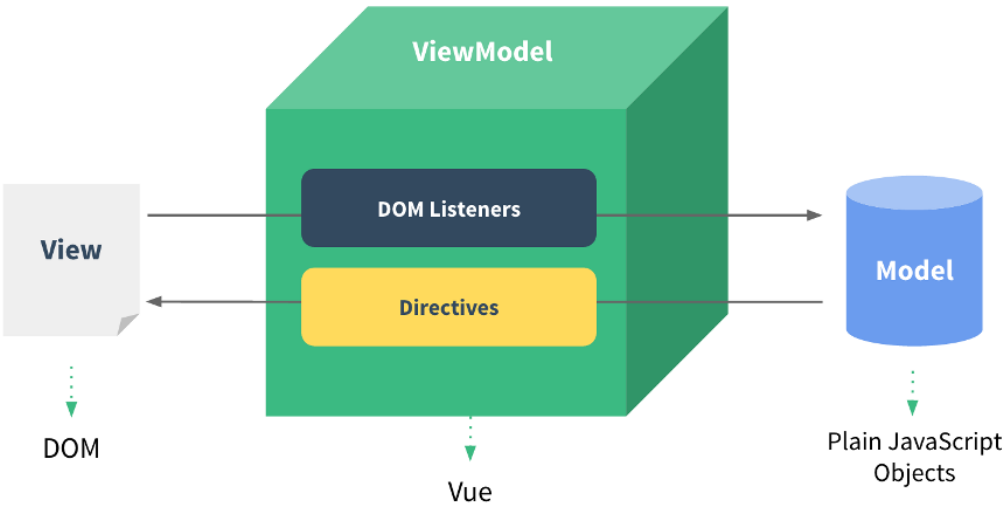


Figure 3.3 *Vue.js Model–View–ViewModel architecture [23].*

Vue is also a progressive Javascript framework [23]. This framework is referred to as a progressive framework due to its unique design that allows for increased adaptability over time. As a result, it is relatively straightforward to incorporate into diverse frameworks and libraries [23]. When used in conjunction with contemporary tools and supportive libraries, Vue is capable of providing comprehensive support for intricate single-page applications. It was introduced by Evan Yu, who worked on various AngularJS projects while working at Google. Sharing his stories about Vue on GitHub, Evan Yu said: "I started Vue as a personal project when I worked at Google Creative Labs in 2013" [24]. Following his involvement in creating user interface prototypes using both Vanilla JavaScript and Angular 1, Evan Yu conceptualized a notion centered on addressing issues within Angular in a more straightforward, less complicated, and more user-friendly manner[24].

3.4.4 Other development frameworks

In addition to the previously mentioned frameworks, there are other development frameworks that a quickly gaining interest in the developer community. This includes Svelte, Solid, Elm and Ember.js. Svelte is a lightweight framework with API similar to React. However, unlike most modern frontend frameworks, Svelte does not have a virtual DOM which results in a faster rendering or re-rendering speed. Elm uses a strictly functional approach to develop applications that are void of runtime errors and are super fast. Ember.js was introduced as an all-in-one solution for developing modern web applications. Ember.js comes with state management,

routing and optimized rendering solutions out of the box.

3.5 State management patterns

Having to manage the state of an application in an unorganised manner can quickly get ambiguous or even unmaintainable. As a result of this problem, different state management approaches have been developed to not only simplify state management but also optimize the process and make the scalability of an application easily achieved. Using the wrong design could make the application more rigid, fragile, immobile or viscous [26]. It is important to understand the state management approach suitable for an application in order to choose state management tools that support the implementation of such a design. Some of the most commonly adopted architectures and patterns for managing states in modern frontend web applications are introduced.

3.5.1 Elm Architecture

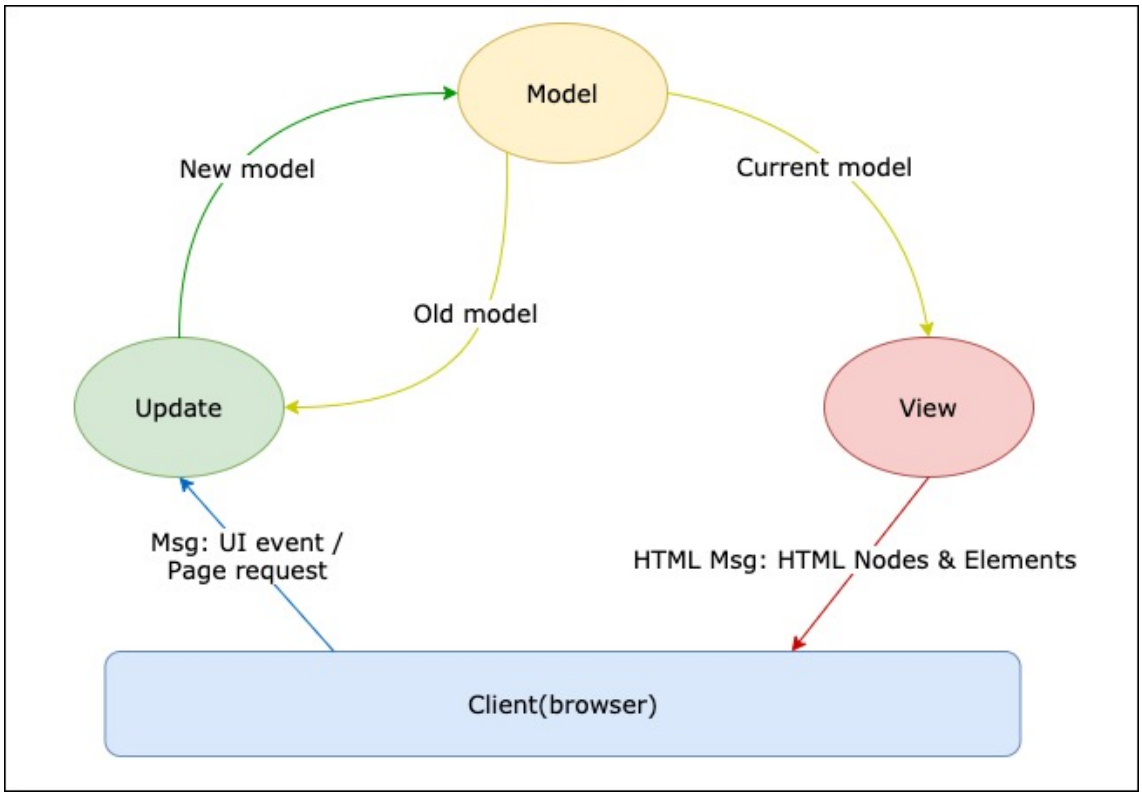


Figure 3.4 *The Elm Architecture: Model-View-Update [27]*

Elm is an architecture and programming language invented by Evan Czaplicki as a functional reactive language [28]. It is important to note that the Elm architecture

inspired many other popular state management architectures and patterns. This can be attributed to its simplicity and predictability.

The Elm architecture is referred to as Model-View-Update (MVU) pattern [27] as shown in Figure 3.4. The Model-View-Update pattern inherits the Model-View-Controller (MVC) design pattern [27] by having a model which holds the application's current state at any given time; a view function which renders the application and can reflect the state of the model; and an update function which handles actions that require modification to the application state.

Model

The model represents the application state as well as shows its data structure [29]. The model can be seen as the single source of truth for the application state at any point in time holding the current values of the application state [28], [30]. The model is set with the initial state of the application and then gets updated based on actions made to application [28].

View

View is a function with the task of generating the User Interface (UI) [29]. It contains the markup for the UI [28]. Using the view function, Elm is able to generate the HTML DOM elements or nodes such as div, button, h1 etc. In addition to creating the UI elements or nodes, the update function also renders the model [27] which is fed to it as an argument and updates the UI accordingly if changes occur to the model.

Update

Update is a function which handles user interaction or events emitted by the UI. The update function is called when an event occurs in the UI [27]. The purpose of the update function is to make changes to the model based on the actions made to the UI [29]. To make this possible, the update function takes the Msg (UI event) and the model as its arguments and returns the new model [30].

As shown in Figure 3.4, the app life cycle starts with the initialization of the model with the starting state of the app. This state is then presented by the View function (e.g. 3.5.1) to build the UI. The view also provides methods for interacting with the app in the form of a message that is defined for the app. When a user interacts with the UI, the view then sends the message (information) to the update function e.g. 3.5.1, which has access to the current model and can make changes to it. The changes made to the model are then sent to the view which updates the UI accordingly.

3.5.2 Flux architecture

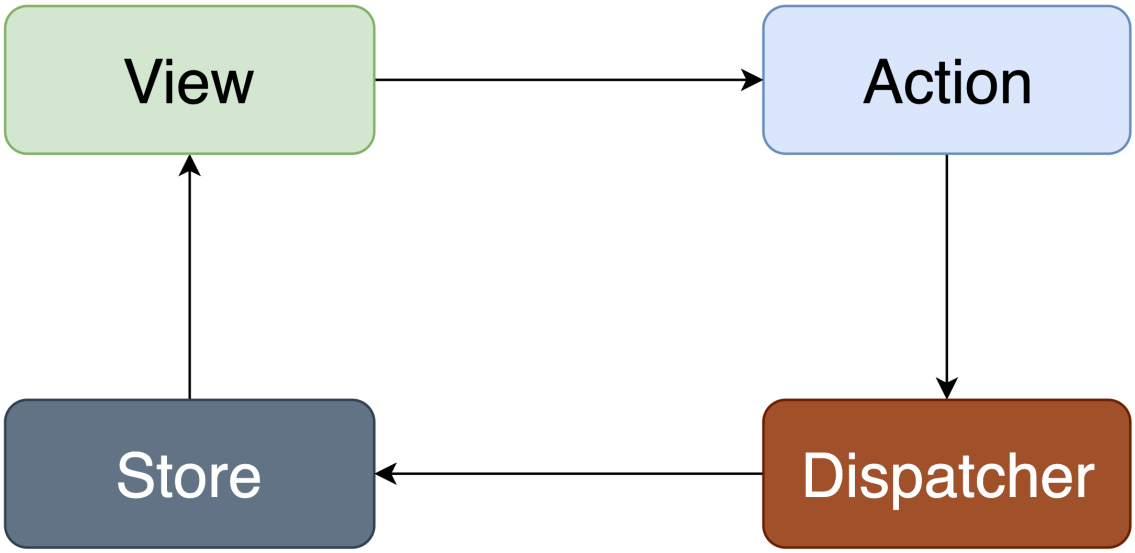


Figure 3.5 *The Flux architecture [31].*

Flux is an application architecture introduced by Facebook as a set of patterns for managing complex client-side applications in such a way that a tangled weave of data flow and unpredictable results is avoided [31]. The architecture enforces unidirectional data flow and ensures that state changes occur in a synchronous and predictable manner [32] as shown in Figure 3.5. This synchronous approach makes it possible to know when and control how the state changes occur [32].

The Flux architecture is motivated by the View-Update-Model pattern and has three major parts: the dispatcher, the stores, and the views. In addition, there are **the actions** which are dispatcher helper functions. The components are loosely coupled making it easy to scale and adapt to most architectures.

Dispatcher

The dispatcher serves as the central station through which all access or actions to the store are managed [32]. In simpler terms, the dispatcher is where actions to be executed on the store are defined. To perform certain actions or modifications to the store, identifiable callbacks are registered to the dispatcher [32]. When an action is triggered, the dispatcher is able to identify the action and execute the appropriate callback on the store. The dispatcher works as a traffic controller to the store in the sense that it ensures that only one action is performed on the store at a given time.

Store

The store handles the state of the application. This includes the client-side data and logic. There can be multiple stores in one application, each managing the application state for a particular domain.

Stores register themselves with the dispatcher and provide it with a callback that is used to access and modify the stores. When an action is triggered, the dispatcher informs all the stores registered to it about the action through the callback that stores provided on registration. The callback is then executed on the store's data, modifying the state of the application. When the state is modified, it triggers an update function on the view to update the data that is displayed by the view.

View

The view in flux architecture serves primarily two purposes: initiate actions and utilize store data for the UI. To maintain a unidirectional data flow, views cannot modify the application state data directly but are required to initiate an action [32]. For example, when a delete button is clicked, the view can invoke a DELETE_ITEM action, which is sent to the dispatcher and then handled by the store. On the other end, the view consumes and uses the data in store to render the UI of the application. When the state data in the store changes, the view is informed about the change so that it can update the data it uses to render the UI.

Action

The Flux pattern tends to be more explicit by describing everything that can happen to the store. These are grouped as actions. Actions are like transporters of events that require some sort of response by the application from the view to the dispatcher [32]. Every modification to the store starts with an action. An action typically has an identifier and specific predictable impact on the store which is explicitly defined in the store logic. The dispatcher exposes a dispatch function which is used to inform the stores of actions to take [31]. The dispatch function takes an action with an identifier and an optional payload which is then broadcasted to the stores registered to the dispatcher. The stores which expect such actions are able to identify the action via the action's identifier and then perform the respective modification to the store data.

3.5.3 Redux architecture

Redux architecture is used for creating a predictable state container [34]. Redux architecture was inspired by Flux and Elm architectures [35]. Redux is similar to

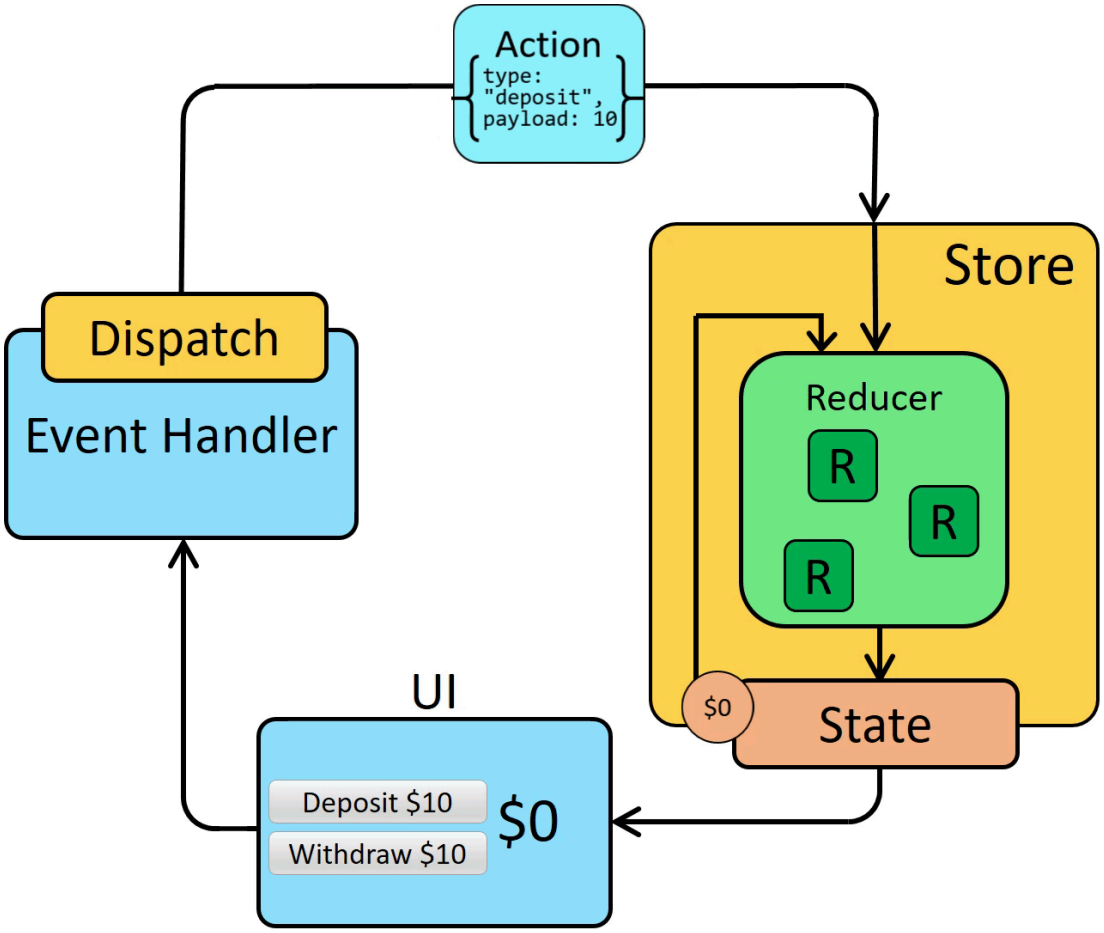


Figure 3.6 *Redux data flow diagram [33].*

Flux in that it has a unidirectional data flow, making use of Actions and a store called Reducer as shown in Figure 3.6. Unlike Flux, Redux favours a centralised store where all the states of the application are managed. Also, Redux does not use the same concept of Dispatcher as in Flux but favours the use of pure functions instead of event emitters. Pure functions are composable and easier to manage. Also, in Redux architecture, the state is immutable. Redux is explicit and declarative on how the state of the application is modified.

Reducer

Reducer is a function that is unique to the Redux architecture and differentiates it from Flux. it is a function used to calculate the new state of the application. It is inspired by Elm in that it receives the old state and an action and returns the new state of the application which is $(oldState, action) => newState$. Reducers handle all the actions that are related to the state modification similar to the store in the Flux pattern. Asynchronous logic or side effects are not allowed in the reducer

to keep it predictable. To modify the state, a completely new state needs to be returned. This can be achieved by either making a copy of the previous state first and then modifying it or a new state can be created from scratch with the changes included.

Dispatch

The Redux store exposes a method called dispatch *store.dispatch*(). This is the only way to inform Redux to update the state data. It takes an action as its argument and passes it to the reducer. The store will then execute the reducer function and save returned data as the new state value inside. The store also exposes a function *store.getState*() to retrieve and update the state of the application

Action

Actions in Redux are similar to that in Flux. They are objects that contain information about what the reducer should do. They can either be a plain object or a function that returns an object [33]. They typically contain an identifier or the type of the action and an optional payload or data to be used to execute the action.

Redux approach to state management is based on four (4) main principles [36]:

1. Data flow is strictly unidirectional
2. A single source of truth
3. The read-only nature of the state
4. The reducer principle

3.5.4 Vuex Pattern

Vuex is both a state management pattern and a library designed for Vue.js applications [37]. The fundamental concept of Vuex takes inspiration from Flux, Redux, and The Elm Architecture [37]. It is designed to serve as a centralized state management system for all the components in an application [38]. By following the pattern's rules, the state mutation is ensured to happen in a predictable fashion.

The state management system is subdivided into four (4) parts as shown in Figure 3.7:

- **State**: The state constitutes a unified entity that accommodates all the state data at the application level and serves as the definitive reference for all state-related queries also regarded as the "single source of truth". The utilization of a single state tree in Vuex simplifies the process of locating a particular state

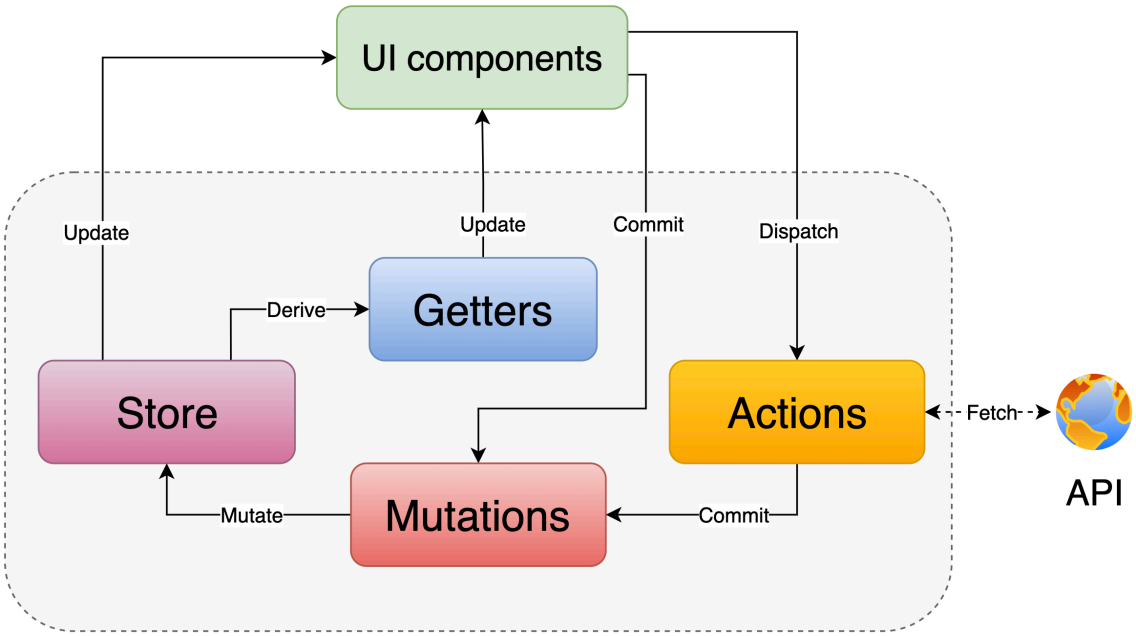


Figure 3.7 *Vuex pattern [37]*

element and enables the effortless creation of snapshots of the current state of the application, which can be utilized for debugging objectives [37].

- **Getters**: The getters are used to create a derived or computed state. They are functions that receive the state as an argument and return processed state values.
- **Mutations**: This is the only place the state can be modified. The store exposes a *commit* function which accepts a string i.e. the mutation identifier and an optional payload. Mutations are functions that take the state as well as the payload as arguments and can directly modify the state of the application. Vuex is able to track and update the state modifications made in the Mutation functions. Mutation functions must be synchronous.
- **Actions**: Actions are similar to Mutations, the difference being that actions be either synchronous or asynchronous. Also, actions cannot mutate the state directly but have to commit the mutations to be handled by the Mutation functions. To trigger an action, the store exposes a function called *dispatch*. The dispatch function is similar to the *commit* function in that it takes the action identifier and the optional payload for the action.

In summary, Vuex pattern enforces the following guidelines [39]:

- The state is stored and managed in a centralised location at the application level.

- The state can exclusively be updated through mutations, which are synchronous transactions.
- Actions are employed for performing asynchronous transactions.
- Computation of derived data based on the store state is facilitated through the getters.
- The store can be modularized, where each module has its own state, mutations, actions, and getters.

3.5.5 Model View Controller Architecture

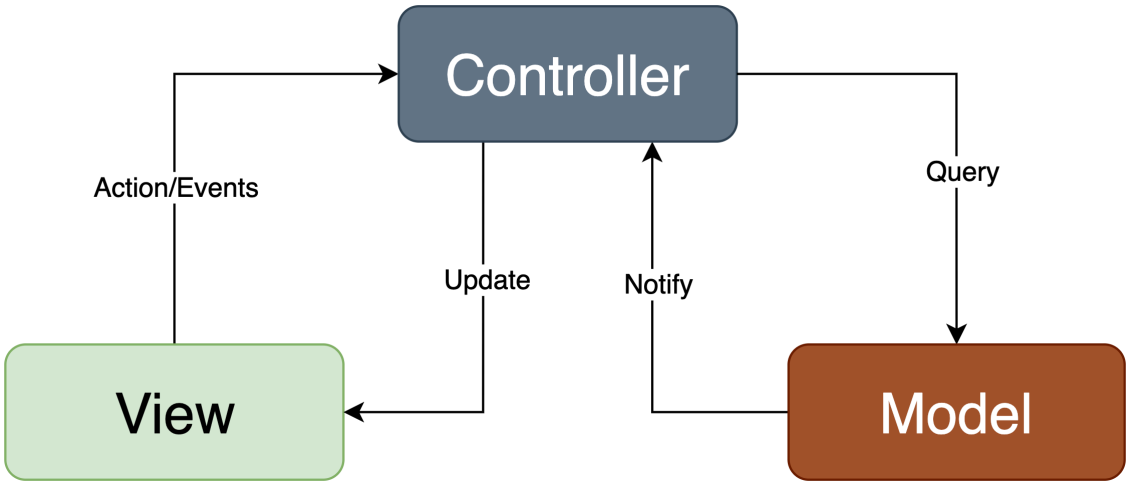


Figure 3.8 *MVC architecture [11]*

The Model View Controller (MVC) pattern was introduced as a design pattern that focuses on data and code structure. The architecture enforces the breakdown of the application into three (3) components as shown in Figure 3.8: View, Model, and Controller. The separation of concern makes the development and management of an application more reasonable to go about and easier to collaborate and work in parallel, improving productivity [11].

- **Model**: The model is the sole storage and manager of the application data. It handles everything related to the data in the application. This includes specifying, modifying, saving or updating the application's state. The model does not usually communicate to the View directly but through the controller.
- **Controller**: The controller links the View to the model. It does every other thing that does not have to do with managing the application data or displaying them. It listens for events or actions on the view and informs the model

to take the proper action. it also supplies the view with the latest data to be used for updating the UI. The controller can also be seen as the component that handles the business logic of the application interacting with the view, the outside world and the model.

- **View**: The view is responsible for rendering the UI and displaying the data. The view interprets user actions for the controller. The view generates the HTML and CSS code of the application.

3.5.6 Observer pattern

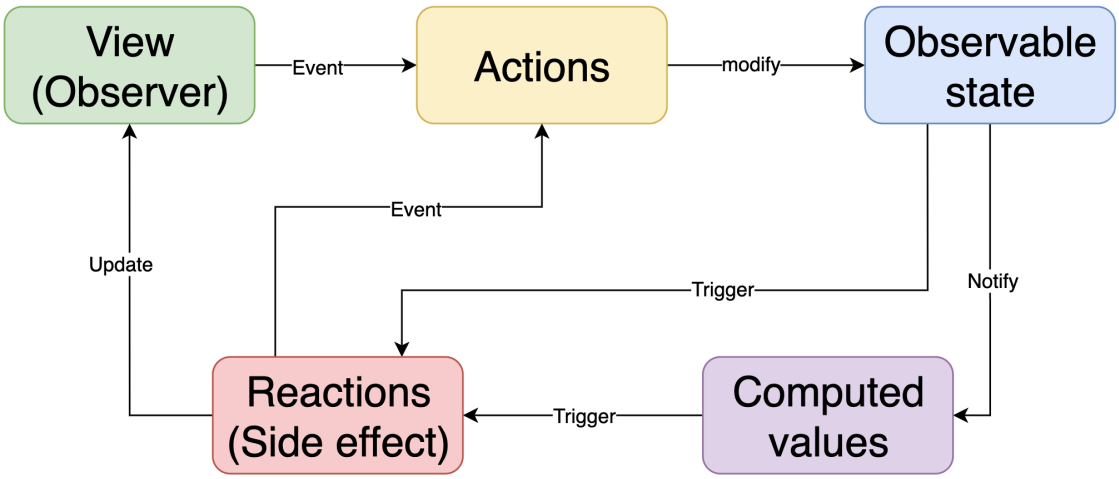


Figure 3.9 *Observer pattern [10]*

The observer pattern is used when there is a one-to-many relationship between states of an application such as if one state is modified, its dependent components are to be notified automatically. The Observer pattern is functionally reactive in nature and mutating an observable state is automatically detected by the observers of that observable. When a state data is registered as an observable (or provider), subscribers to that observable (also known as observers or sometimes consumers) get automatically notified when the state data is modified or whenever a predefined condition is met [40]. Observables or Providers are usually application state data that can change whereas observers could be data or functions that depend on the provider or observable. Observer patterns are usually un-opinionated and can be implemented using other patterns as well. In Figure 3.9, the components of the observer pattern include [10]:

- **View**: the view is the application UI, reflecting the application's state. The view is an observer to all the observables used to render the View. That way, the view gets updated by re-rendering when the observables are modified.

- **Actions**: Events that occur during user interaction with the view cause actions to be emitted. Actions are functions that modify observable states and are usually grouped for a specific event.
- **Computed values**: Computed values as the name suggests are processed observable data and obtained after a function (usually a pure function) is executed on the observable data to return new or derived data. These are observers as well as observables. They depend on the observables but other observers can also depend on them.
- **Reactions**: Reactions are side-effect functions that are executed when subscribed observables to meet a certain condition. Reactions are also types of observers. The side effect could be a network call or updating the view.

3.6 State management tools

Besides the internal solutions for state management provided by various frontend development frameworks such as React, Vue, Angular, etc., there are other external standalone solutions which can be integrated into these development frameworks. These external solutions are solely focused on state management and not only simplify the process of managing the state of an application but also aim at efficiently updating the changes to the state.

Some of these state management solutions are introduced based on their popularity, support, maturity, size, compatibility and features offered by the frameworks or libraries.

3.6.1 Redux

With over 7 million weekly downloads, Redux is one of the most popular state management solutions for frontend web applications. Redux is known as a predictable state container for JavaScript applications [34]. It implements an opinionated approach to state management making code more readable and easier to manage especially for large applications. Although Redux can be integrated with most frontend and even some backend development frameworks, it is mostly used by React developers. The core features of the Redux state management tool include:

- Predictability: Redux aims at ensuring that the application behaves the same irrespective of the environment. This ensures that the developer can trust the test result to be consistent with what the user gets.
- Centralized: State management easily gets complicated as the application grows. To solve this problem, Redux implements a centralised state management approach so that developers can see how the state changes are handled

in one place. This approach reduces the complexity of state management and improves the time to fix bugs.

- Debuggable: Redux's powerful debugger makes tracking where, when and how the state changes a breeze. This is very useful when it comes to finding and fixing bugs in an application. The debugger comes with the functionality of under/redo state change which helps the developer to study the state system more efficiently.

- Flexible: Redux was designed as a standalone state management solution. It can be used to manage the states of various frontend development frameworks such as React, Angular, Vue, etc. In addition, various add-ons exist which can be integrated into Redux thereby increasing its capability as well as tuning it to the needs of the application.

3.6.2 Zustand

One of the fast-growing state management tools that are similar to Redux is Zustand which is a small, fast and scalable barebones state-management solution [41]. It proposes an opinionated solution to state management which is easy to use with only a few lines of code when compared to Redux. In addition, Zustand can be highly customized, making the library usage very flexible and giving the developers the freedom to choose how to organise the state management structure. Zustand integrates well with most Javascript frameworks like react, Vue, Angular and VanillaJs as the store is managed externally from the development framework.

At the core of Zustand is the *create* function which is used to create an instance of a store. The *create* function also provides a *set* function that can be used to update the store. Multiple stores can be created for an application, giving the developer the possibility of modularising the stores to serve different purposes. The store is created as a react hook after which it can then be used within the components of the application as needed. Whenever the store data is updated, any component of the application using the store data is also updated. Unlike Redux which requires a *Provider* to wrap the section of the application where the store is accessible, Zustand does not need a *Provider* making it much easier to integrate and decoupled from an application. A simple use case of Zustand can be seen in Figure 3.10.

3.6.3 Vuex

Although every component of Vue is capable of managing its state, there are cases where this solution is not optimal. Managing the states of medium to large applications with shared states by relying on component-based state management solutions

```
import { create } from 'zustand';  3.2k (gzipped: 1.4k)

// Create Store
const useStore = create((set) => ({
  bears: 0,
  increasePopulation: () => set((state) => ({ bears: state.bears + 1 })),
  removeAllBears: () => set({ bears: 0 }),
}));

// Use Strore in components
function BearCounter() {
  // Access store property => Read property
  const bears = useStore((state) => state.bears);
  return <h1>{bears} around here...</h1>;
}

function Controls() {
    // Access store property => Set property
  const increasePopulation = useStore((state) => state.increasePopulation);
  return <button onClick={increasePopulation}>one up</button>;
}

function App() {
  return (
    <div>
      <BearCounter />
      <Controls />
    </div>
  );
}
```

Figure 3.10 *Using Zustand in React [41]*

easily gets complicated and hard to debug [38]. For this purpose, the Vue team developed a more robust, scalable, stronger conventional for team collaboration, centralised and long-term productivity state management solution called Vuex. Vuex offers a more intuitive and efficient approach to managing and sharing states between components of Vue applications. Vuex offers an excellent debugger tool which is capable of time-travel debugging. Pinia was developed as a replacement for Vuex and the recommended state management solution for Vue applications [37]. Pinia is considered super light with only a size of about 1kb [42]. Pinia is designed to be modular. This means that there can be multiple stores in one application which are independent and can be dedicated to handling specific types of states.

3.6.4 NgRx

NgRx is a reactive state management solution for Angular applications inspired by Redux. NgRx is one of the most popular state management solutions in the angular community. It contains a set of reactive-oriented modules for Angular [43]. Utilising a reactive programming paradigm using RxJs at its core, NgRx aims to provide common features for the Angular platform.

NgRx consists of several components that work together to solve the state management needs of an Angular application. These components include [44]:

- *@ngrx/store*: Store is a container where the global state of the application is managed. The store is a controlled state container. It is designed with the performance, consistency and predictability of angular applications in mind.
- *@ngrx/effect*: The effects provide a means to handle side effects. Effects handle events such as network requests or other asynchronous or synchronous actions that return new actions to the store.
- *@ngrx/router-store*: This helps the store to be away from the route state. It binds the store with the route and can dispatch multiple actions based on the route state.
- *@ngrx/store-devtool*: This provides the debug functionality for the state management of an angular application. Helps developers view and track changes to the state of the application.

3.6.5 MobX

MobX is a simple and scalable state management library that is implemented using reactive functional programming [45]. MobX is un-opinionated and allows the developer to manage the state of the application outside of the UI frameworks which allow for separation of concern [45]. MobX can be integrated with any UI framework and can be easily adapted to any architecture. Being minimalist and boilerplate-free, MobX can fit into any project size from small to enterprise-level applications.

At the heart of MobX are the **observables** which serve as the building block or unit of a reactive state. MobX can track all changes and uses of data registered as observable at runtime. Another important component of MobX is the **observer**. This tells MobX which application component is interested in an observable or multiple observables. The observers are notified of changes to the observable they are interested in. Because different components might be interested in an observable in different ways, MobX provides different functions for the components to express the type of dependency they have on an observable. This include *autorun*, *reaction*

and *when* [10]. *autorun* API takes in a function called an effect that is executed immediately after the application is started. Whenever any of the observables used in this function change, the function is executed again. *reaction* is similar to the *autorun* but accepts two arguments, a track function and an effect function. The track function is where the observable that is of interest is specified whereas the effect function is a function that is executed if the observables change. Furthermore, the *reaction* is executed when change occurs to the observable and not at the start of the application. *when* observer is quite similar to *reaction* API but this time, the first argument expects a Boolean value. When the Boolean resolves to true, the effect function is executed.

3.6.6 Recoil

Recoil is a state management solution developed by Facebook. Recoil has a lot of similarities with Mobx. For example, in Recoil atoms replace observable as reactive data. Recoil was developed with React in mind and in some way mimics the *useState* react hook. To subscribe to atoms, Recoil provides the *useRecoilState* and *useRecoilValue*. The *useRecoilValue* is used to subscribe to a computed value which could be through a synchronous or asynchronous operation. Recoil was developed to optimize and simplify sharing of state values between components in multiple layers of a react application without re-rendering unnecessary components when the state values change [46].

3.6.7 TanStack Query

Unlike most state management tools introduced above that are focused on managing the internal state data used by the application, TanStack Query is more focused on managing state data from external sources. TanStack Query consists of four (4) libraries namely React query, Solid query, Vue query and Svelte query as it aims to support various development frameworks. Currently, React query is the most mature and popular of the four. React query is currently in version four (4) with over one (1) million weekly downloads from npm.

As with React, some modern development frameworks lack well-defined inbuilt support for managing state data from external sources [47]. React query presents an efficient state management solution for data that come from sources such as a server, database or other sources. React query is opinionated and intuitive. Some of the features of React query include [47]:

- React query is able to persist data to prevent making the same request multiple times when the page is refreshed or as long as the data is needed. In addition,

react query is able to update outdated data to always keep the application with the latest data.

- React query can improve the performance of an application through its caching techniques. Data that is requested from different parts of the application can be cached and supplied from the cache than spending more time reaching the external source. This improves the response time and performance of an application.
- React query can smartly determine and request only the data that is needed by the current page which is also known as lazy loading. This greatly reduces the load time and improves the user experience of the application. This technique is useful for pagination.
- React query can efficiently manage the memory used to cache data. Knowing when to free up the memory and when to cache data that is frequently requested by the application.
- React query's opinionated approach simplifies the management of external state data.

3.6.8 SWR

State While Revalidate (SWR) is a lightweight alternative library to React query. SWR works in such a way that when a request is made, SWR provides the existing or previous data for that request meanwhile making the new request in the background [48]. When the response is got, it updates the state of the application with the new data. SWR is able to cache data just like React query but lacks a lot of the extra functionality that comes with React query such as selectors, render batching, auto garbage collection, offline mutation, query cancellation, stale time configuration, etc. [49].

3.6.9 Apollo Client

Apollo client was developed to support architectures that use GraphQL protocol to communicate with external data sources. Although react-query can be configured to work with any asynchronous function including GraphQL and REST protocols, Apollo client was designed to be much more suitable for GraphQL data endpoints. In addition to remote state management, Apollo client is also able to manage and store the local states of a frontend web application in its cache. This makes it possible to request a combination of both client and remote data through one call. This is more efficient than making multiple requests. Having a simplistic declarative

data fetching style, the developer only needs to describe the data they need and Apollo client handles the optimization and caching behind the scene. Besides the queries and mutation API which does the network calls, Apollo client provides a subscription API which maintains the connection to a server and is useful for real-time communication. Furthermore, Apollo client is able to handle errors that occur as a result of executing GraphQL operations. It is important to note that to get the best out of Apollo client, the server is recommended to use Apollo server, which handles the server-side operations.

3.7 Functionality Analysis of Selected Tools

When the tools for managing the state of a web frontend application have been selected, the next step is to properly utilize the resources in managing the state of the application.

In order to take full advantage of the resources available for state management, the developer needs to fully understand and analyse the selected tools in such as way as to provide the expected functionalities as specified in the functional requirements as well as meet up with the non-functional requirements of the application. To achieve this goal, the developer needs to have in-depth knowledge of how and when to use the functionalities offered by the resources. A guide to analyzing the selected tools includes:

- grouping the solutions (i.e., functions or methods) offered by the selected state management tools to the class of state (e.g., global state, local state, etc.) they can manage,
- comparing the advantages and disadvantages of using each solution,
- selecting the most suitable solution for the app need from each group.

Most state management tools provide different methods to manage specific types of states. Being able to understand how to properly utilize these methods benefits the application.

3.8 Conclusion

A clear understanding of the requirements for an application as well as good planning, analysis and consideration before selecting the tools for managing the state of a frontend application will set the application development on a strong foundation. This can be gotten by answering the questions which influence the state management of an application as specified at the beginning of this Chapter 3. As a result

Table 3.2 *State management tools comparison.*

Package name	**Architecture or pattern**	**Framework compatibility**	**downloads (npm/week)**	**State management**	**Size (KBs)**
Redux	Redux, Flux	most	9.3 mil.	client	1.6
MobX	Observer	most	0.9 mil.	client	16.3
Vuex	Vuex	Vue	1.7 mil.	client	6.5
Pinia	Vuex	Vue	0.3 mil.	client	7.4
Recoil	Observer	React	0.3 mil.	client	23
NgRx	Observer, Redux	Angular	0.5 mil.	client	5.5
SWR	Flux	React	1.0 mil.	server	4.2
React-query	Flux	React	1.5 mil.	server	13
RxJs	Observer	most	39.5 mil.	client	17.6
Akita	Akita	most	0.04 mil.	client	13.3
NgXs	NgXs	Angular	0.1 mil.	client	5.2
Apollo Client	Flux	most	2.3 mil	client, server	40
RTK	Redux	most	1.7 mil.	client, server	12.8
Zustand	flux	most	0.6 mil.	client	1.17
Elf	Observer	most	7k	client, server	2
Jotai	Observer	React	0.2 mil.	client	3.4

benefits such as performance, maintainability, scalability, security, time and efficient resources management are only a few of which come with proper assessment and selection of tools for managing the state of a frontend application.

A thorough analysis of how and when to use the selected tools ensures that the implementation makes the best use of the resources which as a result benefits the application.

The strategy tries to minimize the number of unattended requirements of the application by maximizing the utilization of available resources

Based on popular solutions that exist, Table 3.2 is compiled containing useful information that can serve as a guideline for choosing tools for state management.

4 Case study

As a case study, the state management of Nokia Test Automation Platform (NTAP), a test automation tool developed at Nokia is analyzed, redesigned, and re-implemented.

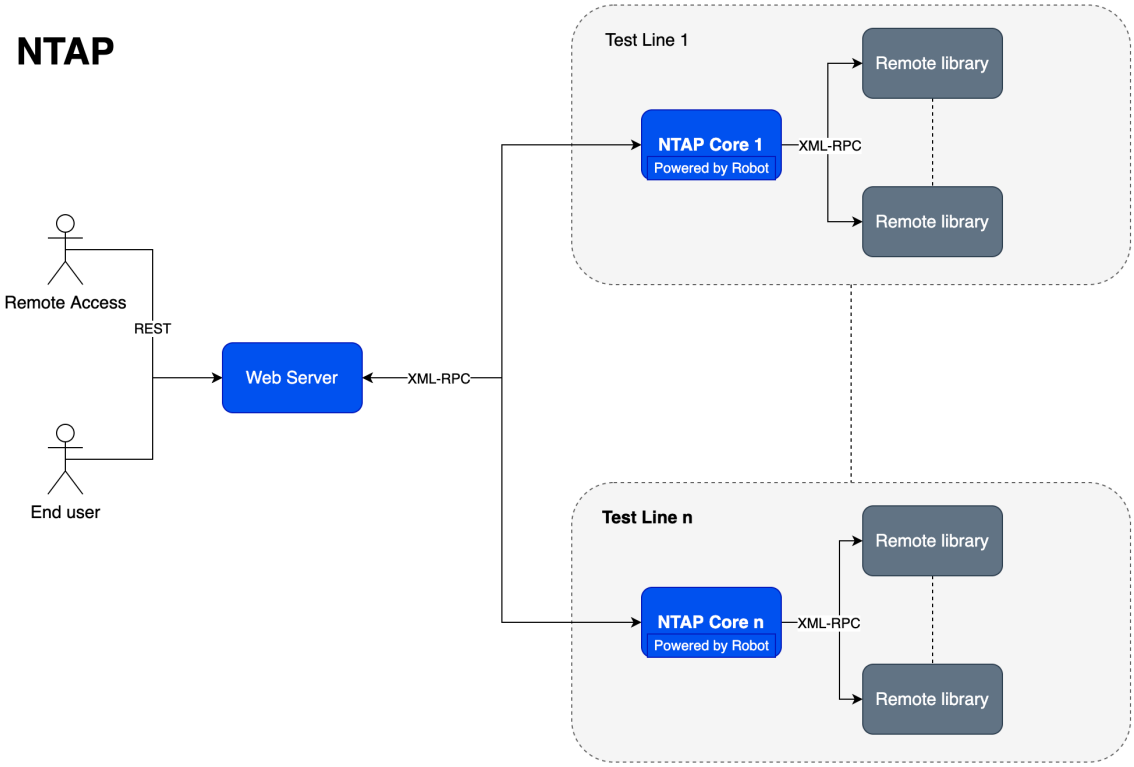


Figure 4.1 *NTAP Architecture*

NTAP is developed on top of Robot Framework (a popular automation tool) to simplify and automate test case execution. NTAP serves as a user-friendly platform that is highly customisable to work with different software and hardware based on user preference. NTAP has a centralized control panel used to create, edit, analyze, and report robot test cases in an intuitive manner, and remote test lines where the test cases are executed as shown in Figure 4.1.

Currently, Nokia plans to productize NTAP for commercial use. The current state of NTAP requires some improvements before its release to meet company standards after some analyses were conducted. These improvements include state management, security hardening, containerization, etc.

The primary means of interacting with NTAP is through a web application GUI known as NTAP Web Server. However, NTAP also exposes various API endpoints for integration with other tools and applications through which it can communicate or be controlled. The case study work focuses on NTAP Web Server state

management improvements.

To improve the state management system of the NTAP Web Server, it is necessary to go through the following steps:

1. Study and analyse the current state of NTAP Web Server.
2. Identify issues with the current implementation.
3. Redesign the state management architecture of NTAP Web Server.
4. Choose tools for state management improvement.
5. Analyse the state of NTAP Web Server and resources available.
6. Utilize the available resource properly and efficiently.

4.1 NTAP Web Server

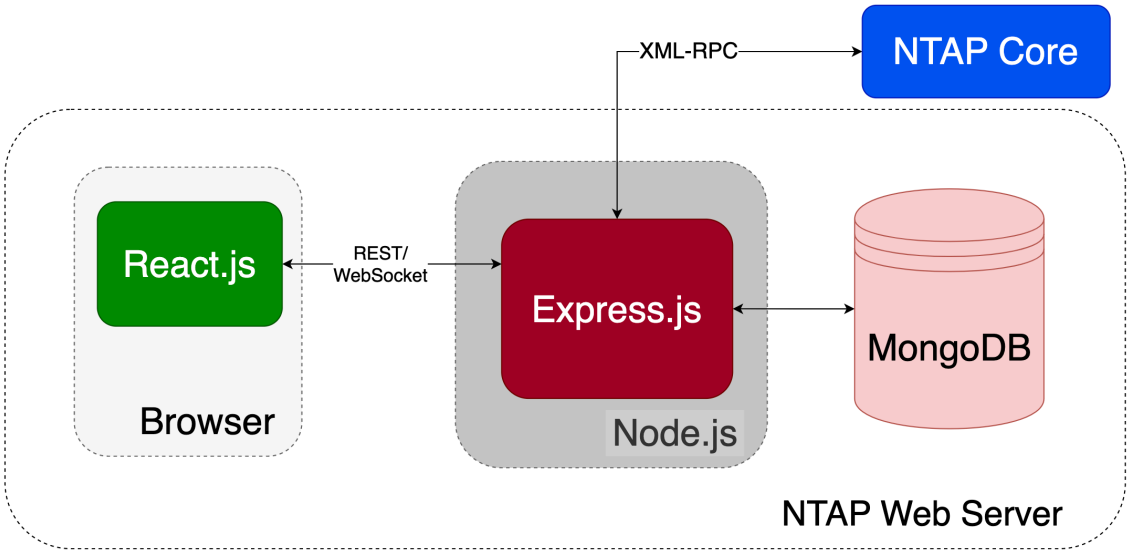


Figure 4.2 *NTAP Web Server Architecture*

NTAP Web Server consists of 3 main components as shown in Figure 4.2:

1. React.js app - the client side which runs on a web browser
2. Express.js app running on Node environment which acts as the server communicating with NTAP core, database and the web GUI.
3. Mongo DB - the persistent data storage

React.js is used to develop the client side of the NTAP Web Server. React.js has already been introduced in this thesis (see 3.4.1). The React.js application runs on a browser and communicates with the server using either RESTful APIs or WebSocket. The Server consists of an Express.js application running on a Node.js environment and a MongoDB database.

Express.js is a minimalistic, lightweight, flexible, and highly customizable Node.js web application framework that can be used to develop server applications. It is unopinionated, giving the developer the freedom to structure the application as they like. Node.js is a cross-platform, open-source, runtime environment for JavaScript and is designed for building scalable network applications [50]. Operating on the V8 engine, it is able to execute JavaScript code outside of a web browser [50].

The state management improvement is focused on the Web Server GUI which is a React.js application. Currently, the states of NTAP Web Server GUI are managed using React internal state management solutions and JumpState state which depends on Redux state management tool.

JumpState is a Redux state management utility that is lightweight at 2kb and minimalistic to use. Unlike using pure Redux, JumpState provides a means of using Redux without the verbose need of declaring action creators, action constants and dispatchers. With few lines of code, JumpState can be easily integrated into a React.js application. It is readable and easy to learn and teach thereby making it easy to collaborate and maintain the codebase.

Since NTAP Web Server is a very large application with hundreds (100) of state values, the case study will be focusing on the Execution or Main section of the application as shown in Figure 4.3.

4.1.1 Current Implementation

Based on the current implementation, the functionalities of the Main section as shown in Figure 4.3 includes:

- create, select, modify and delete workspace
- setup, modify, delete and configure the system under test
- setup, modify, delete and configure the automation environment
- setup, modify, delete and configure test sets
- starts and stops the test execution based on the selected configurations and views the progress.

Furthermore, the following observations were found:

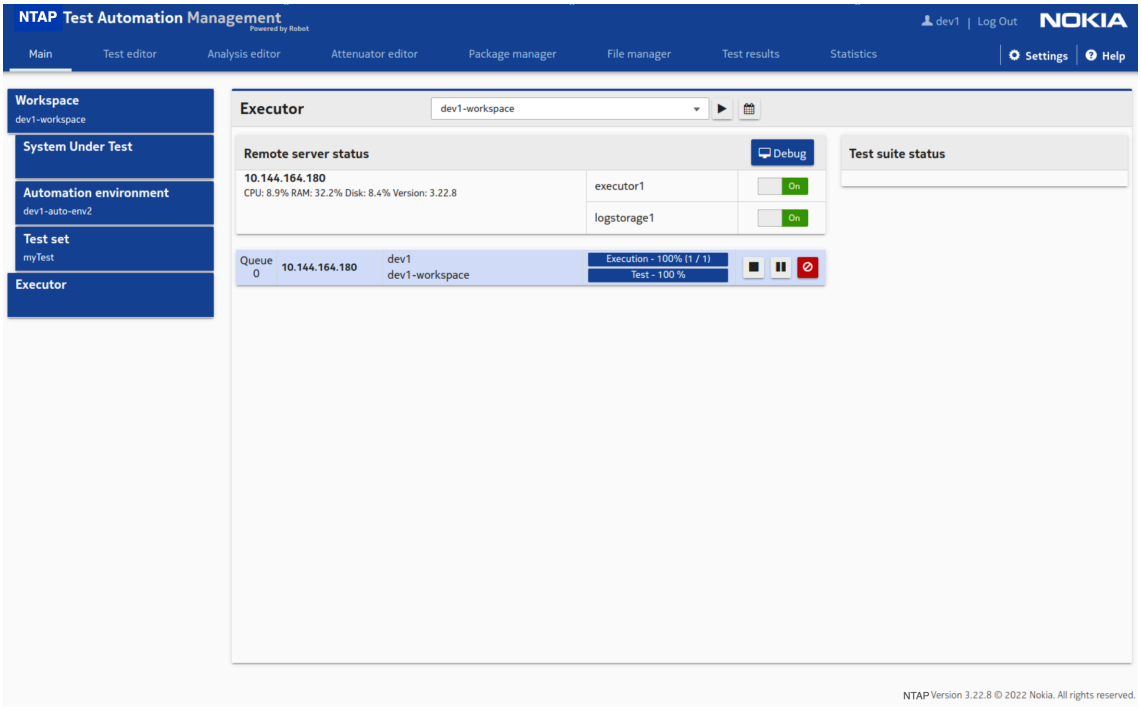


Figure 4.3 *NTAP Web Server Main Section*

1. Main Content tab contains 4 direct children components (Workspace, System under test, Automation environment, Test set and Executor components) and 34 sub-components in total, with the deepest component nine (9) generations away.

2. Workspace component depends on 17 state values from the global store and 4 local state values

3. Automation environment, System Under Test component, and Test Set components also depend on 18 global state values and have 4 local state values

4. Most of the states are managed as a global or local state.

5. Requires 12 network calls to load all components

For analysis purpose NTAP web server is accessed on a Chrome browser (version 106.0.5249) running on a Linux Ubuntu 22.04 machine with an Intel® Core™ i5-10310U CPU @ 1.70GHz × 8 Processor, 32 Gigabyte RAM, and a 512 Gigabyte Disk Capacity. The Chrome DevTools Performance monitor is used to collect performance-related statistics such as rendering time, re-rendering count, etc. on loading the Main Page of NTAP web server and then the execution tab as shown in Table 4.1.

The application makes a total of 45 network requests while taking up 17,6 Megabytes of the browser's resource space.

Table 4.1 *NTAP Main Section Statictics*

Component name	**Total Render time (ms)**	**Total Render time (%)**	**Render Count**
MainContent	86,5	42,5	14
Workspace	25,6	12,6	14
System Under Test	18,1	13	11
Auto. Environment	18,6	13,6	11
Test set	12,5	9	11
Execution	11,7	5,8	14

4.1.2 Issues

From the observations made on the current implementation, the following issues of concern both to the state management and productization of NTAP were found:

1. JumpState library used for state management is deprecated
2. Components re-render multiple times as shown in Table 4.1. This affects the performance of the application in a negative way.
3. The server state is poorly managed with no caching technique implemented nor any server state management tool used resulting in repeated network calls.
4. Code structure was not well organized and therefore hard to maintain. Most of the states of the application are managed in one file, resulting in a huge file which is difficult to read or maintain as shown in 4.4.
5. The load time, as well as the time to set up an execution, is quite long at about ten (10) seconds.

4.1.3 Re-implementation strategy

To improve the state management system of NTAP web server, the proposed strategy as specified in Chapter 3 is used as a guideline.

Software requirements

As NTAP is not just going to be an internal tool for Nokia but will be productized and used by external companies, the security and performance of the tool will be of high importance. Furthermore, the memory size and resources used by the tool should be minimal. This implies that well-trusted state management tools with a

```
import { State } from 'jumpstate';

export const mainPageState = State({
  initial: {
    user: {},
    loading: false,
    selectedView: '',

    groups: [],
    testSuites: [],
    parameters: [],
    elements: [],

    selectedParameterSet: {},

    workspaceItems: [],
    parameterValues: [],
    elementValues: [],

    testSuiteSet: {},

    workspaces: [],
    workspace: {},
    selectedWorkspace: '',
    workspaceComponentsModified: false,

    openNodes: [],
    viewTouched: {},
    validForm: true,

    users: [],
    userGroups: [],

    saveLastVisitedPage: false,
    showExportDialog: {},
    showImportDialog: {},
    execution: {
      signals: []
    },
    apiCallTimeout: 30000
  },
  // ...
})
```

Figure 4.4 *NTAP JumpState store*

good security reputation, highly performant and of a considerable size would be a suitable fit for the application. The functional requirements of NTAP web server are specified in Section 4.1.1. The state management systems should make it easy and intuitive to create, modify and delete a workspace, test case, test suite, test automation environment and system under test. Also, the execution of tests is according to the configuration of the user.

Architectural design

The architecture of NTAP as seen in Figure 4.1 shows that NTAP web server GUI requires communication through RESTful APIs to a server where data is processed, retrieved and stored. The existence of an external source of data or communication can be seen as a dependency on the frontend side of the software. This implies that

the state management system is required to efficiently handle such external state data which is regarded as a server state.

State classification

From the requirements and architecture of NTAP, the existence of workspaces signifies that a global state will be needed to properly group and identify data. Also, having in mind that data is stored and accessed from a remote database suggests that the server state will be involved in the state of the application.

Furthermore, it is expected that NTAP web server will have many local state data which will be responsible to handle an elegant UI for creating, deleting, and editing data as well as other configurations in NTAP. In addition to the local state, a computed or derived state will be needed to extract data from the global state such as extracting the Automation environment data from a Workspace. As a result of this, the selected state management system must be well capable of handling server, global, derived, and local state data in an efficient way.

Development framework

NTAP is developed using React.js as the development framework. Therefore, the state management system should be easily integrable with a React.js application. Also, the state management tool should be capable of working with React.js efficiently and optimally. The version of NTAP that will be shipped out will be developed using Typescript. Typescript is a superscript of JavaScript. It is regarded as a typed version of JavaScript. The state management system needs to support Typescript.

State management approach

NTAP web server will need a state management pattern that will be able to cope with a medium to large application. Also, it should be easy to read, understand and maintain. The flux-like pattern is adopted for the state management architecture. The pattern is highly supported by React.js. Due to its predictability, scalability as well as readability, the flux pattern is highly beneficial for medium to large applications. A large store for managing the global state will be needed on a page basis. This makes the source of data easy to locate and identify. For local states, a decentralised store in the form of state hooks is used. Hooks in React.js is a concept which involves decoupling parts of an application as stand-alone reusable functions. Hooks are independent of the application and as a result, can be tested in an isolated environment.

State management tools selection

As a result of the considerations made on selecting the tools for the state management system of NTAP web server, **React-query** is chosen for the server state management whereas **Zustand**, as well as React.js internal state management solution, are used for the client state management. The selected tools are widely adopted state management tools with no found vulnerability issues that could affect NTAP web server. Also, the selected tools support the implementation of the flux pattern which brings additional benefits such as readability, maintainability, scalability, etc.

State management tools usage

React-query is used because of its advanced caching and updating mechanism which is efficient and suitable for managing the server state needs of NTAP. SWR would have been an alternative to React-query but lacks some important features such as selectors, query cancellation and mutation hooks important to the server state need of NTAP web server [49].

Zustand is used because of its un-opinionated approach to state management. This makes it flexible to adopt a global store which can be more verbose and readable as well as a smaller store in form of hooks which can be used to manage local states in a reusable manner. Although Redux could serve as an alternative to Zustand, Zustand on the other hand has the advantage of being lightweight in size and does not require the whole application to be wrapped by a Provider but can be nicely integrated only into the component where it is needed.

Furthermore, React.js internal state management is used to manage some micro local states, i.e., when a component requires three (3) or fewer local state values. This is to reduce the verbosity of creating hooks for every state as well as maintain code readability.

4.1.4 Re-implementation

Based on the improvement strategy, the new implementation design is represented in Figure 4.5. The re-implementation design aims to reduce redundancy and duplication of state by bringing the state data closer to components where they are used as well as providing components with exactly what they need. Furthermore, the new implementation carefully utilizes the best methods offered by the state management tools to handle each state's data.

React-query now manages all the server states. This means that for a new network request to be made, the previously available data has to somehow be invalidated regardless of where and when it was made otherwise the previously retrieved

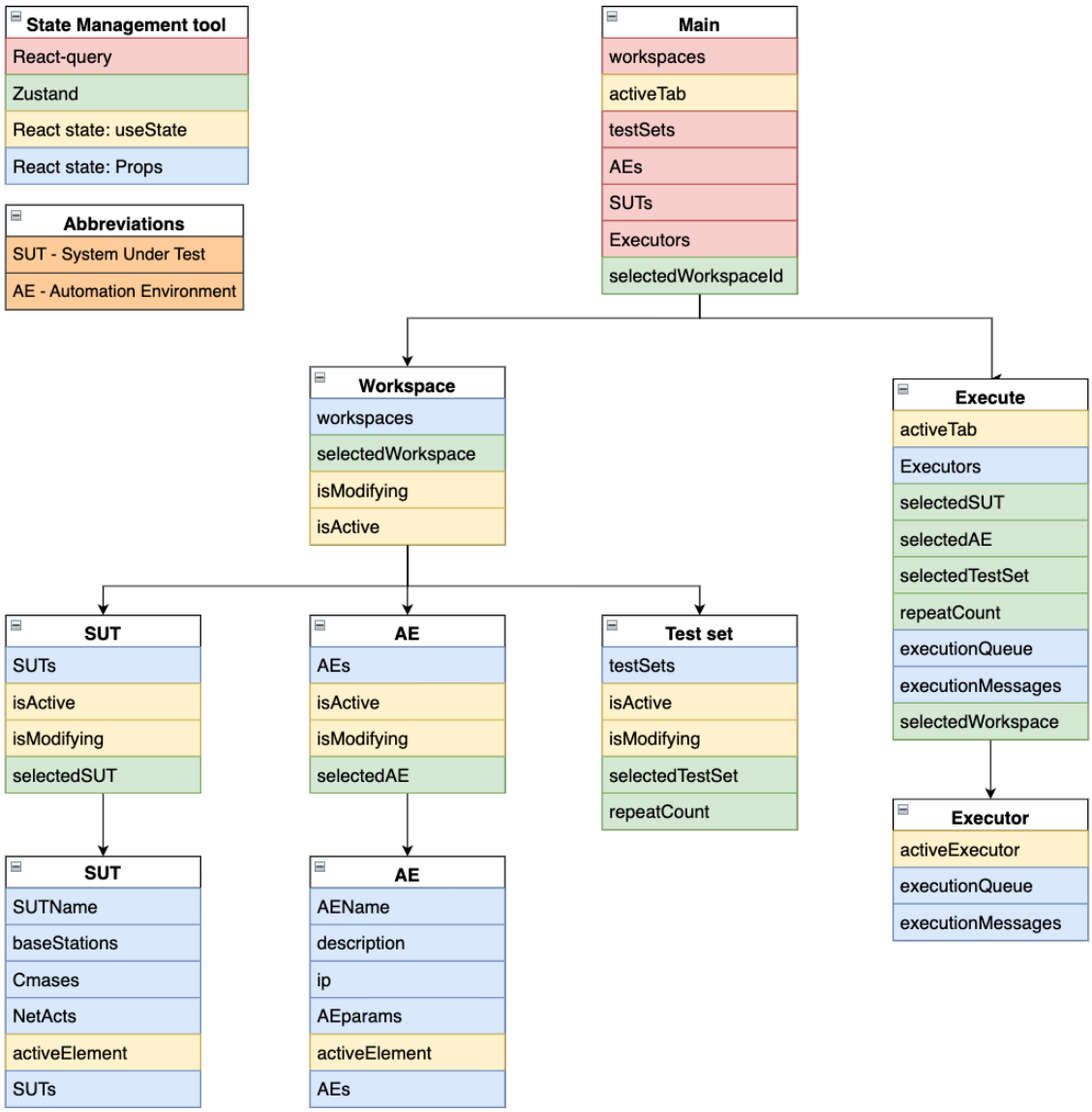


Figure 4.5 *NTAP Web Server Main Section State Management architecture*

data is used. This saves a lot of time to reach the actual server and performance caused by re-rendering the related components and their children.

Zustand is used to manage client state that is used across multiple sibling components. This makes it easier to manage the states in one place whereas the state data can be provided only to the parts of the application where they are needed. React props are used to provide closely related components (e.g., children components) with derived or computed state values from their parent component. As these props are monitored by React.js and their values are cached, the components only re-render when the values of the props change. This saves some performance for the application as values used to render these components are not re-calculated.

Taking the workspace as an example, data related to the workspace is extracted

```
2  import { useMutation, useQuery, useQueryClient } from 'react-query';  18.8k (gzipped: 5.3k)
3  import Create from 'zustand';  2.6k (gzipped: 1.2k)
4  import { persist } from 'zustand/middleware';  2.5k (gzipped: 1.1k)
5  import { deleteWorkspaces, getWorkspaces, postWorkspaces, putWorkspaces } from '../services';
6
7  // Workspace Client State Hook with Zustand
8  export const useWorkspaceClient = Create<{
9    selectedWorkspaceId: string;
10   setSelectedWorkspaceId: Function;
11 }>()(
12   persist(
13     (get) => ({
14       selectedWorkspaceId: '',
15       setSelectedWorkspaceId: (payload) => set({ selectedWorkspaceId: payload }),
16     }),
17     { name: 'selectedWorkspaceId' },
18   ),
19 );
20
21 // Workspace Server State Hook with React Query
22 export const useWorkspaceQuery = () => {
23   const queryClient = useQueryClient();
24
25   // Get all workspaces
26   const workspacesGetQuery = useQuery('workspaces', getWorkspaces);
27
28   // Create new workspace
29   const workspacesCreateQuery = useMutation(postWorkspaces, {
30     onSuccess: (resData) => {
31       const workspaces = queryClient.getQueryData('workspaces');
32       queryClient.setQueryData('workspaces', [...workspaces, resData]);
33     },
34   });
35
36   // Delete a workspace
37   const workspacesDeleteQuery = useMutation(deleteWorkspaces, {
38     onSuccess: (resData) => {
39       const workspaces = queryClient.getQueryData('workspaces');
40       queryClient.setQueryData(
41         'workspaces',
42         workspaces.filter((item) => item.id !== resData.id),
43       );
44     },
45   });
46
47   // Update a workspace
48   const workspacesUpdateQuery = useMutation(putWorkspaces, {
49     onSuccess: (resData) => {
50       const workspaces = queryClient.getQueryData('workspaces');
51       queryClient.setQueryData(
52         'workspaces',
53         workspaces.map((item) => (item.id === resData.id ? resData : item)),
54       );
55     },
56   });
57   return {
58     workspacesGetQuery, workspacesCreateQuery, workspacesDeleteQuery, workspacesUpdateQuery,
59   };
60 };
```

Figure 4.6 *NTAP Web server workspace state hook*

into its own hook. This can be seen in Fig. 4.6. The modularity of this structure improves reusability, testability, and code maintainability. Zustand manages the *selectedWorkspaceId* as can be seen in lines 8 - 16 of Fig. 4.6. Since *selectedWorkspaceId* is a client-only needed information and also used by several components, it is therefore most suitable to be managed by Zustand. React *useState* is not used as it would then require a context to wrap the application or pass it as props which are more difficult to maintain. For the server state of the workspace, React query is used for handling the data related to fetching, creating, updating, and deleting workspaces as can be seen from lines 22 - 60 of Fig. 4.6. The *useWorkspaceQuery* hooks make it

easy to access and modify the server data of the workspace as well as ensure that any action made on the workspace is reflected on other parts of the application making use of the workspace data. Furthermore, whenever any section of the application tries to make a fetch request for the workspace data using the hook, React query returns the already cached data. This ensures that the fetch request is made only once to the server, thereafter, React query responds to subsequent requests with the existing data saving time and performance for the application.

```
2   import shallow from 'zustand/shallow';  544 (gzipped: 316)
3   import { useWorkspaceClient, useWorkspaceQuery } from '../stores/useWorkspace';
4
5   const Workspace = () => {
6     // Get the worksapce query
7     const { workspacesGetQuery } = useWorkspaceQuery();
8     // Get workspace data from query
9     const { isLoading, isError, data: workspaces, error } = workspacesGetQuery;
10    // Get selected workspace ID from Zustand store (saved in localstorage)
11    const [selectedWorkspaceId, setSelectedWorkspaceId] = useWorkspaceClient(
12      (state) => [state.selectedWorkspaceId, state.setSelectedWorkspaceId],
13      shallow,
14    );
15
16    return (
17      <select
18        className="custom-select"
19        id="workspace-select-dropdown"
20        onChange={(e) => setSelectedWorkspaceId(e.target.value)}
21      >
22        <option disabled hidden>
23          Select a workspace
24        </option>
25        {workspaces.map((workspace) => (
26          <option value={workspace.id} selected={workspace.id === selectedWorkspaceId}>
27            {workspace.name}
28          </option>
29        ))}
30      </select>
31    );
32  };
33
34  export default Workspace;
```

Figure 4.7 *Workspace component*

Figure 4.7 shows how to access both the data managed by Zustand (i.e., line 11) and those handled by React query (i.e., line 7, 9). In the same way, any component that needs any of this data can easily import the hook. Any modification made to the data stored in these stores is reflected efficiently in other components using the data.

4.1.5 Re-implementation result

To check the impact of the improvement on NTAP web server, statistics of a similar procedure were taken as in Table 4.1, but this time on the newly re-implemented version. The results of the test are shown in Table 4.2. In addition, the number of network requests was reduced from forty-five (45) to thirty-three (33) on accessing

the execution tab of the Main page. Whereas the application now occupies approximately 15,8 Megabytes of the browser's resources memory space compared to the 17,6 Megabytes it used up previously.

Table 4.2 *Re-implement NTAP Main Section Statictics*

Component name	**Total Render time (ms)**		**Total Render time (%)**		**Render Count**	
	before	**after**	**before**	**after**	**before**	**after**
MainContent	86,5	52,6	42,5	27,9	14	9
Workspace	25,6	19,1	12,6	10,1	14	9
System Under Test	18,1	13,3	13	7,1	11	6
Auto. Environment	18,6	12,9	13,6	6,8	11	6
Test set	12,5	9,5	9	5,7	11	6
Execution	11,7	8,5	5,8	4,5	14	9

5 Discussion

The state management system in an application plays a huge role in the performance, speed and resource requirements such as the memory size and processing power of the application. It would seem easy to use the internal state management solutions provided by the development framework. However, the consequences of an unplanned approach to state management would have a negative impact on the application as it expands. In some cases, this could result in the application suffering a lot of performance issues, thereby, taking up a lot of processing power of the client's device. Take, for example, an application using the same server state data in different components or parts of the application and there is no means of caching these values or distributing the data to the components that need it. This will require each component that uses this data to make a separate request to the server for the data. If this data needs some processing it would have to be recomputed multiple times. This example already shows some performance issues as a result of duplicate operations which with a good strategy to state management can be mitigated. Furthermore, the components in the example will take some time to load which will worsen the user experience. However, having an application that is lightweight, fast, and does not require a lot of processing power will play a great role in improving the user experience. Also, the lesser the resource requirements such as memory space or processing power needed by the application, the likelier it is to be compatible with more devices, thus reaching more users.

From the case study, if we compare the result before the state management implementation, i.e., Table 4.1, to the result collected after the implementation as seen in Table 4.2, it can be seen that the components re-rendering decreased about five (5) times. This can be attributed to the use of Zustand for distributing the client states in such a way that only components needing a state get access to the store. Also, components update only when the state data used by these components change. This is in contrast to Redux's Provider which was used in the previous implementation to wrap the whole application. The provider can sometimes update the components inefficiently when the store data changes regardless of whether the state data used by the components changed or not. The problem caused by this inefficiency leads to unnecessary components re-rendering, and the data used to render the components are re-evaluated or sometimes even recreated from scratch leading to performance issues. So replacing JumpState with Zustand alone resulted in a high improvement in efficiency and performance. Considering the MainComponent which depends on some server states such as data for the workspace, test set, etc., if the component re-renders fourteen (14) times and the server state is not cached as

in the previous implementation, the MainComponent will have to reach the server fourteen (14) times to get the state data. This sometimes leads to noticeable delays in the rendering time making the application act slower for the users. However, in the re-implementation, as the server state is cached, the application only needs to fetch data from the server once saving a lot of time and performance for the application as the cached data is reused on subsequent re-rendering. The choice of tools as well as the architecture for managing the state pay off which can be seen in the result of the new implementation from Table 4.2.

It is important to highlight the security impact of the new state management system. Prior to the re-implementation, NTAP used an outdated state management library called Jumpstate. Jumpstate is no longer maintained, meaning that if any vulnerability is found with the library, it will not be fixed and therefore exposes applications such as NTAP using the library to serious security issues. The replacement of the JumpState with React Query and Zustand which are both actively maintained with no known vulnerability proves beneficial to the security of the application. If we consider the large community that watches, supports and uses these packages, then the likelihood of unknown vulnerability is slim and the speed at which any found security threat will be fixed is high. This increases the reliability of the new state management system and as a result, the security of NTAP is further hardened.

As a result of using the proposed state management strategy in Section 4.1.3, every decision made concerning the state management of NTAP is justified. This results in NTAP web server performing more efficiently. In addition, the management setup can be easily maintained by the team, the state management system is more decoupled from the application and can be easily replaced, and the state management system is more scalable as a result of the decoupling approach. Based on the analysis conducted, the use of the proposed state management strategy offered an improvement to NTAP web server as a whole. Therefore, the proposed guidelines can be said to be of benefit to the state management of a modern frontend application.

6 Conclusion

The aim of this thesis was to develop a strategy for managing the state of a modern frontend web application. The strategy was intended to cover a wide range of factors that affect the state management of a modern frontend application.

To choose a state management solution, the strategy proposes that the developer should be able to:

- extract the necessary information from the application's requirements and architecture that would affect its state management;
- identify and classify the types of states needed by the application;
- consider and select patterns suitable for the application's state management;
- understand other factors that could influence the choice of tools for state management such as the development framework;
- analyze the available state management tools to point out their properties and use cases and then select the most suitable ones.

After the tools have been selected, it is essential for the methods provided by the chosen state management tools to be carefully analyzed and matched with the appropriate type of state they were designed for.

To determine the effectiveness of the strategy, a case study was presented. This involved re-implementing the state management system of Nokia Test Automation Platform (NTAP), a test automation software developed at Nokia. The re-implementation required studying the previous implementation to understand its issues. Using the proposed strategy, Zustand and React-query were selected as the new state management tools for NTAP. Before and after the implementation of the new state management system, some state-related statistics were collected and compared. The statistics that were gathered include the rendering time, rendering count and rendering percentage as well as the network call counts and resource memory of the Main Page of NTAP. From the results, the newly implemented state management system showed improvements; the number of network calls and re-rendering count of the components lessened. In addition, the re-rendering time was reduced. This proved that the developed strategy was effective for the state management of a modern frontend web application.

For future work, it will be useful to understand where a decentralised state management architecture is suitable over a centralised one. This could lead to an increase

in the performance of the application. The update time could be further reduced by managing a state close to where it is used. This, however, could cause the re-calculation of certain state data multiple times which would be avoided in a global store. This shows that both centralised and decentralised systems have their pros and cons. Understanding when to use the models will improve the overall state management system of an application.

References

[1] graphql.org. "Graphql - a query language for your api." (2022), [Online]. Available: https://graphql.org/ (visited on 08/27/2022).

[2] E. Wilde and C. Pautasso, *REST: from research to practice*. Springer Science & Business Media, 2011, ISBN: 978-1-4419-8302-2. DOI: 10.1007/978-1-4419-8303-9.

[3] B. Nelson, *Getting to Know Vue.Js: Learn to Build Single Page Applications in Vue from Scratch*. Berkeley, CA: Apress L. P, 2018.

[4] C. S. Langdon, "The state of web services," *Computer Science*, vol. 36, no. 7, pp. 93–94, 2003, IEEE.

[5] X. Song, N. Jeong, P. Hutto, U. Ramachandran, and J. Rehg, "State management in .net web services," in *Proceedings. 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2004. FTDCS 2004.*, 2004, pp. 21–27. DOI: 10.1109/FTDCS.2004.1316589.

[6] A. N. Erdanto and A. Sujarwo, "Improving effectiveness of state management using prop drilling pattern on jala tech's financial feature," in *Proceeding International Conference on Religion, Science and Education*, vol. 1, 2022, pp. 651–655.

[7] K. Szymanek and B. Pańczyk, "Comparison of web application state management tools," *Journal of Computer Sciences Institute*, vol. 20, pp. 183–188, Sep. 2021. DOI: 10.35784/jcsi.2675.

[8] I. P. Vuksanovic and B. Sudarevic, "Use of web application frameworks in the development of small applications," in *2011 Proceedings of the 34th International Convention MIPRO*, IEEE, 2011, pp. 458–462.

[9] S. Ivanova and G. Georgiev, "Using modern web frameworks when developing an education application: A practical approach," in *2019 Proceedings of the 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 1485–1491. DOI: 10.23919/MIPRO.2019.8756914.

[10] P. Podila and M. Weststrate, *MobX Quick Start Guide: Supercharge the client state in your React apps with MobX*. Packt Publishing Ltd, 2018, ISBN: 978-1-78934-483-7.

[11] D. P. Voorhees, *Guide to Efficient Software Design: An MVC Approach to Concepts, Structures, and Models* (Texts in Computer Science), eng. Cham: Springer International Publishing AG, 2020, ISBN: 3030285006.

[12] J. Lee, T. Wei, and S. K. Mukhiya, *Redux Quick Start Guide: A beginner's guide to managing app state with Redux*. Packt Publishing Ltd, 2019, ISBN: 978-1-78961-008-6.

[13] Kudiabor and D. Travis, "State management with react-redux," Bachelor's Thesis, Centria University of Applied Sciences, 2020. [Online]. Available: https://urn.fi/URN:NBN:fi:amk-2020121829663.

[14] T. McFarlane, "Managing state in react applications with redux," Bachelor's Thesis, Tampere University of Applied Sciences, 2019. [Online]. Available: https://urn.fi/URN:NBN:fi:amk-2019121025636.

[15] K. E. Wiegers, *Software Requirements* (Pro-Best Practices), eng, 2nd ed. Sebastopol: Microsoft Press, 2009, ISBN: 0-7356-3518-8.

[16] V. S. P. Vidanapathirana and K. H. M. R. Peiris, "Software metrics for identifying software size in software development projects," English, *Compusoft*, vol. 4, no. 8, pp. 1960–1965, Aug. 2015, Copyright - Copyright COMPUSOFT, An International Journal of Advanced Computer Technology Aug 2015; Document feature - Tables; ; Graphs; Diagrams; Charts; Last updated - 2016-04-18. [Online]. Available: https://libproxy.tuni.fi/login?url=https://www.proquest.com/scholarly-journals/software-metrics-identifying-size-development/docview/1781594868/se-2?accountid=14242.

[17] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Trans. Internet Technol.*, vol. 2, no. 2, pp. 115–150, May 2002, ISSN: 1533-5399. DOI: 10.1145/514183.514185. [Online]. Available: https://doi-org.libproxy.tuni.fi/10.1145/514183.514185.

[18] Stack Overflow. "Most popular technologies: Web frameworks and technologies by professional developers." (May 2022), [Online]. Available: https://survey.stackoverflow.co/2022/#most-popular-technologies-webframe-prof (visited on 07/05/2022).

[19] A. Chiarelli, *Beginning React: Simplify Your Frontend Development Workflow and Enhance the User Experience of Your Applications with React*, eng. Birmingham: Packt Publishing, Limited, 2018, ISBN: 1789530520.

[20] J. Larsen, *React Hooks in Action: With Suspense and Concurrent Mode*, eng. New York: Manning Publications Co. LLC, 2021, ISBN: 9781617297632.

[21] T. Dresher, A. Zuker, and S. Friedman, *Hands-On Full-Stack Web Development with ASP. NET Core: Learn End-To-end Web Development with Leading Frontend Frameworks, Such As Angular, React, and Vue*, eng. Birmingham: Packt Publishing, Limited, 2018, ISBN: 9781788622882.

[22] B. Dayley, *Node.js, MongoDB and Angular Web Development, 2nd Edition*, eng, 1st edition. Addison-Wesley Professional, 2017, ISBN: 0-13-465564-8.

[23] vuejs.org. "Introduction - vue.js." (2022), [Online]. Available: https://v2.vuejs.org/v2/guide/index.html (visited on 07/17/2022).

[24] H. A. Pham, "Developing mobile application with vue. js framework: Case: Osmi application and superapp oy," Bachelor's Thesis, Lahti University of Applied Sciences, 2019. [Online]. Available: https://urn.fi/URN:NBN:fi:amk-2019112522507.

[25] J. Gossman, *Introduction to the model-view-viewmodel pattern*, https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm, 2022. (visited on 11/25/2022).

[26] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[27] S. Fowler, "Model-view-update-communicate: Session types meet the elm architecture," *CoRR*, vol. abs/1910.11108, 2019. arXiv: 1910.11108. [Online]. Available: http://arxiv.org/abs/1910.11108.

[28] W. Loder, *Web Applications with Elm: Functional Programming for the Web*, eng. Berkeley, CA: Apress L. P, 2018, ISBN: 1484226097.

[29] E. Salmi, "Comparing the use of purely functional programming language to event-driven javascript in modern web application development," Master's Thesis, Tampere University, 2020. [Online]. Available: https://urn.fi/URN:NBN:fi:tuni-202004294357.

[30] A. Holappa, "Web frontend development with elm," Bachelor's thesis, Oulu University of Applied Sciences, 2018. [Online]. Available: http://urn.fi/URN:NBN:fi:amk-201805148013.

[31] facebook.github.io. "In-depth overview | flux." (2022), [Online]. Available: https://facebook.github.io/flux/docs/in-depth-overview (visited on 07/24/2022).

[32] A. Boduch, *Flux architecture: learn to build powerful and scalable applications with Flux, the architecture that serves billions of Facebook users every day* (Community experience distilled.), eng, 1st ed. PACKT Publishing, 2016, ISBN: 9781786465818.

[33] redux.js.org. "Redux essentials, part 1: Redux overview and concepts | redux." (2022), [Online]. Available: https://redux.js.org/tutorials/essentials/part-1-overview-concepts#terminology (visited on 07/29/2022).

[34] redux.js.org. “Redux - a predictable state container for javascript apps.” (2022), [Online]. Available: https://redux.js.org/ (visited on 07/18/2022).

[35] redux.js.org. “Prior art | redux.” (2022), [Online]. Available: https://redux.js.org/understanding/history-and-design/prior-art (visited on 07/29/2022).

[36] D. Bugl, *Learning Redux*, eng. Birmingham: Packt Publishing, Limited, 2017, ISBN: 1786462397.

[37] vuex.vuejs.org. “What is vuex | vuex.” (2022), [Online]. Available: https://vuex.vuejs.org/ (visited on 07/18/2022).

[38] vuex.vuejs.org. “When should i use it | vuex.” (2022), [Online]. Available: https://vuex.vuejs.org/#when-should-i-use-it (visited on 07/18/2022).

[39] S. Goel. “Reactive and simple state management with vuex.” (2022), [Online]. Available: https://techblog.geekyants.com/reactive-and-simple-state-management-with-vuex (visited on 12/17/2022).

[40] mobx.js.org. “Observer design pattern | microsoft docs.” (2022), [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern (visited on 08/02/2022).

[41] D. Kato. “Zustand documentation.” (2022), [Online]. Available: https://docs.pmnd.rs/zustand/getting-started/introduction (visited on 12/20/2022).

[42] pinia.vuejs.org. “Home | pinia.” (2022), [Online]. Available: https://pinia.vuejs.org/ (visited on 07/18/2022).

[43] O. Farhi, *Reactive Programming with Angular and Ngrx: Learn to Harness the Power of Reactive Programming with RxJS and Ngrx Extensions*, eng. Berkeley, CA: Apress L. P, 2017, ISBN: 1484226194.

[44] C. Noring, *Architecting Angular Applications with Redux, RxJS, and NgRx*, eng, 1st edition. Packt Publishing, 2018, ISBN: 1-78712-240-9.

[45] mobx.js.org. “Readme - mobx.” (2022), [Online]. Available: https://mobx.js.org/README.html (visited on 07/19/2022).

[46] recoiljs.org. “Motivation | recoil.” (2022), [Online]. Available: https://recoiljs.org/docs/introduction/motivation (visited on 07/20/2022).

[47] tanstack.com. “Overview | tenstack query docs.” (2022), [Online]. Available: https://tanstack.com/query/v4/docs/overview (visited on 07/20/2022).

[48] swr.vercel.app. “React hooks for data fetching – swr.” (2022), [Online]. Available: https://swr.vercel.app/ (visited on 07/20/2022).

[49] react-query-v3.tanstack.com. “Comparison | react query vs swr vs apollo vs rtk query vs react router | react query | tanstack.” (2022), [Online]. Available: https://react-query-v3.tanstack.com/comparison (visited on 07/20/2022).

[50] S. Tilkov and S. Vinoski, “Node.js: Using javascript to build high-performance network programs,” *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010. DOI: 10.1109/MIC.2010.145.