TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TUUKKA KATAJA, JUHO TEPERI

IMPROVING AND EVALUATING CLOJURE WEB APPLICATION ARCHITECTURE

Master of Science Thesis

Examiner: Prof. Hannu-Matti Järvinen
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 3rd June, 2015

ABSTRACT

TUUKKA KATAJA, JUHO TEPERI: Improving and Evaluating Clojure Web Application Architecture
Tampere University of Technology
Master of Science Thesis, 78 pages, 10 Appendix pages
June 2016
Master's Degree Programme in Information Technology
Major: Software Engineering, Pervasive Systems
Examiner: Prof. Hannu-Matti Järvinen
Keywords: software architecture, web applications, Decision-Centric Architecture Review, Clojure

In this study, the software architecture of Clojure-based web applications was improved, based on findings in previous projects where the authors have been involved. These projects are briefly introduced to provide context for the proposed improvements. Then, potential solutions were evaluated and a web application was built based on these solutions. In addition, some new fundamental architecture-related ideas were studied to gain more understanding of how to implement forthcoming projects. As a result, a reference architecture was created based on the implemented example application.

Decisions related to the reference architecture were then evaluated using the Decision-Centric Architecture Review method. This method produced structured documentation about the feasibility of architectural choices. Based on the results, some decisions were considered beneficial and should be taken into account in future projects whereas some were considered to have minor impact or even major negative impact.

TUUKKA KATAJA, JUHO TEPERI: Clojure web-sovellusten arkkitehtuurin kehitys ja arviointi
Tampereen teknillinen yliopisto
Diplomityö, 78 sivua, 10 liitesivua
Kesäkuu 2016
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto, Pervasive Systems
Tarkastajat: Prof. Hannu-Matti Järvinen
Avainsanat: ohjelmistoarkkitehtuuri, web-sovellukset, Decision-Centric Architecture Review, Clojure

Tässä työssä kehitettiin Clojure-pohjaisten web-sovellusten ohjelmistoarkkitehtuuria perustuen havaintoihin projekteissa, joissa työn tekijät ovat olleet mukana. Projektit esitellään lyhyesti taustatietona tarvittaville parannuksille. Työssä käydään läpi erilaisia mahdollisia ratkaisuja ja työn tuloksena syntyi esimerkkisovellus. Kokemusten perusteella tehtyjen parannusten lisäksi työssä käsitellään uusia perustavanlaatuisia arkkitehtuuriratkaisuita, joita voidaan mahdollisesti hyödyntää tulevissa projekteissa. Työn tuloksena syntyi viitearkkitehtuuri, joka pohjautuu toteutettuun esimerkkisovellukseen.

Työssä tehdyt arkkitehtuuripäätökset arvioidaan DCAR-menetelmällä. Kyseinen menetelmän avulla päätöksistä saatiin jäsennettyä dokumentaatiota arkkitehtuuriparannusten kelvollisuudesta. Tulosten perusteella joitakin hyödyllisiä arkkitehtuuripäätöksiä kannattaisi ottaa huomioon tulevissa projekteissa. Jotkin päätöksistä olivat vähäpätöisempiä tai niillä oli jopa merkittävä negatiivinen vaikutus.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PROGRAMS

LIST OF ABBREVIATIONS AND TERMS

ACID	Atomicity, Consistency, Isolation and Durability properties of a database
API	Application Programming Interface provides means to interact with the software component.
AST	Abstract Syntax Tree is an internal data representation of parsed code.
ATAM	Architecture Tradeoff Analysis Method is an architecture review method.
CQRS	Command-Query Responsibility Segregation is an API design pattern.
DCAR	Decision-Centric Architecture Review is an architecture review method.
DOM	Document Object Model is an interface to manipulate browser document contents.
EDN	Extensible Data Notation is an extensible data format.
FRP	Functional Reactive Programming is a functional programming pattern.
JSON	JavaScript Object Notation is a data format.
JSONB	PostgreSQL's internal format of JSON that allow e.g. efficient indexing.
JSX	Javascript syntax extension that enables inlined markup for React.
JVM	Java Virtual Machine
MVC	Model-View-Controller is an architecture pattern designed for UI programming.
REPL	Read-Eval-Print-Loop allows interaction with compiler/interpreter.
REST	Representational State Transfer is an architectural style for distributed (hypermedia) applications
RPC	Remote Procedure Call allows invoking procedures over network.
SPA	Single-Page Application is a browser application written in JavaScript and requiring only initial pageload.
VDOM	Virtual presentation of DOM that e.g. React uses.
WS	WebSockets allow TCP/IP like communication in web application.
Avro	Data serialization system
Backend	Server application that usually provides API for client(s).
Frontend	User-facing client application that often communicates with backend.
Audit Logging	Keeping track of events to e.g. identify who did what and when.
React	Javascript rendering library
S-expression	Notation for expressing nested lists
Lisp	One of the oldest high-level programming languages where code and data are expressed similarly using S-expressions.

1. INTRODUCTION

Modern web development techniques are evolving rapidly. New ways to develop web applications are introduced by companies such as Netflix or Facebook in form of libraries such as React, Relay and Falcor to name a few [1] [2] [3]. These libraries are trying to address the increasing complexity of applications running in browsers. In addition, compile-to-JavaScript languages bring flexibility to client-side development. Languages such as Scala or Clojure offer production-ready implementations targeting the JavaScript execution environments, bringing powerful tools within the reach of frontend developers [4] [5]. Our tool of choice for both server-side and client-side development has been Clojure, meaning that developers can benefit from their Clojure knowledge as much as possible.

The web development community continues to look for better ways to implement non-trivial client-side applications. For instance, there exists many architectural patterns designed around React alone. As mentioned, it is not uncommon to use other languages than JavaScript to implement client-side applications and finding a better way to structure applications is desired since best practices have not yet been established. The purpose of this thesis is to improve the architecture of web-based Clojure applications. The need for architectural improvements stem from observed shortcomings in previous projects the authors of this thesis have been involved in. During these projects, there has usually been a shortage of time due to constrained resources, making it difficult to make a major study on architecture. This thesis is a great opportunity for such a study.

With better architecture, agility of development and easier maintenance of software can be pursued. Moreover, architecture that is flexible or extensible provides potential leverage. Since ClojureScript has been available for serious production use only for a few years, the interoperation between Clojure and ClojureScript can be improved. Also, problems like user interface rendering and data fetching that React, Relay and Falcor are trying to address, are also present in the Clojure ecosystem. However, there are some solutions such as React for rendering and Om.next for data fetching. Even if server side development can

be considered more matured, it has room for improvement also. A simple and robust way to persist and query data combined with easy-to-build and easy-to-consume APIs would be welcomed enhancements.

The first step towards an improved architecture involves examining the projects where authors have been part of the team. Secondly, solutions to improving the technology stack or architecture are evaluated. Thirdly, an example application is implemented and the resulting reference architecture is evaluated using the Decision-Centric Architecture Review method to produce analyzed documentation about the architectural decisions.

The reader of this thesis is expected to have basic knowledge of modern web development. This thesis has two authors and their individual contributions are available in the Appendix A.

Chapter 2 provides background information such as technology used and introduces the case projects. It also describes the focal points for the architectural improvements based on the case projects and has a specification for the example project. The Decision-Centric Architecture Review method is introduced in Chapter 3.

Chapter 4 covers the evaluation of data persistence and API solutions. Databases used in the case projects are examined as well as some potential choices that have not been in use. The API portion of the chapter covers a previously used API library and a more recent one that has been introduced in some projects. In addition, real-time updates are addressed in this chapter.

Chapter 5 addresses frontend-related matters. Rendering, state management and data fetching are discussed and some existing solutions are introduced. Chapter 6 is about the implementation of both the backend and the frontend. Chapter 7 concludes the architecture evaluation session that was held and its results. Appendix B contains the documented decisions that worked as a basis for the evaluation session. Chapter 8 concludes the thesis.

2. BACKGROUND

This chapter contains some prerequisite information for this thesis such as the introduction of example projects and technology used. Example projects are referred to in subsequent chapters. Also, focal points for the architectural improvements are stated and specification of example application is presented.

2.1 About Metosin Ltd

Metosin is a software consulting company specializing in agile development and Clojure programming language. The projects done are mostly web applications. Those include a backend software and a frontend user interface. The backend often communicates with external systems, implemented by other parties.

2.2 About Clojure

Clojure is a Lisp dialect targeting Java Virtual Machine (JVM). Clojure is a functional dynamically typed programming language with strong focus on immutable data and concurrency [6].

Clojure was written to make Lisp available for a matured platform such as JVM. Immutable data structures and functional programming are core concepts of Clojure. Another important design goal for the multi-core era was to make concurrent programming easier. Immutable data can be freely shared between the threads but when changing state is crucial, Clojure's software transactional memory implementation is designed to help the programmer. Clojure discourages the use of mutable state and mutable objects are considered harmful because of e.g. more difficult testing and problems related to them in concurrent applications. However, polymorphism part of Object-oriented programming (OOP) is considered beneficial. Instead of inheritance it is achieved with language feature called multimethods and protocols. [6]

In Clojure, the concept of state and identity is separated opposed to what is typical in object-oriented languages. Identity is "a stable logical entity associated with a series of different values over time" and "a value is something that does not change" [7] While this is the basis of Clojure's approach on concurrency, it is also beneficial when reasoning about single-threaded applications. Pure functions operating on pure values, producing new pure values, are easier to test than stateful objects. Clojure has a wide offering of data structures (sets, maps, vectors etc.) with a variety of core functions that can be used to operate on them.

Another important characteristic of Clojure is its interactive development flow. Clojure is a compiled language and its compiler is made available at runtime. Read-eval-print loop (REPL) can be used to define or redefine Clojure constructs when running the application [8, ch10]. For example, one can setup a project running a web server and interactively develop the application with instant feedback. Compared to some other technology stack where there could be a rather time-consuming compilation and restart phase of the whole software after every change, this is a rather productive way of developing software.

Clojure also compiles to JavaScript with ClojureScript compiler. This allows using the same language to write code targeting both the backend and the frontend environment. Some parts of the application logic that are not platform-specific can be shared between the different environments. Being able to use Clojure in the frontend development is beneficial since the same mindset can be used when developing the backend and the frontend. Clojure is also available for .NET ecosystem with an implementation for the Common Language Runtime.

2.3 About ClojureScript

JavaScript is the only language which is supported by all web browsers. It is also used on mobile devices and for server and command line programming through a JavaScript engines such as Node.js. As a language JavaScript is however not very modern.

There are ongoing efforts to modernize JavaScript (ECMAScript 6 and 7 standards). It is argued that JavaScript cannot be improved in timely manner because it is so widespread [9]. It would take a rather long time to ensure that new standards are available in e.g. all the mobile and desktop browsers.

ClojureScript targets common JavaScript environments like browsers and Node.js. JavaScript is also usable in mobile devices. [9]

To create efficient, small and compatible (with all browsers) code ClojureScript leverages Google Closure[1] compiler. Closure compiler emits code which works even in older browsers. The compiler has many advanced optimization methods like the whole program dead code elimination and function inlining.

Using regular JavaScript libraries with ClojureScript (because of Closure Compiler optimizations) requires the use of extern files describing the interface published by JavaScript library. Community maintains a collection[2] of these files for common JavaScript libraries.

2.4 Case Projects

Some previously encountered problems are described in this section.

While the Clojure(Script) stack and tooling have evolved greatly during its use at Metosin there are some fundamental architecture-related questions:

- How to implement efficient and simple backend-frontend communication for an application with non-trivial business logic?
- What data persistence solution would best support this communication model and implementation?

Some ad hoc solutions have already been invented and used on some projects. Diverging solutions between projects results in an increased maintenance burden and it becomes harder for a person to shift between projects. Proper research is needed to find common problems and to try to find more general solutions.

Next, some cases are introduced. In every project some of these challenges have been occurring and those have been solved in some ways. The projects are presented in the order that they were implemented, to see the progress.

2.4.1 Case X

System X is a part of a distributed system and works as an API service and reporting UI between several other services. System X is responsible for receiving and mediating

[1] Despite having a similar name to Clojure and being leveraged by ClojureScript Compiler, Google Closure project is not related to the Clojure language.

[2] `http://cljsjs.github.io`

orders for other systems. There is one external master system where orders are created and managed. This master system sends the orders to the system X. These orders are then routed based on postal codes to several other systems, managed by other companies. Other systems also send updates to orders which are sent back to the master system. System X stores incoming and outgoing messages (subsequently referred to as events) between the systems to achieve e.g. traceability. Any event can result in multiple messages being sent to other systems. Figure 2.1 shows a high-level overview of the system X and related systems.

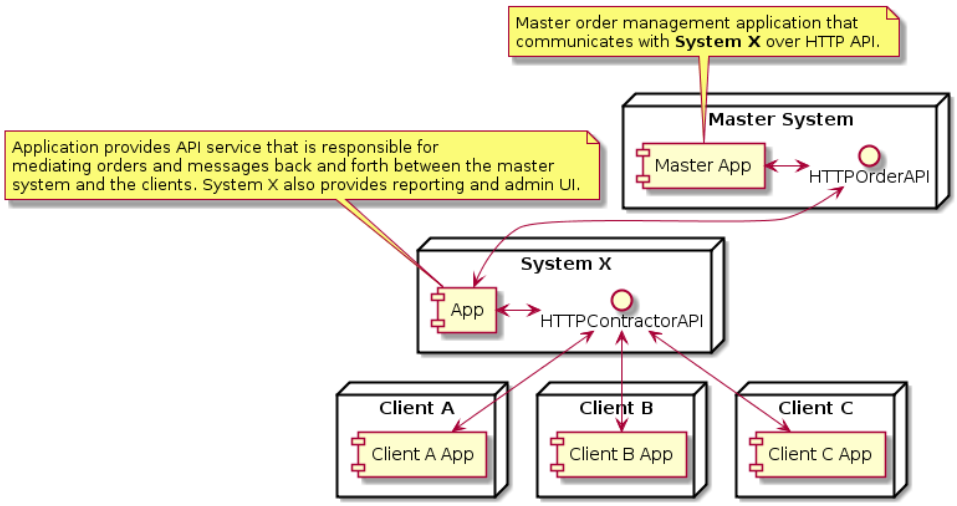

Figure 2.1: *High-level overview of the system X.*

System X uses MongoDB [10] as a document storage. Orders are stored as one large document which contains the current state of order. In addition to order collection, all the events are stored in a separate collection. Each event contains identity for corresponding order which relates to information about its origin system and recipient. System X is always either the origin or the recipient.

The event collection has the following use cases:

1. It is displayed on the user interface to manually ensure that the system is working correctly, providing traceability.
2. It is used in a couple of places for business logic.
3. It is used as a master data source if the orders are missing some data.

Point 1 should be self-explanatory. It is used for checking that correct messages are being sent to correct recipients.

Point 2 is a result of adding a feature without careful planning. In few cases, the processing logic depends on information if a message has already been sent to a specific system. Instead of querying the event collection, a better alternative would be to update the order (effectively a view) when these previous events are processed. Then the processing logic would only depend on the state of the order, which would be simpler and more clear.

Point 3 is probably the most interesting. Because event collection contains all the unprocessed messages that have affected the state of the system it is always possible to go through the events to find some information that was not originally saved to the order documents. This is very useful for database migrations.

In several cases it has been noticed that the order documents do not contain some field which previously was not interesting. Because reporting is one of the main purposes of the system, it is beneficial that when a new field is added to orders, the field can also be added to all the orders in the system based on the historical events. One example of harnessing the event data is:

1. A new feature is added to store a new field on the order document.
2. Find the orders that do not have the new field (created before the feature).
3. For each order, find the first message originating from the master system where the new field is derived.
4. Update each order by adding the new field value derived from the historical event.

In case of the system X, storing the events has proven out to be valuable for e.g. achieving traceability and adding new reporting capabilities afterwards.

The frontend of the system X is built using Google's AngularJS framework [11] and is planned to be rewritten with ClojureScript since all of other maintained frontend projects have been written using it. As frontend part is rather small, a rewrite is considered beneficial since new development will be faster and maintenance will require less effort. This is due to the potential reuse of code in form of libraries and developers being more familiar with the stack.

AngularJS is a large and complete JavaScript framework. It includes great amount of functionality which is required because JavaScript is a low-level language and lacks many

modern features. AngularJS tries to evade JavaScript's problems by providing implementations for many of these missing features: modules, dependency injection, easier asynchronous programming model and application state management [11]. Many of these features are available in the ClojureScript language or there are some libraries offering the functionality.

Important points:

- storing history as events
- using events to debug the system
- using events to derive new valuable business data
- plans to harmonize the frontend technologies used between projects.

2.4.2 Case Y

Y is a project where sensitive data is created and processed. One of the essential requirements related to data was that there must be audit log for several database tables. Information such as the last modifier and related timestamp are stored in the audited tables. Some tables have the whole history preserved. Project Y is written using Clojure and ClojureScript and uses a relational database to store the data.

History tables can be used to store the changes related to any row in a given table. Database triggers can be used to append a row to history table after each change to the original row. Project Y uses this strategy to preserve the history.

Project Y involves multiple user roles that define what actions are available to the user and what data can be seen. The user interface has to adapt for different roles and actions need to be restricted or made available based on some business rules. Since the frontend code was also written in Clojure, some logic could be shared between the backend and frontend.

The backend and the frontend communicate over HTTP. Some parts of the API can be described being RESTful[3] meaning that the domain objects are modeled as resources with URI mappings. These resources are manipulated using the HTTP verbs. The API

[3]Using Richardson Maturity Model, which describes to what degree the web technologies are applied [12, pp. 18–20], RESTful parts of the API in this application are level two in a scale of zero to three.

also has parts which are more like Remote Procedure Calls (RPC) over the HTTP[4]. This means that functions are made invokable by exposing them via HTTP POST endpoints and arguments are given in an HTTP POST method body.

Initially, RESTful APIs did work rather well for some parts since some entities were independent resources with simple create, read, update and delete operations. Fundamental problems began to show up when updates to a resource caused side effects to another resource. The problem was solved on the client side by reloading the data that was altered by the side effects. However, the solution does not work when there is a need for functionality where multiple users can simultaneously modify entities and changes have to be reflected in real time to other users. This is because the changes are only loaded after user's own actions and it does not take into account the actions of the other users. Another API-related observation was that wrapping some backend functionality behind HTTP API required more work than it should since the built frontend application is the only client and invoking actions should be made more straightforward.

Case Y was the first major project where ClojureScript was used to build the frontend. The project has been going on for over a year and during this time there have been major changes in ClojureScript ecosystem. Initially, the frontend was built using Om library [13] but after half a year the code was ported to use Reagent library [14]. As both are ClojureScript wrappers for React JavaScript library, the porting was not hard, but because of it, the code base still contains some idiosyncrasies caused by Om.

The libraries guide towards a good model of rendering the data received from backend, but they do not address the problems such as how to handle the side-effects, for example, how to save form data to the backend. It is simple enough to send HTTP requests to the backend over the API, but there are many requirements that add to the complexity. Some errors can only be detected by the backend so the response might contain an error which needs to be displayed to the user. Some backend calls are quite slow and user needs to see when an operation is in progress. These were solved per case using ad hoc solutions which add considerable complexity to the project.

Important points:

- storing history using history tables for audit log purposes worked OK
- shared (validation) logic between the server and the client

[4]RPC over HTTP is discussed in more detail in section 4.3.3

- optimal way to handle the side effects is still unknown
- more straightforward client-server interaction mechanism needed
- simultaneous editing for multiple users requires real-time communication support.

2.4.3 Case Z

This project is one of the systems integrated into project X. Whereas project X works as a message mediator and reporting interface, this system works as a client to project X. Project Z is used by multiple businesses to manage their data within project X's system.

The application is implemented using Clojure and ClojureScript. For this project PostgreSQL was selected as a database system instead of MongoDB. This is because PostgreSQL provides transactions and is fully ACID-compliant [15].

As the experiences with storing all events were good with previous projects, storing events was also implemented in this project. Because the event schema is quite flexible and dynamic, PostgreSQL document database features were chosen to store the event data as JavaScript Object Notation (JSON). This works well as event data does not usually need to be queried. PostgreSQL also allows indexing and querying JSON data if needed [16].

For this project, the event logging was somewhat enriched when compared to the Project X. Now both the request and response are logged as one event, instead of separate *received* and *responded* events. This helps with reading and debugging the logged data.

The data in the system belongs always to a single tenant (usually a company) and the tenants can only access their own data. User has multiple roles to manage permissions. An administrator role exists to debug the system state and to create new tenants and users. For tenant users there are two roles, and tenant administrator role which can add new users and normal tenant user.

The communication between the client and the backend is implemented using a Command Query Responsibility Segregation (CQRS) model using Kekkonen Clojure library[5] implemented by Metosin. CQRS is a pattern where operations that update the data (commands) are separated from the operations that read the data (queries) [17]. The API is modeled using queries which retrieve some data from the system, without side-effects, and commands which cause side-effects in data. The commands themselves do not return any results, but instead the client can run some queries again to retrieve the changed state.

[5] `https://github.com/metosin/kekkonen`

In this project, the frontend was built from the beginning using the latest version of Reagent library[6]. This helped to write more idiomatic code but the same fundamental problems of handling side-effects from Project Y still exist and are solved ad hoc.

Important points:

- improved stored event data compared to Project X
- using PostgreSQL's document features (JSON)
- communication using CQRS model helps, but it is still frontend code's responsibility to know what side effects a command causes
- user role and tenancy logic are implemented ad hoc inside queries and commands.

2.5 Focal Points for Architectural Improvements

The three cases introduced previously have some common unaddressed architectural challenges that need to be resolved. These challenges are addressed in the example application that is built to test architectural solutions. The architecture of the example application will be referred to as *reference architecture*.

Audit logging has been present in two of the case projects and it will be addressed in the reference architecture. History has been preserved in the case projects in various ways. Project X used event storing that was successfully used for debugging or deriving new valuable data from history. Project Y used history tables to store full history of some tables due to customer requirements. REST-style APIs have been used in the past. One reason for that is that previously frontends were built using various JavaScript frameworks such as AngularJS and generic interface was not considered a disadvantage. Using Clojure(Script) in the frontend together with more straightforward interoperation with backend is improved by using CQRS-style APIs instead of RESTful APIs. This means that instead of wrapping the domain data and functionality in RESTful resources, they are made more directly available via simple commands and queries. Reference architecture will also address live updates (or real time updates) as it would have been very beneficial to have some solutions available in the case project Y.

Databases used in the case projects are MongoDB and PostgreSQL. Project X uses only document model but projects Y and Z are mostly relational with some document-like

[6] `https://reagent-project.github.io`

data where suitable. MongoDB has some use cases and is more pleasant to work with Clojure data structures than PostgreSQL. On the other hand, it lacks e.g. transactions and PostgreSQL is improving its document-storing and querying capabilities. There are also promising databases such as Datomic available, which will be examined more closely.

As ClojureScript has only been available for quite a short period, there is a room for improvement on the client side of the stack. As mentioned above, shifting from REST to CQRS means that the frontend must also be improved in that area. Addressing the communication between the backend and the frontend is important but as user interfaces become more and more complex, how the application state is stored is another important concern. How to store the frontend state needs further studying.

Picking technology for use in customer projects is not only a matter of picking the one which is technically the most promising. For example, there might be some requirement that forces selecting a specific database solution due to e.g. support provided for it. In contrast, some startup prototyping a product with a potentially short lifespan could make less conservative choices to gain more agility.

Table 2.1 introduces features in case projects and in the reference architecture. Checkmarks (✓) are used to annotate the presence of a feature in each project. Ballot marks (✗) indicate where the feature is not present but would have been beneficial. Functionality/features will be explained in more detail in forthcoming chapters.

Table 2.1: *Architecture-related patterns and their presence in each case project and the reference architecture*

Functionality/feature	Case X	Case Y	Case Z	Reference architecture
Backend				
Audit logging		✓	✓	✓
Preserving history	✓	✓	✓	✓
REST-style API	✓	✓		
CQRS-style API		✓	✓	✓
Live updates		✗		✓
Document data model	✓	✓	✓	✓
Relational data model		✓	✓	✓
Frontend				
CQRS-client			✓	✓
App state as tree	✓	✓		
App state as graph			✓	✓
State management conventions				✓
Live updates		✗		✓

2.6 Specification of an Example Project

For purpose of this thesis, an example application is built to address aforementioned problems and to study different kind of software stacks and architectural choices. A lightweight specification of this application is introduced in this section. The example application is a time-tracking service that allows logging of working hours for various projects that user is working on. It supports multiple users. Some reporting views are also implemented. Other features include e.g. real time updates to UI. For example, if someone is looking at a monthly report and users are modifying data, changes are visible without a page reload.

Planned features:

1. User management
    1. User accounts can be registered

    2. User accounts can be suspended
    3. User login/logout
    4. Password retrieval

2. Project management
    1. Projects can be created
    2. Project attributes can be edited
    3. Projects can be archived
    4. Users can be assigned to projects
3. Logging of hours
    1. User can create entries to specific project
    2. User can modify created entries
    3. User can mark entries as removed
4. Monthly reporting view of logged hours
5. Real time changes in UI
    1. Real time sync of entries
6. Audit trail

During the building of the example project, some technical spikes are done to test feasibility of technology choices introduced later on this thesis, without consuming too much resources on them.

3. DECISION-CENTRIC ARCHITECTURE REVIEW

As mentioned in the background chapter, based on the observations on earlier projects, the aim of this thesis is to improve the architecture of the forthcoming projects. Therefore, some architecture evaluation method is necessary to produce useful information about possible improvements. Authors of this thesis have been familiarized with two methods during the university course:

- Architecture Tradeoff Analysis Method (ATAM)
- Decision-Centric Architecture Review

Decision-Centric Architecture Review is chosen as an evaluation method and is briefly introduced in this chapter. It was selected due to the agility of the method and e.g. the documentation that it produces were considered a great fit for this thesis.

The main motivation behind architecture evaluation is to systematically find flaws in the architecture as early as possible to avoid higher costs in finding them in the later phase of software lifecycle. Software development being more and more agile, thus resource-consuming analysis is avoided and there existed a need for a more agile evaluation method. DCAR was developed together with industry to satisfy the requirement of being more lightweight than some other alternatives. [18, p.69]

Being lightweight (half-day session with 3-5 members), DCAR aims to choose a specific set of decisions for analysis instead of more thorough analysis in other methods. Relationships between architectural decisions are recognized and decision forces behind them are identified. These decision forces can contradict with each other. [18, p.70] For example, some bleeding edge document database could be selected for performance reasons and to allow more agile development. However, that particular choice could lead to weaker data integrity and reliability. After finding out these forces, DCAR participants will evaluate the decisions and check if those are still valid given the current situation [18, p.71].

Preparation Step

A session date is chosen, participants are invited, the lead architect prepares an architecture presentation with e.g. high-level views, architectural patterns, technologies. Another presentation with management and customer perspective is prepared with details about the product, its domain and other important business-related matters. Material is given to participants before the DCAR session. [18, p.71]

Introducing DCAR

The DCAR evaluation method is introduced by showing the session schedule, steps involved, the scope of the evaluation, what is produced during the session and responsibilities and roles for each participant. [18, p.71]

Business-Related Representation

15-20 minute representation about business-related matters is given to expose decision forces related to business. Review team collects these forces and might ask additional questions to find out more. [18, p.72]

Architecture Representation

45-60 minutes is used for highly interactive architecture representation given by the lead architect. Review team will complete the architecture decision list that was produced in the first step. In addition, a list of decision forces is completed. [18, p.72]

Finalizing The List of Decisions and Forces

One person is responsible to produce a relationship diagram during the previous steps that has all the gathered decisions with *depends on* or *caused by* relationships marked between them. Motivation for the relationship diagram is to aid the reviewers to estimate the importance of each decision and to find out if they are also decision forces for some other decisions. [18, p.73]

Decision Priorization

Decisions must be prioritized since usually the number of the decisions found is too large to go through within a session. Prioritization depends on context but usually selection criteria involve e.g. those causing high risk or high cost. Each reviewer has 100 points that they can allocate freely to any decisions based on the selection criteria. During the

process, each point allocation is discussed and decision evaluation ordering is achieved. Typical reasonable amount of decisions for half-day evaluation varies between 7 to 10. [18, p.74]

Decision Documentation

Decisions picked to evaluation in the previous step are documented by each person selecting two to three decisions they can reason about. Architectural problem, solution, known alternative solutions, existing or new forces in favor or against related to decision are documented using given template. [18, p.74]

Decision Evaluation

15-20 minutes is used for each documented decision to review it, to discuss about it, to identify more related forces. Meantime, the decision documentation and the decision-relationship diagram is updated. Forces in favor and against are discussed and finally reviewers rate the evaluated decision and give their rationale behind the rating. Rating and the rationale are documented into the template mentioned in the previous step. [18, p.74]

Reporting and Retrospective

The Decision-Centric Architecture Review method produces artifacts during the steps (Diagram 3.1) and review team will use those to a write report which is then discussed with the architect. It is beneficial to arrange this retrospective as soon as possible when the session is still in the recent memory of the participants. [18, p.75]

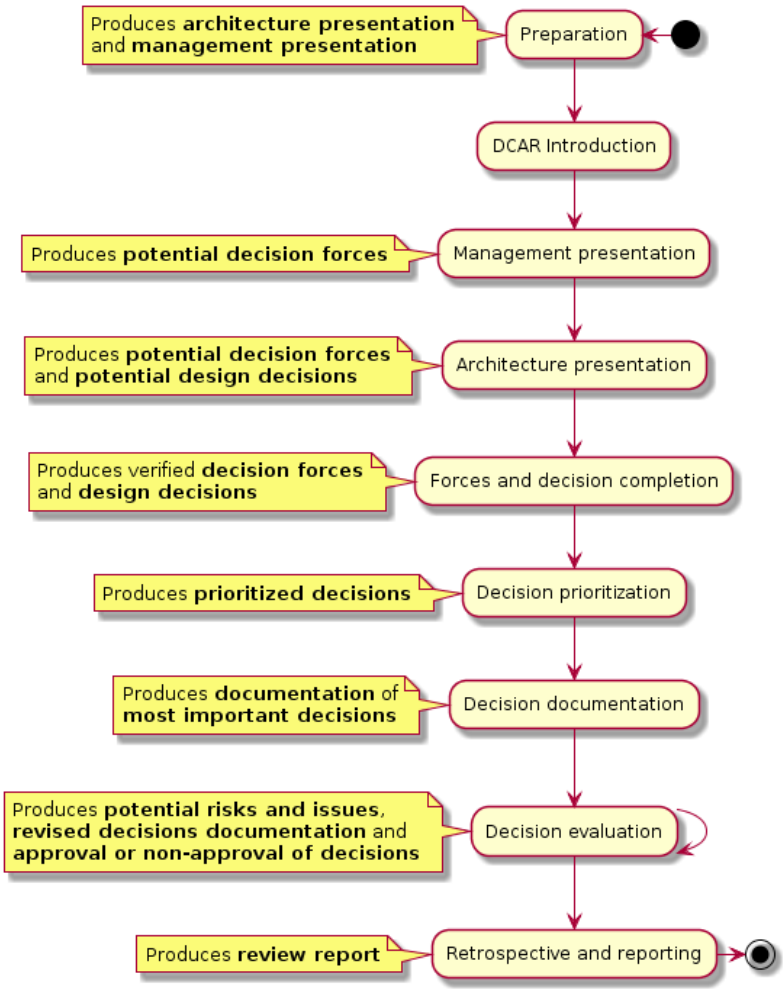

Figure 3.1: *Summarization of DCAR review steps and artifacts produced during each step. Adapted from [18, p.72].*

Some parts of the evaluation method might be omitted for the purposes of this thesis. For example, there are no business aspects related to the example project and DCAR contains some business-related steps and aspects.

4. EVALUATING DATA PERSISTENCE AND API SOLUTIONS

In this chapter, the used backend-related technologies are discussed and evaluated based on the real-world project experience. As mentioned in the background section, the tool of choice, when writing backend code, has mostly been Clojure but different database solutions have been in use. In addition, API-related choices and technologies are addressed in this section.

4.1 Data Persistence

There is abundance of databases to choose from. Various logical data models exist such as relational, document, key-value, graph etc. The scope of the study is limited to familiar models and one new interesting model that has aroused interest. Used database solutions are evaluated based on the experience gained from projects. Some important criteria for selecting a database and associated libraries in no particular order:

- preserving Clojure types (dates, sets, vectors)
- ability to store documents
- easiness of generating queries programmatically
- transactions.

4.1.1 Document-model (MongoDB)

By now, a couple of different database solutions have been in use with differing data models. For example, project X was based on a document-oriented database called MongoDB. Schema-less solution has proven to work rather well when persisting Clojure data structures such as maps. This requires less work compared with mapping Clojure data

back and forth between, for instance, a relational database and an application. On the other hand, flexibility coming with schema-lessness means that the application itself is responsible for ensuring data constraints.

Unfortunately, MongoDB is not fully ACID-compliant and enables atomicity only on document-level [19]. MongoDB has only recently (available for version 3.2) added support for (left outer) joins [20]. With earlier versions, data was denormalized or joins were required to be implemented on the application level. MongoDB is also rather new technology and is not as proven as, for example, some relational database solutions.

The used MongoDB client is a library called Monger [21] which makes it easy to store Clojure maps into the database. Clojure types are widely supported and many of the core data types can be easily persisted [22]. Queries are formed by using plain Clojure data structures [23]. Thus, generating queries programmatically is possible.

To conclude, MongoDB has proven out to be suitable for relatively simple domains where non-relational data needs to be easily persisted and other than trivial document-level atomicity is not necessary. Used together with Monger library, it allows Clojure data structures to be persisted without too much of additional work. Querying is easy since those are formed from Clojure data structures. If non-trivial transactions are required or the data model is not document-like, there are more suitable data storage solutions.

Experienced strengths:

- lack of schema allows more agility during the development
- less burdensome to map the data between the DB and the Clojure application than e.g. with relational databases.

Experienced weaknesses:

- the application is responsible for most of the data validation
- missing joins (before version 3.2) leading to denormalization or application-level joins
- no transactions (only document-level atomicity)
- rather new technology compared to more conservative data storage solutions.

4.1.2 Relational Mixed with Document Model (PostgreSQL)

Recently, the chosen technology in projects for storing relational data has been PostgreSQL. The development of PostgreSQL has been going on for almost two decades, it is a mature and stable ACID-compliant database. [24] It is rather feature-rich and offers great amount of functionality. It is quite flexible when it comes to the data model. PostgreSQL is an object-relational database, having even document-oriented properties since recently introduced JavaScript Object Notation (JSON) data type.

Compared with MongoDB, PostgreSQL is fully ACID-compliant and supports transactions meaning that changes made inside a transaction are persisted (written permanently on disk) completely or not at all. Also, wide variety of joins are supported to combine data between tables in contrast to recently introduced left outer join in MongoDB.

Where PostgreSQL is not on par with MongoDB is the easiness of persisting Clojure data. Some additional work is required if (potentially nested) Clojure maps are going to be persisted into normalized tables adhering to a relational model. Similarly, reading the values from PostgreSQL might require joining the data from multiple tables and then converting the result to Clojure data. One potentially interesting approach would be to use PostgreSQL as a document database by utilizing its JSON data type.

JSON data type was introduced in version 9.2 (released in 2012) and the development around JSON is still ongoing and active [25], [26], [27]. The interesting major features of PostgreSQL regarding JSON are introduced in the Table 4.1.

Table 4.1: *JSON data type improvements in PostgreSQL database*

Version	Changes
PostgreSQL 9.2 [25]	Introduced JSON data type meant for storing JSON-data Validation of JSON string Built-in functions that can convert rows and arrays to JSON
PostgreSQL 9.3 [26]	Operators to access JSON object fields within given path
PostgreSQL 9.4 [27]	JSONB data type that is stored as a binary instead of text JSONB indexing
PostgreSQL 9.5 [28]	JSONB data modifying functions

Introduction of JSONB data format was something that could make PostgreSQL a serious competitor for document stores such as MongoDB performance-wise [29]. Using PostgreSQL as a document storage could be now combined with its transactional features that

are inadequate in MongoDB for many purposes. However, two criteria are still behind when compared with MongoDB and Monger. First, storing JSON loses type information since it supports numbers, strings, booleans, nulls, arrays, and objects [30]. Second, library support for programmatically querying JSON is not on the same level as Monger queries used with MongoDB. If these problems could be addressed, it would be a proper replacement for MongoDB in forthcoming projects.

Experienced strengths:

- maturity
- enables different data models (relational, object-relational, document)
- ACID-compliant.

Experienced weaknesses:

- suitability for document storage (MongoDB replacement) needs more examination
- lack of types when used as document storage with JSON
- when using relational model, Clojure data mapping needs extra work
- when denormalized data is required, it is not as easy as with MongoDB.

4.1.3 Fact-Based Temporal Database (Datomic)

Whereas MongoDB and PostgreSQL are already in use in Metosin's production software, Datomic is something that needs more exploration before any decision can be made of using it. Datomic is an ACID-compliant database that has interesting and potentially valuable properties compared with previously mentioned databases. Datomic is a database where immutable facts are stored over time superseded by facts written in the future [31]. Allowing queries with time specified, it is a temporal database which preserves all the history of everything ever written into it. This property of involving time might have potential business value or it might help with requirements such as keeping the history of the modifications for auditing purposes etc. If this kind of functionality is required, one has to build these on top of the previously mentioned database solutions.

Datomic stores datoms, facts about some entities, which are the basic unit of stored information. Entities are collections of related facts. In every Datomic database, a schema

must exist. The schema defines what kind of attributes can exist in the database. These attributes can be freely associated to any entities in the database. Datomic does not dictate what kind of attributes can be related to an entity, leaving this as the responsibility of the application itself. [32], [33] This offers a rather flexible way to model the data, which can still be constrained by the schema. One strong point of Datomic within our context is the use of Clojure language since it is already being used to write almost all backend and frontend code. Clojure can be used to write custom aggregates, predicates etc. to be used within queries.

Datomic query format is based on Datalog query system. Queries are written in Extensible Data Notation (EDN) format. Queries can be thus created programmatically since EDN consists of data structures such as e.g. strings, lists and maps. Datalog is declarative like SQL. One fundamental difference between SQL and Datalog is that whereas the former requires explicit joins, the latter combined with query engine makes them implicit. [34]

Compared with PostgreSQL, Datomic has a rather limited set of types and cannot be extended. This thesis and the example project is a good ground for testing Datomic's suitability for our purposes compared with two other mentioned databases.

Datomic has a feature called transaction notifications which allows every peer that is connected to system's transactor to be notified of new facts written into the database. Connected peers will receive transaction reports that consists of value of the database before and after the transaction. Also, reports contain a set of facts written in a transaction. [35] This feature can be used to implement the real-time features of the example project. Whenever some entity is changed, all the connected clients will get changes related to entities that are subject of interest of each client.

Potential strengths:

- built-in history-preserving capabilities
- potential benefit from existing Clojure knowledge
- fully ACID-compliant.

Potential weaknesses:

- rather new technology
- closed source
- low-level API, might need some library to make usage more comfortable [36].

4.1.4 Storing Events and Event Sourcing

With MongoDB and PostgreSQL, the obvious way to store some state would be inserting documents (in case of document stores) or corresponding rows into tables (in case of relational DBs) that represent some value. Modifying the state would be done with updates to documents or rows as well as deleting would be mapped to delete operations of those databases. In this case, the database would only be able to store the current state. However, there are alternatives to modifying state in-place. As was described in two of the example projects, some events were stored to have the history of the interaction between the systems. Storing events and using them can be taken to the next level and the application state could be determined by processing events in order.

The main principle of the event sourcing is to ensure that the application state is captured as a sequence of stored events. The application state (and all the past states) can be constructed by processing the stored log of events. Events are created after successfully processing commands. For each modification related to a modeled domain object there is a corresponding event object. For example, this enables complete rebuild of the application state by reprocessing the events on the initial application state. Another interesting example that event sourcing enables is temporal queries which are achieved by rerunning events until a specific point at history. This also allows e.g. speculation by inserting and processing some sequence of events at that point and observing the resultant state. [37]

When the application state that is persisted into a database is changed by e.g. object-relational mapper frameworks such as Hibernate or Entity framework, change is considered to be *implicit* [38]. For example, domain objects are mutated in the application code and finally persisted into the database by the framework. Change can be made *explicit* by using domain events that explicitly state the change [38]. A domain event can be modeled with some object such as *ItemRemoved* which could e.g. contain information about when a specific item was removed. By not storing the current state, coupling between the representation and storing mechanism is removed [38]. This means that various kinds

of representations can be achieved by processing the explicit changes which is a very powerful concept.

One way to delete things when using implicit changes is to simply delete the object in the database or at least mark it as deleted. If history tracking is necessary, some other technique must be used. However, deletion as a concept is rather different when using event sourcing. There must exist an event that reverses some earlier event. *Reversal transactions* will not lose the history and on the other hand it simplifies the storage mechanism (e.g. easier partitioning) because it can be append-only [38].

The efficiency of calculating the current state with a large number of events might become a concern. The processed state can be persisted, for example, after a certain threshold in number of events is exceeded. These rolling snapshots are used to prevent a loading of too many events, but it is only necessary to be used heuristically when performance causes problems [38], [39].

Undoing the effects of the faulty events can be handled with a few different strategies. One would be to roll back to some previous state and replaying the events without incorrect ones in the sequence. This can be also done with reversing events. A reversing event is a type of event that undoes the effect of some other event. This is merely an optimization since rollback and replay can be used to the same purpose [37].

As mentioned in the project case section, projects have usually been integrated into various external systems. When an event-sourced system system resends messages during the replaying of the events, external systems probably do not work very well since they were not designed for replays. There are a few ways to tackle this problem. During the replays, those messages that are not safe to resent should be disabled. Another solution would be to buffer external messages so that events can be reprocessed freely before external changes have been occurred. External queries can also be problematic if replaying the events cause queries to external systems. This might be problematic because the query result might be different or the query might not work at all during the replay.

Changing the application logic of an event-sourced system itself has its own challenges. For example, if there is a new feature that adds a new mandatory data field. Then there either must be code that is compatible with both the old and new data or migration code that sets some sensible default value to previous data. Also, these changes might have to be reflected to external systems as well.

While event sourcing has interesting properties and possibilities, the whole application has to be designed around it. If the application has many integrations to external systems, using event sourcing might make the system rather complicated or at least more expensive to implement. As Greg Young states in his talk [39], careful and potentially expensive analysis is required and event sourcing is not suitable for every domain. Event Store is a database specifically built for storing events [40] but since event stores are rather simple append-only storages, an event store could be built on top of existing storage solutions.

As mentioned in Young's talk [39], business value can still be gained without using the event sourcing. This is due to preserving potentially valuable historical data. As was noticed in the example case earlier, if some event log is available, it can be valuable. It requires more thorough analysis in our case to determine whether to just refine the concept of a plain event log or would it be useful to take a more radical approach of adopting event sourcing.

Potential strengths:

- data is not lost, whole history is preserved
- decoupling the views from storing mechanisms.

Potential weaknesses:

- not suitable for every domain
- many new unfamiliar concepts
- low number of libraries and various event storages.

4.1.5 Capturing Data Changes with PostgreSQL and Bottled Water

Bottled Water is a tool that uses PostgreSQL's logical decoding feature to produce a stream of persistent changes based on the database modifications [41]. Logical decoding feature allows the extracting of the database changes, based on the PostgreSQL's transaction log (write-ahead log) [42]. For each committed insert, update or delete, an event is produced. This event is then encoded in Avro format and sent to Kafka messaging system. Avro is a data serialization system that has JSON-based schema declaration [43]. Bottled Water uses it to lossless type conversion [41].

The Kafka messaging system maintains the feeds of messages that belong to some topic. Some processes can work as producers by publishing messages to Kafka topics whereas some processes can work as consumers, processing the messages. Kafka guarantees that messages sent appear in the same order as they were sent to consumers. Typical use cases for Kafka are using it as message broker or using it for log aggregation, stream processing or *event sourcing*. [44]

For each table present in PostgreSQL, there will be a topic in Kafka. For each committed insert, update or delete operation to some row, there will be a message with corresponding topic in Kafka. Bottled Water also consists of a client that receives Avro-encoded data from the database and forwards it to Kafka. [41]

Now any number of consumers could be set up to listen to changes in PostgreSQL. PostgreSQL with Bottled Water provides interesting possibilities to achieve some of the properties discussed earlier, such as audit logging, preserving history etc. Now, every change in the database could be consumed by multiple processes. If there is a need to persist a history e.g. for audit logging purposes, a consumer can be setup to listen to changes to tables that need audit logging and store the changes using some other durable storage. Also in case of event stores, there was an argument in favor related to decoupling views from data storage. With this mechanism, one could use more traditional architecture and database such as PostgreSQL and still build interesting views based on the change data produced by the Bottled Water.

To conclude, Bottled Water is a very interesting approach to get the change data out of PostgreSQL. Choosing the event sourcing approach would be indeed interesting but it requires a total mindset shift compared with more typical architectures. With Bottled Water, some of the benefits of the event-sourced model such as audit logging or power to build various views could be achievable with more familiar overall architecture.

In Figure 4.1 the communication between PostgreSQL, Bottled Water extension, Kafka and the application backend is illustrated and explained.

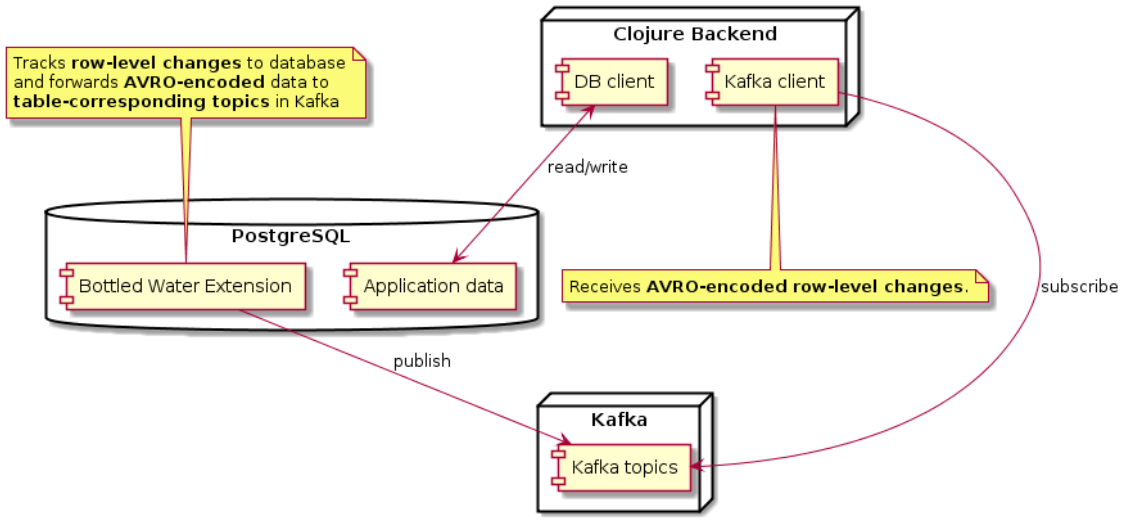

Figure 4.1: *Overview of the interaction between PostgreSQL, Bottled Water extension, Kafka and the application backend.*

Potential strengths:

- keep familiar model to query and mutate the data while having the capability of preserving the history
- could be plugged into existing software to produce change data.

Potential weaknesses:

- Bottled Water is still in alpha-stage
- more complicated infrastructure (requires Kafka, PostgreSQL extensions etc.)

4.2 Database Choices for the Example Application

After briefly evaluating the databases, the features of those, the potential and experienced strengths and weaknesses, some of them has to be chosen for further testing with the example application.

MongoDB was previously used in a few of Metosin's projects. While it is rather easy to use with Clojure, the lack of transactions is considered a severe deficiency. Meanwhile,

PostgreSQL's document-storing features are improving, making MongoDB less and less interesting choice for future projects. Thus, MongoDB is left out as a choice for the example application.

PostgreSQL and relational model is not that easy to use with Clojure. One of the reasons is that mapping between Clojure data structures and relational data requires some additional work. As mentioned above, PostgreSQL's improving JSON support makes it more and more suitable choice for document-storing purposes. PostgreSQL is thus rather flexible solution as data storage while being a very conservative choice due to near two decades of development. There is also an interesting ongoing project called Bottled Water which extends PostgreSQL to enable the tracking of the changes. PostgreSQL is in use in some of Metosin's projects and has been a recent choice for new projects. Thus, it is picked for further evaluation in the example project.

There is no popular event sourcing library available for Clojure. One open source library called Rill [45] is available in GitHub but it seems to be in alpha-stage without much documentation available. Event sourcing could be prototyped rather easily. Events could be appended into some in-memory location but while there is nothing (at least documented) available, everything should be created from the beginning. Thus, the event sourcing option is not evaluated within this thesis.

Datomic is generally a rather unknown choice for a database while it is quite known in the Clojure community due to being invented by the designer of the Clojure language. Datomic is very different compared with MongoDB and PostgreSQL (data model, query language to mention a few). It is picked for evaluation in the example project, since it is a good place to test this interesting but still bleeding edge technology, instead of some real customer projects where picking it might be too risky.

4.3 Application Programming Interface (API)

PostgreSQL and Datomic were picked as the databases for the example project. Another important concern about the backend is to what kind of API is exposed to the frontend and what is the communication mechanism. Typically, some sort of RESTful approach has been used over HTTP where modeling the domain works in resource-centric manner. If some parts of the API are more easily modeled as actions, RPC-style API has been favored.

4.3.1 HTTP, Representational State Transfer, REST HTTP APIs

Hypertext Transfer Protocol (HTTP) is a request-response protocol initially designed for simple raw data transfer over the Internet. HTTP usually works on top of TCP/IP connections. Current protocol version (HTTP/1.1) points out multiplicity of concerns such as content-type negotiation, available methods, status codes, connection persistence, caching etc. [46] HTTP/1.1 is a text-based protocol and HTTP/2 (currently proposed draft) is binary-based and mainly addresses the performance concerns occurring in current version but also introduces new functionality such as server-pushed responses to a client. [47]

Representational State Transfer (REST) is an architectural style with emphasis on stateless client-server communication with good caching capabilities and uniform interface. Client-server communication improves the portability of the user interface. Stateless client-server communication means that every request contains every bit of required information to operate, without having some related context on the server. Stateless communication improves e.g. scalability due to the server not having to manage resources over requests. Caching improves network efficiency by requiring requests to be marked as cacheable or non-cacheable. [48]

REST includes a concept of a uniform interface which is designed for purposes of hypermedia data transfer and is optimized for the common case of the web. The uniform interface consists of the identifications of the resources, the manipulation of the resources through presentations, self-descriptive messages and hypermedia as the engine of application state (HATEOAS). [48]

REST HTTP APIs are based on the REST architectural style. One important concept is a resource. Resources such as videos, documents, business processes or even devices are exposed to the web [12, p.4]. To access or manipulate resources over some protocol such as HTTP, unique identifiers are assigned to them [12, p.5].

Another important concept is the representation of a resource. Representation is a view of some resource at some point in time. Software components are communicating by transferring run-time negotiated representations of some resources. There might be many different representations for the same resource such as JSON, XML, image, plain text etc. While the multitude of available representations for some resources is natural for human consumers, a more limited set of structural formats is more suitable for system-to-system interaction. [12, p.7-8]

Resources are being manipulated and accessed via uniform interface consisting of small number or verbs [12, p.11]. In the case of REST HTTP APIs, these verbs are HTTP methods such as GET, POST, PUT and DELETE to name a few. When some of the verbs are used on some resource, a response with descriptive HTTP status code is returned.

Hypermedia as the engine of application state (HATEOAS) is a central concept in REST architectural style. It is analogous to human browsing some web pages and then navigating to other places using hyperlinks. HATEOAS enables the application state to be transitioned from one state to another by using links that are not possibly known in advance by the software client [12, p.13].

One way to classify services that are exposed on the web is called Richardson Maturity Model. This model introduces four levels that indicate how much of the web technology is used to implement the service. [12, p.18-19]:

Level zero services expose single URI which is accessed using an HTTP POST method. Payload is sent in the body of a request. [12, p.19]

Level one services expose multiple URIs that are accessed using a single HTTP verb such as HTTP GET. For example, operation names and parameters can be part of the URI. [12, p.19-20]

Level two services allow the manipulation of the resources with a larger set of HTTP verbs and resources are mapped to numerous URIs. [12, p.20]

Level three services use previously mentioned HATEOAS concept to inform clients about possible state transitions.

RESTful APIs in the example cases can be considered level two in the above classification system. Level three APIs have not been built since the benefits of the HATEOAS or even the possibility of implementing it in practice in our cases have been uncertain.

RESTful APIs have been proven out to be successful when the server and API users (frontend and other services) are completely separated. One proof of the success of RESTful APIs is that large portion of public APIs (e.g. Twitter, GitHub) have been built using this methodology. However, the benefits of the RESTful model are not that obvious when both backend and frontend are closely tied together and no public API is exposed. When the objective is to solve a business problem using as little effort and development resources as

possible, this kind of mapping from a business domain to well-constructed RESTful API resources can be both unnecessary and costly. For example, in our projects it has been noticed that designing a good RESTful API requires:

- mapping of business domain to some sensible resources
- designing URIs for the resources
- choosing which (HTTP) verbs map to which operations
- picking descriptive status codes for different situations.

To conclude, we have built some RESTful APIs that can be considered level two in Richardson Maturity Model. The benefits of the clear and intuitive RESTful APIs cannot be denied when building public API for multiple possible client software, but that is not the common case for us and a more suitable way for tightly-coupled client-server communication is sought.

4.3.2 Compojure-api Library for Building Web APIs

One important feature for some cases (public API) is to provide documentation for the API. The most difficult part of creating the documentation is that it should always be kept up-to-date. One solution to this is to automatically generate the documentation from the source code; this way the documentation should always represent the current version of the API. This is possible with Clojure, and for example Metosin has built several libraries with a support for auto-generated API documentation. Compojure-api[1] is one of those libraries. It is a library that imitates a common Clojure routing library named Compojure[2] and adds functionality on top of it.

Compojure-api widely utilizes a library called Schema[3] which allows describing data declaratively with schemas. It also allows annotating types of parameters and return values with optional run-time validation. Other capabilities are data coercion and data generation for tests to name a few. [49] Compojure-api also uses the defined schemas to generate API documentation using OpenAPI specification (formerly Swagger API specification).

[1] `https://github.com/metosin/compojure-api`
[2] `https://github.com/weavejester/compojure`
[3] `https://github.com/plumatic/schema`

Routes in Compojure and Compojure-api are defined per HTTP terms: Each route corresponds to a specific HTTP method and endpoint (uri).

```
1  (ns example
2    (:require [compojure.api.sweet :refer :all]
3              [ring.util.http-response :refer :all]
4              [schema.core :as s]))
5
6  (def pizzas (atom []))
7
8  (s/defschema Pizza
9    {:name s/Str
10    :size (s/enum :L :M :S)
11    :origin {:country (s/enum :FI :PO)
12             :city s/Str}})
13
14 (def app
15   (api
16    {:swagger
17     {:ui "/api-docs"
18      :spec "/swagger.json"
19      :data {:info {:title "Sample API"
20                    :description "Compojure Api example"}
21             :tags [{:name "api", :description "sample api"}]}}}
22   (context "/api" []
23     :tags ["api"]
24     (GET "/pizza" []
25       :return [Pizza]
26       (ok @pizzas))
27     (POST "/pizza" []
28       :return Pizza
29       :body [pizza Pizza]
30       :summary "echoes a pizza"
31       (swap! pizza conj pizza)
32       (ok pizza))))
```

Program 4.1: Basic Compojure-api example application

In Program 4.1, atom is used as an in-memory store (line 6). The schema for application data is described using the Schema library (line 8). Application has a GET route (line 24) to allow clients to requests all the pizzas stored in the atom. There is also a POST method (line 27) defined for the same endpoint. As seen in the example, return type of the data is

annotated with Pizza schema. Also, the request body must conform to the same schema. Compojure-api validates the schema and if the data does not validate against the given schema, the client automatically receives an error response. JSON has a very limited set of types and it lacks, for example, type to represent dates. For example, if application has defined some field to be a date in a schema, Compojure-api coerces a predefined date-representing string to a date object.

Using Compojure-api library, the API is build by defining endpoints and available HTTP methods. While it does not force to build the API using RESTful architecture, it certainly makes it easy and thus encourages developers to use that approach. As was mentioned earlier, alternative approaches are under examination since RESTful API design might require some additional effort. More straightforward communication between the server and the client is desirable e.g. in the cases where frontend and backend are tightly-coupled and public API for multiple consumers is not a goal.

Important points:

- always up-to-date API documentation
- automatic input data validation (and coercion)
- routes/endpoints defined using the HTTP terms (methods, URI)
- RESTful API design is not always the right approach.

4.3.3 Remote Procedure Calls (RPC) over HTTP

Remote Procedure Call (RPC) can be defined as a "synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is narrow channel" [50, p.9]. One motivation of RPC is to make calling some procedure in a remote machine as simple as it would be on a local machine [51].

JSON-RPC (version 2.0) is a stateless protocol that uses JSON to encode request and response objects. The protocol does not define what the transport mechanism should be. It can be used for example between multiple processes over a socket connection or it can be used on top of HTTP.

Requests are JSON objects that consist of:

- protocol version (e.g. "2.0")

- the name of the operation (method)
- parameters array holding the argument objects
- id that will be the same as in a related response object.

Responses are JSON objects that consist of:

- protocol version
- result only present in successful invocations
- error only present in case of failures
- id that was sent in the request object.

Protocol also describes notifications (requests without id), the format of error object, some error codes and batching. [52]

Using RPC-like communication over HTTP seems noteworthy when implementing the communication between the backend and the frontend. Not only because the specification and the idea are simple but also because resource-centric RESTful API does not always map that well to problem domain or even the programming paradigm.

If the problem domain consists of many actions or operations on some data, it is not that natural fit for RESTful API that emphasizes resources that are manipulated with a limited set of verbs combined with error code mapping to appropriate HTTP status codes. For example, to decide if registering a new user is an HTTP POST to some specific URI with some specific response status code is more complicated than invoking a *register-user* RPC operation with either a result or some error returned.

Also, it is typical of functional programming is to use functions that operate on some values and produce new values. These functions map more easily to a procedure that is executed with some input than into a resource that is manipulated with predefined vocabulary.

4.3.4 Kekkonen Library

The Kekkonen library is Metosin's attempt to provide easy way to define APIs using e.g. CQRS model instead of RESTful architecture. Parts of the Compojure-api that were

considered good, have also been built into Kekkonen. Some examples of such parts are live API documentation and automatic input validation and coercion. Instead of focusing on resources, endpoints and HTTP methods, plain Clojure data and domain functions are important.

Specifically tagged Clojure functions (called handlers) are organized into virtual namespaces. These namespaces are registered into a dispatcher. Clients invoke handlers by creating invocation context that is passed to the dispatcher. This organization is illustrated in Figure 4.2. The client uses a fully qualified name of the handler consisting of the namespace part and the handler name. [53]

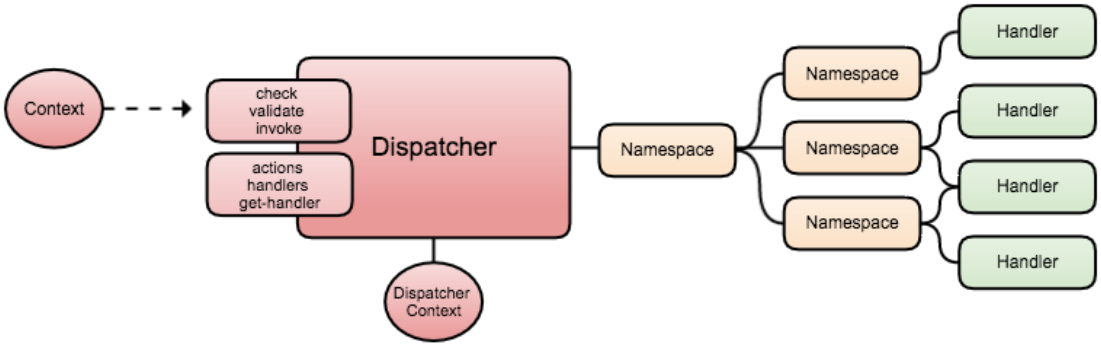

Figure 4.2: *High-level conceptual view of Kekkonen library*

The core of Kekkonen does not dictate used transport mechanism and provides mainly means to define handlers, organize them into virtual namespaces and to create the dispatcher that can be used to invoke the handlers. The library is written in this way to allow using e.g. HTTP, WebSockets as a transport layer. The library provides CQRS implementation on top of HTTP that allows tagging Clojure functions either as commands or queries. Commands use HTTP POST to invoke remote execution and queries use HTTP GET. Kekkonen also provides ClojureScript client to enable very simple interaction between the backend and the client.

```
1  (ns example
2    (:require [kekkonen.cqrs :refer :all]
3              [plumbing.core :refer [defnk]]
4              [schema.core :as s]))
5
6  (s/defschema Pizza
7    {:name s/Str
8     :size (s/enum :L :M :S)
9     :origin {:country (s/enum :FI :PO)
10             :city s/Str}})
11
12 (defnk ^:query get-pizzas [pizzas]
13   (success @pizzas))
14
15 (defnk ^:command add-pizza! [pizzas [:data pizza :- Pizza]]
16   (swap! pizzas conj pizza)
17   (success pizza))
18
19 (def api
20   (cqrs-api {:swagger {:info {:title "Kekkonen Sample API"}}
21              :swagger-ui {:path "/api-docs"}
22              :core {:handlers {:pizza [#'get-pizzas #'add-pizza!]}
23                     :context {:pizzas (atom [])}}}))
```

Program 4.2: Basic Kekkonen example using the CQRS model

Program 4.2 shows Kekkonen API providing similar functionality as Compojure-api example Program 4.1. Instead of defining a route for getting pizzas with the corresponding HTTP method, function in line 12 is annotated as a query and return value is wrapped in a success response. Similarly, side-effecting mutation to an in-memory database of pizzas is tagged as a command in line 15. CQRS-API is defined using *cqrs-api* helper function, in line 19, allowing the (remote) invocation of given commands and queries.

Important points:

- focus on domain functions and data
- good parts from the Compojure-api such as live API documentation and automatic input data validation (and coercion)
- supports e.g. CQRS model
- very simple interaction between ClojureScript client and Clojure server.

4.3.5 Side Effects of Commands and Real-Time Events

Blog post *Hybrid Microservices* introduces an idea where REST microservices are augmented with eventing mechanism between components [54]. Components publish events whenever side effects (changes) are occurring and other components can subscribe to event streams.

This idea can be modified to fit this thesis purposes. Since the CQRS model is used, commands of the backend producing side effects are viable source for events. The backend is the event producing component whereas the frontend is a component subscribing to the produced event streams. Command responses could then indicate immediate side-effects with the response whereas events could indicate the effects of long-running tasks. Eventing mechanism also allows, for instance, push notifications to be sent to the client. This idea is illustrated in the Figure 4.3.

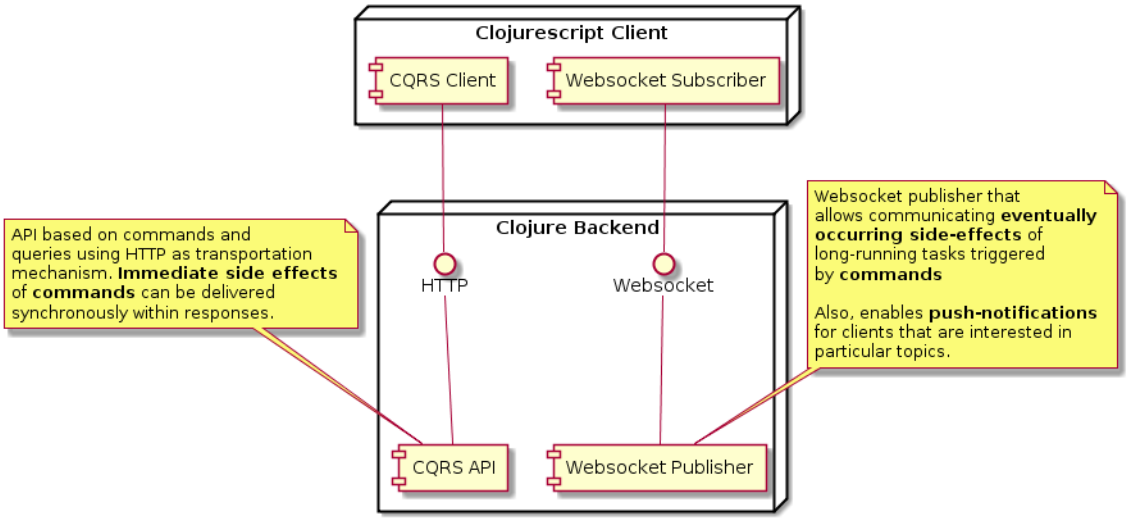

Figure 4.3: *Diagram of interaction between the backend and the frontend over HTTP and WebSocket.*

Like in the *Hybrid Microservices* article, events would have fields that identify the event, the event payload itself and a timestamp. Backend should be able to offer event streams to the frontend per event identifier basis. Possible viable (emulated or real) push mechanisms to implement event streams include HTTP polling, HTTP long polling, HTTP streaming and WebSockets.

HTTP polling is an emulated push mechanism where clients poll the server periodically to receive asynchronous events. It can be implemented with standard HTTP requests. Continuous polling will consume bandwidth since there will be a request and a response for each periodically-made request and corresponding response. Continuously polling has a tradeoff between responsiveness and used resources such as bandwidth. If the polling interval is increased, responsiveness also increases but more resources are consumed and the other way around. [55, p.2]

To address the latency problems with continuous polling, HTTP Long Polling was introduced. The server waits until it has something to deliver to the client for each request. After delivering the event to the client within the response, the client initiates a new long polling request. This means that the server has always one long-polling connection open for the client and data is delivered as soon as possible. [55, p.3-4] Problems related to HTTP Long Polling include such as HTTP protocol overhead and latency related to initiate new long-polling request. [55, p.4-5]

HTTP Streaming is a mechanism that does not terminate the connection after the server has delivered data to the client. Client initiates communication and waits for a response. The server can then send chunks of data to the client without the termination of the connection. [55, p.6-7] One of the main problems related to the HTTP Streaming include network intermediaries such as proxies and gateways that can delay the forwarding of the response as allowed by the HTTP protocol. [55, p.8]

WebSocket is a protocol designed to resemble raw TCP connections in the context of HTTP infrastructure [56, p.9]. The protocol is separate from HTTP protocol built on top of the TCP layer, avoiding for example overhead related to transferring HTTP headers. However, it uses HTTP protocol to do the handshake (or to *upgrade* the connection in WebSocket terminology). It is designed to be functional on HTTP(s) ports 80 and 443 and to support HTTP proxies and intermediaries. [56, p.3] WebSocket is widely supported (having multiple libraries or server implementations) in popular server-side languages such as Java or C#. Also, modern browsers support it via the WebSocket API [57]. There are also many options to use WebSockets on Clojure backend and ClojureScript frontend. Drawbacks of WebSockets include that it makes the server stateful. Stateless services are more trivial to scale horizontally (i.e. to add more machines to serve more users).

However, a single node setup should cover quite many use cases before scaling becomes

a real problem. Httpkit[4] is an event-driven HTTP server for Clojure that can handle up to 600,000 concurrent HTTP connections on a modern desktop machine [58]. Httpkit also offers a WebSocket server that is not benchmarked but probably should handle similar order of magnitude of concurrency since it is rather minimal protocol on top of raw TCP.

4.4 Backend Technology Choices for Example Application

The backend technology stack was evaluated in this chapter based on the experiences gained so far and potential strengths and weaknesses of yet untested but promising solutions. Feasibility of these choices will be experimented against the specification of the application build for the thesis. Final architecture will then be evaluated using the DCAR method to document and understand its pros and cons.

Databases to experiment with are PostgreSQL and Datomic. The former is proven production-suitable solution. However, combined with Bottled Water's change-tracking capability, it is something that need further analysis. The latter is more unknown technology. However, its synergies with Clojure backend are something that requires experimenting with together with different kind of data model, query language etc.

In addition to the database layer, the API layer was also evaluated. Compojure-api has been used to offer mostly RESTful APIs but a common use case being such that the only API user is a tightly-coupled frontend, shift to Kekkonen library aims at more straightforward interaction. The API layer is completed with possibility to subscribe event streams e.g. for real-time changes or the results of long-running tasks initiated by commands, provided by WebSocket based API.

[4] `http://www.http-kit.org`

5. EVALUATING FRONTEND TECHNOLOGIES

This chapter describes the currently used frontend solutions and explores new technologies related to the observed problems. Since focusing on Clojure the technologies are those that can be used with ClojureScript. Most of ClojureScript web technologies are built on top of React JavaScript library, this is evident from the latest Clojure user survey which shows that eighty percent of people using ClojureScript use React [59].

Before going into the workings of React, the reasons for its existence are explored. After the currently used libraries have been described, some more recent and unproven technologies are introduced.

5.1 Web Development Background

The web development has evolved alongside regular UI programming and has taken influence from architecture models used there. Before the emergence of Ajax (asynchronous JavaScript and XML) practice the web applications were mostly backend applications which generated HTML. JavaScript was only used to enhance experience. [60, p.1-2]

Backend applications were often build using Model-View-Controller model [61, p.6]. When web application UIs moved from the server to the browser, the same architecture models were brought along [61, p.6]. These new browser-centric web applications are often built as Single Page Applications (SPA). In a Single Page Application the page is loaded only once. This means that all the content has to be rendered using JavaScript and Document Object Model (DOM). DOM is an interface allowing the manipulation of the page contents using JavaScript.

Common JavaScript frameworks using MVC are AngularJS and Ember.js. Other similar patterns are Model-View-ViewModel used by KnockoutJS and Model-View-Presenter used by Backbone.js.

5.1.1 Model-View-Controller

Model-View-Controller (MVC) is a common architecture pattern used in user interface software. MVC was developed at Xerox PARC in 1970's by Trygve Reenskaug. The general concept was first described by Pope and Krasner [62].

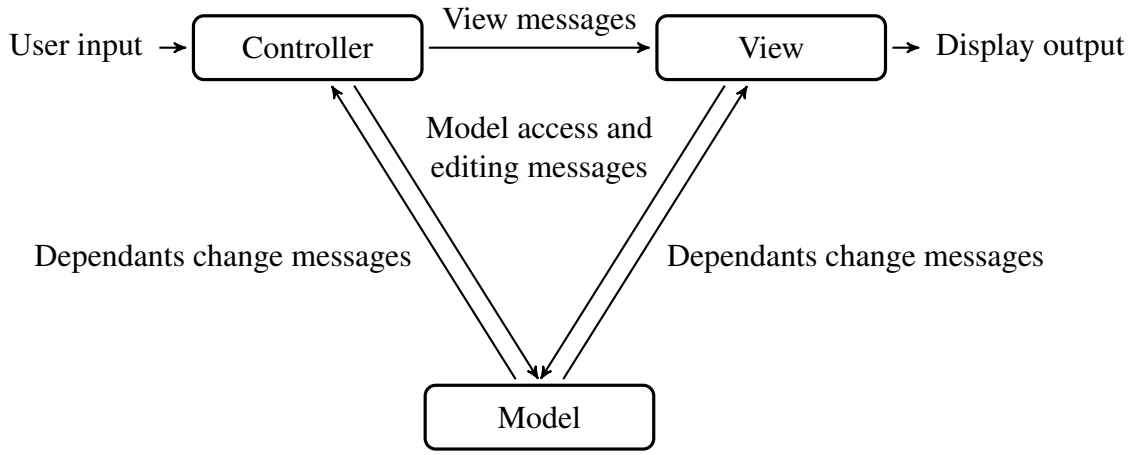

Figure 5.1: *Message flow between parts of the MVC model. Adapted from [62, p.5].*

In MVC, the application is separated into three parts, as shown in Figure 5.1. The parts are decoupled for flexibility and better reuse. [63, p.4]

Models correspond to domain-specific entities and manage their state. Models are separated from views and controllers. This allows using the single model with multiple views and controllers. [63, p.4]

Controller handles the user input, for example, from an input device such as mouse. The controller sends messages to models to update the state. Controllers can use the model state to decide how they work, for example some input could be disabled in certain application state. [63, p.5]

View displays the entities represented by the models for the user. The model notifies related views about its updates. View uses these update messages to refresh the view seen by the user. [63, p.4]

MVC model is usually implemented using Object-Oriented Programming (OOP). According to David Nolen [64], MVC offers sound separation of concerns. However, Nolen says that the implementations leaves to be desired. One of the problems is the stateful objects. According to Nolen, it should be possible to separate useful properties from

MVC and get rid of the stateful objects. This is possible using immutable data structures provided by Clojure.

5.2 Rendering and State Management

This section introduces React and some wrappers for ClojureScript. These libraries mostly address rendering the view, and might have some opinions on state management.

5.2.1 React

React is a JavaScript library for building user interfaces. React is created at Facebook and is now used by many large companies such as Netflix, Instagram and Airbnb. [61, p.3]

React is a library with one main responsibility: it renders the user interface from data. It does not dictate how to structure applications or how to fetch or how to handle application state. [61, p.5]

React is quite different from traditional practices in web development. React does not embrace Model View Controller (MVC) architecture. In React the application is structured using UI components. The idea of an UI component is similar to functions in functional programming. Components should be reusable like functions. Some think that React should be considered as the view part of the MVC architecture, but Freddy Rangel disagrees. [61, p.5-6]

Rangel writes that using React only as a view layer would negate many benefits of React [61, p.6-8]. He justifies this by arguing that instead of separating markup and display logic they belong together. React Components combine both markup and logic. According to Rangel, it is possible to achieve better separation than with MVC by leaving the separation to developer. Developers can separate logic and markup by creating small reusable components. The simplest components are idempotent functions so they are easy to unit test.

```
1 var HelloWorld = React.createClass({
2   render: function (props) {
3     return React.DOM.div(null, "Hello " + props.name + "!");
4   }
5 });
```

Program 5.1: Simple React component

Program 5.1 shows a simple pure React component. This component uses the name property it received to build the text it displays to user. The interesting bit about the function is that it returns a special data structure defining the markup of the component. In this example, the data is constructed using functions but React also provides a JavaScript language extension called JSX where the markup can be defined as embedded HTML inside the JS code. Using JSX requires preprocessing the code before browsers can evaluate it.

In addition to components using basic HTML nodes in the markup, the markup can also contain other components. This allows reusing components anywhere they are needed. Components and their children HTML nodes and other components form a component tree.

Components use two types of data. The first type of data is properties the components received from their parents. Another source of data is the local state contained by the component. [61, p.12]

Traditionally, JavaScript frameworks try to synchronize data and state using data-binding. Rangel argues that this has not worked well as the frameworks try to hide data-binding behind leaky abstractions. React tries to solve this by providing data-binding using plain old JavaScript functions, so that the leaks in the abstraction are easy to understand and predict. [61, p.9-10].

This same simple approach is used both to update the state when user interacts with the application, and to update view when state changes. When component's state changes, the whole component is re-rendered. This is fast because rendering a component only needs to create the data presenting the markup, it does not really need to update the DOM.

Usually updating DOM is the bottleneck of JavaScript application. To optimize DOM updates, React utilizes technique called Virtual DOM. The values returned from a component render methods are used to construct the representation of desired the DOM tree. This desired DOM tree and current DOM tree are compared to find optimal strategy to update the real DOM to match the desired DOM. [61, p.16]

The comparison can be performed quickly as the DOM tree has some restrictions which allows making some assumptions for the comparison. This optimized tree comparison can be done in $O(n)$ time. [61, p.17-22]

To avoid the need to use DOM event handlers, React instead only sets a single event handler on the document itself. React then reimplements event handling by passing the

events from document to correct components and elements itself. [61, p.22-23]

In addition to the render method, components can have other methods. These lifecycle methods allow setting the initial data of the component and customizing the component for better performance or to integrate a traditional JavaScript library with React.

Important points:

- component local state
- explicit state updates
- data is passed from parents to children
- virtual DOM comparison to calculate optimal DOM updates.

5.2.2 Om

Om is a React wrapper for ClojureScript. It does not try to hide underlying the React object model. Components are defined as objects implementing one or multiple Om interfaces which correspond directly to the React component methods. Macros can be used to simplify the creation of these objects.

Though Om does not try to hide React, it offers some additional features. One of these is global state management. Program 5.2 contains an example Om application.

The application state is managed using Clojure `atom` which is a data structure used to manage shared state. Though browser environment is single-threaded, atoms are useful because their state can be observed for changes (watched). Om uses this to trigger re-render of the application. In the example application state in line 4 contains a Clojure map with two properties and the values of the properties are also maps. The state is managed using a single atom. This has several benefits:

- a single source of truth
- solves synchronization between stateful components
- easy transactional updates
- undo and redo.

```
1  (ns dippa.om
2    (:require [om.core :as om]))
3
4  (def app-state (atom {:player {:name "Juho"} :game {:score 9000}}))
5
6  (defn player-component [state owner]
7    (reify
8      om/IRender
9      (render [this]
10       (om/div
11         (om/h1 "Hello World " (:name @state))
12         (om/label "Name")
13         (om/input #js {:value (:name @state)
14                        :on-change (fn [e]
15                                     (let [v (.. e -target -value)]
16                                       (om/transact state assoc :name v)))}))))))
17
18 (defn game-component [state owner]
19   (reify
20     om/IRender
21     (render [this]
22       (om/div
23         (om/h1 "Score " (:score @state))
24         (om/button #js {:on-click #(om/transact state update :score inc)}
25                    "Play!")))))
26
27 (defn main-component [app-state owner]
28   (reify
29     om/IRender
30     (render [this]
31       (om/div
32         (om/build player-component (:player app-state))
33         (om/build game-component (:game app-state))))))
34
35 (om/root main-component app-state {:target (js/document.getElementById "app")})
```

Program 5.2: Basic Om example

A single source of truth makes it easier to reason about the state and helps with debugging. Keeping the state in a single atom solves the synchronization between stateful components by making it unnecessary. It is also easier to update the state transactionally as all the updates can be made in a single operation. Using a single atom allows the implementation

of undo and redo trivially by keeping track of the previous states. This is cheap because of immutable data structures and structural sharing.

Not all components are interested in the complete application state but only in parts of it. For this use Om provides *cursors*. Cursors have a reference to the original atom and a path which the cursor is interested in. Cursor's value is the value of the atom in a given path. Writes update the atom in the given path. This way the components do not need to consider what is the absolute path of their data in the application state. Om cursors are similar to functional lenses.

CircleCI who is a prominent Om-using company has written about the problems in the Om and especially those related to the cursors [65]. The problems they encountered were the same as what we encountered in a project using Om. The main problems are related to the application state and the cursors. Cursors make structuring application state hard and while cursors work fine in simple cases, they make non-trivial cases really complex.

Figure 5.2 depicts the application state, components and the cursors passed to components of the example application. The root component receives the complete application state and splits it to two cursors it passes down to `AuthorList` and `Game` components.

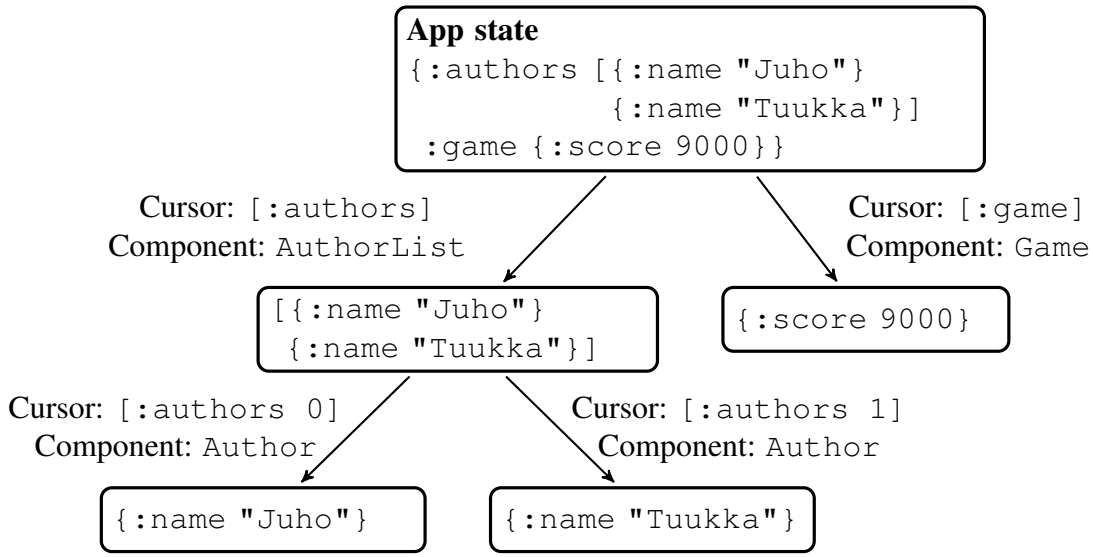

Figure 5.2: *Diagram of how Om cursors can be used to acccess subtrees of application state.*

In this simple case, the application state is structured as a tree where every node corresponds to a node in the component tree. In real-world cases, this is problematic because of two reasons: a component might require data from two branches, and it might be hard to know all data needed by all the descendants beforehand. An example would be a user

name autocomplete component in the leaf of the component tree which requires a list of all users in the system. In this case, the list of the users would need to be passed through all the components from the root component to the autocomplete component, even though the components in the path do not need the data.

Another problem with cursors is that while simple updates are easy, more complex updates are difficult to implement. For instance, `Author` component can easily read and update the author name. Besides simple updates, a common operation in a list component would be to remove items. It would make sense to render the remove button in `Author` component, so that each item has its own button. But as the `Author` component has only cursor to a single item, it cannot remove the single item from the complete list. Because of this, the remove remove functionality needs to be implemented in the `AuthorList` component and somehow triggered by remove button in `Author` component.

Important points:

- single application state
- exposes underlying React object model
- cursors are problematic.

5.2.3 Reagent

Reagent is a React wrapper for ClojureScript. Unlike Om, Reagent tries to represent idiomatic Clojure interface. This means that it hides React Component methods and the components are implemented as normal functions. Reagent provides some tools for handling the application state, but it does not have as strict conventions as Om. Reagent supports keeping the application state in a single atom, but does not enforce that. Reagent has support for cursors but also offers other solutions.

```
1  (ns dippa.reagent
2    (:require [reagent.core :as r]))
3
4  (def app-state (r/atom {:player {:name "Juho"} :game {:score 9000}}))
5
6  (defn player-component [data]
7    [:div
8     [:h1 "Hello World " (:name data)]
9     [:label "Name"]
10    [:input {:value (:name data)
11             :on-change (fn [e]
12                          (let [v (.. e -target -value)]
13                            (swap! app-state update :player assoc :name v)))}]])
14
15 (defn game-component [data]
16   [:div
17    [:h1 "Score " (:score data)]
18    [:button
19     {:on-click #(swap! app-state update-in [:game :score] inc)}
20     "Play!"]])
21
22 (defn main-component []
23   [:div
24    [player-component (:player @app-state)]
25    [game-component (:game @app-state)]])
26
27 (r/render-component [main-component] (js/document.getElementById "app"))
```

Program 5.3: Basic Reagent example

Program 5.3 contains simple Reagent sample application. In Reagent, the component contents are defined using Clojure data structures. The vector-based markup format is

called Hiccup and it is often used in Clojure to construct HTML. This format has a few benefits in comparison with the functions used by Om:

- empty attribute maps can be skipped
- attributes are defined as Clojure maps instead of JavaScript objects
- attribute names can be written in idiomatic Clojure using name-case instead of camelCase used by React.

```
1 (defn component-2 [data]
2   (let [state (r/atom {:name "Foo Bar"})]
3     (fn [data]
4       [:div
5        [:h1 "Hello " @name]
6        [:input {:type "text"
7                 :value @name
8                 :on-change #(reset! state (.. % -target -value))}]])))
```

Program 5.4: Reagent local state example

Similar to Om, the state in Reagent is managed using Clojure atoms. But Reagent has its own custom atom implementation which must be used instead of the Clojure built-in atom. In Reagent, the components can use the state from any atoms they can access, e.g. from local or other namespaces. This allows splitting the state to multiple atoms. Components can also contain local state by creating an atom in function closure, as shown in Program 5.4.

Reagent also has Functional Reactive Programming (FRP) features. As mentioned, the Reagent has custom an atom implementation often referred as ratom. Another FRP-like tool is `reaction` which wraps a computation so that it is re-run when any of its inputs change. [66]

Program 5.5 shows a simple example of using reactions. Value of reaction `b` depends on ratom `a`. When the value of `a` changes, the reaction computation is re-run and value of `b` changes. If value of `a` stays the same, e.g. `(reset! a 1)`, the computation is not run.

```
1 (def a (r/atom 5))
2 (def b (r/reaction (+ @a 2)))
3 @b ;; => 7
4 (reset! a 1)
5 @b ;; => 3
```

Program 5.5: Reagent reaction example

Input to reaction can also be another reaction. This allows chaining multiple reactions together. This can be used to run expensive computations when the inputs really change.

Important points:

- hides React model behind idiomatic Clojure code
- FRP style features

5.2.4 Re-frame

Re-frame is an architecture model and example implementation for Reagent. Reagent itself does not enforce any architecture choices and Reagent application can be structured around multiple atoms or a single atom. Re-frame forces the application state to be stored in single atom and enforces ideas how to manage state. [66]

Using a single atom for the application state is similar to Om. The benefits are the same. Most importantly the single source of truth makes it easier to reason about the state and makes the updates easier.

Re-frame provides two core features: subscriptions for reading data from the application state, and events and handlers for updating the application state. Subscriptions are an additional layer on top of Reagent reactions. Events and handlers provide clear convention how to update the application state. These features build on top of Reagent's FRP features and seek to provide higher-level utilities for using the FRP features.

The idea behind these features is to enforce unidirectional data flow. The data flow is demonstrated in Figure 5.3. The data flows in one direction forming a cycle. It can also be thought as two separate flows, *render flow* where view is constructed from application state and shown to user, and *event flow* where user actions update application state. [66]

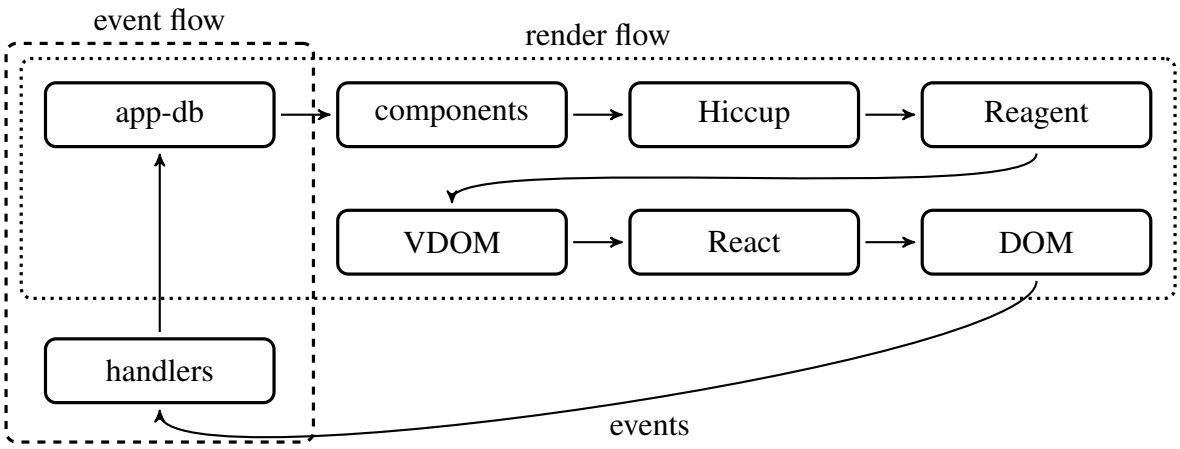

Figure 5.3: *Visualization of dataflow in Re-frame architecture. Adapted from [66].*

The first step of the render flow is the that the components depend on the application state. Components use the application state directly or through reactions and subscriptions. It is possible that the data flows through multiple computations (in reactions and subscriptions) before it is passed into a component. The rest of the render-flow happens inside Reagent. Components return the Hiccup style representation of desired DOM, Reagent turns that data into React DOM elements and React will then synchronize the changes to real DOM.

Benefits of the re-frame subscriptions over pure reactions are that they can be defined without using macros and that they allow using middleware pattern to reuse common logic.

The second flow, event flow, starts from the DOM events sent by React. In re-frame model the DOM event handlers are written to dispatch re-frame events which get handled by re-frame handlers. The advantage of handling all the events in re-frame handlers is that instead of directly accessing and updating the application state atom, the handlers are pure functions from a current state to a new state. This makes them easier to test and understand [66]. Similar to subscriptions, handlers can also use middleware pattern to reuse common logic.

Because handlers are pure functions, they cannot directly update the application state asynchronously. Instead a handler can asynchronously dispatch a new event to a second handler. The second handler will update the application state using the response from the asynchronous operation. This model is used when communicating with a server. The first handler starts an HTTP request and the second handles the response. [66] In this case, the

model simplifies the testing of the handler responsible for handling the response.

The handlers should make it easier to talk with the server and handle errors in a unified way. Still re-frame does not offer much help with how to select which data to load from the server and leaves it up to the user to decide structure for the application state.

Important points:

- two unidirectional data flows
- event handlers are pure functions
- asynchronous handlers can dispatch new events.

5.3 Fetching Data

Above libraries deal mostly with rendering data and managing the application state. The libraries do not try to provide solutions on how to fetch data from external services. In this section, the libraries which offer solutions to this are explored.

Metosin projects thus far have used REST-influenced communication where entities in the database are published as resources. In projects with CQRS style APIs, the queries are also very similar to REST model. With highly denormalized data, such as MongoDB documents this can work relatively well. When a page is loaded in the optimal case, only one document or list of documents is loaded. However, data should probably not be modeled based on the API requirements of server-client communication but based on the domain logic. In some cases, these lead to the same result.

Even when an API is modeled after the domain logic, one page in an application can require data from multiple queries. In the example project Y, one page is making seven HTTP requests to load different resources needed to show the page. This is suboptimal.

Next, some methods to fetch data are introduced. First, some previously used approaches are described followed by two declarative solutions that have not been tested yet. There are also other solutions like Netflix's Falcor, but that does not directly integrate with React so it is left out of this study.

5.3.1 Previously Used Approaches

The data fetch has to be triggered by something. Some data is used by many parts of the application and can be loaded when the application is opened. Most of the data is related to current view. The view is selected based on UI route, which often uses URI hash. This allows changing the view using normal hyperlinks. Hash change events can be listened to and they can be used to trigger the data fetches.

In the previous applications, this has been implemented using a multimethod, which has an implementation for each route. The code calls either HTTP client or Kekkonen client to start an asynchronous request to the server. After a response is received, it is stored in the application state. Usually each view has its own path inside the application state, so that data of different views does not clash.

In this model, the route change handlers define what data is loaded, and the components of the view are coupled together. If the components require some new data, the route change handler has to be updated. There are also other complications. The data does not necessarily have to be loaded each time the route changes, so the code often has to take into account what parameter changes require loading new data.

Another perspective to data fetching is how to update the application state after some backend mutations. Four different approaches can be distinguished based on the responsibilities of the backend and the frontend:

1. Frontend is responsible for knowing what was changed and how to load changed data.
2. Backend is responsible for knowing what was changed and how to pass the data to frontend, frontend updates the state.
3. Backend is responsible for knowing what queries were changed and executing the queries and passing the response to frontend which updates the state.
4. Backend is responsible for knowing what queries were changed and passes a list of the queries to frontend which executes interesting queries.

The different approaches differ also on the number and contents of the responses. In the approach 1 the mutations do not have to respond with other data besides response status. This keeps all the frontend-related code decoupled from the backend code. This is the approach used in most of the previous projects.

In the approaches 2-4 the backend has to be coupled in some degree with the frontend. The problem is that backend cannot really know what data the frontend is interested in. Making assumptions will be really hard or impossible if there are multiple clients using the same backend.

The approach 4 should be the most flexible as the backend is only passing information about changes to the frontend instead of the real changes. In this approach, the frontend can select what changes it is interested in, and only execute those queries.

If each query is loaded using a separate HTTP request, the approach might have performance issues. This could be solved by batching several queries into a single request, but this further complicates the implementation.

Previous approaches only addressed updating the UI view of the user who makes the change. Sometimes it is desired that certain updates are propagated to all the users. A WebSocket connection can be used in this case to provide a low-latency two-way communication.

To broadcast update notifications to interested users, the backend needs to keep track of the users and their subscriptions. Frontend has to define the subscription and pass it to backend. This could be done in route change handlers, where data fetching occurs.

The problem is that all these solutions are complicated to implement and requires some co-operation between backend and frontend. In the following chapter two existing solutions are explored.

5.3.2 Relay

Relay is a React architecture model and implementation from Facebook. The problems, it tries to solve is how frontend application should communicate with backend.

This is further complicated by several different client applications which might require different data: a mobile client does not need all the data used by a desktop client. HTTP request round-trips are also slow over a mobile connection so the amount of requests should be minimized.

Simple solution would be to have React root component to load all the data for all its children. This has a problem with coupling all the components to the root component. To solve this Relay introduces three concepts: [67]

- declarative data requirements
- colocated data requirements
- mutations.

Declarative data requirements mean that developer only declares the data requirements for the components and Relay will take care of loading the data automatically. This means that one does not need to write imperative code to communicate with the server. [2]

Colocation means that the queries are written next to the component code itself. This is a logical place for the data requirements and will help understanding the application. All the data should be loaded in a single request and the loading should be started before the component tree has been rendered the first time. To allow this the data requirements are defined as static methods of the components. This way the component tree can be introspected to determine the data requirements without rendering the components. Colocation is implemented by providing higher order React components: Relay Containers. Containers are a way to wrap regular React components and attach queries to them. [67, 2]

Mutations are a way to send updates from frontend to backend. They declare what data might change in response to mutation, and the data declaration can be used to automatically load changed data. Mutations also provide optimistic updates and error handling. [2]

The queries and mutations are written in a language called GraphQL. The language allows describing precisely the data requirements. This means that it is, for example, possible to select which fields of an object are required, so that no unnecessary data is loaded. It is also possible to join objects, where in a REST style API loading a list of objects could take $n + 1$ calls to load a list and object related to each list item. In GraphQL, the same data can be loaded with one query. [68]

Relay forces the components to have explicit data requirements, that is, components cannot access data even if it was required by another component. This is to prevent problems of changing the data requirements of a component affecting other components. [67]

Relay requires a backend which can respond to the GraphQL queries and mutations. The project provides a Node.js library. It is possible to either create compatible server implementation or to implement custom network implementation for a client which can talk with different server implementations. The default implementation does not use HTTP cache but instead manually caches query responses in local state. [68, 69]

Important points:

- declarative data requirements
- all data can be loaded in single request.

5.3.3 Om.next

Om.next is a ClojureScript library and architecture model which takes inspiration from Datomic database as well as Relay and Falcor libraries.

Similar to Relay, data requirements are colocated with the components as static methods so that the query can be determined without instantiating the components. Also similar to Relay, it is possible to determine the complete query without rendering the application, and load all the data in a single query. [70]

Data requirements are represented using S-expressions, that is, they can be written in Clojure. This query syntax is very similar to Datomic pull syntax. Some query features include: [70]

- selecting the fields of an object to load
- parameterized reads, for example `(:user/pic {:size :small})`
- joins.

In Relay, the GraphQL language specifies how queries work. Om.next, however, requires the developer to implement query evaluation themselves. Om.next includes a parser which turns the query into simple Abstract Syntax Tree (AST) which will be evaluated by developer provided read-function. The read function must return a value based on the given query AST. [70]

The read function can also specify that query should be sent to a server. A part of a query can be loaded from a local state while the query could also load other data from a server.

Om.next parser can be used in both ClojureScript frontend and Clojure backend. This means that the server will also implement a read function that is very similar to one in frontend. In backend, the read function will retrieve data from a database instead of local application state. [70]

The state is updated using transactions. Transactions consist of mutations and queries that will change in response to the mutation. Similar to read function, the developer will implement mutate-function that will do the real work. Mutations can both update the local application state and send the mutation to a remote server. It is possible to implement optimistic updates easily by doing both in the mutate-function: local application state is updated immediately and the same query is sent to the remote, when the remote responds, the application state is updated again if needed. [70]

Remote mutations can describe what changes they will cause, but that does not force the client to load everything that changed. Client will only load the queries included in the transaction. [70]

```
1 {:list/one [{:name "John" :points 0}
2             {:name "Mary" :points 0}]
3  :list/two [{:name "Mary" :points 0 :age 27}
4             {:name "Gwen" :points 0}]}
```

Program 5.6: Code demonstrating data in a tree format with duplicated entities. [70]

```
1 {:list/one [[:person/by-name "John"]
2             [:person/by-name "Mary"]]
3  :list/two [[:person/by-name "Mary"]
4             [:person/by-name "Gwen"]]
5  :person/by-name {"John" {:name "John" :points 0}
6                   "Mary" {:name "Mary" :points 0 :age 27}
7                   "Gwen" {:name "Gwen" :points 0}}}
```

Program 5.7: Code demonstrating normalized data in a graph format. [70]

Usually, the data is in a tree model, like in Program 5.6. While this data is easy to render, it is hard to update. In the tree model the same data might be duplicated in multiple branches. By normalizing the data, it is possible to de-duplicate the data. An example of this is displayed in Program 5.7. Om.next can automatically normalize data based on the queries of the components. [70]

For inserting new items Om.next provides a feature called temporary ids. This is copied from Datomic. Temporary ids are useful when inserting multiple items in a transaction that might need to refer to one-another. Once the transaction is executed in remote, the remote will respond with a mapping table from temporary id to real id and based on this data, the frontend will replace temporary ids in the local application state. David Nolen says this will make it easy to create applications that work in offline mode and synchronize data to remote when possible. [70]

It is also possible to update the local application state of the frontend manually outside mutate function. This allows loading streaming updates from backend, for example, over WebSocket. [70] Using the complete application query might be useful for subscribing to update notifications.

Testing frontend applications has been traditionally hard and it is often hard to test frontend on many devices. In Om.next, it is possible to use unit testing to prove that the queries and mutations work as desired. This is enough, because parsing creates a UI data tree and components are pure functions from UI data tree to DOM. This testing approach works well with property-based testing. Using test.check library, it is possible to generate transactions, similar to those triggered by a real user. [70]

Huey Petersen has used both Relay and Om.next. He wrote about his experiences with both libraries [71]. His conclusion is that Om.next does not directly compare to Relay. This is because of Om.next leaving much of the functionality up to the developer to implement, while Relay includes existing implementations. This means that Om.next is more extensible but is also a lower level tool than Relay. Petersen predicts that developers will build frameworks on top of Om.next to provide the features included in Relay.

Important points:

- normalized data in graph allows easy updates in local-state
- Datalog-like query language
- flexible but low-level
- unit testable state transitions.

5.4 Frontend Technology Choices for Example Application

Previous choices for projects have been AngularJS, Knockout, Om and Reagent. The first two being pure JavaScript solutions have already been deemed uninteresting. Om

was the first used ClojureScript frontend solution and several applications were built with it. During a larger project, many problems mentioned above were encountered with Om. The larger project was refactored to use Reagent and it proved to be more flexible. All of the Om projects maintained since have been ported to use Reagent.

While Reagent is flexible, it also has a downside that is easy to do things wrong with it. Conventions of re-frame look promising and worth to try out.

While Om.next provides very interesting features, based on a few small prototypes, it is still very unfinished. Constant breaking changes and missing documentation make it still an ineligible choice for real projects. It is, however, a project worth following closely. If the ideas in Om.next prove out to be good, they will probably be adapted for use in Reagent and re-frame and some new libraries.

Using normalized data, like in Om.next, in the application state should be easy to adapt to a Reagent application. This will be tested with the example project.

6. IMPLEMENTATION

This chapter is an overview of the implemented application in which architectural choices were tested in practice. The backend and the frontend are described in their own sections.

6.1 Backend

The planning of the backend architecture involved issues such as how to store the data and how to structure the API to allow easy interaction with ClojureScript frontend. In addition, there were previously unaddressed issues to resolve such as feeding the real-time changes to the frontend, initiated by the side-effects of commands invoked by the frontend.

6.1.1 Data Persistence

A couple of data storage solutions were evaluated in the Chapter 4. Some of them were familiar from previously implemented projects and some of them were completely unfamiliar. MongoDB and PostgreSQL are in use on multiple projects whereas event sourcing based data model and Datomic have not been tested at all. Of the tested solutions, MongoDB was dismissed mainly due to the lack of transactions and on the other hand improving document-storage capabilities of the PostgreSQL. Event sourcing based data model was rejected because of lacking libraries (or at least lacking documentation on the very few existing). Also, it would have required plenty of more time and development effort since it is rather different way to approach data modeling and how to implement software around the concept.

Datomic and PostgreSQL were picked as database choices, but the former was dropped out after having difficulties with it in some common use cases. When using Datomic, all data must have a defined schema. The Datomic Entities consists of attributes adhering to the schema. There is no easy way to store documents like it is in MongoDB and

PostgreSQL using JSON. Many projects have a use case for it such as storing varying kind of token data. Typical way to implement workflows that do not rely on login are modeled as presented in the Table 6.1. User navigates to URI where *token id* is present. The frontend then fetches the data and dispatches it to a specific handler based on the *token type*. *Token data* can be anything that frontend needs for example rendering the view.

Table 6.1: *Example of token data*

token_id	token_type	token_data
B2viiotW5R3LyU5dIuKX	password-reset	`{"account-id": 1}`
ojH1Q9ln6mO6VggkeDEr	password-reset	`{"account-id": 2}`

Another use case would be the logging of the events where the data column can be anything. Of course, it would be possible to define a schema for every token and event type but that would reduce the flexibility significantly making the implementation more burdensome.

Datomic has a predefined set of types available and for example it lacks very common types such as local dates or plain dates. So far, there is no mechanism to extend types even though plans to implement type extension were mentioned in the September 2012 [72].

During the early phase of implementation, a decision was made to use PostgreSQL since it satisfies the above-mentioned requirement of storing document-like data in a flexible manner. Also, it has plenty of types available and allows defining new data types. Bottled Water, an extension to capturing change data for PostgreSQL, was experimented with to preserve the history.

The Bottled Water project is at alpha stage and setting it up required some experimenting even when the project page offered quick-start Docker images for easier setup. Documentation was rather minimalistic but through trial and error, setup and usage succeeded. For example, setting up required debug-level logging of third party libraries to figure out the options that must be given when running Docker images to get the system running properly.

First observation was that the Bottled Water does not support PostgreSQL schemas[1]. PostgreSQL schemas can be used to group e.g. database tables to separate logical groups with their own privileges [73]. For example, application domain related data could be in its own schema whereas some system-level data with more restricted privileges could reside in a dedicated schema. Another example would be the separation of the data on a multi-tenant application. This allows using the same database instance for several (groups of) users isolated from each other. With Bottled Water, this becomes impossible since schema part is left out whenever change is forwarded to Kafka.

Second observation was that with Bottled Water the infrastructure becomes rather complex. To make use of the Bottled water, following parts are required:

- Bottled Water extension to PostgreSQL
- Apache Kafka messaging system
- Apache ZooKeeper for centralized coordination
- Confluent Schema Registry for Avro schema retrieval.

In addition to a relatively great number of infrastructure-related dependencies, the application software requires a Kafka client and it needs to communicate with the Schema Registry to get the Avro schema for Kafka message decoding. Also, mapping Avro-decoded data back to Clojure data structures increases complexity.

However, after setting up the infrastructure and writing little amount of code to read changes from Kafka topics, change data is available to use. Implemented software can be configured to listen to any Kafka topic. For each topic, a handler can be implemented that will be invoked when reading a message for that particular topic. The message is coerced to Clojure data before invoking the handler with the message as a parameter.

[1] PostgreSQL schema is a somewhat confusingly named feature where tables, functions etc. can be located in different namespaces. Not to be mixed with a term *database schema* that might be used to refer the organization of the database in general.

```
1  CREATE TABLE account (
2    account_id SERIAL PRIMARY KEY,
3    name TEXT
4  );
5
6  INSERT INTO account(name) VALUES ('tuukka.t.kataja@student.tut.fi');
7
8  UPDATE account
9  SET name='tuukka.kataja@metosin.fi'
10 WHERE account_id=1;
```

Program 6.1: Example SQL to demonstrate change data capturing.

When executing SQL presented in Program 6.1, a handler for the *account* topic would be called with the following coerced Clojure data structures given as a parameter:

- `{:account_id 1 :name "tuukka.t.kataja@student.tut.fi"}` after the SQL insert
- `{:account_id 1 :name "tuukka.kataja@metosin.fi"}` after the SQL update

For example, in the project built for this thesis, this data is broadcasted to connected clients using WebSockets to enable real-time updates in the user interface.

While Bottled Water works for communicating the side-effects of commands, it makes the infrastructure and architecture more complex. Instead of using Bottled Water for sending change data, changes could be simply manually emitted in the handler code.

```
1  (defnk ^:command add-account!
2    {:requires-session true}
3    [[:ctx db]
4     account :- domain/Account]
5    (let [added-account (first (jdbc/insert! db :app.account account))]
6      (success added-account)))
```

Program 6.2: Example of a Kekkonen handler for adding new account.

In the Program 6.2, handler for adding an account into the system is shown. The handler function gets an account map as one parameter and it inserts it into the database. The new account is immediately returned to the client. Behind the scenes, Bottled Water

extension sends the data to Kafka and the implemented system consumes it and sends it over WebSocket to the connected clients.

```
1 (defnk ^:command add-account!
2   {:requires-session true}
3   [[:ctx db ws]
4    account :- domain/Account]
5   (let [added-account (first (jdbc/insert! db :app.account account))]
6     (websocket/notify! ws :account-added added-account)
7     (success added-account)))
```

Program 6.3: Example of a Kekkonen handler for adding new account with explicit WebSocket broadcast.

In the Program 6.3, an alternative implementation for `add-account!` -command is shown. Instead of attempting to automatically communicate the changes, a notify function is called so that everyone listening to the `:account-added` topic will receive the created account. This solution would remove the infrastructure dependencies (Kafka, ZooKeeper etc.) and make the software simpler. Also, this mechanism does not depend on PostgreSQL and Bottled Water at all. While Bottled Water seemed interesting at first, it does not only make the infrastructure more complicated, but it either is not really suitable where it was thought to be a good fit.

The Bottled Water approach could still be useful in some other use cases. For example, if the data needs to be stored in some other data storage, Kafka consumer could listen to the changes and reconstruct the data accordingly. For example, if there were a separate data storage for reporting purposes, changes could be consumed in the reporting system.

6.1.2 API

API of the implemented application is built using the Kekkonen library instead of the Compojure-api that has been used in many previous projects. One motivation for this was to allow the more straightforward exposal of domain data and functions to the frontend without the additional burden of mapping them to the resource-centric model.

API is separated into commands and queries. Side-effecting operations such as adding a new project and creating a user are annotated as commands whereas operations that e.g. return data without side-effects are annotated as queries. These operations are plain

Clojure functions residing in namespaces that are given to Kekkonen making them available for clients to invoke.

Data is transmitted between the server and clients in transit format, making a richer set of data structures available than, for instance, plain JSON offers. This makes e.g. transferring of the Clojure data structures such as keywords, sets, maps between the server and the client easy.

Kekkonen has an extension mechanism called interceptors. Interceptors are defined as maps that have *enter* and *leave* functions taking invocation context as a parameter. Invocation context includes e.g. information about:

- type of the invoked handler (command or query)
- the fully qualified name of the handler
- request map presenting the HTTP request.

Interceptors can be chained and there is an example of an API that is constructed using two interceptors in the Program 6.4.

The enter functions of interceptors are executed in the order as defined in the map given to *cqrs-api* function in the Program 6.4 and leave methods in reverse order. Interceptors are a powerful extension mechanism and for example simple audit logging was trivial to implement since invocation context contains information about the invoked handler such as the name of the handler and data passed to it.

```
1  (def hello-interceptor
2    {:enter (fnk [:as ctx] (logging/infof "Hello from interceptor!") ctx)
3     :leave (fnk [:as ctx] (logging/infof "Bye from interceptor!") ctx)})
4
5  (def audit-interceptor
6    {:enter (fnk [:as ctx]
7              (assoc ctx :audit/start-time (System/currentTimeMillis)))
8     :leave (fnk [[:kekkonen.core/handler action type] data request :as ctx]
9              (let [account-id (get-in request [:identity :account-id])
10                   now (System/currentTimeMillis)
11                   duration-ms (- now (:audit/start-time ctx))]
12               (logging/infof "AUDIT: %s" {:account-id account-id
13                                           :duration-ms duration-ms
14                                           :type type
15                                           :action action
16                                           :data data})
17               ctx))})
18
19 (kekkonen/cqrs-api {:core {:handlers {:accounts 'backend.accounts}
20                           :interceptors [hello-interceptor
21                                          audit-interceptor]}})
```

Program 6.4: Example of Kekkonen CQRS API and interceptors

```
2016-03-13 17:28:51.532 +0200 INFO  [backend.handler] - Hello from interceptor!
2016-03-13 17:28:51.534 +0200 INFO  [backend.handler] - AUDIT: {:account-id 1,
  ↪  :duration-ms 2, :type :query, :action :accounts/get-entries, :data
  ↪  {:account-id 1}}
2016-03-13 17:28:51.534 +0200 INFO  [backend.handler] - Bye from interceptor!
```

Program 6.5: Log output produced by interceptors when invoking Kekkonen handler defined in the listing 6.4

In addition to CQRS API, there is a WebSocket-based streaming API to push real-time changes to the frontend. As mentioned in the description of database implementation, database changes are pushed to connected clients using WebSockets. When a client has initiated a WebSocket connection to the server, changes are sent as transit-encoded Clojure maps with a *type* tag to allow frontend to act based on the type of the message. For example, if the project is modified by the command invoked by the client, change is streamed:

```
{:type :project-modified :project_id 1 :name "Diplomityö" :archived nil}
```

and the client can e.g. update its state to render the updates.

6.1.3 Backend Implementation Conclusion

Multiple alternative data storing solutions were considered. Previously used MongoDB was dismissed in favor of a more robust ACID-compliant database. PostgreSQL was considered a robust choice for many needs. It could replace MongoDB even when storing mostly document-like data due to its improving JSON-storing capabilities. Datomic was chosen as a more experimental alternative but during the technical spike it proved out to be rather challenging in some quite typical use cases. Event Sourcing was also considered but it would have been a radically different approach and at least currently lacks well-documented libraries for Clojure. Bottled Water extension to PostgreSQL was experimented to capture data changes which could be used to store history and emit real-time events to clients. However, it made the infrastructure complicated and did not fit the use case that well.

In addition to testing different data storage solutions, new API library Kekkonen was also tested. It brought more streamlined development experience and interaction between the

server and the client while taking the good parts out of the previously used Compojure-api library such as documentation generation. Interceptors allow extensibility and for instance implementing simple audit logging was rather trivial. CQRS API was augmented with WebSocket API to provide real time capabilities between the client and the server.

6.2 Frontend

The technologies for the example application were chosen from technologies introduced in Chapter 5. Relay was rejected as it does not fit into our Clojure ecosystem. Though it is possible to wrap Relay for ClojureScript use, it does not leverage Clojure data structures. Om.next was rejected as we felt that it is not yet ready for production use, based previous experiments by the second author. Re-frame was not previously used comprehensively so that was chosen for the test application. Especially the conventions re-frame provides over Reagent are interesting, as that was found to be lacking with the previous projects.

Because Relay and Om.next were rejected, data fetching was mostly implemented as before. However, based on the ideas learned from Relay and Om.next, some conventions were introduced to the application state structure.

Live updates were also found to be an area which required more study, as one project would have benefited greatly from the knowledge. Because of this, live updates were tested in the example application.

Re-frame provides conventions for state management by enforcing keeping the application state in a single atom. It also recommends using subscriptions for accessing the application state. However, it is still possible to directly access the application state atom. This is useful if one wants to use new track-feature introduced in Reagent 0.6.0, which is in some ways replacement for re-frame subscriptions. Handlers provide a convention for updating the application state.

6.2.1 Data Fetching Implementation

Frontend uses the same kind of routing implementation as previous projects. URI hash fragment is used to select a view and the view has a associated route change function. The function loads the data required by the view.

The application state was modeled as a graph, like in Om.next. Entities are stored in maps

and indexed by identifiers. The identifier can be used to refer to the entity from elsewhere in the application state. Because the entity only exists on one place in the state, it is easy to update.

```
1  (register-handler :load-account-entries
2    (fn [db [_ account-id]]
3      (a/go
4        (let [resp (a/<! (cqrs/query :accounts/get-entries {:account-id account-id}))]
5          (if (cqrs/success? resp)
6            (dispatch [:set-account-entries account-id (:body resp)])))))
7      db))
8
9  (register-handler :set-account-entries
10   (fn [db [_ account-id account-entries]]
11     (let [entries (index-by :account-project-entry-id account-entries)]
12       (assoc-in db [:account-entries account-id] entries))))
```

Program 6.6: Example of data fetching frontend code

Program 6.6 shows an example of two event handlers. The first event handler starts a request using Kekkonen to the backend. The request is made asynchronously using Core.async. The dispatch call in line 6 is called asynchronously after the response has been retrieved. The second handler updates the application state using the response. In line 11, the account-entries are converted to a map, this is to enable the data normalization.

When a command is executed, the frontend is responsible for determining and loading the data that was changed. Figure 6.1 shows the high-level view of what parts of the system take part in handling a command from the frontend.

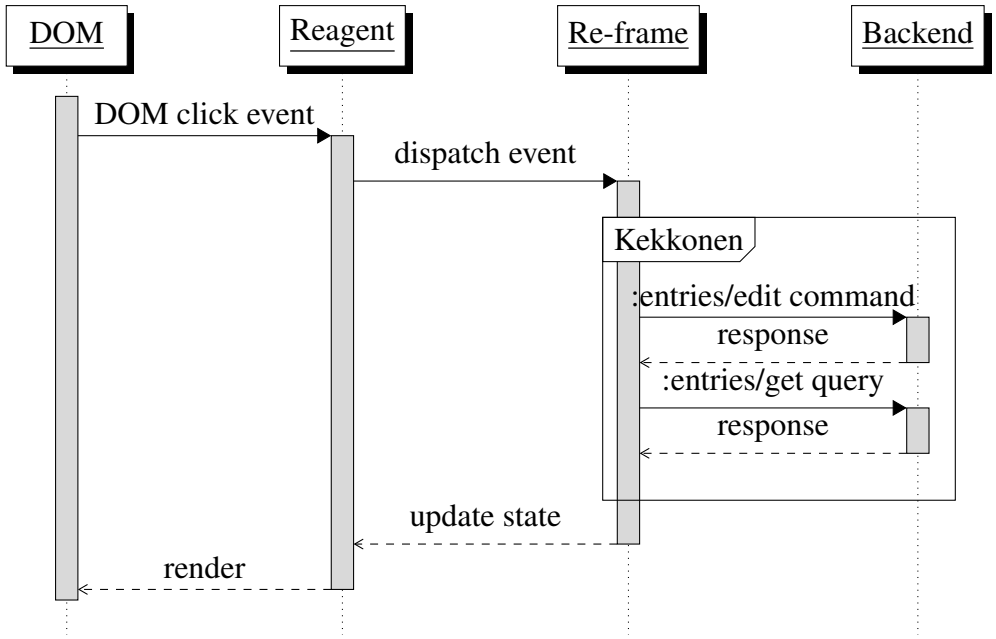

Figure 6.1: *Sequence diagram depicting how command triggered from user interface is processed.*

```
1  (register-handler :edit-entry
2    (fn [db [_ {:keys [data]}]]
3      (a/go
4        (let [resp (a/<! (cqrs/command :entries/edit data))]
5          (when (cqrs/success? resp)
6            (dispatch [:load-entry (:entry-id data)])))))
7      db))
8
9  (register-handler :load-entry
10   (fn [db [_ entry-id]]
11     (a/go
12       (let [resp (a/<! (cqrs/query :entries/get {:entry-id entry-id}))]
13         (when (cqrs/success? resp)
14           (dispatch [:update-entry-to-db (:body resp)])))))
15     db))
16
17 (register-handler :update-entry-to-db
18   (fn [db [_ {:keys [entry-id] :as entry}]]
19     (let [account-id (get-in db [:session :account])]
20       (assoc-in db [:account-entries account-id entry-id] entry))))
```

Program 6.7: Example frontend code for remote mutation implementation

The Program 6.7 shows the event handlers used to save the modified entries to the backend. The first handler, `edit-entry`, starts an asynchronous HTTP request to the backend using Kekkonen. After the response is received another event is triggered. The first handler does not touch the application state and just returns the untouched application state forward. It would be possible to implement optimistic updates here by updating the application state using the parameters of the event. The second handler is triggered by the success of the response and will reload the changed entry using a query. The third handler will save the response into the application state. Updating a single entry in the application state is easy because the entries are stored in a map indexed by an identifier.

While the normalized application state allows easy updates, it makes the reads somewhat harder. Each time the state is read for rendering, the possible links need to be followed. Thanks to Reagent reactions and re-frame subscriptions, this is however easy to implement outside of the components.

```
1  (register-sub :selected-entries
2    (fn [db _]
3      (let [calendar-account (subscribe [:selected-calendar-account])
4            account-entries (subscribe [:account-entries])]
5        (reaction (get @account-entries @calendar-account)))))
6
7  (register-sub :entries-by-date
8    (fn [db _]
9      (let [entries (subscribe [:selected-entries])]
10       (reaction (group-by :entry-date (vals @entries))))))
```

Program 6.8: Example of re-frame subscription used built data for components from application state

Program 6.8 contains code which is responsible for taking the application state and using it to create the data for components. The first subscription depends on the selected calendar user, which is either current logged-in user or user selected from a dropdown. It also depends on the entries of all the users and selects only the entries of the selected user. Second subscription groups the entries to a map indexed by entry dates.

6.2.2 Live Updates

WebSocket-based implementation providing live updates was tested. In this solution, the backend sends update notifications to every client. This was implemented using Post-

greSQL together with Bottled Water which passes updates through Kafka to the Clojure backend. The backend retrieves the data associated with the update and broadcasts this to all clients. In the frontend a re-frame handler is called based on the updated entity and this handler updates the application state accordingly.

In this simple test, this approach worked successfully. However, this solution has multiple constraints. The backend implementation depends on the database for update events. Updates are based on the database schema, that is, the update events correspond to database rows. The data, the frontend is, interested in is often in different form. Also, in this example application, all the updates were sent to every client. In other applications, every client would not be interested in every change, and it might even be important that clients do not see data they are not authorized to access.

6.2.3 Component Schema Validation

One minor annoyance with Reagent is that it is easy to provide components with parameters of a wrong kind. As Clojure is not statically typed language, the errors are not found until runtime and can cause UI breakage.

Prismatic Schema is used heavily in the backend to validate e.g. the user input. Schema can also be used to annotate functions, their parameters and their return value. As components in Reagent are plain functions, it is possible to use this to validate component parameters.

```
1 (p/defnk entry
2   [entry :- Entry
3    account-id :- s/Str]
4   ...)
```

Program 6.9: Example of Schema annotated Reagent Component function

Program 6.9 contains an example of Schema-annotated Reagent component. The component function is defined here using an alternative `defnk`-macro, which allows annotating functions taking a single map as an argument more easily. This function takes as a parameter a map with three required keys: entry with the value which must be of an `Entry`-type and account identifier of the current user.

When Schema validation fails, the default error messages in the browser are hard to decipher. This was solved in the example project by catching the validation exceptions and displaying the error in a prettier way. This helps catching bad component parameters faster. A problem remains that validation errors break React rendering loop so that the application has to be completely reloaded after such an error.

6.2.4 Frontend Implementation Conclusion

While Relay and Om.next are promising, it seems that they are only the first solutions to the problem. It will take time until they are production ready and new solutions might still arise.

In addition to Om.next being unfinished, based on the previous prototypes using it, it also seems to have a long learning curve. This is because the developer has to implement so much of the functionality.

The ideas in Om.next and findings from the example project have generated some new ideas about data fetching. In Relay and Om.next, the data requirements are colocated with the components. This causes additional complexity in the implementation. The reason is that the requirements need to be static properties of the components.

In Relay and Om.next, all the steps in rendering the UI are pure functions. In those cases, the data requirements are a function of Component tree. By defining the data requirements as a function of route data it could be possible to implement simple declarative data fetching.

In this thesis, the research focused on Clojure(Script) ecosystem and possibly this might have filtered out some good approaches. FRP community and especially Elm programming language community are good candidates for ideas worth exploring.

Re-frame was rather pleasant to use and proved that re-frame handlers are really useful in comparison with writing event handlers inline or in ad-hoc functions. Re-frame subscriptions are not as useful. Most use cases are better handled by track function introduced in Reagent 0.6.0.

7. ARCHITECTURE EVALUATION RESULTS

Architecture-related decisions were evaluated using the DCAR method presented in the Chapter 3 with minor modifications:

- Business-related parts were left out since there was no business aspect related to example project build for this thesis.
- Decisions were not prioritized since their number was constrained.
- Decision documentation was prepared beforehand to make the session as lightweight as possible. Documentation was finalized during the review.

7.1 Evaluation Session and Results

All of the major architectural decisions were evaluated using the DCAR method. The method was even more lightweight than it would have been in a real-life project since some parts were omitted. In a real project, business matters would have been taken into account. Also, a real project would probably have greater amount of decisions that should have been prioritized.

Three software developers from Metosin participated in the session. One of the reviewers was the second author of this thesis and two were participating outside the project. Decisions were documented before the session and participants had the draft of this thesis available before the session to allow them to familiarize themselves with the architecture, the decisions and to some extent the considered alternatives.

During the session, decision documentation was augmented while discussing about the decisions. After a brief conversation about the decisions, each participant gave their rationale and outcome. Detailed decision documentation can be found in the Appendix B and the summary of the evaluation session is on the Table 7.1.

As seen on the Table 7.1, three out of four of the backend-related decisions were considered positive (✓) and one out of four was negative (✗). In the frontend-related decisions,

Table 7.1: *Summary of the results of the Decision-Centric Architecture Review*

Decision	Outcome	Conclusion
Backend		
PostgreSQL as database	✓	ACID-compliant, feature-rich database was considered as only viable option out of the alternatives.
Bottled Water extension	✗	Infrastructure-complicating, unproved alpha-quality software with no fit for the intended use case.
Kekkonen CQRS API	✓	Simple backend-frontend communication due to simplicity compared to RESTful API.
WebSocket API	✓	With little more complexity, adds real-time capability for backend-frontend communication.
Frontend		
Normalized application state	✓	Prefer easier updates over easier reads to avoid e.g. buggy reads and hard-to-reason logic.
Re-frame dispatchers & handlers	✓	Good documentation and conventions makes it easier for new developers to join teams.
Re-frame subscriptions	=	Neutral outcome since alternatives offer easier solutions but this one comes with the library anyway.
Component Schema validation	=	Could be useful since validation could be removed from production build. Should be librarized.

two out of four decisions were considered as positive and the rest were considered neutral (=).

Backend-related decisions and the evaluation results show that the usage of PostgreSQL as a database was considered the most reasonable choice among the alternatives. However, Bottled Water extension to PostgreSQL proved out to be too immature for real usage and it was not suitable for the purposed use. During the projects, more streamlined backend-frontend communication has been desired and Kekkonen CQRS API provides that. During the writing of this thesis it has been introduced in production usage and the development will be continued. As mentioned, the Bottled Water did not work very well

when serving the frontend real-time changes based on the database changes. However, WebSocket APIs should be used to augment the HTTP APIs with real-time capabilities.

Frontend-related decisions and corresponding evaluation results indicate that normalized application state should be embraced. This helps avoiding duplicated data in the frontend state. Duplicated data can lead to stale data being present in the application state. For example, if the same object is present in two paths but the other one is not updated for some reason. In previous projects stale data has caused some confusing problems.

Experiences from the previous projects are that more documented solutions and improved conventions are required in the frontend development to e.g. alleviate the difficulties when new developers join the project. Re-frame proved out to be a solution offering both the documentation and conventions and it should be used. However, some parts of it could be replaced with better solutions. One example is re-frame subscriptions that let components to subscribe for state changes that could be substituted by Reagent's track functionality. However, more important is to choose one way and keep using it. Component schema validation was considered useful for improving the development experience. However, it adds some complexity and should be librarized.

8. CONCLUSION

The purpose of this thesis was to create improved architecture for web-based Clojure applications, based on the shortcomings found in previous projects. Focal points for architectural improvements originated from these past projects. After setting goals for improvements, different solutions and techniques were evaluated. Then an example application was implemented by using the selected approaches. The architecture of the example application was reviewed and the decisions made were documented and evaluated. The result of this thesis was a collection of documented decisions that can be adapted to future projects.

During the writing of this thesis, some of the ideas were adapted into on-going projects proving the concrete usefulness of some results. It is recommended that forthcoming projects adapt even more decisions since that would provide additional value for a little effort. The results would have had only minor impact on two of the cases projects whereas in one case there would have been a notable improvement in the development process and the overall architecture.

The resulting reference architecture was surprisingly similar to those in existing projects. The expectations were set high for some of the evaluated technologies but for various reasons those had to be dismissed. However, it was still valuable to evaluate those technologies during the writing process even if they were not considered useful. In the future, further development in those technologies might make them suitable for use. In addition, lightweight architecture evaluation was considered useful and should be done in real projects whenever possible. The reference architecture and decisions can be further maintained during future projects. Based on these findings new and existing open-source libraries will be developed.

BIBLIOGRAPHY

[1] *React | A JavaScript library for building user interfaces*. [Accessed 28.4.2016]. Available: https://facebook.github.io/react/.

[2] *Relay | A JavaScript framework for building data-driven React applications*. [Accessed 25.3.2016]. Available: https://facebook.github.io/relay/.

[3] *Falcor | A JavaScript library for efficient data fetching*. [Accessed 28.4.2016]. Available: http://netflix.github.io/falcor/.

[4] *Scala.js - A safer way to build robust front-end web applications!* [Accessed 28.4.2016]. Available: https://www.scala-js.org.

[5] *ClojureScript*. [Accessed 28.4.2016]. Available: http://clojure.org/about/clojurescript.

[6] *Clojure - Rationale*. [Accessed 18.1.2016]. Available: http://clojure.org/about/rationale.

[7] *Clojure - State*. [Accessed 18.1.2016]. Available: http://clojure.org/about/state.

[8] C. Emerick, B. Carper & C. Grand. *Clojure Programming*. O'Reilly, 2012, p. 630. ISBN: 9781449394707.

[9] *Rationale · clojure/clojurescript Wiki*. [Accessed 25.1.2016]. Available: https://github.com/clojure/clojurescript/wiki/Rationale.

[10] *Introduction to MongoDB*. [Accessed 28.4.2016]. Available: https://docs.mongodb.org/manual/introduction/.

[11] *AngularJS — Superheroic JavaScript MVW Framework*. [Accessed 28.3.2016]. Available: https://angularjs.org/.

[12] J. Webber, S. Parastatidis & I. Robinson. *REST in Practice: Hypermedia and Systems Architecture*. 1st. O'Reilly Media, Inc., 2010. ISBN: 9780596805821.

[13] *Om*. [Accessed 28.4.2016]. Available: https://github.com/omcljs/om.

[14] *Reagent: Minimalistic React for ClojureScript*. [Accessed 15.1.2016]. Available: http://reagent-project.github.io/.

[15] *About*. [Accessed 28.4.2016]. Available: http://www.postgresql.org/about/.

[16] *PostgreSQL 9.4.5 Documentation - JSON Types - jsonb Indexing*. [Accessed 14.1.2016]. Available: `http://www.postgresql.org/docs/9.4/static/datatype-json.html#JSON-INDEXING`.

[17] *CQRS*. [Accessed 28.4.2016]. Available: `http://martinfowler.com/bliki/CQRS.html`.

[18] U. van Heesch et al. Decision-Centric Architecture Reviews. *IEEE Softw.* 31.1 (Jan. 2014), pp. 69–76. ISSN: 0740-7459.

[19] *Atomicity and Transactions — MongoDB Manual 3.2*. [Accessed 15.1.2016]. Available: `https://docs.mongodb.org/manual/core/write-operations-atomicity/#transaction-like-semantics`.

[20] *lookup (aggregation) — MongoDB Manual 3.2*. [Accessed 15.1.2016]. Available: `https://docs.mongodb.org/manual/reference/operator/aggregation/lookup/#pipe._S_lookup`.

[21] *Monger, a MongoDB Clojure client for a more civilized age | MongoDB library for Clojure*. [Accessed 15.1.2016]. Available: `http://clojuremongodb.info`.

[22] *Monger, a Clojure MongoDB client: Inserting documents | MongoDB library for Clojure*. [Accessed 15.1.2016]. Available: `http://clojuremongodb.info/articles/inserting.html`.

[23] *Monger, a Clojure MongoDB client: querying the database | MongoDB library for Clojure*. [Accessed 15.1.2016]. Available: `http://clojuremongodb.info/articles/querying.html`.

[24] *PostgreSQL: About*. [Accessed 15.1.2016]. Available: `http://www.postgresql.org/about/`.

[25] *What's new in PostgreSQL 9.2 - PostgreSQL wiki*. [Accessed 15.1.2016]. Available: `https://wiki.postgresql.org/wiki/What's_new_in_PostgreSQL_9.2#JSON_datatype`.

[26] *What's new in PostgreSQL 9.3 - PostgreSQL wiki*. [Accessed 15.1.2016]. Available: `https://wiki.postgresql.org/wiki/What's_new_in_PostgreSQL_9.3#JSON:_Additional_functionality`.

[27] *What's new in PostgreSQL 9.4 - PostgreSQL wiki*. [Accessed 15.1.2016]. Available: `https://wiki.postgresql.org/wiki/What's_new_in_PostgreSQL_9.4#JSONB_Binary_JSON_storage`.

[28] *What's new in PostgreSQL 9.5 - PostgreSQL wiki*. [Accessed 15.1.2016]. Available: https://wiki.postgresql.org/wiki/What's_new_in_PostgreSQL_9.5#JSONB-modifying_operators_and_functions.

[29] M. Linster. *Postgres Plus: the EDB blog | EnterpriseDB*. [Accessed 15.1.2016]. Available: http://www.enterprisedb.com/postgres-plus-edb-blog/marc-linster/postgres-outperforms-mongodb-and-ushers-new-developer-reality.

[30] *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627. June 2003. 10 pp.

[31] *Architecture Overview | Datomic*. [Accessed 25.1.2016]. Available: http://docs.datomic.com/architecture.html.

[32] *Entities | Datomic*. [Accessed 25.1.2016]. Available: http://docs.datomic.com/entities.html.

[33] *Schema | Datomic*. [Accessed 25.1.2016]. Available: http://docs.datomic.com/schema.html.

[34] *Datomic Queries and Rules | Datomic*. [Accessed 25.1.2016]. Available: http://docs.datomic.com/query.html.

[35] *Transactions | Datomic*. [Accessed 25.1.2016]. Available: http://docs.datomic.com/transactions.html.

[36] H. Hübner. *Datomic in Practice*. [Accessed 16.2.2016]. Available: https://skillsmatter.com/skillscasts/7228-datomic-in-practice.

[37] M. Fowler. *Event Sourcing*. [Accessed 18.1.2016]. Available: http://martinfowler.com/eaaDev/EventSourcing.html.

[38] *Event Sourcing Basics — Event Store*. [Accessed 20.1.2016]. Available: http://docs.geteventstore.com/introduction/event-sourcing-basics/.

[39] G. Young. *CQRS/DDD by Greg Young*. [Accessed 20.1.2016]. Available: https://www.youtube.com/watch?v=KXqrBySgX-s.

[40] *Event Store*. [Accessed 20.1.2016]. Available: https://geteventstore.com/.

[41] M. Kleppmann. *Bottled Water: Real-time integration of PostgreSQL and Kafka*. [Accessed 15.1.2016]. Available: `http://www.confluent.io/blog/bottled-water-real-time-integration-of-postgresql-and-kafka/`.

[42] *Logical Decoding Concepts*. [Accessed 15.1.2016]. Available: `http://www.postgresql.org/docs/9.4/static/logicaldecoding-explanation.html`.

[43] *Apache Avro™ 1.8.0 Specification*. [Accessed 20.1.2016]. Available: `https://avro.apache.org/docs/current/spec.html`.

[44] *Apache Kafka*. [Accessed 20.1.2016]. Available: `http://kafka.apache.org/documentation.html#introduction`.

[45] *rill-event-sourcing/rill: Clojure Event Sourcing toolkit*. [Accessed 25.1.2016]. Available: `https://github.com/rill-event-sourcing/rill`.

[46] *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. 176 pp.

[47] *Hypertext Transfer Protocol version 2*. RFC 7540. May 2015. 96 pp.

[48] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures". AAI9980887. PhD thesis. 2000.

[49] *plumatic/schema - Clojure(Script) library for declarative data description and validation*. [Accessed 3.2.2016]. Available: `https://github.com/plumatic/schema`.

[50] B. Nelson. *Remote Procedure Call*. CMU-CS. Xerox Palo Alto Research Center, 1981.

[51] R. H. Arpaci-Dusseau & A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. [Accessed 3.2.2016]. Arpaci-Dusseau Books, May 2015, p. 686. ISBN: 9781105979125.

[52] *JSON-RPC 2.0 Specification*. RFC. [Accessed 3.2.2016]. Available: `http://www.jsonrpc.org/specification`.

[53] T. Reiman & J. Teperi. *Basics*. [Accessed 17.2.2016]. Available: `https://github.com/metosin/kekkonen/wiki/Basics`.

[54] O. Lewis. *Hybrid Microservices*. [Accessed 27.1.2016]. Available: `http://owainlewis.com/articles/hybrid-microservices`.

[55] S. Loreto et al. *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*. RFC 6202. 2011.

[56] I. Fette & A. Melnikov. *The Websocket Protocol*. RFC 6455. 2011.

[57] *Web Sockets*. [Accessed 18.2.2016]. Available: `http://caniuse.com/#feat=websockets`.

[58] *600k concurrent HTTP connections, with Clojure & http-kit*. [Accessed 19.2.2016]. Available: `http://www.http-kit.org/600k-concurrent-connection-http-kit.html`.

[59] *State of Clojure 2015 Survey Results*. [Accessed 28.4.2016]. Available: `http://blog.cognitect.com/blog/2016/1/28/state-of-clojure-2015-survey-results`.

[60] M. Mikowski & J. Powell. *Single Page Web Applications: JavaScript End-to-end*. 1st. Manning Publications Co., 2013, p. 432. ISBN: 9781617290756.

[61] F. Rangel. *React Under the Hood*. Leanpub, Nov. 2015.

[62] G. E. Krasner & S. T. Pope. A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. *J. Object Oriented Program.* 1.3 (Aug. 1988), pp. 26–49. ISSN: 0896-8438. Available: `http://dl.acm.org/citation.cfm?id=50757.50759`.

[63] J. Vlissides et al. *Design patterns: Elements of reusable object-oriented software*. 18th ed. Reading: Addison-Wesley, 1995, p. 395. ISBN: 0201633612.

[64] D. Nolen. *The Functional Final Frontier, Clojure/West 2014*. [Accessed 2.2.2016]. Mar. 2014. Available: `https://www.youtube.com/watch?v=DMtwq3QtddY`.

[65] P. Jaros. *Why We Use Om, and Why We're Excited for Om Next | The CircleCI Blog*. [Accessed 25.1.2016]. Available: `http://blog.circleci.com/why-we-use-om-and-why-were-excited-for-om-next/`.

[66] *re-frame: Derived Values, Flowing*. [Accessed 25.1.2016]. Available: `https://github.com/Day8/re-frame/blob/master/README.md`.

[67] *Thinking In Relay | Relay Docs*. [Accessed 25.3.2016]. Available: `https://facebook.github.io/relay/docs/thinking-in-relay.html`.

[68] *Thinking In GraphQL | Relay Docs*. [Accessed 25.3.2016]. Available: `https://facebook.github.io/relay/docs/thinking-in-graphql.html`.

[69] *Network Layer | Relay Docs*. [Accessed 26.3.2016]. Available: `https://facebook.github.io/relay/docs/guides-network-layer.html`.

[70] D. Nolen. *Om Next, Clojure/conj 2015*. [Accessed 26.3.2016]. Nov. 2015. Available: `https://www.youtube.com/watch?v=MDZpSIngwm4`.

[71] H. Petersen. *om.next from a Relay / GraphQL Perspective*. [Accessed 25.3.2016]. Available: `http://hueypetersen.com/posts/2016/02/13/om-next-from-a-relay-graphql-perspective/`.

[72] *Dates with timezones*. [Accessed 28.2.2016]. Available: `https://groups.google.com/forum/#!msg/datomic/0V5Ima9fw88/qNzjQBZEF5kJ`.

[73] *Schemas*. [Accessed 12.3.2016]. Available: `http://www.postgresql.org/docs/9.4/static/ddl-schemas.html`.

A. CONTRIBUTIONS

1. Introduction **[both]**
2. Background **[both]**
3. Decision-Centric Architecture Review **[both]**
4. Evaluating Data Persistence and API Solutions **[Tuukka, 22 pages]**
5. Evaluating Frontend Technologies **[Juho, 20 pages]**
6. Implementation
   (a) Backend Implementation **[Tuukka, 9 pages]**
   (b) Frontend Implementation **[Juho, 6 pages]**
7. Architecture Evaluation **[both]**
8. Conclusion **[both]**

B. ARCHITECTURE DECISIONS

Architecture decisions are documented here using the DCAR template as basis. Format of the documentation for single decision is shown below.

Name	*Name of the decision*		
Problem	*Description of the problem*		
Solution / description of decision	*Description of the solution*		
Considered alternative solutions	*Description of alternative solutions*		
Argument in favor of decision	*List of arguments in favor of decision*		
Argument against the decision	*List of arguments against the decision*		
Outcome and rationale	*Rationale for positive outcome*	*Rationale for neutral outcome*	*Rationale for negative outcome*

Example documentation for decision

Name	PostgreSQL as database		
Problem	Selecting a database that enables flexible data modeling. Robustness, reliability, ACID properties are high priority. Storing and retrieving Clojure domain objects should be as straightforward as possible.		
Solution / description of decision	PostgreSQL is a proven ACID-compliant relational database with improving support for JSON to enable storage of document-like data. Mapping Clojure data to relational model and vice versa needs some additional work compared to working with e.g. MongoDB but being otherwise more solid choice, PostgreSQL is preferred over MongoDB. However, if solved domain problem requires storing simple document-like data without a need for complex transactions, MongoDB should be considered as a choice. Also, if the problem naturally maps to e.g. graph, look for a graph database instead.		
Considered alternative solutions	Using MongoDB or Datomic as a database.		
Argument in favor of decision	• Proven, feature-rich ACID-compliant open-source database • Bottled Water extension to allow streaming of change data		
Argument against the decision	• Working with Clojure data not as streamlined as it could be		
Outcome and rationale	Mapping dynamic Clojure data to relational database is errorprone but PostgreSQL offers most features anyhow.	Only viable option as Datomic is not Open Source and Mongo is not ACID compliant.	Doesn't properly solve using Clojure types, but neither do alternatives.

Name	Bottled Water extension		
Problem	With most of the databases, data is updated in-place which leads to losing history related to stored entities. In addition to solving problems related to history preserving, access to emitted change data enables real-time notifications of data changes where necessary.		
Solution / description of decision	Bottled Water extension is enabled on the PostgreSQL database. It provides the change data to Kafka where it can be consumed. Change data could be consumed to store the history to some persistent database. To implement real-time change notifications, Kafka consumer was implemented to read changes and forward them to connected websocket clients.		
Considered alternative solutions	To store the history of any database table could have been implemented using history tables and triggers that would have been at least infrastructure-wise more simpler approach. Real time change notifications could been implemented by manually notifying the clients whenever necessary.		
Argument in favor of decision	• Bottled Water enables many use cases such as preserving history or notifying users on changes.		
Argument against the decision	• Experimental alpha software • Adds lot of complexity to the infrastructure		
Outcome and rationale	Complicates infrastructure, alpha quality and doesn't fit the use case perfectly.	Interesting, but adding complexity & yet unproved. On hold.	Too complicated installation and maintenance. Brings more complexity than added value

Name	Kekkonen CQRS API		
Problem	Modelling the problem domain data and functionality has been done using plain Clojure data and functions. Exposing these to the web UI clients have been previously done using REST like APIs. Mapping Clojure data and functions to a resource-centric API just to consume them again in Clojure(Script) code becomes additional burden. Typical use case for us is to implement web UI for the backend we are providing without a massive public APIs for generic consumer.		
Solution / description of decision	Instead of mapping the problem domain and related functionality into REST like API, use more streamlined approach to invoke operations and transfer data between the server and the web UI. This is achieved with Kekkonen library where API can be constructed commands and queries. Commands and queries are defined as plain Clojure functions and data is transferred as Clojure data structures between the backend and the frontend.		
Considered alternative solutions	If the API would be open to more consumers REST like mapping with various available content types would be another alternative. However, since Kekkonen generates always up-to-date API documentation there is no reason why CQRS api could not be used by other consumers as well. All-in on Relay and GraphQL.		
Argument in favor of decision	• CQRS API without mapping to resources allows more straightforward interoperation between Clojure backend and Clojurescript frontend • Auto-generated documentation would help other consumers integrate as well • Possibly usable with Om.next		
Argument against the decision	• CQRS API could be too specialised for third party consumers		
Outcome and rationale	Simpler than REST and makes development more agile.	KISS, WASP, VENOM +1 likes.	REST and CQRS both have their own use cases and their usage in a project is situational.

Name	Websocket API		
Problem	Applications that have multiple simultaneous users that require real-time updates or notifications need a mechanism to deliver those changes to the browser. For example in case project Y there were some use cases where this kind of functionality would have been beneficial.		
Solution / description of decision	Backend provides a Websocket API to stream changes to other clients. In this example implementation, some of the changes originated from Bottled Water are published to the browser clients as well. This ensures that if multiple clients are in the same view, everyone will see up-to-date data if some client makes modifications.		
Considered alternative solutions	Stateless alternative could have been possible where clients would simply poll the data by calling the API periodically.		
Argument in favor of decision	• WebSockets are most common real-time commucation method for browsers • Low-latency duplex communication • Allows the user to select data format (e.g. Transit) • Good existing Clojure(Script) wrappers • Many libraries provide some fallbacks if WebSocket transit doesn't work.		
Argument against the decision	• Still might have some problem in enterprise envinronments (firewalls)		
Outcome and rationale	Useful, but adds some complexity.	Use a library, which fallbacks to polling. Real-time FTW!	Better than periodical polling or long polling but requires an easy to use Clojure API. Could be integrated to Kekkonen library.

Name	Normalized application state		
Problem	If application state is modelled as a tree, it contains denormalized data. This means that same entity might reside in multiple branches of state tree. This makes updates harder because entity has to be updated in all the places.		
Solution / description of decision	By normalizing the data and and modelling state as a graph the updates become easy. Nodes can use "links" to refer to other nodes. The normalization will be done manually.		
Considered alternative solutions	• To ignore the prolem and keep data in tree & denormalized • Use Relay & automatically normalized data • Use Om.next & automatically normalized data		
Argument in favor of decision	• Easy updates. • Relay and Om.next have proved this works.		
Argument against the decision	• Data has to be denormalized for rendering. • Reading the data is a bit harder		
Outcome and rationale	Reagent reaction/-track allows easy reads anyhow.	I have felt the mess with denormalized data, causing buggy reads and hard-to-reason logic. Normalization should be used.	No-brainer. Reduces unnecessary HTTP traffic.

Name	Re-frame dispatch & handlers		
Problem	No conventions for writing event handlers. Often event handler code has been written inline inside the components. This makes it harder to read the code. Some handler code has been written as normal functions in the same namespace as the component.		
Solution / description of decision	Use well-documented convention from Re-frame: all handlers are registered via Re-frame. All events are dispatched to these handlers.		
Considered alternative solutions	• Own implementation		
Argument in favor of decision	• Re-frame is well documented		
Argument against the decision	• Re-frame introduces some global state to manage the registered handlers.		
Outcome and rationale	Convention is good	re-frame (or similar) should be used, despite not perfect, good docs give a baseline for new developers to learn it.	Good idea. Makes it possible to clearly separate the event logic from global state changes.

Name	Re-frame subscriptions		
Problem	Reagent code has usually used many ‘reactions‘ and most components have been written in form-2 (they use closure). Closures make the code harder to understand.		
Solution / description of decision	Re-frame subscriptions allows components to register named queries over the application state.		
Considered alternative solutions	• Using Reagent 0.6 ‘track‘ • Writing something similar ourselves		
Argument in favor of decision	• Have some performance pros against bare reactions (no duplication) • Middleware pattern can allow sharing common stuff		
Argument against the decision	• Subscriptions still require using form-2 components (closures) • Reagent track works without Closures and does the same		
Outcome and rationale	Using track is easier and does the same	track vs subscriptions. Choose one, stick to it. Neutral.	Track and subscriptions both have marginal benefits. Use whatever you like.

Name	Component Schema validation		
Problem	If components are called with wrong parameters, the errors might not show instantly. The errors caused might be confusing.		
Solution / description of decision	Attach Schema declarations to Components. This can be used to validate the parameters. Errors can be catched to show clean error messages to developer.		
Considered alternative solutions	• Static type analysis with Core.typed. • React.js component validation • Elm		
Argument in favor of decision	• Better error messages • Faster feedback		
Argument against the decision	• More complexity		
Outcome and rationale	Useful but should be implemented in a library	Validation code can be fully removed from prod-build, need to test pros vs cons. Neutral.	Adds too many lines of code for the added value. Components are kept pure and simple so testing them is unnecessary.