Calderwood: A CQRS / Event Sourcing application template

Presented by:
Oliver Powell
Dept. of Electrical and Electronics Engineering
University of Cape Town

Prepared for:
Dr. Simon Winberg
Dept. of Electrical and Electronics Engineering
University of Cape Town

Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfillment of the academic requirements for a Bachelor of Science degree in Electrical and Computer Engineering.

30 January 2017

Keywords: Web applications, CQRS, Event Sourcing

Abstract

Calderwood is a web application template designed for building applications that are architected for data science. Traditional Create, Read, Update, Delete (CRUD) based web applications lose data, since they only store the newest state of the world. This makes data analysis after the fact tedious, as the information of interest is often sitting in obscure places, such as database logs. Research was done on Event Sourcing, CQRS (Command Query Responsibility Segregation), Datomic (an immutable database), among others, in order to come up with a solution. The project was successfully completed, and demonstrates how to build a web application which puts data analysis first, while exhibiting a flexible and maintainable architecture.

Contents

1 Introduction

1.1 Project Background

This project was originally inspired by the work done by Bobby Calderwood, in his 2015 Clojure Conj talk titled “From REST to CQRS with Clojure, Kafka, and Datomic”. In the talk he enumerated a number of problems we currently face in the field of web development, along with what he thought was one possible implementation of a solution. His presentation introduced the ideas of Event Sourcing and CQRS, as ways of combating some of these problems. [1]

A second project, by a local Cape Town company, Yuppiechef also served as motivation for exploring this topic. Yuppiechef built a small prototype of an Event Sourced / CQRS system which used Onyx, Kafka, Datomic and Amazon's DynamoDB. Their code included a very through writeup, describing the problem they were solving, along with the actual design. [2]

One of the central ideas in both Bobby Calderwood's talk, and Yuppiechef's writeup is that we are losing valuable data. Most web systems today are centered around the model of Create, Read, Update, Delete, otherwise known as CRUD. Whenever a record in the database is updated or deleted, this is done in place - whatever the old value was is lost, or banished to obscure logs. CRUD based systems effectively only remember the present, with the past being lost. However much of the interesting data exists in the past.

To make these ideas more concrete, let's examine what happens to a shopping cart in a typical CRUD application. fig. 1 illustrates this. As we add or remove items from our cart, past states of the shopping cart are lost. CRUD based systems keep only the latest state. You'll notice towards the end that the potential customer initially had oranges in their shopping cart, only to remove them towards the end. Wouldn't an ecommerce based business want to know that?

Harvard Business Review labeled Data Science as the "sexist" career of the 21st century, and for good reason. [3] More and more businesses are able to extract value from data to improve existing products and processes, and even to create new products. An example of this is something we use everyday - Google. Of course being able to analyze data means actually having that data available, and this is where Event Sourced systems fit in.

Instead of storing the state of the shopping cart, why not simply store the events that caused it to get into that state? fig. 2 shows this in action. Note that now we have a specific event for removing an item from the cart. That data is now preserved, and has been made a first class citizen. Not only that, but it's now possible to "time travel" and put the shopping cart into any previous state; providing that each event is ordered in time.

In light of all the above, this project was born out of the desire to build a template for an application that was architected for data science, where

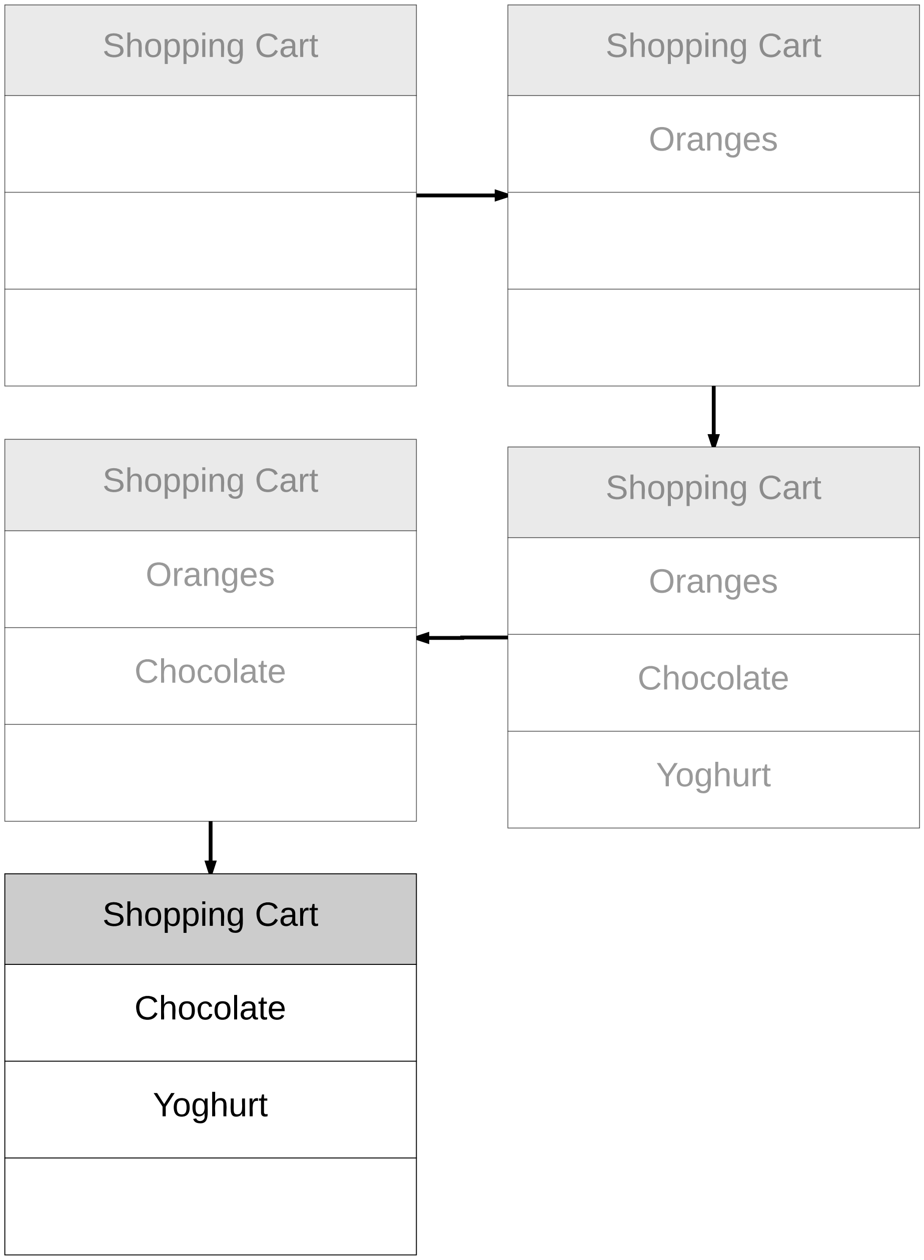


Figure 1: A CRUD based shopping cart, past states are lost, and only the latest state is kept.

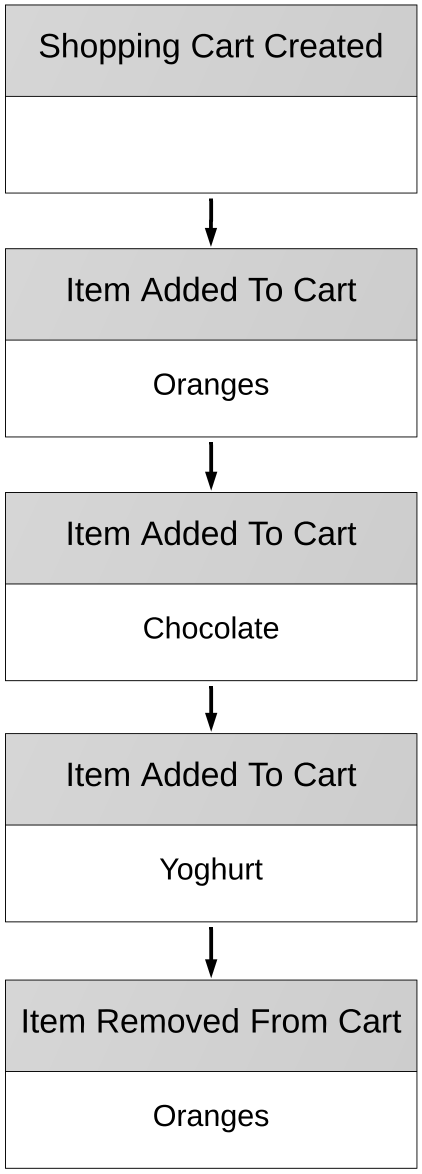


Figure 2: Using events to capture shopping cart actions.

it wouldn't be a struggle to mine the data needed for data analysts to do their jobs. At the same time there were plenty of side benefits from an Event Sourced / CQRS system, some of which included looser coupling between services, and simpler debugging.

Many frameworks for CRUD based systems exist, such as the ever popular Ruby on Rails web framework. There are very few Event Sourcing / CQRS frameworks, and barely any of them have much traction. This is most likely because Event Sourcing is a different way of thinking about applications, but it's also because Event Sourcing as a whole offers a lot of flexibility in how your application grows and evolves. Greg Young, an Event Sourcing expert, advised the public against creating another Event Sourced framework. [4] This project heeds his advice, and aims to rather be a reference, or a template, on which to build an Event Sourced/CQRS application. It does not intend to impose any boundaries on the engineers using it, and in encourages them to freely modify everything to suit their application.

1.2 Project Objectives

1.2.1 Problems to be Investigated

The following problems are to be investigated and researched.

- How to build Event Sourced / CQRS systems, and their benefits and tradeoffs.
- The importance of data science, and how applications can make data analysis simpler.
- Asynchronous communication on the web and how to decouple requests from responses.
- How to architect flexible and scalable systems.
- How to reduce HTTP API surface area, and build smaller, simpler APIs.

1.2.2 List of Objectives

The goal of this project is to design and implement a template for the back-end of a web based application, with the following objectives:

- First class support for data science.
- Fully asynchronous.
- Flexible architecture which is able to adapt to change and scale.
- Initial support for 500-1000 concurrent users on modest hardware.
- Audit trail support.
- Simple to test, debug, and maintain.

1.2.3 Purpose of the Project

The purpose of this project is to provide a foundation for a small to medium sized team that would like to architect a modern web application with first class support for data science, that is flexible, scalable, maintainable, and performant enough.

1.3 Scope and Limitations

This project is limited to research within the fields of web applications, and distributed systems only. Any related fields will not be included.

Research sources for this project include books, online articles, blog posts, and non-academic software conference talks in addition to peer reviewed materials. Software, and web applications in particular, is a rapid moving field, and such much of the available peer-reviewed literature becomes outdated soon after it is published. In order to access the most relevant information it is thus essential that these sources are consulted.

It is assumed that the reader is reasonably well versed in traditional back-end web development, and that terms such as threads, HTTP, asynchronous communication, etc do not need to be explained.

It is further assumed that the reader is familiar with the technologies chosen for implementation of this project, or that they are able to find tutorials or books to teach them the basics. Due to the short time frame given for this project, it is impossible to include a crash course in the technologies used, in addition to the project itself. Information on all technology used in this project is readily available for free from numerous online sources.

This project will be tested for performance, however the results should be taken with a pinch of salt, given that it is very difficult to predict the performance of a real world application built on top of this framework without actually testing that exact application. Also time constraints have resulted in performance testing being a minor concern, and a better test can certainly be written.

Finally, the project includes an additional component, a web console for interacting with the back-end, and testing it. This front-end application does not form part of the scope of this project, and therefore it's implementation, or any details surrounding it will not be discussed at all.

1.4 Terminology

- *UUID*, Universally unique identifier, a 128 bit number used to identify information in computer systems.

1.5 Plan of Development

This project report has the following structure:

Chapter 1 introduces the research project, it's objectives, scope, limitations and motivation.

Chapter 2 consists of a literature review of relevant sources both in academia and industry.

Chapter 3 gives an overview of the project methodology.

Chapter 4 details the design of the application template.

Chapter 5 presents the implementation of the application template design.

Chapter 6 reviews both functional and performance tests performed, and the results of such tests.

Chapter 7 is a discussion of the testing results, and overall implementation in light of the project objectives.

Chapter 8 lists recommendations for future work.

2 Literature Review

2.1 Place-orientated Programming

Information consists of facts. A fact is something that is known to happen. It is precise, immutable and it has a time dimension - it is known to have happened at a particular point in time.

A place is a particular portion of space, used for a particular purpose. In computer systems this would correlate to an address in memory, or a sector on disk.

Information systems today generally operate by replacing old information with new information. A fact is associated with a particular place and new facts overwrite old facts. Rich Hickey refers to this as place-orientated programming. [5]

This was not always the case. Before computers our information systems were always accumulating. Accountants do not use erasers, they use double ledger entry. Places were not important, facts were. This was born out of the limitations of early computers, but storage capacity has increased a million fold since then. When the scale of something increases so dramatically, do the same rules still apply? [5]

Place-orientated programming gives us the current view of the world, only the very newest facts. However, making good decisions tends to require past facts, in addition to the newer facts. Often to build a coherent picture of the world requires digging through database logs, a very suboptimal way of finding past facts. [5]

A well designed information system would have first class support for facts and operate in a very similar fashion to pre-computer record keeping systems.

2.2 CRUD Flavored REST

REST or REpresentational State Transfer is the brain child of Rory Fielding, and was first described in his doctoral thesis. It is effectively a generalization of the Web's architectural principles into an architectural style that can support almost any kind of application. His work led to a new perspective on the Web and how it could be used for purposes other than simple information retrieval and storage. [6]

CRUD is an acronym for create, read, update and delete. It defines the four basic functions of peristant storage. It is currently the dominant way of modifying resources in a web application, usually exposed via a RESTful interface. Resources can be created, read, updated and deleted. Web application frameworks, such as Rails (Ruby), Play (Java), Express (node.js), have popularised this approach, by providing good support for creation of these end points. This can be viewed as "CRUD" flavoured REST. [1]

Table 1: A typical "CRUD" flavoured REST API

HTTP Verb	Path	Used For	CRUD letter
GET	/articles	List articles	R
POST	/articles	Create a new article	C
GET	/articles/:id	Get article with :id	R
PUT	/articles/:id	Update article with :id	U
DELETE	/articles/:id	Delete article with :id	D

CRUD typically gives us only the current view of the world, and emphasizes place-orientated programming where new facts replace old. Update and delete cause old facts to be lost. If I update or delete an article in tbl. 1 then the past content of that article is essentially lost. This might not be of great importance in a content based domain, but it could be very important for a shopping cart.

Steve Yegge wrote an excellent rant against Java (and Object-orientated programming) and it's over emphasis on nouns rather than verbs. [7] Which makes more sense? Paying your credit card bill by updating a bill sub-resource of a credit card resource, or simply calling a pay-card-card-bill API end point? CRUD flavoured REST is really the kingdom of nouns for distributed systems.

[1] The description of what actually happened is lost in a haze of resources and sub-resources.

Bobby Calderwood lists some other negative aspects of CRUD based REST: [1]

- It causes a proliferation of HTTP API end points since every resource now needs its own path, along with it's CRUD operations. Add sub-resources and the problem is further compounded.
- It burdens the client with resource orchestration. The client now needs to deal with resource orchestration, and has to be aware of operational complexity that should be hidden within the backend. This detracts from the clients responsibility - the user.
- It's difficult to scale since reads and writes are typically coupled together in the same set of API endpoints.

He also lists some positive aspects: [1]

- Hiring for applications built this way tends to be easier.
- Tooling and support is generally good, so it's easy to get something up into production.
- It may be superior for certain domains, for example, domains that are heavily content based.
- It may also be a better choice if you cannot afford extra operational costs of an added messaging and event tracking system.

2.3 Essential REST

Essential REST exemplifies the heart of REST, and it's good parts. These include: [1]

- Clear communication semantics for big distributed systems in low-trust, low-coordination contexts.

- Proxying and caching of various types of requests over an unknown network path.
- Clear error codes when requests fail.
- Loose coupling from the underlying backend implementation.
- An emphasis on being self describing and on ease of consumption, exploration and navigation.
- A data orientated approach, requests are responses are just data, and are often encoded in formats such as JSON or EDN.

“CRUD” flavored REST is merely one way of building a RESTful API, but it is not the only way. It is certainly possible to build API end-points which are far more descriptive and verb orientated, such as “/pay-credit-card-bill” as opposed to “/credit-card/:id/bill/:id”.

2.4 Architecting for Data Science

Data science can be broadly defined as “the transformation of data using mathematics and statistics into valuable insights, decisions, and products.” It is the same as, or very closely related to terms such as business analytics, operations research, business intelligence, competitive intelligence, data analysis and modeling, and knowledge extraction. [8] In essence the purpose of data science is to extract value from data.

Data science enables the creation of data products. This differs from simply consuming data like many web applications do - data applications directly acquire their value from the data itself, and create more data as an output. The quintessential example of this is Google search which was the first search engine to realize that one could use input data other than text on the page. The PageRank algorithm that now powers Google search was one of the first to start using external data, such as the number of inbound links, in order to rank the page. There are numerous other examples from other companies such as Amazon, Linkedin etc. [9]

It is not just huge companies with massive amounts of data that can benefit from data science. Increasingly these techniques are being used in smaller sized datasets too in order to make better decisions in business and other domains. [8]

In CRUD based systems, data science is often an after thought, and generally logs have to be mined in order to obtain the data necessary to gain insight, since databases in these systems only store the newest facts. This essentially creates more barriers to entry for data analysis. [1]

What tends to matter is a system is what your user tried to do, and how you helped them to do it. CRUD heavy systems tend to obscure these important details by not storing all the information.

2.5 Event Sourcing

Martin Fowler describes Event Sourcing as follows: [10]

> We can query an application's state to find out the current state of the world, and this answers many questions. However there are times when we don't just want to see where we are, we also want to know how we got there.
>
> Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes.

Event Sourcing is an architectural pattern in which changes to an applications state are captured as a time ordered series of event objects. It is then possible to reconstitute the application state, to any point in time, simply by replaying these events, along with the initial application state. Unlike CRUD based systems, no data is lost, and Event Sourced applications naturally support data science. [1]

The history of Event Sourced applications goes back as far the IBM IMS TM transaction manager which was used as part of an inventory system to manage the bill of materials for the Apollo space program. The system was capable of 2300 transactions per second, on 1960s hardware, an impressive feat, even by todays standards. [11]

Events describe something that has happened in the past and typically have names such as UserLoggedIn, OrderShipped, and PageViewed. Events are created by Commands, which operate in the present tense. For example, UserLogin and ShipOrder. Once events are created they are essentially immutable facts, that cannot be changed. [2]

Events are stored in an Event Store, which can be implemented in any type of persistent storage, though a database is always preferable. The Event Store then acts as the system of record, tracking all changes to the application state over time. Event Stores are typically optimized for fast time based range queries, for example, finding all the OrderShipped events between two dates. [12]

Events are aggregated into derived views, or read models, which can result in very efficient queries for data later. These aggregates can be rebuilt on the fly as events are processed and stored. Event Listeners can also be attached, which perform external actions, such as sending email, or creating assets for downloading. [2]

Event Sourcing hasn't seen mainstream popularity in general, though custom built Event Sourced applications are popular in the financial industry, especially in high speed trading. A notable example is the LMAX architecture, which used a single thread to process incoming trade events with exceptionally low latency and high throughput. [13]

The downsides of Event Sourcing are best found by talking to practitioners that use it in industry. Some of these are the following:

- Event Sourcing is a different way of thinking about applications, and it may be difficult to hire for as a result, as most people wouldn't have worked with such an application in the past.
- Versioning of events can be difficult, given that events are immutable. If you need to make changes to events that apply retroactively, this is can be quite a challenge, and will usually involve correcting events, or rebuilding a new event stream, which can be costly if there are millions of events.
- Certain domains are better suited to CRUD based designs, such as content based applications, like blogs. If your application does not require you to care about past data, and you only need to care about the

present moment, then Event Sourcing will probably only add complexity to your system.

2.6 CQRS

CQRS or Command and Query Responsibility Segregation is an intimidating acronym with a very simple meaning - separate your reads from your writes. It was popularized by Greg Young. [14]

In typical CRUD based APIs reading and writing are intertwined. Typically the same server will serve both updates to resources in addition to queries on them. Yet in most applications read and write loads are seldom equal.

The simplest reason for CQRS is scaling. If your reads and your writes are separated then it's possible to independently scale them. If your application is write heavy you may choose to have more servers servicing writes, likewise, if your application is more read heavy, you may want to add a few more servers for queries. [14]

Bobby Calderwood suggests a simple way to achieve this, by defining only three end points. tbl. 2 lists them. Three endpoints greatly simplifies the HTTP API endpoint proliferation typically experienced in CRUD systems. [1]

Table 2: CQRS HTTP API endpoints

Path	Used For
/command	Issuing a command
/query	Performing a query (pull)
/update	Real-time push via WebSockets or SSE

Martin Fowler advises caution when implementing CQRS, as it can add plenty of additional, unneeded complexity if the domain it is applied to does not benefit from it. [14] Bobby Calderwood argues that CQRS complexity mostly occurs when it is applied on the micro level, particularly with OOP principles. He does not see the same problems occurring when it is applied at the macro level, i.e HTTP API level, with a functional programming approach. [1]

2.7 Avoiding the Tarpit

According to Mosely and Marks in their iconic 2006 paper, "Out of the Tarpit", complexity is the root cause of most problems in software development, because understanding of the software, they argue, is greatly undermined by complexity. [15]

They categorize complexity into two types, essential and accidental. Essential complexity is inherent in the problem, whereas accidental complexity is additional complexity created by the programmer, and orthogonal to solving the problem at hand. Accidental complexity is the enemy of the programmer.

Complexity in software they argue is primarily caused by mutable state, and control. It is difficult to keep track of all possible states that a system can be in. Likewise control is difficult because order is important. If one statement is incorrectly placed before another then the program is typically incorrect too.

Object-orientated programming has been one approach to handling this complexity. However it fails for two primary reasons. OOP conflates the notion of intensional identity (an object's identity) and extensional identity (the object's attributes). This increases the number of states which need to be considered. Secondly each object method accessing the object's state needs to enforce integrity constraints, it is also akward to enforce multi-object integrity constraints. OOP also uses traditional flow control structures, and thus has no solution to taming complexity associated with control. [15]

They further state:

> The bottom line is that all forms of OOP rely on state (contained within objects) and in general all behaviour is affected by this state. As a result of this, OOP suffers directly from the problems associated with state, and as such we believe that it does not provide an adequate foundation for avoiding complexity.

Functional programming in it's purest form eschews mutable state and side effects, and thus avoids much of the complexity associated with state. Higher level functions such as filter and map, allow a slighly better form of control, though order still matters to a large degree. The biggest weakness of functional programming is it's strength too, since all non-trivial programs require a certain amount of essential state. [15]

Mosely and Marks recommend two strategies for dealing with complexity. Avoid it, accept only essential complexity, and if you can't avoid it, for reasons of ease of use, or performance, then separate it out in order to better manage it.

2.8 Service-orientated Architecture and Microservices

Service-orientated architectural or SOA is a design paradigm centered around the idea of multiple services, which collaborate to provide functionality to the overall system. Each service is autonomous, and operates as a separate operating system process. All communication between these services has be occur through a network protocol, rather than direct method calls within the same process. No memory is shared directly between services, and each service usually has it's own database. [16]

SOA developed as an alternative to large monolithic applications, where different parts of the application were often tightly coupled, and difficult replace, or rewrite. SOA aimed to address this by building an application out of composable services, with strict interface boundaries. [16]

Microservices emerged from real-world use as one specific way of implementing SOA. It involves much finer grained services, which focus on a doing single tasks well. In general the goal for each service is to have a small amount of code, that is easy to understand, change and maintain. [16]

Microservices are said to have a number of benefits: [16]

- They promote technology heterogeneity. Services can be written using any language or technology stack. This means that if one part of the system require more performance then it's possible to rewrite it later in a faster language, or more suitable technology stack.
- Services can be independently scaled. In a large monolithic application, everything must scale together, but in a Microservice architecture the cardinality of each service is independent.
- Services are optimized for "replaceability". As long as a service maintains its interface contract, it's possible to rewrite it, without affecting any other parts of the system.

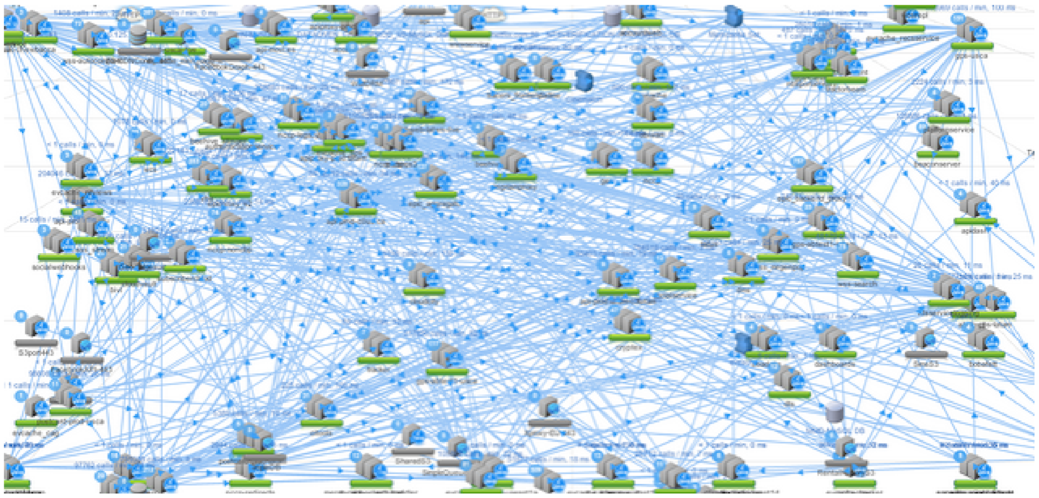


Figure 3: Netflix topology

But at the same time, they also have a number of tradeoffs: [1]

- There is an explosion of HTTP API endpoints. A graph of connections between Netflix services is shown in fig. 3. [17] So many end points quickly become a nightmare to reason about.
- Teams tend to reinvent the wheel constantly. Each service has to solve the same set of problems, but often in a different technology stack. Databases, cache, deployment, authentication, and authorization to name a few.
- All end points are subtly different, which makes building clients for these services more difficult. It creates an additional problem to the front-end teams, which now need to be concerned with both user experience, and how to orchestrate resources across a large number of varying HTTP API endpoints.

2.9 Stream Processing

Stream processing typically involves processing large amounts of events in sequential order. These event form an event stream. As new events are generates, they are added to the stream, which is usually a persistent queue of some type, that guarantees order, such as Apache Kafka. It is very closely related to Event Sourcing, and big Internet companies such as Linkedin typically have teams of data analysts consuming these events in order to improve their products. [18]

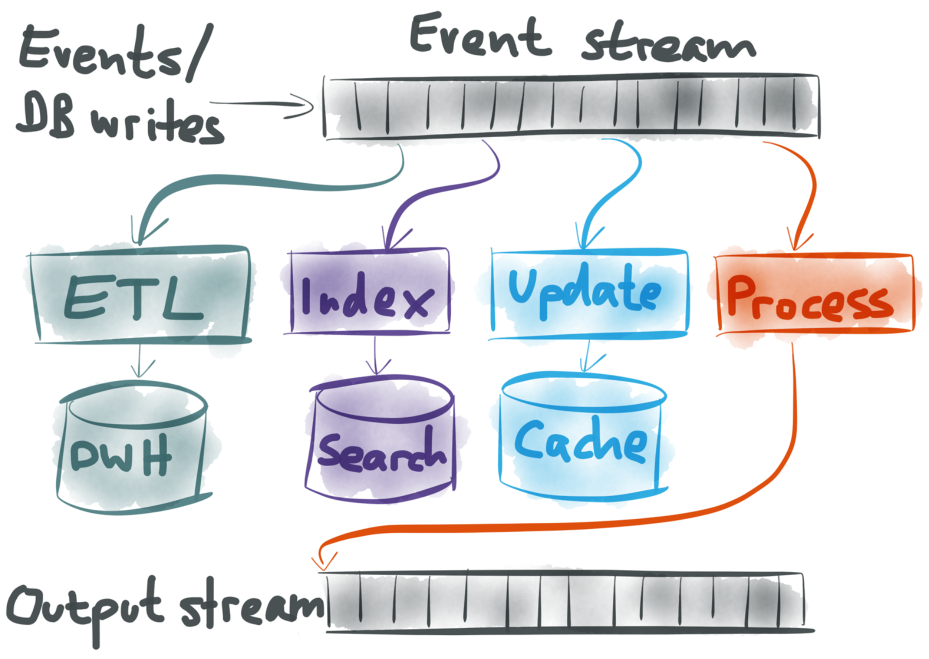


Figure 4: Stream Processing [18]

Event streams can be consumed by multiple consumers in parallel, and offer a unidirectional flow of data through the system. As seen in fig. 4 the flow of events can be used to update full-text search indexes, such as Solr, build read models, rebuild caches, and even produce new output streams which can be joined with other event streams, or consumed by other downstream consumers. [18]

Event streams are often connected to distributed stream processing frameworks, which attempt to shield developers from the operation complexity of distributing work across a cluster of machines. Such frameworks include Samza, Storm, Onyx, Spark, and Flink. [18]

2.10 Websockets

Todd L. Montgomery gave an excellent talk at React 2014 on the need to embrace asynchrony in our web applications. He argues that synchronous (or blocking) calls tend to produce systems that are both coupled, and less performant. Asynchronous systems allow requests to be processed independently of responses, which encourages loose coupling. In addition to this, while an asynchronous request is being made, other work can be done on the client while it waits for a response. While it is possible to make HTTP requests asynchronously via threads or callbacks, in practice a large amount of developers tend to wait for a response in order to do error handling, before continuing. [19]

In the past clients have had to result to solutions such as HTTP long polling in order to break the coupling between HTTP requests and responses. The HTML5 standard sought to include a better solution to the problem, in the form of the Websocket Protocol (RFC 6455), which allows a full duplex, bidirectional connection from client to server. Websockets piggy back on top of the HTTP protocol, and can be seen as thin layer of abstraction on top the underlying TCP connection. The results in much higher performance, saving bandwidth, CPU power, and latency, and making Websockets ideal for real-time, asynchronous applications. [20]

2.11 CAP Theorem

Eric Brewer first introduced the CAP Theorem in 2000. The theorem centers around the idea that there is a fundamental tradeoff between consistency, availability, and partition tolerance in distributed systems. [21]

An informal definition for consistency is simply that that each server returns the right response to each request. Right in this context means, correct according to the desired service specification. [21]

Availability means that each request to a server will eventually receive a response. [21]

Partition tolerance refers to the underlying system. Networks are unreliable and it is highly likely that servers are partitioned into groups that cannot communicate with each other for extended periods of time. [21]

In broad terms the CAP Theorem implies that you can either have a consistent and partition tolerant system (CP), or a an available and partition tolerant system (AP). It is impossible to have a CA system, unless that system all within in a single node, and is not networked.

Seth and Nancy offer a proof for the CAP Theorem that is relatively straight forward: [21]

> Consider an execution in which the servers are partitioned into two disjoint sets: $\{p_1\}$ and $\{p_2, ..., p_n\}$. Some client sends a read request to server $p2$. Since $p1$ is in a different component of the partition from $p2$, every message from $p1$ to $p2$ is lost. Thus, it is impossible for $p2$ to distinguish the following two cases:
>
> - There has been a previous write of value $v1$ requested of $p1$, and $p1$ has sent an ok response.
> - There has been a previous write of value $v2$ requested of $p1$, and $p1$ has sent an ok response.
>
> No matter how long $p2$ waits, it cannot distinguish these two cases, and hence it cannot determine whether to return response $v1$ or response $v2$. It has the choice to either eventually return a response (and risk returning the wrong response) or to never return a response.

Event Sourced systems typically favor the AP side of the CAP Theorem, trading consistency for high availability. [11]

2.12 Functional programming

Fogus and Houser provide a workable definition for functional programming: [22]

> Whether your own definition of functional programming hinges on the lambda calculus, monadic I/O, delegates, or java.lang.Runnable, your basic unit of currency is likely some form of procedure, function, or method—herein lies the root.

> Functional programming concerns and facilitates the application and composition of functions. Further, for a language to be considered functional, its notion of function must be first- class. First-class functions can be stored, passed, and returned just like any other piece of data. Beyond this core concept, the definitions branch toward infinity; but, thankfully, it's enough to start.

Hughes argues that other functional programming features, such as higher-order functions, and lazy evaluation, lead to better structured and more modular programs, with less lines of code and less flow control statements. [23]

Functional programming tends to focus on minimizing state by emphasizing the use of pure functions. Pure functions resemble mathematical functions in the sense that always return the exact same output, given the same inputs. This is known as referential transparency. In simpler terms, pure functions do not have side effects. Programs that consist primarily of pure functions have minimal state, and are thus easy to reason about and test. [15]

2.13 Clojure

A good description of Clojure can be taken from it's website: [24]

> Clojure is a dynamic, general-purpose programming language, combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming. Clojure is a compiled language, yet remains completely dynamic – every feature supported by Clojure is supported at runtime. Clojure provides easy access to the Java frameworks, with optional type hints and type inference, to ensure that calls to Java can avoid reflection.
>
> Clojure is a dialect of Lisp, and shares with Lisp the code-as-data philosophy and a powerful macro system. Clojure is predominantly a functional programming language, and features a rich set of immutable, persistent data structures. When mutable state is needed, Clojure offers a software transactional memory system and reactive Agent system that ensure clean, correct, multithreaded designs.

Clojure has seen steady industry adoption in the last few years, and continues to gain popularity. It has been marked as “Adopt” on the ThoughtWorks technology radar since October 2012. [25]

The language itself has a rich set of data structure literals, and strongly emphasizes programming with values. Clojure also defines extensible data notation, or EDN, to represent Clojure programs. This is effectively a way for Clojure to serialize programs, and in particular, data structures, and pass them over the network to communicate with other Clojure processes natively. [26]

Clojurescript is a variant of Clojure that targets Javascript as a host language. This makes it possible to write fully isomorphic web applications, with Clojure on both the client (web browser) and server. [22]

2.14 Datomic

Datomic is a distributed, immutable, cloud-ready database which supports ACID transactions, joins, and boasts a powerful logical query language - Datalog. What makes Datomic particular interesting is it’s notion of the database as a value. Since Datomic databases consist of a time ordered series of immutable facts, known as datoms, it is possible to return an exact immutable value of the database at any point in time. [27]

The Datomic team argues that databases have traditionally been called up to deliver numerous services, such as coordination, consistency, indexing, storage, and queries. Datomic breaks these services apart into separate components, which each do a single task well. This, among other benefits, allows the moving of query capabilities from the database to the application, effectively giving applications scalable and elastic intelligence. [27]

The architecture of Datomic can be seen in fig. 5 it consists of the following parts:

- **Peers.** These are a library which gets embedded into an application. Peers submit transaction to the transactor, and accept changes from it. They also provide data access, caching and query capability to the application, and can directly read from storage as needed. [27]

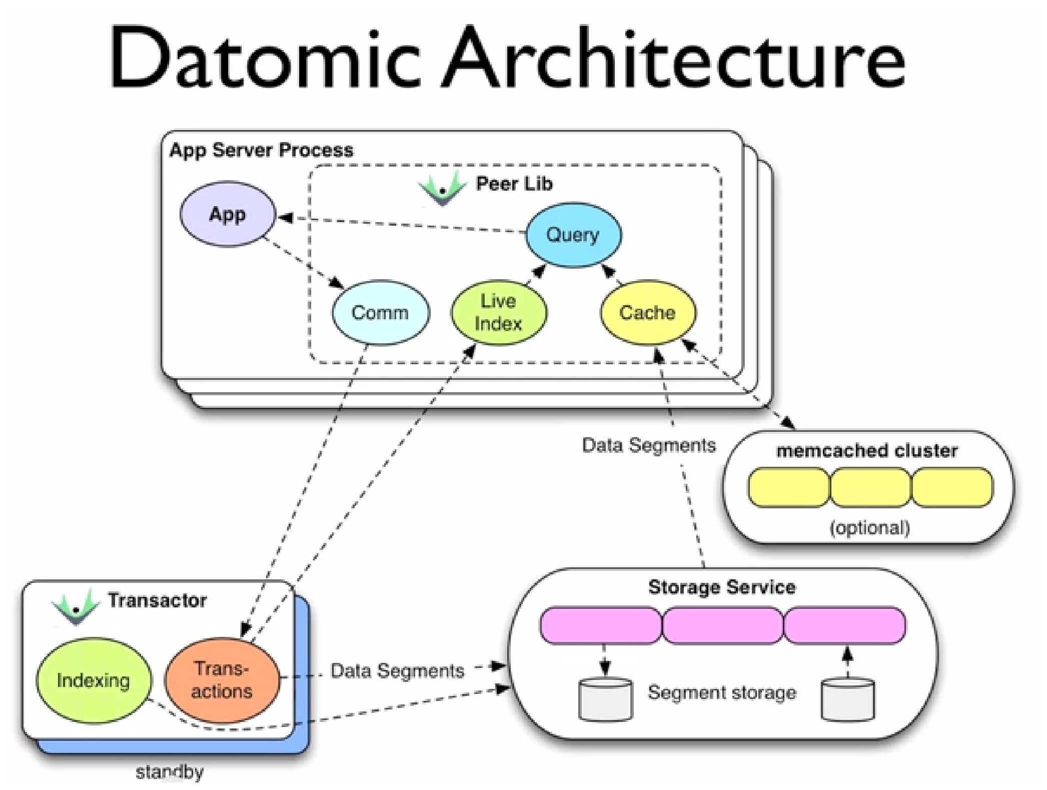


Figure 5: Datomic's architecture [27]

- **Transactor.** Accepts transactions, and processes them serially, and commits the results to storage. It then transmits any changes to the peers. The transactor is also responsible for building indexes in the background. [27]
- **Storage.** This provides an interaface to highly reliable and redunant storage, and includes backends for a number of databases, such as Amazon DynamoDB, and PostgresSQL. There is also an optional Memcached layer, which provides further caching for production use. [27]

Datomic is proprietary software, but offer a free version, limited to two peers, which uses the H2 database engine and is suitable for small production deployments. It also offers a professional version which provides a year of access and updates for free.

3 Methodology

3.1 Research

The initial phase of the project consisted of research into relevant literature and conference talks in the field of web development and distributed systems.

Topics included:

- Event Sourcing.
- CQRS.
- Microservices.
- Data science.
- Steam processing.
- Software complexity.
- The limitations of CRUD based REST APIs.
- Functional Programming.

- Websockets.
- Datomic, an immutable database.

A variety of sources were consulted, including sources from industry.

3.2 Design

The design phase of the project primarily focused on high level design from a birds eye view. This consisted of the following:

- Establishing the flow of data throughout the system.
- Dividing the system into components.
- Identifying tasks for each component.
- Considering possible error points in the system, and things that can go wrong.
- The structure of events and commands.
- The HTTP API routes.

In addition the this primary task, there were a number of secondary design tasks such as:

- Designing an application level protocol for asynchronous communication over Websockets.
- Designing an algorithm for retrying database transactions.
- Choosing a method for user authentication and authorization.
- Coming up with a strategy for performance testing.

3.3 Implementation

The project implementation phase primarily consisted of choosing libraries and writing code to implement the design.

During implementation it was natural to re-design certain components and algorithms when practicality forced it. Certain problems in design can only be fully understood when an attempt at implementation is made.

3.4 Testing

The testing phase involved two sub phases, functionality testing, and performance testing.

Functionality tests were done by using a custom built Clojurescript console application, which allows commands and queries to be submitted to the back-end system. The details of the construction of this application are beyond the scope of this project, but it was necessary for more sane testing of the application from a graphical interface. The console was also protected by the back-end authentication / authorization scheme, which allowed testing of user login and logout too.

Performance testing consisted of a latency and throughput test which exercised the entire flow of data through the system by focusing on a single event, and Websocket connection. The goal was to find the maximum throughput attainable with acceptable latency, which was defined as under 100ms for the 99.9th percentile.

3.5 Results and Conclusions

In this phase of the project, results of the testing phase are analyzed together with non-quantifiable project features and conclusions are drawn in light of the project objectives. Was the project a success?

3.6 Recommendations for the Future

Finally, recommendations for future work, such as exploring the use of Apache Kafka in the system, are made.

4 Design

4.1 High-level overview

The high level overview of the design is shown in fig. 6.

The design begins with the *SPA client*, a Javascript application running in a modern web browser. It is assumed that this application has access to the Websocket API. Websocket support in modern browsers is generally very good.

The *SPA client* is connected to the Websocket server via the bidirectional connection that the Websocket protocol affords. This is represents by the two blue queues, since the communication is fully duplex. Due to this we are able to have fully asynchronous communication, with requests decoupled from responses. The client is able to send commands, and receive updates. Websockets are effectively a thin layer of abstraction above TCP, and hence like TCP, they offer no way for the application to acknowledge data once it has been received. Once the application is delivered a packet of data, it is then the applications problem. It is up to the application to develop a protocol that will allow it to identify when a command has been successfully submitted for processing. That protocol forms part of the design process, and will be discussed in more detail later.

The *Websocket Server* is responsible primarily for establishing, and maintaining Websocket connections, and secondly for validating and putting commands onto the *Command Queue*. Invalid command should result in the client receiving a message indicating that there was an error, and likewise an acknowledgment if everything was validated successfully. This component can be used by other services, such as the *Update Handler* to send updates back to client after commands have been processed.

The *Command Queue* can either be a local concurrent queue, since there are potentially many Websocket connections putting commands onto it, or it could be a distributed queue which maintains order, like Apache Kafka, or NATS Streaming. The command queue effectively keeps the *Websocket Server* and *Command Processor* decoupled, and ensures there is some ordering of incoming commands.

The *Command Processor* is responsible for converting commands to events,

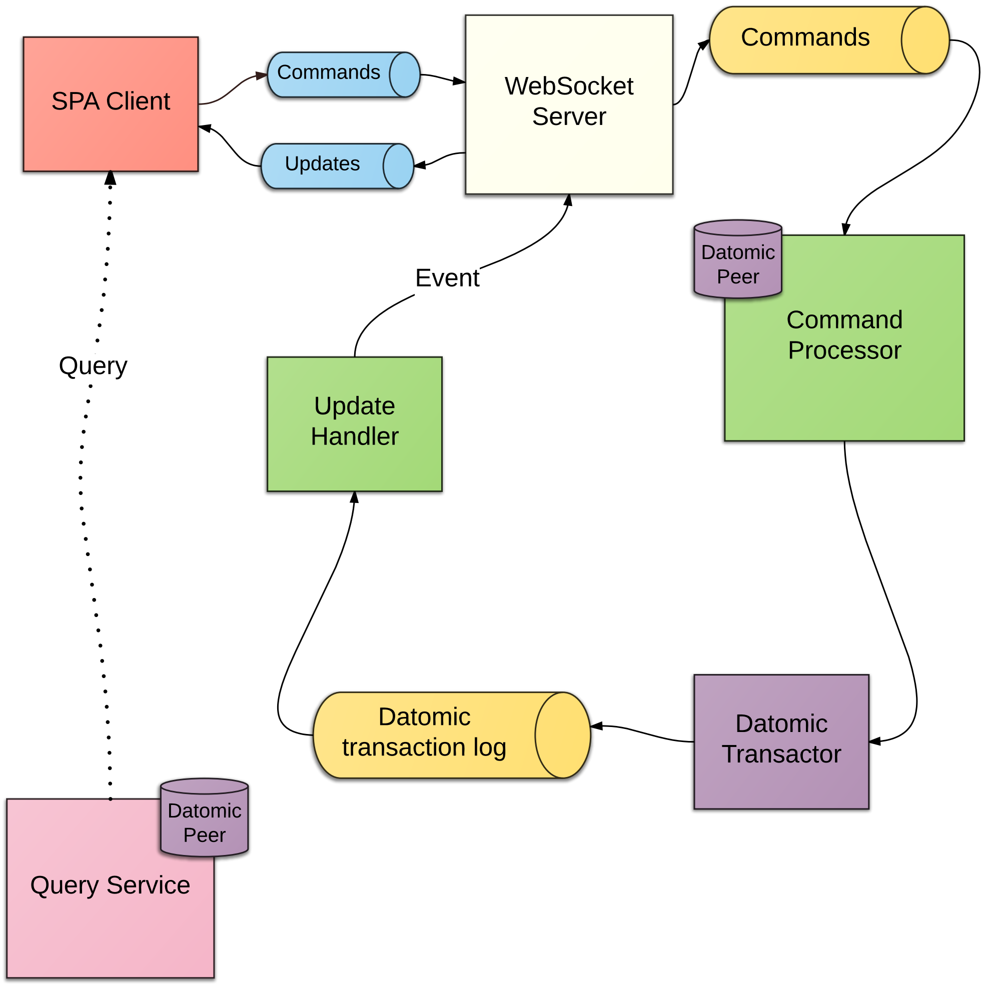


Figure 6: Calderwood

and also for aggregating those events, and transacting them to the Datomic database. It uses a Datomic peer to read the current value of the database at the time of receiving the command from the *Command Queue*, and uses that database value to produce events. These events are then aggregated and transacted. Datomic allows database transactions to be annotated with additional information. By default Datomic only annotates each transaction with a timestamp of when that transaction was processed. In our case we also include the event data. This allows downstream processors, such as the *Update Handler* to easily obtain the event, and pass it back to the client. Since transactions can timeout, it is essential that there is some sort of retry policy in place.

The *Datomic transaction log* is a log of all transactions that have happened. These transactions are stored as data, and Datomic provides a useful API to access it. For our purpose we will use the transaction report queue, which gives a real-time view into transactions as they happen. As each transaction occurred it is pushed onto a queue, which can be read off by a consumer.

The *Update Handler* watches the *Datomic transaction log* and reads off each incoming transaction's data. Since each transaction is annotated with the event which is was caused by, the *Update Handler* can simply extract the event, and send it back to the *Websocket Server*, which will in turn send it on to the client.

Finally there is the *Query Service* which enjoys a high degree of decoupling from the rest of the system. It's primary purpose is to process queries received by the client, and return the relevant data from the aggregated events. It does this by querying Datomic, use either it's indexes, or Datalog query language. Since the overall design is a CQRS system, there can be a number of *Query Services*, which can together support excellent read scaling. Read scaling is also a big strength of Datomic.

4.2 Commands

Commands represent a present tense action, and are handled and converted into past tense events, which are facts. A command can produce one or more events.

Commands can be represented by a map or object, and have the following fields:

Field	Type	Description
UUID	UUID	The command's unique identifier.
Name	Clojure Keyword	The name of the command.
Data	Clojure Map	The command data, key/value pairs.
Meta	Clojure Map	Command metadata, key/value pairs.
User UUID	UUID	The user who issued the command's unique identifier.
Client ID	String	The client where the command came from's ID.
Client Seq	Long	The sequence number of client.

Only UUID, Name, and Data are always required, the rest are optional.

An example command in Clojure EDN:

```
{:command/uuid #uuid "588d97e8-c456-47a0-bffd-84af06223387"
 :command/name :view-page
 :command/data {:page-view/url "http://www.example.com"}
 :command/meta {:send-timestamp #inst "2017-01-02T07:17:03.944-00:00"}
 :command/user-uuid #uuid "588d96ef-173e-422c-b89b-bb2de199322c"
 :command/client-id  "588d96ef-4c26-4769-9b86-bf893aebfa72"
 :command/client-seq 0}
```

4.3 Events

Events represent a past action that as occurred. Events are facts. They are aggregated into entities, that are useful to the domain of the application, and than can be queried for.

Events can be represented by a map or object, and have the following fields:

Field	Type	Description
UUID	UUID	The event's unique identifier.
Command UUID	UUID	The parent command's unique identifier.
Name	Clojure Keyword	The name of the event.
Data	Clojure Map	The event data, key/value pairs.
Meta	Clojure Map	Event metadata, key/value pairs.
Source User UUID	UUID	The user who issued the event's unique identifier.
Client ID	String	The client where the event came from's ID.
Client Seq	Long	The sequence number of client.

Only UUID, Command UUID, Name, and Data are always required, the rest are optional. Command UUID is useful for matching an event with a logged command and can be very handy for debugging purposes.

An example event in Clojure EDN:

```
{:event/uuid #uuid "588d97e8-c456-47a0-bffd-84af06223399"
 :event/command-uuid #uuid "588d97e8-c456-47a0-bffd-84af06223387"
 :event/name :page-view
 :event/data {:page-view/url "http://www.example.com"}
 :event/meta {:send-timestamp #inst "2017-01-02T07:17:03.944-00:00"}
 :event/source-user-uuid #uuid "588d96ef-173e-422c-b89b-bb2de199322c"
 :event/client-id  "588d96ef-4c26-4769-9b86-bf893aebfa72"
 :event/client-seq 0}
```

4.4 Datomic Transaction Annotation

If the reader is unfamiliar with Datomic, a good primer was written by Daniel Higginbotham, and is available online. [28]

Datomic bases its information model around facts. A fact is Datomic is known as a *datom* and consists of a tuple of entity, attribute, value, time, and whether is was an addition, or a retraction. Time is a first class citizen. A *datom* in Clojure is represented as follows:

```
[10 :person/name "John" 65 true]
```

The entity is just an integer identifier. An attribute and value can be seen as a key/value pair. Underneath the hood, they are just integers too. The time component is represented by a special kind of entity, called a transaction entity. The integer identifiers of these entities provide a logical time, decoupled from wall clock time, making working with time simpler. Facts can be either asserted, or retracted. If John changed his name to Jake, Datomic would retract the datom above, and issue a new datom:

```
[[10 :person/name "John" 66 false]
 [10 :person/name "Jake" 66 true]]
```

Entities are not required to have the same attributes, and can have a mix of attributes. All datoms with the same entity identifier are grouped together, and the entities value at any point in time can be established. In the snippet below we see two entities, John and Jill, John has an age attribute, while Jill does not.

```
[[10 :person/name "John" 65 true]
 [10 :person/age 30 65 true]
 [11 :person/name "Jill" 65 true]]
```

Datoms are very low level, and working with them in production, as the author has for a few years, can be tedious. Even though all facts are stored, we are often more interested in collections of facts being asserted at the same time, rather than individual facts. We are more interested in actual events that happened, like a user being created, as opposed to a particular attribute of that user being asserted.

In light of this, this project uses Datomic in a way which gives an emphasis on collections of datoms, otherwise known as transactions. Since the time component of each datom is just itself an entity, it is possible to annotate groups of datoms being transacted with additional data. For example:

```
[[10 :person/name "John" 65 true]
 [10 :person/age 30 65 true]
 [65 :db/txInstant 65]
 [65 :event/name :user-created 65 true]]]
```

This now gives us a clear way to label transactions as a particular event, which is meaningful to our domain. We can query any entity attribute, find it's transaction entity id, and establish which event caused this change to happen. We can also attach all the other fields of the event to the transaction entity- this gives us a very clear picture of any details.

4.5 HTTP API endpoints

The goal of this project is to minimize API surface area. Thanks to CQRS, this is possible by limiting ourselves to two endpoints only. This makes it much easier for clients to interface with our system, and prevents a proliferation of endpoints from occurring.

HTTP Verb	Path	Used For
GET	/ws	Websocket connection for sending commands / receiving updates.
POST	/query	Performing queries and getting query results.

4.6 Authentication and Authorization

Authentication and authorization tend to be very application specific, but the project has included a usable form of user authentication, since most applications will require it.

Authentication is achieved through the use of an encrypted session browser cookie, which holds a session UUID. Whenever a user logs in successfully, a session entity in the database is created, which contains their user UUID, and this session UUID is stored in the cookie returned to them.

When the user makes future requests, the system decrypts the cookie, finds their session UUID, queries the database for it, and from that, it's possible to find the user's UUID and attach it to the HTTP request. Authorization can then be performed by checking that the user UUID has access to the particular resource it is attempting to access.

The project contains only basic authorization, which simply checks that the user has in fact authenticated, and is an actual existing user an the system, otherwise a 401 response is returned, indicating that the user is denied access.

4.7 Websocket Application Protocol

As explained in the high level overview, the Websocket protocol is only responsible for delivery of a message, there is no way to first do additional processing before acknowledging the message. The onus is therefore on the application to develop it's own protocol for acknowledgment.

A protocol for this purpose was developed using Clojure's EDN data structures:

```
;; Sending only
[:cmd <client-id> <client-seq> <data>]

;; Receiving only
```

```
[:cmd-ack <client-id> <client-seq>]

[:error <client-id> <client-seq> <err-msg>]

[:update <client-id> <client-seq> <data>]
```

When a client wants to send a command, it sends it as a Clojure vector beginning with the keyword *:cmd*, and followed by the client's id, the sequence number of this message, and the data representing the command. The client id, together with the client sequence number enable the client to later know which message was acknowledged, or failed.

An example of sending a command:

```
[:cmd "12345" 0 {:command/name :view-page
                 :command/data {:page-view/url "http://www.example.com"}}]
```

There are two possible responses to this command, either a Clojure vector beginning with *:cmd-ack*, indicating that the command was successfully accepted by the server, or otherwise a Clojure vector beginning with *:error* if the command failed. The error vector includes an error message, which gives a reason for the failure.

Finally, updates can be pushed to the client. These are fully processed commands, which have been returned as events that the initial command produced. These come in the form of a Clojure vector beginning with *:update*, and include the client id, and client sequence number if the client wants to correlate an event with a previously issued command. This is useful for user feedback, since a command acknowledgment only indicates that the command was accepted for processing, not that is has already been processed.

4.8 Websocket Server

The *Websocket Server* has a few responsibilities:

- It needs to accept incoming Websocket connections, and ensure they are authenticated.

- It needs to index incoming connection by their user UUID, so that update pushes can look up users in order to send them updates.
- It needs to validate commands, and acknowledge commands, or otherwise return an error to the client.

Websocket connections need to be managed, and when the server wishes to send updates to clients, it needs to be able to find those clients connections. This is where indexing comes in. No doubt this is application domain specific. For example, a product which has company entities may wish to index incoming connections by company id instead, so that updates can easily to sent to all users within a particular organization. This project has elected to show one way of indexing users, by their UUID. This ties in nicely with the fact that events contain a source user UUID field, which allows the Update Handler to send updates back to a specific user, or simply broadcast to all users, or a subset of users only. Indexing is typically achieved by using a map data structure, where the keys are UUIDs, and the values are the actual connection object.

4.9 Command Processor

The *Command Processor* is the heart of the system. It accepts validated commands, produces events, and ensures those events are aggregated, and transacted to Datomic.

There are two stages to the command processing process, handling commands, and aggregates events. These are both handled by functions which dynamically dispatch on the command or events name. It is possible to implement this as either a switch statement (known as a *case* statement in Clojure), but a better choice is to use Clojure's multi-methods, which offer an elegant solution to the expression problem. [29]

The handle command function takes the current Datomic database value, along with a command, and returns a vector of one or more events. The current database value can be queried in order to ensure constraints, or find data that the events will need.

$$HandleCommand(DatabaseValue, Command) \rightarrow [Event...]$$

The aggregate event function takes the current Datomic database value and an event, and produces a Datomic transaction, which is annotated with the event data. Datomic transactions are just data structures themselves, which makes this step essentially one of data transformation.

$$AggregateEvent(DatabaseValue, Event) \rightarrow DatabaseTransactionData$$

The *Command Processor* can exist independently of the main system, in it's own process if need be, provided an external distributed persistent queue is used. In this case of this project it was elected to simply run the *Command Processor* in it's own thread.

4.10 Datomic Transaction Retry

It is possible for Datomic transaction to time out, and thus it is essential to have a retry strategy in place to combat this. It is further necessary to only retry on timeout, or any other such recoverable situation. It would be pointless to keep retrying on an application error, which will never correct itself.

An exponential backoff algorithm was used to retry up till a maximum of 60s. The algorithm is a slightly modified version recommended by Google when retrying requests on its cloud platform. [30] It consists of the following steps:

- Attempt the transaction.
- If the transaction fails, wait 1 + a random number of milliseconds and retry
- If the transaction fails, wait 2 + a random number of milliseconds and retry
- If the transaction fails, wait 4 + a random number of milliseconds and retry
- Repeat until the maximum back-of time of 60s, and continue making requests at 60s.

The random number of milliseconds is an integer between 0 and 1000.

4.11 Update Handler

Datomic provides it's peers with access to a transaction report queue, which provides live updates of any transactions currently happening on the system.

Transactions are simple data structures, and thus they are trivial to parse and extract the transaction entity, which contains all the event data.

In this project the *Update Handler* illustrates a real-time server to client broadcast. For each transaction received the event data is extracted, and broadcast to all connected clients.

Different applications will have different strategies for updates. Sensitive events might be visible to only as select group of users, and thus it would be up to the developers to implement logic which checks the incoming event's data, and user identifier, and determine which other connected users to send events to.

The *Update Handler* does it's push via the *Websocket Server*, which contains the an index of user identifier to Websocket connection. It thus must be co-located with the *Websocket Server*. It can however run in it's own dedicated thread.

4.12 Query Service

Queries enable access to database entries produced by aggregating events.

The project emphasizes a single query endpoint. This may seem strange at first, but it has the benefit of keeping the query service simple.

The query endpoint accepts queries in EDN format, as a Clojure map with the following fields:

Field	Type	Description
Name	Clojure Keyword	The name of the query.
Data	Clojure Map	The query parameters.

An example query:

```
{:query/name list-page-views :query/data {}}
```

Queries consist of two stages. Validating the incoming query data, and then performing the query and returning the results. Both of these cases are covered by Clojure multi-methods and dispatching on the name of the query.

Thanks to Datomics excellent read scaling it is possible to have multiple query services running independently. This is made possible by the Datomic's design choice to keep a portion of the database (or the entire database if the is enough memory available) in memory via the Datomic Peer. It is thus simple for the application to handle increased read load, and scale appropriately.

5 Implementation

This section details the implementation of the design, and consists primarily of actual code snippets with some minor commentary.

5.1 Project Setup and Dependencies

The project uses *Leiningen*, the standard Clojure build tool to manage build the project and management dependencies. The *project.clj* file is shown below. It offers a declarative way of describing a project.

```
(defproject calderwood "0.1.0-SNAPSHOT"
  :description "Calderwood: An opinionated reference for
                event sourced applications."

  :license {:name "MIT"
            :url  "http://"}

  :dependencies [[org.clojure/clojure "1.9.0-alpha14"]
                 [org.clojure/clojurescript "1.9.293"]
                 [org.clojure/core.async "0.2.385"]
                 [org.clojure/java.classpath "0.2.3"]

                 [http-kit "2.2.0"]
                 [ring/ring-core "1.5.0"]
                 [com.datomic/datomic-free "0.9.5544"
                  :exclusions [com.google.guava/guava]]

                 [io.rkn/conformity "0.4.0"]

                 [bultitude "0.2.8"]

                 [clj-time "0.12.0"]

                 [com.taoensso/timbre       "4.8.0"]

                 [clojurewerkz/scrypt "1.2.0"]

                 [compojure                 "1.5.1"]
                 [hiccup                    "1.0.5"]

                 [rum "0.10.7"]
                 [bidi "2.0.9"]
                 [cljs-http "0.1.42"]]

  :plugins [[lein-cljsbuild "1.1.5"]]

  :profiles {:dev {:dependencies [[org.clojure/tools.namespace "0.2.11"]
                                  [stylefruits/gniazdo "1.0.0"]
                                  [org.hdrhistogram/HdrHistogram "2.1.7"]
                                  [incanter/incanter-core "1.5.6"]
                                  [incanter/incanter-charts "1.5.6"]]
                   :source-paths ["dev" "test"]}}

  :cljsbuild {:builds [{:id           "console-dev"
                        :source-paths ["src/calderwood/console"]
                        :compiler     {:output-to
                                       "resources/public/js/console.js"
                                       :optimizations :whitespace
                                       :pretty-print  true}}]}})
```

Dependencies of interest:

- *Http-kit*, a high performance Clojure web server, with good Websockets support, and a simple clean Websocket API.
- *Ring*, a Clojure HTTP abstraction, similar to Ruby's *Rack*, or Python's *WSGI*.

- *Datomic free*, the Datomic peer library for the free version.
- *Compojure*, an HTTP routing library.
- *Scrypt*, an library for generating Scrypt based password hashes.

5.2 Component system

The project relies on ideas from Stuart Sierra's component library, but does not actually include the library, since we have so few components to manage, however all the principles suggested by him apply. [31]

Each component is defined as a Clojure record, which is essentially a class which supports hash map like operations, such as *assoc*, annd *dissoc*. Behind the hood it actually does compile to a Java class, unlike normal Clojure maps. Records also allow a protocol to be implemented, which is essentially like a Java interface. Thus each component also implements the Lifecycle protocol, which has the methods start and stop. The system itself is just a record which includes all the other components, and is a component itself, which can be started and stopped.

Below we see each component being initialized and started. The order in which components are started and stopped does matter, and hence a bit of boilerplate code is needed to get everything wired up.

We can see all the components in the design:

- *datomic*
- *ws-channels* is the component which holds the Websocket connections.
- *update-handler*
- *command-processor*
- *app-handler* is the component which contains the *Query Service*.

The component model allows us to store all application state in a single place, greatly reducing the amount of global state, and allows us to predictably start and restart our application during interactive development.

We also add a shutdown hook, which will stop the system, and clean up any open connections etc, before the Java Virtual Machine (JVM) shuts down.

```
(defrecord DevSystem [datomic
                      local-command-queue
                      ws-channels
                      app-handler
                      http-kit-web-server
                      update-handler
                      command-processor]
  Lifecycle
  (start [component]
    (let [datomic* (start datomic)
          app-handler* (-> app-handler
                           (assoc :datomic datomic*
                                  :ws-channels ws-channels
                                  :command-queue local-command-queue)
                           start)
          http-kit-web-server* (-> http-kit-web-server
                                   (assoc :app-handler app-handler*)
                                   start)
          update-handler (-> update-handler
                             (assoc :datomic datomic*
                                    :ws-channels ws-channels)
                             start)
          command-processor (-> command-processor
                                (assoc :datomic datomic*
                                       :local-command-queue
                                       local-command-queue)
                                start)]

      (.addShutdownHook (Runtime/getRuntime)
                        (Thread. (fn []
                                   (timbre/info "Shutdown hook")
                                   (stop component))))
      (assoc component
             :datomic datomic*
             :ws-channels ws-channels
             :app-handler app-handler*
             :http-kit-web-server http-kit-web-server*
             :update-handler update-handler
             :command-processor command-processor)))
  (stop [component]
    (timbre/info "Shutting down")
    (assoc component
           :datomic (stop datomic)
           :http-kit-web-server (stop http-kit-web-server)
           :update-hander (stop update-handler)
           :command-processor (stop command-processor))))

(defn dev-system [{:keys [db-uri]}]
  (map->DevSystem {:datomic (temp-datomic db-uri)
                   :ws-channels (ws-channels)
                   :local-command-queue (local-command-queue 1000)
                   :app-handler (app-handler)
                   :http-kit-web-server (http-kit-web-server 8080)
                   :update-handler (update-handler)
                   :command-processor (command-processor)}))
```

5.3 HTTP API endpoints

We define HTTP routes using the Compojure library, which gives a declarative way of expressing all our application routes. Each route takes a request, and returns a response, and conforms the Ring Specification. Requests and responses are simply Clojure maps. [32]

The Ring Specification also describes middleware, which are essentially higher order functions, which wrap request handlers, and are able to inject information into the request before or after the handler receives it. Or it could modify the response of the handler, or it can choose not to call the handler at all and simply return an error response, there are many possibilities. [32]

Our routes include some standard Ring middleware, such as *wrap-keyword-params* which automatically converts incoming form data via a GET or POST into a more usable Clojure map. We also include some of our own middleware, such as *wrap-identity*, which attempts to add a user's UUID to the request, assuming there is a valid session UUID in the session cookie.

The routes are then encapsulated by the *AppHandler* component, which gets passed to the web sever when the system is started.

```
(defn create-handler [datomic command-queue ws-channels]
  (-> (compojure/routes
       (compojure/GET "/console" request (console-handler request))
       (compojure/GET "/ws" request (ws-handler ws-channels
                                                command-queue
```

```
                                               request))
       (compojure/POST "/query" request (query-handler request))
       (compojure/GET "/login" request (view-login-handler request))
       (compojure/POST "/login" request (login-handler request))
       (compojure/GET "/logout" request (logout-handler request))
       (compojure.route/resources "/"))
      (wrap-impersonate)
      (wrap-identity)
      (wrap-session)
      (wrap-keyword-params)
      (wrap-params)
      (wrap-components datomic ws-channels)))


(defrecord AppHandler [datomic command-queue ws-channels handler]
  Lifecycle
  (start [component]
    (if-not handler
      (assoc component :handler (create-handler
                                 datomic
                                 command-queue
                                 ws-channels))
      component))
  (stop [component]
    (assoc :handler nil)))

(defn app-handler []
  (map->AppHandler {}))
```

5.4 Authentication and Authorization

The implementation follows the recommendations in the design chapter. The login handler takes care of authenticating the user's credentials, creating a session in the database, and returning an encrypted session cookie with the created session UUID.

We also see the implementation of the *wrap-identify* middleware, which tries to find a valid session in the database and a matching user. If it finds a valid user UUID, it then attaches it to the request under an *:identity* key.

To logout, we simply clear the session cookie, by setting it to *nil*.

```clojure
(defn user-credentials-valid? [db user password]
  (util/password-hash-valid? password
                             (:user/password-digest user)))

(defn create-session-tx-map [user-uuid remote-address]
  {:db/id (d/tempid :db.part/user)
   :session/uuid (d/squuid)
   :session/user [:user/uuid user-uuid]
   :session/remote-address remote-address})

(defn create-session! [conn user-uuid remote-address]
  (let [session-tx-map (create-session-tx-map
                        user-uuid
                        remote-address)
        session-uuid (:session/uuid session-tx-map)
        db-after (:db-after
                  @(d/transact conn [session-tx-map]))]
    (d/entity db-after [:session/uuid session-uuid])))

(defn login-handler [{:keys [db conn params] :as request}]
  (let [{:keys [email password]} params
        email* (-> email
                   (string/lower-case)
                   (string/trim))]
    (if-let [user (d/entity db [:user/email email*])]
      (if (user-credentials-valid? db user password)
        (let [session (create-session! conn
                                       (:user/uuid user)
                                       (:remote-addr request))]
          (-> (ring.response/redirect "/console" :see-other)
              (assoc-in [:session :session-uuid]
                        (:session/uuid session))
              (update :session
                      (fn [s] (vary-meta s assoc :recreate true)))))
        (ring.response/redirect "/login?error=true" :see-other))
      (ring.response/redirect "/login?error=true" :see-other))))

(defn logout-handler [request]
  (-> (ring.response/redirect "/login" :see-other)
      (assoc :session nil)))

(defn wrap-identity [handler]
  (fn [{:keys [db session] :as request}]
    (if-let [session (d/entity db
                               [:session/uuid (:session-uuid session)])]
      (handler (assoc request
                      :identity
                      (-> session :session/user :user/uuid)))
      (handler request))))
```

5.5 Websocket Server

The *Websocket Server* relies directly on the command-queue, and Websocket channels.

When the *ws-handler* function initially gets called we need to first check if the request has been authenticated, by looking for the *:identity* key in the request, if this is missing we immediately return an *unauthorized* response.

If the request is indeed authenticated, then we add the user's UUID to the user UUID to Websocket connection index.

HTTP-Kit provides a simple API for further dealing with Websocket connections. There are two callbacks that need to be registered:

- *On close*, when the Websocket connection closes, in this case we need to remove that user from the index of user UUID to Websocket connection.
- *On receive*, gets called when a message is received from the connection. The message needs to be validated, and either put on the command queue, and a command acknowledge returned, or an error message should be sent to the client.

```
(defn ws-handler [ws-channels command-queue request]
  (timbre/debug request)
  (if-let [id (:identity request)]
    (http-kit/with-channel request channel
      (timbre/debug "Websocket channel opened for user id:" id)
      (swap! (:channels ws-channels) assoc id channel)
      (http-kit/on-close
       channel
       (fn [status]
         (timbre/debug "Websocket channel closed for user id:" id)
         (swap! (:channels ws-channels) dissoc id)))
      (http-kit/on-receive
       channel
       (fn [msg]
         (timbre/debug "Received WS message:" msg)
         (let [[tp client-id client-seq data]
               (clojure.edn/read-string msg)]
           (when (and (= tp :cmd)
                      client-id
                      client-seq)
             (let [[ok-or-err cmd err]
                   (-> data
                       (assoc :command/user-uuid id
                              :command/client-id client-id
                              :command/client-seq client-seq)
                       (coerce-command))]
               (if (= :error ok-or-err)
                 (http-kit/send! channel (pr-str [:error
                                                  client-id
                                                  client-seq
                                                  err]))
                 (do (put-command command-queue cmd)
                     (http-kit/send! channel
                                     (pr-str [:cmd-ack
                                              client-id
                                              client-seq]))))))))))
    (not-authorized "Not authorized")))
```

5.5.1 Command Queue

We specify a protocol called CommandQueue, which *Command Queue*-like components can implement. There is only one method, *put-command*, which adds the command given to the queue. A protocol was chosen here to allow other queue implementations possible, without having to modify any code in the queue's producer. This makes it possible to swap in something like Apache Kafka, as opposed to the *ArrayBlockingQueue* implementation we went with below.

```
(defprotocol CommandQueue
  (put-command [queue cmd]))

(defrecord LocalCommandQueue [queue]
  CommandQueue
  (put-command [component cmd]
    (.put (:queue component) cmd)))


(defn local-command-queue [buffer-size]
  (LocalCommandQueue.
   (java.util.concurrent.ArrayBlockingQueue. buffer-size)))
```

5.6 Command Processor

The *Command Processor* runs in it's own thread, and consumes inter-process from the *Command Queue*. We choose to *poll* rather than *take* since we want to respect the *running?* boolean which controls when to stop the while loop, and take would block forever, until a command was put on the queue.

Incoming commands are converted to events by *handle-command* before being aggregated into Datomic transactions by *aggregate-event*. We then attempt to transact with an exponential backoff.

Note how we get a current value of the Datomic database, which we pass to both *handle-command*, and *aggregate-event*.

```
(defrecord CommandProcessor [local-command-queue datomic running?]
  Lifecycle
  (start [component]
    (reset! running? true)
    (.start
     (Thread.
      (fn []
        (timbre/debug "Starting Command Processor thread...")
        (while @running?
          (try
            (let [conn (d/connect (:db-uri datomic))
                  db (d/db conn)
                  cmd (.poll (:queue local-command-queue)
                             1000
                             java.util.concurrent.TimeUnit/MILLISECONDS)]
              (when cmd
                (timbre/info "Command received:" cmd)
                (let [event-txes (mapv (partial aggregate-event db)
                                       (handle-command db cmd))]

                  (doseq [e-tx event-txes]
                     (timbre/debug "Transacting event:" e-tx)
                     (transact-with-exponential-backoff-retry!
                      conn
                      running?
                      e-tx)))))
             (catch Exception err
               (timbre/error "Exception in Command Processor:" err)))))))
    component)
  (stop [component]
    (reset! running? false)
    component))

(defn command-processor []
  (CommandProcessor. nil nil (atom false)))
```

5.7 Datomic Transaction Retry

The transact retry follows the algorithm outlined in the design chapter. We ensure that we only retry on timeout or Transactor unavailable (if it's down) exceptions, and never on application logic related errors that will never be resolved by a retry.

```
(defn transact-with-exponential-backoff-retry! [conn running? txes]
  (let [max-backoff-ms 64000
        rand-sleep-ms 1000]
    (loop [backoff-time 1]
      (when @running?
        (or (try @(d/transact conn txes)
                 (catch java.util.concurrent.ExecutionException e
                   ;; Log an application error here, not a good
                   ;; idea to try again since it won't help.
                   (timbre/error e)
                   true)
                 (catch clojure.lang.ExceptionInfo e
                   ;; Log transaction timeout/unavailable here
                   ;; exponential backoff and it might eventually
                   ;; reconnect.
                   (timbre/error e)
                   nil))
             (let [backoff-time (min backoff-time max-backoff-ms)
                   backoff-time* (+ backoff-time
                                    (rand-int rand-sleep-ms))]
```

```
               (timbre/warn "Transact failed, retrying again in:"
                            backoff-time*
                            "ms")
               (Thread/sleep backoff-time*)
               (recur (* 2 backoff-time))))))))
```

5.8 Update Handler

The *Update Handler* component also runs in it's own thread. During each iteration, we use the *tx-report-queue* function to obtain a reference to the queue. We then use *poll* for similar reasons as the *Command Processor*.

Queue items are maps, which contain keys for the transaction data (*tx-data*) plus references to the value of the database before and after the transaction. after. We pull off the value after, and use it to resolve the transaction entity. To do this we first need to obtain the current *t* value of the after database, and use an API call to convert it to a valid transaction entity id for that point in time. Finally we are able to use Datomic's entity API to fetch the transaction entity itself, and extract the event data. We then broadcast this event data to all connected Websocket clients, completing the command round trip

```
(def MILLISECONDS java.util.concurrent.TimeUnit/MILLISECONDS)

(defrecord UpdateHandler [ws-channels datomic running?]
  Lifecycle
  (start [component]
    (reset! running? true)
    (.start
     (Thread.
      (fn []
        (timbre/debug "Starting Update Handler thread...")
        (while @running?
          (try
            (let [conn      (d/connect (:db-uri datomic))
                  txr       (d/tx-report-queue conn)
                  tx-report (.poll txr
                                   1000
                                   MILLISECONDS
                                   )]
              (when tx-report
                (timbre/debug "Received update:" tx-report)
```

```
                 (let [{:keys [db-after tx-data]} tx-report
                       tx (d/t->tx
                           (d/basis-t db-after))
                       {:keys [event/uuid
                               event/name
                               event/data
                               event/meta
                               event/client-id
                               event/client-seq]
                        :as tx-entity} (d/entity db-after tx)]
                   (when (:event/name tx-entity)
                     (timbre/debug "Update transaction entity:"
                                   (d/touch tx-entity))
                     (doseq [channel (vals @(:channels ws-channels))]
                       (http-kit/send!
                        channel
                        (pr-str
                         [:update
                          client-id
                          client-seq
                          {:event/uuid uuid
                           :event/name name
                           :event/data (clojure.edn/read-string data)
                           :event/meta (clojure.edn/read-string meta)
                           :event/client-seq client-seq}])))))))
             (catch Exception err
               (timbre/error "Exception in Update Handler:" err)))))))
    component)
  (stop [component]
    (reset! running? false)
    component))

(defn update-handler []
  (UpdateHandler. nil nil (atom false)))
```

5.9 Query Service

The *Query Service* is able to exist completely independently of any of the other components, it's only requirement is access to a Datomic Peer. However for our purposes it was fine to simply embed it with our existing routes in the *AppHandler*.

The service expects an incoming *POST* with an EDN map, containing a valid query request. We use a utility function to convert the body *InputStream* into a Clojure map, which is then passed to the coerce-query function for validation.

The validation function returns a tuple of either *:ok*, and the validated data, or *:error* with a possible reason for the failure. We also attach the identity of the user making the request to the query data before it gets validated, in case it is needed for a particular query.

If validation passes, then we can perform the query via the *do-query* function, and return an EDN response with the query results, otherwise we return an EDN based error response.

```
(defn query-handler [request]
  (if-let [id (:identity request)]
    (if-let [query (try (util/input-stream->edn (:body request))
                        (catch Exception e
                          (timbre/error "Error parsing query body:" e)
                          nil))]
      (let [[ok-or-error query* :as query-result]
            (coerce-query (assoc query
                                 :query/user [:user/uuid id]))]
        (if (= :error ok-or-error)
          (edn-response query-result)
          (edn-response [:ok (do-query (:db request)
                                       query*)])))
      (bad-request))
    (not-authorized)))
```

6 Testing

6.1 Functionality Testing

6.2 Adding a new command

Adding a new command for a visiting a page was simply a case of extending three multi-methods.

```
(defmethod -coerce-command :visit-page [{:keys [command/data] :as command}]
  (if (and
       (:command/user-uuid command)
       (s/valid? :command.data/visit-page data))
    [:ok command]
    [:error command :command-validation-failed]))

(defmethod -handle-command :visit-page [db {:keys [command/data
                                                   command/meta]
                                            :as command}]
  [(event command :page-view data meta)])


(defmethod -aggregate-event :page-view [db {:keys [event/data]
                                            :as event}]
  [{:db/id (d/tempid :db.part/user)
    :page-view/url (:page-view/url data)
    :page-view/user (:event/user event)}])
```

6.3 Adding a new query

Adding a query is just as straight forward.

```
(defmethod -coerce-query :list-page-views [query]
  [:ok query])

(defmethod -do-query :list-page-views [db query]
  (->> (d/q '[:find ?p ?tx
              :where [?p :page-view/url _ ?tx]]
            db)
       (mapv (fn [[p tx]]
               (let [{:keys [page-view/url
                             page-view/user]} (d/entity db p)]
                 {:page-view/timestamp (:db/txInstant (d/entity db tx))
                  :page-view/url url
                  :page-view/user (select-keys
                                   user
                                   [:user/uuid
                                    :user/email])})))))
```

6.3.1 Login

Testing logging in, see fig. 7

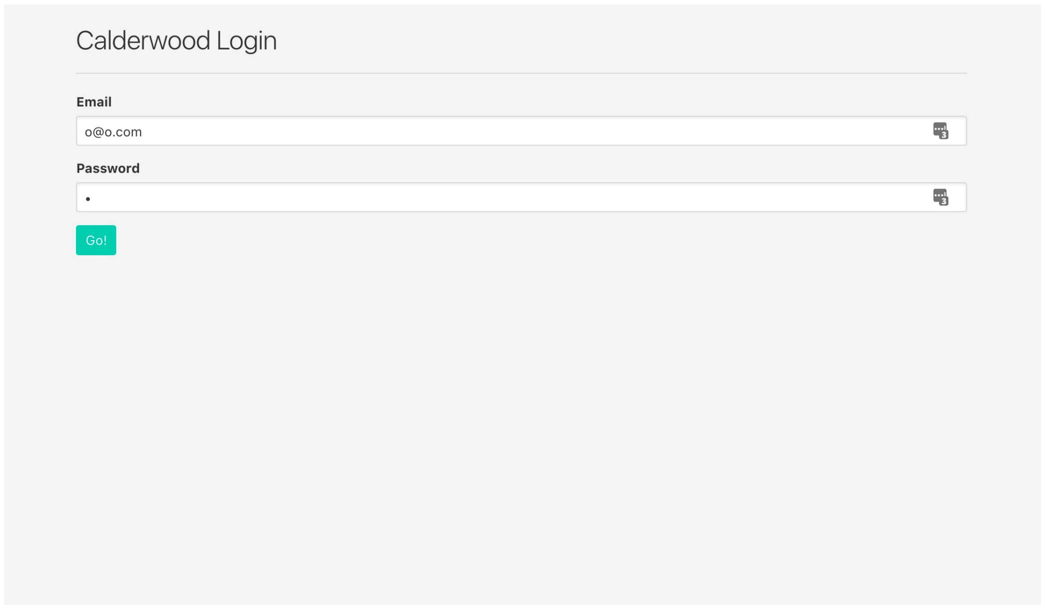


Figure 7: Login test.

6.3.2 Sending a command

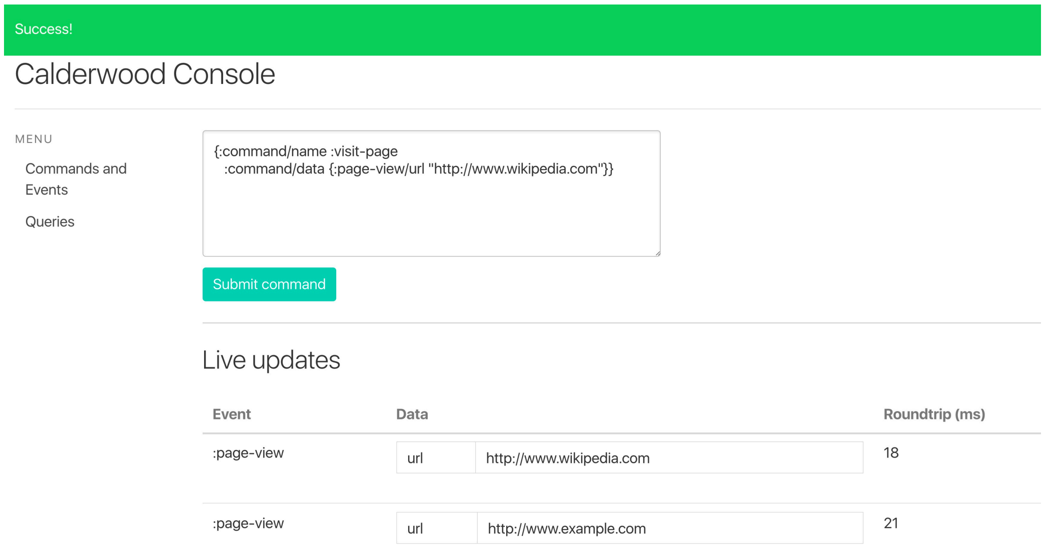
Sending a command using the console. Using a valid command is shown in fig. 8, using an invalid command is shown in fig. 9.

6.3.3 Performing a query

Performing a query using the console. Performing a valid query is shown in fig. 10, using an invalid command is shown in fig. 11.

6.4 Performance Testing

Performance testing was done using a modern laptop with the following specifications:


Success!

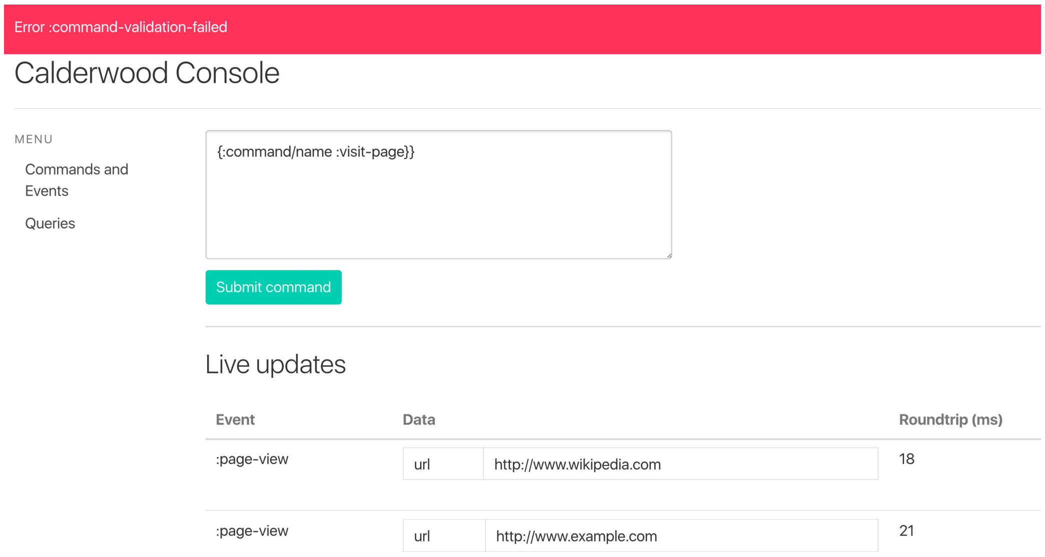
Calderwood Console

```
{:command/name :visit-page
  :command/data {:page-view/url "http://www.wikipedia.com"}}
```

Live updates

Event	Data		Roundtrip (ms)
:page-view	url	http://www.wikipedia.com	18
:page-view	url	http://www.example.com	21

Figure 8: Send command.


Error :command-validation-failed

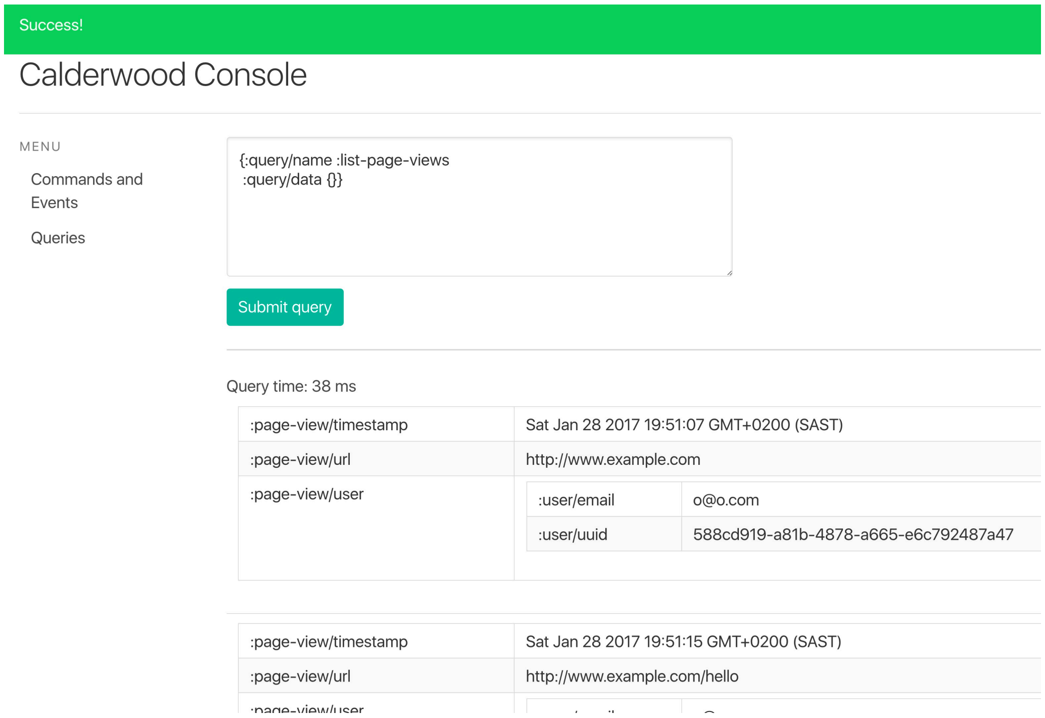
Calderwood Console

```
{:command/name :visit-page}}
```

Live updates

Event	Data		Roundtrip (ms)
:page-view	url	http://www.wikipedia.com	18
:page-view	url	http://www.example.com	21

Figure 9: Send incorrect command.



Success!

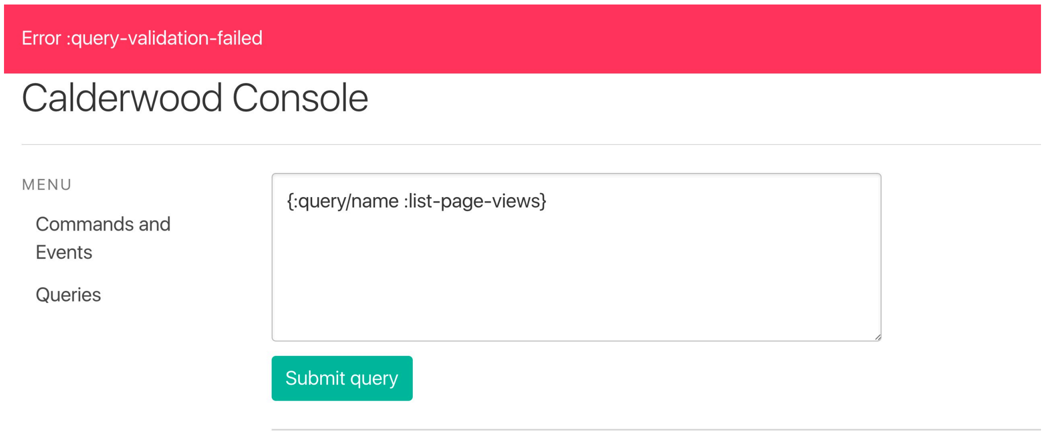
```
{:query/name :list-page-views
 :query/data {}}
```

Query time: 38 ms

:page-view/timestamp	Sat Jan 28 2017 19:51:07 GMT+0200 (SAST)
:page-view/url	http://www.example.com
:page-view/user	:user/email o@o.com :user/uuid 588cd919-a81b-4878-a665-e6c792487a47

:page-view/timestamp	Sat Jan 28 2017 19:51:15 GMT+0200 (SAST)
:page-view/url	http://www.example.com/hello

Figure 10: Send query.



Error :query-validation-failed

```
{:query/name :list-page-views}
```

Figure 11: Send incorrect query.

- Operating System: OSX (10.11.6)
- CPU: 2.6 Ghz Intel Core i7
- Memory: 16 GB 1600 MHZ DDR3
- Disk: 500 GB SSD

The code used for performance testing is shown below.

```
(def CLIENT-ID "12345")

(defn ws-uri-for-email [email]
  (str "ws://localhost:8080/ws?login-email=" email))

(defn output-results [results]
  (->> results
       (map (fn [{:keys [arrive-ms msg]}]
              (let [[tp _ _ data] (read-string msg)]
                (when (= tp :update)
                  (- arrive-ms
                     (-> data
                         :event/meta
                         :sent-ms))))))
       (remove nil?)
       (output-percentile-distribution)))

(defn run-perf-test [iterations interval-us]
  (let [results (volatile! [])
        latch (java.util.concurrent.CountDownLatch. 1)
        ws-socket (ws/connect
                   (ws-uri-for-email "o@o.com")
                   :on-receive
                   (fn [msg]
                     (let [results
                           (vswap! results
                                   conj
                                   {:arrive-ms (System/currentTimeMillis)
                                    :msg msg})]
                       (when (= (count results) (* 2 iterations))
                         (.countDown latch)))))
        start-ms (System/currentTimeMillis)
        _ (doseq [i (range iterations)]
            (busy-wait interval-us)
```

```
             (ws/send-msg ws-socket
                          (pr-str
                           [:cmd CLIENT-ID i
                            {:command/name :visit-page
                             :command/data
                             {:page-view/url "www.example.com"}
                             :command/meta
                             {:sent-ms (System/currentTimeMillis)}}])))
         _ (.await latch)
         ops-per-second
         (int (/  (* iterations 1000)
                  (- (System/currentTimeMillis) start-ms)))]

     {:ops-per-second ops-per-second
      :results @results}))
```

The strategy for the code above is as follows:

- Establish a Websocket connection using a server-side client Websocket library.
- Initialize a CountDownLatch, which can be used as a barrier for completion.
- Register the *on-receieve* handler for that connection, and for each message that arrives, add it to a Clojure *volatile!* which is essentially a faster atom, with less guarantees. We don't need those guarantees though since there is a single thread writing to it. Include the arrival time with each message. When iterations * 2 messages have been received, issue a count down on the latch. We need 2 * iterations here because each send message will have both an acknowledgment response, and an update response.
- Send **iterations** *page-view* messages (10000 in this case) through the Websocket connection, using a busy wait function to control throughput, since *Thread/sleep* is not fine grained enough.
- Await the CountDownLatch.
- Operations per second are measured based on how long it takes to send all iterations.

- The raw results, together with the operation per second are returned.

The results can then be processed using the HDRHistogram tool, and a percentile distribution can be obtained. Inspiration for this method was taken from Gil Tene, of Azul systems (a company that makes it's own proprietary JVM implementation which doesn't pause during garbage collection), and his excellent talk on "How NOT to measure latency" given at Strange Loop in 2015. [33]

We are primarily interested in the max throughput we can achieve while achieving sub 100ms latencies for the 99.9th percentile.

The maximum stable throughput with the above goal is 320/ops, and is shown in fig. 12.

Beyond this point we quickly approach latency values way above our target, see fig. 13.

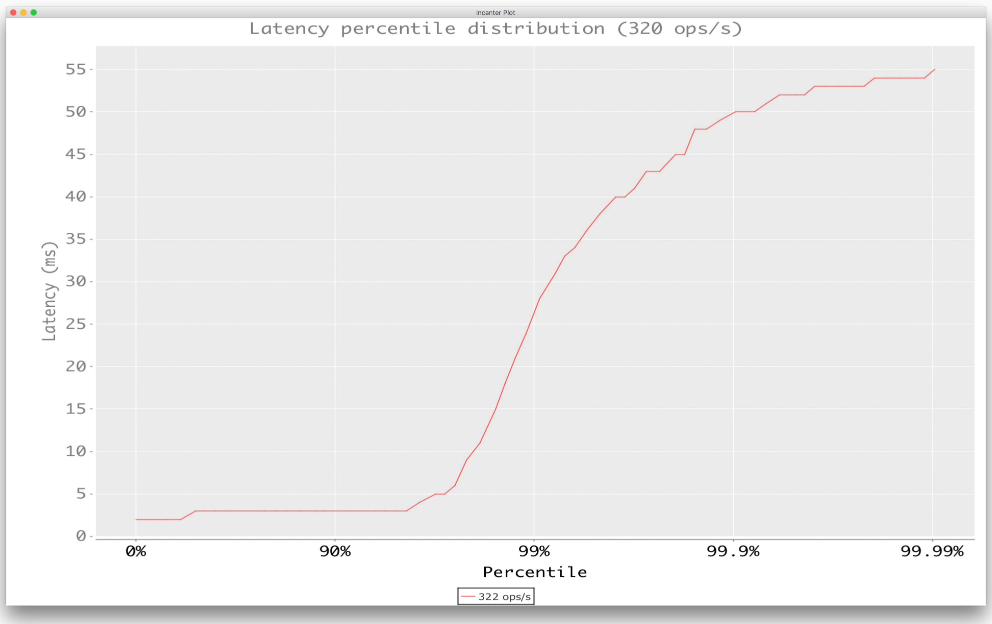


Figure 12: Latency percentile distribution for 320 ops/s.

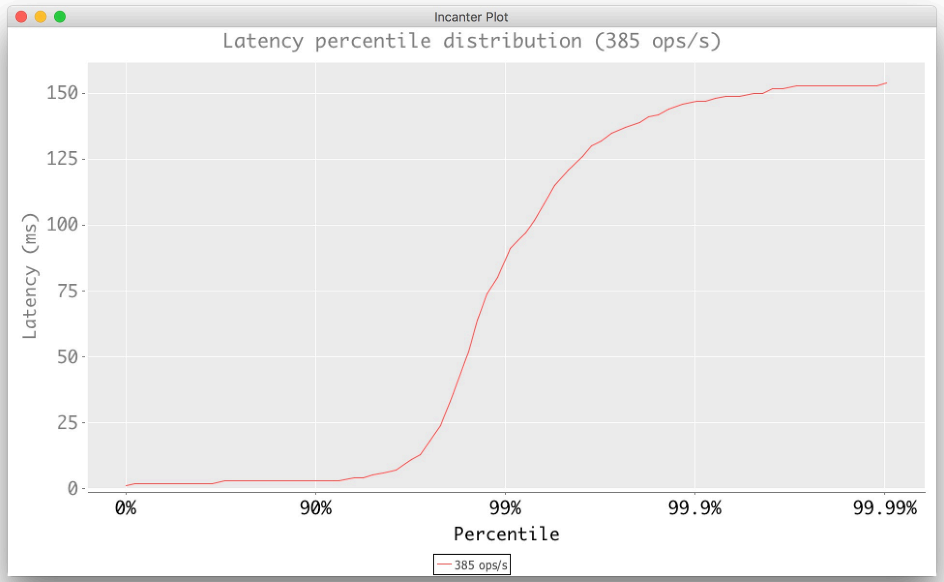


Figure 13: Latency percentile distribution for 385 ops/s.

7 Conclusion

The overall conclusion is that the project was a success. All the project objectives were met, noting the following:

- Since we use an immutable database, with a logical query language, and a fully event driven architecture, we are sure that data analysis will be far simpler on this system compared to traditional web applications. We gain an audit trail as a by-product of this.
- The system supports 320/ops on a single server, with modest hardware. This translates into 600-1200 concurrent users depending on how often users submit commands, if we assume each user takes 2-4 seconds between each command.
- The system is fully asynchronous and requests are decoupled from responses when submitting commands.
- The architecture is flexible, and key components, such as the *Query Service*, and *Command Processor*, can be separated and moved out into their own processes if need be.
- It is simple to add new functionality to the system, in the case of commands it only requires implementing three multimethods, and in the case of queries, two.

We would assume that a small to medium sized team could use and build on this application template to build a real world production system that is architected for data science.

8 Future Work and Recommendations

There are a number of areas which can be explored in future work, these include:

- Allowing for synchronous commands in some instances, instead of being purely asynchronous.

- Dealing with event versioning.
- Moving the *Command Processor* to it's own node, and connecting it to the rest of the system via a distributed queue such as Apache Kafka.
- Exploration of in memory aggregates instead of Datomic.

As for existing recommendations, it would be good to explore means of making the current implementation more performant, perhaps by using Records more instead of Maps, and finding other ways to optimize, this would require profiling the code in order to find bottlenecks.

References

[1] B. Calderwood, "From REST to CQRS with Clojure, Kafka, and Datomic." 2015.

[2] "CQRS-server," *Yuppiechef.github.io*. 2015 [Online]. Available: https://yuppiechef.github.io/cqrs-server/. [Accessed: 08-Sep-2016]

[3] "Data scientist: The sexiest job of the 21st century." 2016 [Online]. Available: https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century. [Accessed: 14-Nov-2016]

[4] G. Young, "A decade of DDD, CQRS, Event Sourcing." 2016.

[5] R. Hickey, "The value of values." 2012.

[6] J. Webber, S. Parastatidis, and I. Robinson, *REST in practice*. O'Reilly Media, 2010.

[7] S. Yegge, "Stevey's blog rants: Execution in the kingdom of nouns," *Steve-yegge.blogspot.co.za*. 2016 [Online]. Available: http://steve-yegge.blogspot.co.za/2006/03/execution-in-kingdom-of-nouns.html. [Accessed: 13-Sep-2016]

[8] J. W. Foreman, *Data smart: Using data science to transform information into insight*. Wiley, 2013.

[9] M. Loukides, "What is data science?" *O'Reilly Media*. 2010 [Online]. Available: https://www.oreilly.com/ideas/what-is-data-science. [Accessed: 24-Sep-2016]

[10] M. Fowler, "Event sourcing," *martinfowler.com*. 2005 [Online]. Available: http://martinfowler.com/eaaDev/EventSourcing.html. [Accessed: 08-Sep-2016]

[11] M. Thompson, "Event sourced architectures for high availability." 2012.

[12] G. Young, "CQRS documents," *CQRS*. 2010 [Online]. Available: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf. [Accessed: 08-Sep-2016]

[13] M. Fowler, "The LMAX architecture," *martinfowler.com*. 2011 [Online]. Available: http://martinfowler.com/articles/lmax.html. [Accessed: 08-Sep-2016]

[14] M. Fowler, "CQRS," *martinfowler.com*. 2011 [Online]. Available: http://martinfowler.com/bliki/CQRS.html. [Accessed: 08-Sep-2016]

[15] B. Moseley and P. Marks, "Out of the tarpit," in *Software practice advancement (SPA)*, 2006.

[16] S. Newman, *Building microservices*. O'Reilly Media, 2015.

[17] *Techblog.netflix.com*. 2016 [Online]. Available: http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html. [Accessed: 13-Sep-2016]

[18] M. Kleppmann, "Making sense of stream processing," *O'Reilly Media*. 2016 [Online]. Available: https://www.oreilly.com/learning/making-sense-of-stream-processing. [Accessed: 13-Sep-2016]

[19] T. Montgomery, "Event-driven, the only way (it's gonna) fly!" 2014.

[20] V. Wang, F. Salim, and P. Moskovits, *The definitive guide to HTML5 WebSocket*. Springer, 2013.

[21] S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[22] M. Fogus and C. Houser, *The joy of Clojure*. Manning Publications, 2014.

[23] J. Hughes, "Why functional programming matters," *The Computer Journal*, vol. 32, no. 2, pp. 98–107, 1989.

[24] *Clojure*. 2016 [Online]. Available: https://clojure.org/. [Accessed: 14-Oct-2016]

[25] "Technology radar," *Clojure*. 2016 [Online]. Available: https://www.thoughtworks.com/radar/languages-and-frameworks/clojure. [Accessed: 14-Oct-2016]

[26] "Extensible Data Notation specification," *Clojure*. 2016 [Online]. Available: https://github.com/edn-format/edn. [Accessed: 14-Oct-2016]

[27] *Datomic*. 2016 [Online]. Available: http://www.datomic.com/rationale.html. [Accessed: 13-Sep-2016]

[28] 2016 [Online]. Available: http://www.flyingmachinestudios.com/programming/datomic-for-five-year-olds. [Accessed: 14-Dec-2016]

[29] 2016 [Online]. Available: https://clojure.org/reference/multimethods. [Accessed: 15-Dec-2016]

[30] 2016 [Online]. Available: https://cloud.google.com/storage/docs/exponential-backoff#example_algorithm. [Accessed: 15-Dec-2016]

[31] 2016 [Online]. Available: https://github.com/stuartsierra/component. [Accessed: 15-Dec-2016]

[32] "The ring specification." 2016 [Online]. Available: https://github.com/ring-clojure/ring/blob/master/SPEC. [Accessed: 15-Dec-2016]

[33] G. Tene, "How NOT to measure latency." 2015.