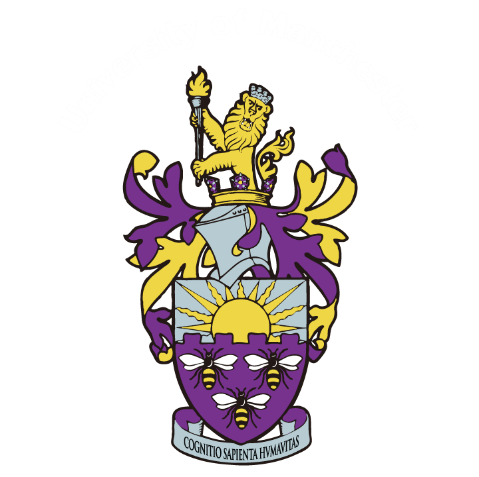
**Maintaining Data Consistency in a Microservice Architecture**

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER FOR THE DEGREE OF MASTER OF SCIENCE IN THE FACULTY OF SCIENCE AND ENGINEERING

Year of Submission

2020



**Michael John Richards**

Student Id Number: 10362882

School of Computer Science

University of Manchester

# Contents

List of figures viii

List of tables xi

1 Introduction

* 1. Motivation
  2. Research Aim
  3. Research Questions
  4. Objectives and Deliverables
  5. Dissertation Structure

“This dissertation is structured in 8 chapters, including the introduction. A brief description of the 7 other chapters is outlined as follows:”

2 Preliminaries – Aim to equip the reader with the necessary knowledge and terminology to understand the rest of the dissertation. (2-3 pages)

3 Some title – background theory/knowledge (origins of the problem) – what approaches have been done in the past? Discuss useful other solutions.

3.1 A Brief History

4-5 more chapters (additional background knowledge on a more technical level). A final chapter on what the dissertation is building and what it attempts to solve.

6 Design and Implementation

6.1 Software architecture (overview)

6.6 Dev environment and tools

6.7 Testing Methodology

7 Evaluation

8 Conclusion

8.1 Summary of Achievements

8.2 Reflection

8.3 Further Work

References 60

# Abstract

This dissertation

The main aim (evaluation) of this research….

# Declaration

I hereby declare that the contents of this dissertation is original except for references made to the work of others, and that no portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning. This dissertation contains the results of my own work and has not been used in collaboration.

# Copyright

1. The author of this dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.
2. Copies of this dissertation, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.
3. The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the dissertation, for example graphs and tables (“Reproductions”), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.
4. Further information on the conditions under which disclosure, publication and commercialisation of this dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/display.aspx?DocID=24420), in any relevant Dissertation restriction declarations deposited in the University Library, and The University Library’s regulations (see https://www.library.

manchester.ac.uk/about/regulations/\_files/Library-regulations.pdf).

# Acknowledgements

// Save until the end

Chapter 1

# Introduction

As the microservice trend continues to grow in popularity, most new enterprise systems are developed with microservices as a key implementation detail with old ones slowly migrating towards them. The number of academic research projects involving the use of microservices has also steadily increased with many aspects explored. From security, domain modelling, database and transaction management, distributed programming, and communication concerns, there are still many other aspects missing a well-defined solution to the technical challenges of maintaining such systems.

Microservice architecture (MSA) attempts to break up a large system into multiple sub-systems called microservices, which can then be individually developed, tested, and deployed in isolation. MSA attempts to improve horizontal scalability whilst providing high availability and reliability with the support of specialised infrastructure platforms and tools (e.g. load balancers with containerization). From a development perspective, this helps alleviate the technical debt of not being able to upgrade or replace old frameworks and databases without the fear of introducing system-wide breaking changes. It also means that separate development teams can work on individual microservices without clashing with other teams. This helps avoid merge conflicts when using version control, and less time is spent running system-wide unit tests because a microservice only needs to contain unit tests relevant to its behaviour.

Unfortunately, MSAs bring new risks and concerns that traditional monolithic applications lacked. With the rise of Domain-Driven Design (DDD), many microservices guard their own portion of the domain model by encapsulating the relevant data they need to work with inside their own datastore, and so microservices rely on specialized strategies to complete queries or transactions that span more than one service. Therefore, high latency is a potential bottleneck for system performance due to microservices experiencing downtime or high traffic. The ability for a microservice to recover from failure, known as fault tolerance, is vital to ensure that the state of a system remains consistent. If a transaction must span several services and one of the services fail, then the system’s data integrity is at risk and must be accounted for.

## Motivation

Many patterns and models for building a MSAs have been proposed to tackle the generic challenges that have been identified. However, all systems based off MSAs have their own unique requirements and use-cases that cannot make use of all proposed solutions, with many alternative approaches being developed as a result. Therefore, we are left with an open-ended list of possibilities that can discourage companies from adopting microservices. This is especially the case when the task is to migrate away from a monolithic architecture because the business risk can be huge, and so the task is often delayed further with the technical debt forever growing. This makes it even more difficult to migrate in the future.

Modern systems are discovering new requirements and patterns for dealing with these challenges, which led to the creation of the reactive manifesto; a set of guidelines agreed by the community for what characteristics a modern system should possess. These characteristics perfectly align with the goals of microservice architecture, but the challenge is designing such a system that complements them.

## Research Aim

These are the main motivating questions behind this research project with the hopes of gaining a better understanding of the technical complexities involved.

## 1.3 Research Questions

How can we implement a responsive, resilient, elastic and message driven system that prevents data inconsistencies when practicing domain-driven design, and still reply to the client that invoked the request in a timely manner? With all the added complexity caused from splitting up the domain and relying on messages to communicate between microservices, what patterns can we employ to reduce the amount of boilerplate code when creating new features whilst designing for change? How can we build fault-tolerant systems that put data consistency concerns first to reduce the business risk involved when migrating from a monolithic architecture to a microservice architecture?

## 1.4 Objectives and Deliverables

We consider 4 types of messages = command, queries, multi-queries and saga commands.

The project is first evaluated on its use of patterns from a qualitative viewpoint to assess the development experiencing and the benefits they provide. Special attention is paid towards the <changability, scalability, etc…) benefits provided. Finally, quantitative evaluation is performed against the number of workers processing data in parallel while recording performance metrics for each type of message. One of the main goals for this research is to ensure that while maintaining data consistency is important, we also need to ensure that the system is reactive and so the system must respond within a timely manner. Therefore, we must record the response times for each message and the memory usage for each worker. We also must evaluate the fault tolerance for when a worker, or the entire system, becomes unresponsive and how effective the system is at recovering from faults without corrupting the domain data.

Chapter 2

# Inter-Process Communication and Domain Modelling

There have been many strategies proposed for handling inter-process communication (IPC) within a distributed system, and many ways for modelling their domain. A controversial debate that is still often discussed, although arguably has died down over the years, has been the move to a network transparent, object-oriented programming model. Developers are initially intrigued with the paradigm of modelling a distributed system in such a way as to mask the underlining network communication involved with passing messages between remote systems. Many frameworks have been proposed over the years to mimic local method invocations, but under the hood are then routed to remote systems to be called on remote objects. Thus, developers have the misconception of believing that there is no difference between local and distributed programming. Unfortunately, there are several key factors where these two paradigms intrinsically differ and fail as a result.

## 2.1 A Brief History of Inter-Process Communnication

Traditionally, all data interactions were conducted locally on the same machine. This is referred to as local computing when computing is restricted to a single address space as opposed to distributed computing where operations are computed across multiple address spaces [1]. Developers have more control over how to react to failure because the dataflow can be easily monitored and we are guaranteed a response, whether that response is successful or not. When we rely on remote services, the network is unpredictable and so we cannot guarantee that a request ever reached its destination. If a remote system carried out our request but failed to send an acknowledgement of any kind, then it becomes a far greater challenge to implement fault tolerance and reliability.

Several middleware strategies have been developed to address the unique requirements of distributed systems. Remote object invocation (also referred to as remote method invocation (RMI)), is an older approach that attempts to implement “the vision of unified objects” [1], where remote method calls as disguised as local ones and the underlining framework handles network specific details. This was seen by developers as the next step to remote procedure invocation (or remote procedure calls, RPC) that achieves the same result except without the native support for object-oriented modelling. A unified object paradigm (also known as a distributed object paradigm) relies on well-defined proxy interfaces, as a way of communicating between objects, declared using an interface definition language. Implementation details are then hidden from the programmer, allowing the same interface to be used regardless of where the object is located (i.e. remotely or locally). The underlining system can then select the appropriate delivery strategy and accommodate the characteristics of the network. The Object Management Group’s Common Object Request Broker Architecture (CORBA) is an early standard that proposes such a system. Other examples include SOAP and Enterprise JavaBeans (ELB) [2].

While there were some benefits to these designs, such as taking away the burden of low-level IPC concerns and allowing developers to focus on the business logic, the characteristics of the network could never truly be ignored. An obvious characteristic of remote messaging is the latency increase as opposed to calling functions locally. This can be salvaged by upgrading hardware to increase processing power, using caching intelligently, and attempting to reduce the number of calls by keeping objects that often collaborate in the same address space. However, less obvious characteristics began to emerge, which created a difficult obstacle for achieving complete network transparency using this paradigm. Developers often fell victim to assumptions about the network and the illusion that they were working on a traditional object-oriented programming model.

Developers using an RPI-styled IPC mechanism would often make the misguided assumption that the invocation would respond within a timely manner. This made it difficult to guarantee responsiveness within their application. Another issue is assuming that all objects had access to shared memory, making pointers to object references difficult to implement. If the technology caters shared object references in a distributed fashion, there is still the issue of handling concurrency. Because any object must potentially handle multiple concurrent method calls, the developer must use synchronous concurrency mechanisms to lock resources where appropriate, thus this breaks the vision of a transparent network. Multiple threads interacting with the same object and its state must be carefully coordinated, which limits concurrency and can become very costly even for modern CPU architectures [3].

RPI-styled middleware designs, such as CORBA, were popular before focus shifted towards newer trends, such as service oriented architecture (SOA) and web service technologies [4]. Newer trends favoured explicit IPC strategies to avoid the locking and blocking of requests caused by transferring threads to maintain object references. Message-driven architectures send serializable messages to remote services that are then picked up and executed on a single thread. There is no need to share object references as the response is sent back to the sender using a separate message rather than as a method’s returned value. This also promotes asynchronous behaviour as the client is not blocked from processing further. The client instead makes use of callback functions to process the result only when the expected message is received.

## 2.2 Message-Driven Architecture

There are several ways to implement a message-driven architecture. Enterprise systems often make use of a message-bus or messaging queue where messages are placed onto the queue via a message channel. A remote service then pulls the messages from the queue that have been assigned to them. This style of pulling messages from a queue differs from the traditional push architectures, such as REST or RPC, because they can be processed asynchronously without blocking trends and facilitates scalability via the use of load balancers to increase the number of workers that can pull from the queue.

However, one disadvantage with this technique is when the client relies on a predefined dataflow. For example, if a request must contact one or more remote services and expects a specific result back, it can feel overly complicated to retrieve the expected response. You can setup a temporary messaging channel for sending the response back to the client, create a separate REST API on the client to handle responses, or broadcast the message to all clients on a shared channel with only the appropriate client pulling the message from it [5].

An alternative solution is to use the publish-subscribe pattern. Instead of the client pushing messages onto a queue, or messaging remote services directly, the remote services decide what messages they are interested in by subscribing to events published by an observable. This simplifies the dependencies between services by giving more control over to the subscribers so that the publisher does not need to coordinate the entire dataflow of the system. Published messages take the form of events, which represent that something has occurred, and the subscribers are free to ignore or react to events based on their own internal logic. Using this paradigm, it is easier to break apart the domain’s data-model by defining boundaries between sub-domains where each service owns a single sub-domain and subscribes to events that directly impact its sub-domain’s data model.

## 2.3 Domain-Driven Design

The term Domain-driven design (DDD) was coined by Eric Evens in 2003 with the publication of his book “Domain-Driven Design: Tackling Complexity in the Heart of Software” [6]. Since then, the term gained huge attention and shaped the way modern architectures are designed. Eric Evans, along with Martin Fowler who helped pioneer the movement, offer guidance on the best practices for implementing DDD by highlighting several tactical and strategic patterns.

Tactical patterns are those that help identify system requirements, through knowledge crunching exercises, to create effective models for a complex domain. Strategic patterns are those that help shape the architecture to support the principles of DDD. Models that have been identified using tactical patterns represent a section of the domain and are separated from one another using what Evans refers to as a bounded context. A bounded context is a strategic pattern that helps control the relationships of models by making them explicit [7].

Usually, a subdomain model takes the form of an aggregate where a single component is said to be the aggregate root. The root defines the relationships between other aggregates so that any other object in the aggregate cannot be referenced directly. This improves the encapsulation between different subdomains and reduces the complexity that can emerge overtime through the relationships between objects [8]. Also, by communicating only with the aggregate root, you ensure the consistency of the domain. If you were to delete a given entity within the domain without deleting or updating other entities that had relationships with that data, then the domain might be left in an inconsistent state. Evans provides an example where if you were to delete a customer record and the address of that customer, but other customers share that same deleted address then you will have objects referencing a deleted record [6]. Thus, weakening the integrity and consistency of the model.

The core philosophy of DDD stems from the need to have separate models to reflect different vocabularies used between different departments of a large organisation. The design should reflect the relationships between processes of the business so that developers and stakeholders can collaborate effectively. By using shared terminology in the form of a ubiquitous language, requirements are easier to capture between the domain experts and the development teams. Some domains may share terms but by using a bounded context, each with their own isolated models, terminology can remain consistent and avoid confusion [7]. For example, if your business model is to sell tickets for events (e.g. a music concert) but another department within the same business deals with IT tickets used internally within a customer relationship management (CRM) system, then both terms must exist within their own isolated models within the system without clashing. It would be impractical to ask the domain experts to change their terminology to benefit the software’s architectural design. By using a shared ubiquitous language during the initial design phases, the barrier between technical terminology and domain terminology is lifted, thus improving collaboration.

## 2.3.1 Microservice Architecture using Domain-Driven Design

DDD fits very well with microservice architecture (MSA) because both philosophies share the goals of enforcing isolation. However, MSA takes it one step further by isolating subdomains using the database per service pattern. Instead of relying on aggregate roots for handling the relationships between subdomains, all communication must be performed using IPC mechanisms, such as a REST API. Thus, no object references can exist between entities belonging between separate subdomains and all subdomain data is persisted inside separate datastores that can only be accessed by the microservice that owns that store. This means that the microservice is in full control on how to modify and maintain that data.

RPI-styled middleware, as previously discussed, does not suit these goals of MSA and DDD because it attempts to make individual objects of an aggregate accessible over the network. Therefore, MSA tends to rely on a layered architecture where the outer layer uses interfaces or adapters to handle requests or incoming messages, which can then be propagated to internal service-layer business logic. The service-layer is usually the only layer allowed to interact with the encapsulated domain model as this promotes many benefits, such as allowing the domain model to change when new requirements are introduced without affecting the infrastructure of the microservice. For example, the microservice API does not need to change to reflect the changes of the domain model, which means that the clients using that API can carry on operating as normal. Some changes might directly impact the client, but version control can be used to maintain the old and the new version of an API, allowing the client to upgrade when they are ready. This requires careful design considerations to ensure that the system is backwards compatible where both versions can function simultaneously.

There are multiple ways to implement version control for an API. One such method is to prefix the request URL with the version number of the API to be used, or by using a query string parameter. For more granular control, JSON/XML data contained in a POST request could represent a change to be committed to the datastore with an extra field representing the version of the API to use. This could signify to the system that the object should be handled in a different way, or maybe the data has been crafted to use a new feature offered by the new API version. This allows different parts of the data contained within the same request to use different API versions if required.

Whatever strategy the architecture uses to separate the request data from the domain, the principle behind the idea is the same. The result is an adapter or façade, acting as an anti-corruption layer (ACL) to preserve the consistency of the encapsulated model. This is another strategic pattern described by Evans and can be used when gradually migrating a monolithic architecture into an MSA [9]. While breaking down a monolithic applications domain into subdomains, the ACL can help with the transition by ensuring that legacy systems are still accessible by the newer features introduced. Legacy systems tend to use obsolete IPC mechanisms or data schemas and so an ACL can be used to ensure that the microservice itself does not need to directly cater towards the legacy system’s requirements. Therefore, when the monolithic architecture has been fully replaced and the ACL is no longer required, no further work is needed to remove obsolete code from the microservice codebase.

Chapter 3

# Data Consistency in Distributed Transactions

A monolithic application using a centralized relational database management system (RDBMS) has a straightforward approach to preserving the consistency of the database. Typically, there are three types of errors that can occur when using a centralized RDBMS within a single address space; the transaction may experience deadlocks caused by concurrent transactions, the system restarts due to system failure resulting in the loss of in-memory data, or database failure. In a distributed system, you also must account for network communication failure such as a remote service being unavailable, loss of messages during transportation, or messages arriving out of order causing the transaction to abort.

In this chapter, we focus on online transaction processing (OLTP) databases for the purposes of building high-throughput, transaction-oriented applications. Online analytics processing (OLAP) database systems typically do not require the most up to date version of the data. They tend to be used for constructing complex queries for the purpose of analysing large historic data and do not require strong transaction consistency. We will also consider the differences between relational and NoSQL database designs and how they can be used in a distributed microservice architecture for implementing concurrent transactions.

## 3.1 A Brief History of Transactions

A transaction translates high-level queries (such as SQL queries) into a set of primitive read/write operations once an optimised execution plan has been chosen, with commands to signal the beginning of the transaction and its termination [10]. During termination, the transaction can be aborted, and rolled-back to undo the changes, or committed to persist the changes. During its execution, the database can be in a temporarily inconsistent state but must be consistent before and after execution. A database is said to be consistent if it obeys all the integrity constraints defined by its schema.

In 1975, the transaction model was first introduced by the IBM System R research project [11]. System R was an experimental database system that consisted of a locking subsystem to ensure that conflicting data value writes caused by concurrent access could be detected and resolved. Later, the acronym ACID was coined by Theo Härden and Andreas Reuter in 1983 [12]. ACID principles define a set of database properties that attempt to provide safety guarantees when using transactions [13]:

1. **Atomicity** – all operations of a single transaction must be fully committed or aborted. This helps to preserve the consistency of the database.
2. **Consistency** – The database must be in a consistent state before and after the transaction has executed.
3. **Isolation** – The state of any given transaction is unknown to any other transaction. Transactions should be independent and able to run concurrently with any other transaction (using currency control mechanisms). Therefore, a transaction should only see one version of the data they are accessing.
4. **Durability** – Changes made to the database are guaranteed to be persisted even in the face of system failure. RDBMSs typically enforce this by logging changes in a log file or log table.

The implementation of transactions can vary between database vendors with each implementation consisting of a set of protocols and algorithms to enforce the ACID principles. A local transaction manager (TM) is given the responsibility of coordinating transactions by communicating with a scheduler. The scheduler uses a concurrency control (CC) method to ensure the correctness of data synchronization between concurrent transactions. Many research studies have been conducted on the use of such methods. They tend to focus on either a lock-based or timestamp-based CC to preserve the isolation of transactions. Both categories can either be implemented using a pessimistic or optimistic model. An optimistic model does not block transaction commits and allows the violation of isolation levels but resolves any conflicting transactions after execution. Pessimistic implementations enforce stronger consistency at the cost of performance by blocking transactions until data is freely available to use.

Isolation helps to avoid strange anomalies such as race conditions, dirty reads/writes, lost updates, and phantom reads. Race conditions are the most common pitfalls caused by two or more concurrent transactions attempting to simultaneously modify to the same data value, or when one transaction tries to read the same data value that another is in the process of modifying. The level of isolation between concurrent transactions can vary between database systems. Some systems implement weaker isolation as a trade-off for improved performance. The original transaction model only defines a standard for serializable isolation, which is considered the strongest level of isolation and is the most researched form of correctness criterion for concurrent transaction execution [14]. Serializable isolation describes the premise that two or more concurrent transactions should behave in the same way as if they were executed serially (as opposed to simultaneously) [15]. Unfortunately, it has some significant performance penalties when horizontally scaling out within distributed systems.

## 3.1.1 Lock-based Concurrency Control

To help preserve isolation, database systems employ the use of concurrency control (CC). Lock-based CC makes use of a lock manager to lock the data required by a transaction. The two-phase locking (2PL) protocol is a lock-based CC method which was the standard for implementing strong serializable isolation for many years. During the first phase, a transaction attempts to obtain a lock on the data it needs to complete the transaction. This prevents no other transactions from accessing it (this includes both reads and writes). Transactions remain blocked until the transactions they depend on release their locks after termination, forcing them to run serially. An obvious downside to this is that if a transaction is particularly large and requires many locks, or is long running, the number of blocked transactions can grow exponentially, which incidentally slows down the entire system.

## 3.1.2 Timestamp-based Concurrency Control

An alternative to lock-based CC is to use timestamp-based CC protocols to preserves the serialization order of concurrent transactions. Each transaction is serialized with a timestamp to construct a dependency graph where newer transactions have dependency on older transactions if data access is to be shared [16]. However, maintaining accurate timestamps can be a challenge in a distributed system. If using the system’s clock time, different sites must have their clock times synchronised using a protocol such as the Network Time Protocol (NTP). Another method is to use simple monotonically increasing time-stamp counter [17], but if one site is less active then another, the differences between one site’s local counter compared to a more active site could be exceptionally large. This means that a transaction that originated from a less active site would be interpreted by another site as an old transaction which could cause problems. Therefore, counters must be synchronised as well.

## 3.1.3 Single-Threaded Transaction Processing

Some database systems avoid the need for CC altogether by forcing transactions to run on a single thread (categorized as single-threaded databases). Because modern computers have a higher capacity of internal memory, serialized transactions can remain in-memory sufficiently. Also, single-threaded transaction processing does not have the lock management overhead that is present in concurrent transaction systems [18]. However, transaction throughput is going to be limited to the use of a single CPU core which means that transactions need to be quick to avoid blocking other transactions for too long. Fortunately, modern CPUs are increasing in processing speeds and single-threaded databases have seen adoption from popular database vendors; One example being PostgreSQL’s query engine although it also uses thread scheduling to avoid blocks caused by accessing data across multiple partitions to continue processing if needed [19]. While they do not work for all types of database requirements, there is certainly some use-cases where they can be an efficient solution.

Parallelism can instead be achieved by partitioning data in a distributed setting where a single thread is in control of a separate partition. Transactions that span multiple partitions should be avoided, otherwise multiple partitions must manage partition-level locks and would suffer in performance costs [18]. If the transactions are short-lived, and transactions have been carefully designed to rarely need a combination of data from separate partitions, this can provide promising results. Stored procedures are a popular method for reducing the lifespan of transactions because the database does not need to communicate with the application to receive the next operation before continuing. Instead, the set of instructions for executing a transaction is contained with the database system rather than the application.

## 3.1.4 Snapshot Isolation

Snapshot isolation was introduced by Hal Berenson and Philip Berenson et al. in 1995 [20]. It is a weaker mechanism for guaranteeing isolation and follows an optimistic model. It is based off the works of multi-versioning concurrency control (MVCC) which is a method of storing different versions of the same data item for each currently execution transaction. MVCC was proposed back in 1981 by Phil Bernstein and Nathan Goodman to address the key differences between read-write and write-write synchronization [21]. The idea is that read operations should not block other operations and write operations should not block read operations. MVCC is used in popular modern database systems such as used in SQL Server, PostgreSQL, Oracle, IBM DB2, and more [22].

Instead of one transaction modifying a data value directly, another version of that data item is created with the new value assigned to it. This means that concurrent transactions can only use consistent values of data without the interference of other transactions, which preserves isolation as each transaction is unaware of the behaviour of other transactions. Conflict resolution must instead be delayed until the end when transactions attempt to commit their changes. Database systems implementing MVCC can also support time-travel based queries to see the changes of data overtime by comparing versions of data.

The goal of snapshot isolation is not to enforce serialisability, but instead to enforce repeatable reads using a consistent snapshot of all data required by a transaction. Instead of creating multiple versions of data items, multiple versions of snapshots are created instead where a snapshot represents the portion of the database that a transaction relies on. This snapshot is taken directly before the transaction executes and can only be seen by that transaction. Read-only transactions avoid the synchronization overhead caused from serialization where even read-only transactions are blocked. During the commit phase, if the version of one snapshot is newer than another which has already committed and updated a shared value, then the transaction using that newer snapshot is aborted and possibly retried. This is referred to as the “First-Committer-Wins” rule [23]. This prevents lost updates from occurring and can perform much better than pessimistic algorithms if the transactions infrequently make use of shared data.

## 3.1.5 Serializable Snapshot Isolation

A relatively new approach to snapshot isolation, referred to as serializable snapshot isolation (SSI), ensures that every execution is serializable whilst still maintaining the benefits of SI. First introduced by Michael J. Cahill et al. in 2009 [23], this method attempts to solve data consistency anomalies caused by traditional snapshot isolation (SI) methods. Previously, explicit application-level locks were required to avoid some of the pitfalls of SI. Write-skew is a known anomaly of SI that occurs when one transaction Ti reads a value before a second transaction Tk updates that same value. Then, Ti uses the original value to decide on how to update another associated value. Unlike the lost-updates anomaly, this is much harder to detect because the isolation mechanism does not consider associated data. Because of this change, the original premise on how Ti reached that decision no longer holds true because Ti was unaware of the changes made by Tk.Because of this, SI is at risk of breaking the integrity constraints of the DBMS.

SSI supports the rules of SI by supporting non-blocking read and write operations whilst also preventing write-skew caused by concurrent transactions. In contrast to other lock-based methods such as 2PL, the performance penalty is relatively small. A major achievement of SSI is that it avoids the need for application developers to be educated on the shortcomings of SI and to compensate for them using explicit locking in application code. SSI detects conflicts caused by non-serializable concurrent transactions at runtime. If a transaction is non-serializable, it means that it cannot safely be serialized without moving the database into an inconsistent state. If SSI detects a pair of non-serializable transactions that conflict with one another, it will only abort the newer transaction rather than other optimistic CC methods that usually aborts both. This is achieved all while maintaining a reasonably small overhead cost for storing metadata required for detecting conflicts [23].

## 3.2 Distributed Concurrency Control

Within a distributed environment, a distributed computing system coordinates with other partitions of the database contained in individual “interconnected autonomous processing elements” [22], referred to as loosely coupled “sites” of a distributed database system [24]. These sites can be geographically distributed across a computer network and vary in their characteristics. They may also be replicated or fragmented using sharding to improve the availability and scalability of the data.

A transaction manager (TM) is selected as the coordinating TM and is put in charge of coordinating the actions of other participating TMs running on other sites. For example, the two-phase commit protocol (2PC) is a highly popular approach at coordinating transactions using a voting strategy between TMs. The simplest implementation of 2PC is known as centralized 2PC whereby only the coordinating TM messages the participating TMs. Then, the coordinating TM waits for the participants to approve the commit before sending out a global commit message during the second phase of the 2PC protocol. This finalises the decision and persists the changes at each site. Alternatively, if a single participant votes to abort the transaction, the coordinating TM receives this decision and sends out a global abort message to all participants. There is an exception to this rule where a participating TM can unilaterally abort the global transaction if a deadlock situation occurs [22].

Another approach is linear 2PC where participating TMs are ordered in a linear fashion with the coordinating TM at the front of the queue. Messages are then forwarded up and down the queue to collect the votes of all participants. This reduces the number of messages but at the cost of not supporting parallel processing. On the other hand, distributed 2PC gives the participants more control over the decision to commit or abort a transaction whilst also supporting parallel processing. This reduces the amount of network communication by making the second phase of 2PC unnecessary as participants can make their own decisions prematurely [25].

Managing locks in a distributed fashion has a similar set of alternative approaches. Two-phase locking (2PL) can be achieved using a centralized or distributed locking protocol. Centralized 2PL uses a single lock manager on a single site to grant locks on data as opposed to distributed 2PL where each site has a lock manager and does not need to wait for a centralised lock manager to grant them permission. However, distributed lock-based CC may cause deadlocks to occur and must rely on a separate distributed deadlock management process [15]. Typically, a wait-for graph (WFG) is produced and maintained either in a distributed fashion where each site has their own local WFG and has the responsibility or detecting deadlocks, or a centralized site maintains a global WFG and takes the responsibility of detecting deadlocks on behalf of all other sites. If using a centralized approach, each individual participating lock managers must send their local WFGs to the coordinating lock manager to be synchronized to form the global WFG [26]. The WFG represents a directed graph of dependencies between transactions. If a cycle occurs, then this means a deadlock has occurred and the appropriate measures can take place to abort the conflicting transaction/s.

Snapshot isolation (SI) largely remains unchanged except for the need to synchronize the monotonically increasing counter or timestamp (using NTP) as previously mentioned. Some studies have focused on improving SI when used in a distributed setting. Carsten Binnig et al. proposed an incremental approach to SI in 2014 [27]. They noticed that SI relies on upfront generation of snapshots which can increase overhead costs, but all known attempts to delaying this had occurred in high abort rates. Their solution was to generate the snapshots incrementally. First, the most recent local snapshot from the first originating site is used as the initial snapshot. Then, each time the global transaction is sent to other sites for further processing, the snapshot is extended using the local snapshots of the visited sites. From their research, incrementally extending the snapshot in this manner performed just as well for sharded or partitioned databases as for centralized, local databases.

Chapter 4

# Implementing Distributed Transactions in Reactive Systems

Like previous methods of partitioning (e.g. sharding) databases across a distributed system as isolated sites, the database per service pattern extends this idea further. Each service has its own database with its own subdomain model and does not use any form of synchronization or concurrency control with other databases. Instead, each database remains completely isolated from other databases to achieve loose coupling between services. A large benefit of this pattern is that services are not constrained on the type of database software. One service might have a different set of hardware constraints or an entirely different data model that is better suited towards a different type of datastore such as a key-value or graph-based datastore. It also enforces domain-driven design (DDD) by keeping separate subdomain models entirely isolated from others, including how they are persisted, modified, and queried. Then, separate development teams are free to change any part of their subdomain or database without affecting any other team or service. A single datastore owns data relevant to a subdomain (within a bounded context) and can still be partitioned using the mechanisms and protocols of the chosen datastore technology. However, within a microservice architecture (MSA), the problems of distributed transactions previously discussed in chapter 2 extend further.

## 4.1 Problems of Distributed Transactions in Microservice Architecture

It is common for an MSA to require distributed transactions to span across multiple microservices and reliably modify each of the participating microservice’s datastores in a consistent manner. Global transactions must still be rolled back if one microservice fails to send confirmation that it has successfully committed the requested change, else the domain may enter an inconsistent state.

To implement distributed transactions across an MSA, we must cater towards the polyglot nature of MSA database technology. For example, NoSQL databases cannot collaborate using the two-phase commit (2PC) protocol and so they may be incompatible with other relational databases such as SQL Server. The X/Open Distributed Transaction Processing Model (X/Open XA) uses 2PC as a way of achieving atomicity across distributed transactions and works across heterogeneous technologies [28]. The problem is that not all database technologies support XA, which limits our flexibility to choose the most appropriate database technology for our microservices. It also means that the limitations of 2PC apply to all databases in the architecture.

We also want microservices to remain isolated from each other so that they can be scaled independently and introducing traditional distributed microservices may cause a bottleneck. For example, if one microservice becomes unresponsive (i.e. the transaction manager goes offline) and does not free the locks its distributed transaction has on other microservice datastores, then this impacts the performance of the entire system. In addition, the performance costs of coordinating locks does not scale well when more external systems are required for each transaction [29]. Therefore, having such a direct, shared dependency between each store defeats the goals of MSA.

According to the CAP theorem published by Eric Brewer in 1999 [30], within a distributed system you can only deliver two of the three properties: consistency, availability, and partition tolerance. Most modern systems prefer to achieve availability and partition tolerance to support scalability as a trade of to weak consistency. Instead, the term “eventual consistency” is used to refer to systems that may return out of date (i.e. stale) data from an out of date partition but is able to maintain high availability and tolerate the loss of a partition without affecting its performance. When data is modified in one partition, the system takes time to synchronize those changes across multiple partitions. Distributed transactions can support strong consistency at the expensive of weaker availability caused by blocking transactions while locks are held on data. This is useful for some systems, such as online banking applications, but usually the user would prefer a reasonable level of stale data as opposed to long response times.

## 4.2 Reactive Systems

By their nature, distributed transactions are a form of synchronous IPC, which has an impact of availability. For MSA implementing eventual consistency, asynchronous IPC can provide a lot of benefits. The reactive manifesto defines a standard for what a reactive system should be [31]:

1. **Responsive** – Systems must respond to user requests within an acceptable timely manner. This principle focuses on providing good user experiences to build user confidence when using the system. By defining reliable methods for notifying the user after a set time limit, and continue processing, if necessary, without the user waiting, the system is said to have an improved quality of service.
2. **Resilient** – The system should support strong fault-tolerance and recovery from system failure in a timely manner. If a system is not resilient, it will also be unresponsive after failure has occurred. Resilience can be achieved through replication and isolation of services and data. By delegating tasks using asynchronous IPC mechanisms, services can be safely restarted and tried later to improve resilience whilst also remaining responsive.
3. **Elastic** – The system should be able to dynamically scale-out based on varying traffic loads and resource requirements to preserve responsiveness. Isolation and delegation can support the elasticity of a system by removing all central points of failure.
4. **Message Driven** – A core principle for reactive systems is to favour asynchronous message-passing IPC mechanisms with fault-tolerance in mind, over synchronous IPC. Message-driven architectures enforce non-blocking communication, which reduces the resource overhead caused by keeping connections open while waiting for results. Instead, subscribers or callback functions are used to process results only when required. It also favours location transparency as a method of supporting loose coupling through isolation and modularity.

MSA using message-driven IPC mechanisms are a natural fit for implementing DDD. By only relying on messages, such as serialized objects, and not sharing object references between microservices, the subdomain models of each microservice can be protected from corruption. When implementing message-driven architecture, developers must approach it with a vastly different mindset from traditional CRUD-like (create, read, update, delete) API designs typically implemented using RESTful principles. Rather than using a single blocking HTTP request to query the system for data, or to perform an action, and returning the result straight away, a message-driven system will continue processing the original request by sending messages to other microservices asynchronously. Each microservice must complete a relevant portion of the original request where each response forms the aggregated result to be sent back to the original client recipient. During this time, the client will either need to poll the system to check if the result is ready for them to read, or other bi-directional forms of communication are needed for the server to send the request to the client directly. Web-sockets are the most common form of bi-directional communication with many implementations supporting fallback methods in case the client’s software (e.g. a web browser) does not support them.

## 4.3 The Saga Pattern

There are various methods for aggregating the results of each microservice before sending the aggregated result to the client. One method is to incrementally build it by passing the previous microservice’s result inside the message sent the next microservice, but this model cannot be parallelized, resulting in higher communication overhead and response times for the client recipient. Another method is to use a centralized aggregator component to send out all requests and wait until all results have been retrieved before aggregating the results and sending it to the recipient. This better supports the separation of concern principle by avoiding the need for each microservice to know about the other microservice participants. Also, microservices only need to respond to simple requests and do not require pre-defined logic on how specific requests must be executed. These benefits come at the expense of a single point of failure caused by the aggregator.

Queries are not the only concern of message-driven architecture. Performing distributed transactions through messaging means that we lose the benefits of concurrency control and commit protocols such as 2PC. Instead, it is up to the application to implement concurrency control mechanisms, such as timestamp ordering or locking resources, and rolling back transactions in the face of failure. Message-driven architecture, unlike previous RPI strategies, make the pitfalls of distributed systems explicit. While it can be a complicated learning curve for developers, it does prevent the misconceptions that the vision of a unified object model had previously led to. It also allows developers to follow the reactive principles defined by the reactive manifesto by having greater control over communication and fault-tolerance to build responsive, resilient systems.

The saga pattern provides a method of simulating distributed transactions within an MSA. It caters towards isolated databases that are not coupled based on previously mentioned protocols and algorithms. Instead, the saga pattern allows us to coordinate our own transactions within a message-driven system. A transaction request is sent to a microservice in the form of a command message. The command contains the necessary data to perform a sub-transaction on an individual microservice with optional meta-data such as a given timestamp or a global transaction ID.

## 4.3.1 Compensating Transactions

If a microservice fails to return a message signally that the sub-transaction was executed successfully on its datastore within a given time limit, then the saga pattern suggests using compensating transactions to undo the global transaction. Because each microservice executes transactions on their isolated datastores, we cannot simply abort and rollback those transactions because they have already been committed. Instead, rolling back a saga is achieved by executing a series of compensating transactions to reverse the changes made to each participating datastore. Compensating transactions typically make us of commands that implement the opposite behaviour of the common executed as part of the failed saga.

A command reflects a real business unit of work such as an application feature or a unit of work that is required to implement that feature. Rather than creating a command to undo a local transaction, the command should implement a real business use-case and should be used by compensating transactions to update state. Therefore, the state of the system does not rollback to the exact same state it was in before performing the saga, but instead it moves forward to a new consistent state to undo the required business-level transaction logic. For example, when a command reduces the stock quantity of an item a customer has bought but their payment method is declined, the compensating transaction could execute a command to add an item to the stock, which may be used in other areas of the system. This supports code-reuse by avoiding the need to create a separate process to handle saga failure, allows audit logging to track the history of commands, and reuses existing business-logic to maintain data consistency. Event stores work well in this scenario but are not required for implementing the saga pattern.

Compensating transactions using the saga pattern are executed in reverse order to maintain data consistency. If a saga executes three out of five transactions but the fourth fails, then the compensating transactions are executed in reverse order where the first one undoes the last successful transaction to have executed as part of the saga. This means that if a concurrent request is made during the rollback of a saga, they see a consistent state where integrity constraints are preserved. Otherwise, you risk other transactions from coming to conclusions based upon a misleading premise, like the write-skew anomaly caused by snapshot isolation we saw in chapter 3. It is worth noting that some transactions do not require a compensating transaction, such as read-only transactions. Other transactions that are not critical for the performance and consistency of the system can be allowed to fail without aborting the whole global transaction if the application logic takes this into consideration without creating a confusing user experience.

## 4.3.2 Event Orchestration and Choreography

The two most popular choices for implementing the saga pattern is to either use orchestration or choreography where the results of commands are emitted as events to say what had occurred. This allows the system to listen out for events of interest and react to those events in an appropriate way to progress to the next step of the saga. The saga pattern does not enforce what type of IPC mechanism should be used. Events can be directly sent to participants, but a reference is required, which creates a dependency between the services. Other location transparent approaches can be used to reduce coupling and allow additional services to participate in the saga without changing any core logic. For example, events could be placed on a queue or emitted as part of a publish-subscribe model.

When using orchestration, a single orchestrator component receives all events and executes the next transaction, or set of parallelizable transactions, in the sequence and returns the result to the client recipient. The orchestrator contains all network communication logic, such as timeouts and retry logic possibly implemented using the circuit breaker pattern. If the orchestrator goes offline during execution, the orchestrator must be able to recover and continue from where it left off while preserving consistency. Therefore, microservices must implement idempotent commands so that triggering the same command more than once with a given transaction ID or timestamp does not corrupt the domain’s state. Ideally, the orchestrator should be able to horizontally scale to avoid becoming a bottleneck.

Implementing the saga pattern with event choreography increases the complexity of the dataflow. Each microservice oversees the execution of the saga whilst also handling compensating transactions. There is no central orchestrator and so each microservice must implement their own fault-tolerant mechanisms previously described. Unlike an orchestrator, it can be difficult to see an overview of how a saga is carried out or what participants are involved without looking through the entire codebase. If separate development teams are working on separate microservices, more team collaboration is required. However, the message overhead is reduced because microservices do not need to send an event back to an orchestrator and can instead be picked up directly by the next participating microservice/s. When using the publish-subscribe pattern, each microservice can be observed by other microservices. An event emitted/published by one microservice can be received by all subscriber microservices, which triggers each subscriber to execute local transactions to progress the saga. If a message is to be sent back to the client, a separate saga-terminated event can be emitted to mark the end of the global transaction so that a separate process can subscribe to this event and notify the client.

One study conducted in 2018 found that event choreography performed much faster in comparison to orchestration [32]. However, as more events increased, the complexity when using choreography became difficult to reason with. Thus, the study suggested that choreography is more suitable when used with fewer events where fast response time is critical. However, orchestration makes it easier to avoid cyclic dependencies between microservices. By keeping microservices simple and using the orchestrator as the only subscriber, it reduces the number of dependencies and avoids microservices accidentally depending on each other for shared domain data to complete their local transactions.

If using DDD, microservices should only need to perform basic CRUD operations and more complex business logic relating to its own subdomain. Sometimes, it is necessary for hold foreign keys relating to another subdomain, which can be acceptable if following the aggregate root pattern, but this should be minimized. If one microservice requires data from another subdomain, causing a dependency, then this is easier to manage with orchestration while avoiding cyclic dependencies. The orchestrator can first request the necessary data from one microservice and then send it as part of a follow-up request to the next microservice in sequential order.

## 4.4 The Actor Model

In 1973, Carl Hewitt, Peter Bishop, and Richard Steiger first proposed a new architectural model based on small units of work, called actors, for simplifying concurrency control [33]. The behaviour of actors relies exclusively on receiving and sending messages to invoke the actor to perform a task. An actor model provides a high-level abstraction of concurrency control within a distributed system. Actors can hold private state and can stay alive for any arbitrary amount of time as required, but it can only perform a task for a given message serially. Multiple actors can be created to execute tasks in parallel but can only modify their own encapsulated private state, which avoids concurrent access and data consistency anomalies. It also avoids the need for explicit lock-based synchronization of resources, allowing for a much simpler development experience.

The main motivation for the actor model is to facilitate highly parallel computing by scaling out the number of actors in an actor system. To support this, many implementations of the actor model support location transparency, clustering, and remote messaging. Although the theory of the actor model has been around for a long time, it never saw wide-spread adoption due to the limitations of technology. However, in recent years it has seen growth in popularity due to the increase in performance provided by modern computer architecture with many frameworks proposed. From as early as 2020, several actor frameworks released new versions for modern languages such as Java, Python, C# and Rust, with new ones currently in development. They have been used in a variety of domains, most notably for games development, web applications, artificial intelligence, multi-agent systems, and for the internet of things (IoT).

## 4.4.1 Fault-Tolerance in Stateful Applications

Object-oriented programming (OOP) models provide features for encapsulation, but all code is executed within the same thread. When multiple threads are executed within the same system, it is common for different threads to operate on shared code, whether that code is a method or stateful data. This is a potential risk to shared data and creates an illusion of encapsulation. Most high-level programming languages provide locking mechanisms to avoid more than one thread having access to the same data, but this can reduce performance and comes with its own technical risks such as deadlocks caused by cyclic dependencies. The CPU cost involved with suspending a thread and restoring it later once a new lock is obtained can be substantial, especially for high performance computing (HPC) systems. In addition, blocking threads reduces response times and wastes resources [3].

A major disadvantage of locking is that is does not scale well; locking data locally can be an acceptable trade-off for data consistency, but when globally distributed locks are required across multiple services for handling distributed transactions, the latency overhead caused by coordinating communication can be extreme. Response times are unreasonable due to high latency, which may cause long-running locks to hold up additional requests and increase the risk of granular deadlocks from executing sub-transactions out of order. If one microservice executes its sub-transaction before another and a second global transaction is executing its own sub-transactions that depends on the first one being complete, it becomes a complex technical challenge to preserve the logical ordering of events and consequently the consistency and integrity of the system.

The support for fault-tolerance is one of many quality-of-life benefits addressed by the actor model, which many actor frameworks typically implement. The idea is to allow actors to create other actors, referred to as child actors, for delegating workloads, potentially in parallel, where the creator actor is said to be the parent who manages their lifetimes. If a child actor experiences a failure, the parent actor can act as a supervisor by deciding which recovery strategy to use on behalf of the child. For example, it could decide to restart the child or to retry the same message multiple times until it reaches some predefined maximum threshold, or possibly to log the error and continue onto the next message for situations where the task was not urgent.

Event sourcing is a pattern used to record events that have occurred in a system by appending them to an event store. This allows the state of an application to be preserved, which becomes especially useful in an actor model when using stateful actors. A stateful actor can represent a materialized view of event data as a form of memorization to improve response times. Instead of querying a database for persisted data, the actor can return values from its own state if needed, reducing the resource overhead caused by establishing connections with the database.

Stateful actors, when used with event sourcing, can recover their state when they experience failure and are forced to restart. If one actor was handling all requests for a specific user and had to maintain state relating to that user’s interactions, then all user data would be lost after an unexpected restart unless persisted. Some data is only short lived as a way of controlling the state of an application, but developers are often forced to store this short-lived data in persistent storage or as session data. A second common requirement is for an application to maintain an audit log representing the series of events that occurred during a user’s interactions. Event sourcing addresses the latter issue by recording all events in an event store, whereas an actor system addressed the former issue where a single actor can represent the short-lived state based on associated event data.

An actor’s state may influence how an actor reacts to incoming messages. Some actor frameworks approach this by representing actors as finite-state machines. If an actor is in one state it may refuse specific message types while allowing others. Other message types might trigger the actor to transition to a new state. For example, if the user is currently adding items to a shopping cart then the actor might be in a unique shopping state. Once the user goes to checkout, the actor may need to change to a new checking out state where no more items can be added to the actor’s private state data. If the actor restarts and recovers their state from the event store, then the user will lose their items in the shopping cart. Actor state should be kept private from other actors in the system and should be immutable to maintain data consistency when handling multiple messages accessing the same state.

By dynamically creating stateful actors to handle long-running user interactions, with their own private state, we avoid deadlocking scenarios by reducing the coupling of shared data. This provides the application with a high-level abstraction of concurrency control in a non-blocking fashion whist supporting parallel processing of incoming messages. By using a hierarchical supervision strategy with fault-tolerance mechanisms such as event sources, the system supports the resilient and elastic principles of the reactive manifesto. Responsiveness is also achieved, especially when using stateful actors to access state rather than directly querying a database, by allowing actors to divide large workloads between child actors to be executed in parallel. The actor model is naturally message-driven and can therefore be distributed across a network cluster for improved scalability. Some actors may be deployed to different actor systems on different nodes within a cluster either explicitly or dynamically depending on the volume of incoming requests. In conclusion, the actor model is a perfect fit for implementing a reactive system and can address complex data consistency concerns.

## 4.4.2 An Overview of Akka.Net

There are many implementations of the actor model for different programming languages and frameworks. Akka.Net is an implementation of the actor model for .NET developers. It consists of a set of open-source libraries to provide many concurrency control features for building distributed reactive systems, such as finite-state machines, hierarchical supervision, mailbox queues for processing messages in serial order, routing strategies to dynamically scale actors, and more. It is based off the original actor model theory proposed by Carl Hewitt but also offers a feature-rich ecosystem for addressing the challenges that modern computer architecture’s face when designing distributed systems [34].

Akka.Net was ported from the popular Akka framework to be used by .NET developers. The original Akka framework, released by Jonas Bonér in 2010 [35], was designed to run on a Java virtual machine (JVM) and supported both the Java and Scala programming languages. Akka attempted to increase the level of abstraction by alleviating the need for developers to manually manage locks and synchronization required by previous Scala concurrency control mechanisms, with the goal of providing a framework for building highly concurrent, event-driven applications. Akka.Net is based on Akka but has evolved over the years to improve the integration with newer .NET Core technologies. While there are many alternative actor model frameworks, this project makes use of Akka.Net with other .Net technologies to build a reactive system.

Chapter 5

# Design and Implementation

As part of this research project, a reactive, distributed MSA was developed using Akka.Net to handle concurrency control and scalability. Each microservice was implemented using the latest version of .NET, which at the time of writing is .NET 5 (renamed from .NET Core 3.0), with its own actor system implemented using Akka.Net. Microservices make use of message-driven IPC by relying on actor references to maintain location transparency. Instead of using object references, Akka.Net handles actor references to coordinate the destination of messages. Actors could then be placed behind a routing actor to distribute the workloads between dynamically created actor instances to handle fluctuations of network traffic.

The purpose of the application is to provide a platform for users to manage projects and discussion groups for the purpose of collaboration. A user can upload a project and invite other users to join as part of a team. The project owner can manage users by assigning them to different teams with various permissions. Discussion groups can be created by any logged in user which can optionally be linked to a project to represent a community for that specific project. Projects and dioscussion groupd

. A project can be where users can join to become members of that group. Then, members can submit issues/bugs relating to the project ho

“As a developer, I want to retrieve X properties for an individual project by project Id.” – Label this card by the context (e.g. the Project Context)

Chapter 6

# Evaluation