BUILDING SCALABLE BUSINESS DOMAIN TRUST

BUILDING SCALABLE BUSINESS DOMAIN TRUST

By ADRIAN BURLACU,

A Thesis Submitted to the School of Graduate Studies in the Partial Fulfillment of the Requirements for the Degree Masters of Applied Science(Software Engineering)

McMaster University MASTER OF APPLIED SCIENCE (2017) Hamilton, Ontario (Software Engineering)
TITLE: Building Scalable Business Domain Trust
AUTHOR: Adrian Burlacu (McMaster University)
SUPERVISOR: Professor Antoine Deza
NUMBER OF PAGES: vii, 36

Abstract

This thesis discusses a minimized number of concepts necessary for creating stateful, asynchronous, and scalable software applications that implement a subject domain. It is shown how domain driven design can be implemented using a minimized set of interfaces and architecture patterns. Further, it is shown how high-level business logic can be exposed as an HTTP service. This is achieved by reviewing requirements, design, and implementation details for shared control of an organizational data structure - a tree. Tree data specific as well as general business logic such as synchronization is identified and testing results are explored. Extensibility design is proposed.

Aknowledgements

Thank you Prof. Deza for your continued patience and support.

Content

1 Introduction

Today's Internet of Things and Software as a Service communities face common goals such as sharing a set of resources in an accessible way. Current software usually consists of being able to model some business domain rules into a system that performs those rules upon an existing information storage. Database systems are easy to take for granted, and thus business domain logic gets trivially modeled calls to such a database. The current paper discusses the minimal concepts necessary to create clean, testable logic, its pitfalls and benefits, and how this architecture can be used to create and extend such business domain logic. Further, it shows a converged implementation that exposes business logic as HTTP request invocations to a web server, wrapping and exposing the database as a representational state transfer(REST) service with the atomicity, consistency, isolation, durability(ACID) properties. Specifically, the example implements a basic, more abstract tree data business logic example. Further, testing and extensibility design is proposed.

(Riel, 1996) describes the techniques for designing object types and (Evans, 2004) describes how to design a domain service. These resources were important for transforming the given problem requirements into scalable service design.

This work's main goals were to review the necessary tools for designing a basic business domain. It builds an architecture that allows for the logic's scalable use, its accurate upgrading, and its readiness for extension. The architecture is detailed and measured against a baseline. The baseline is timing database transactions only.

1.1 Requirements

Requirements with perspectives or viewpoints capture the essence of the interactions between a system and its environment. (Wiegers & Beatty, 2013) states "The goal of requirements development is to accumulate a set of requirements that are *good enough* to allow your team to proceed with design and construction of the next portion of the product at an acceptable level of risk.".

1.2 Design

Describes the chosen interfaces and high-level concepts necessary for solving the problem outlined in the example.

1.3 Implementation

Specific low-level details of how to implement the design are discussed. The resulting implementation is measured and compared to a baseline.

1.4 Verification and Extensibility

One of the main requirements that inspire the design and implementation is accuracy and scalability. A comparison of testing methodologies is given. An extensions review is also presented based on the implemented example.

2 Example

An problem is given to illustrate the use of the technology. The technology is somewhat more flexible than just solving this problem, yet it showcases the functionality necessary.

It is required to know at any one time what country and state or province a user is in. Users or applications can report this information using a list of state identification numbers. Full location data needs to be normalized, State or province cannot be stored with the city in order to save memory space.

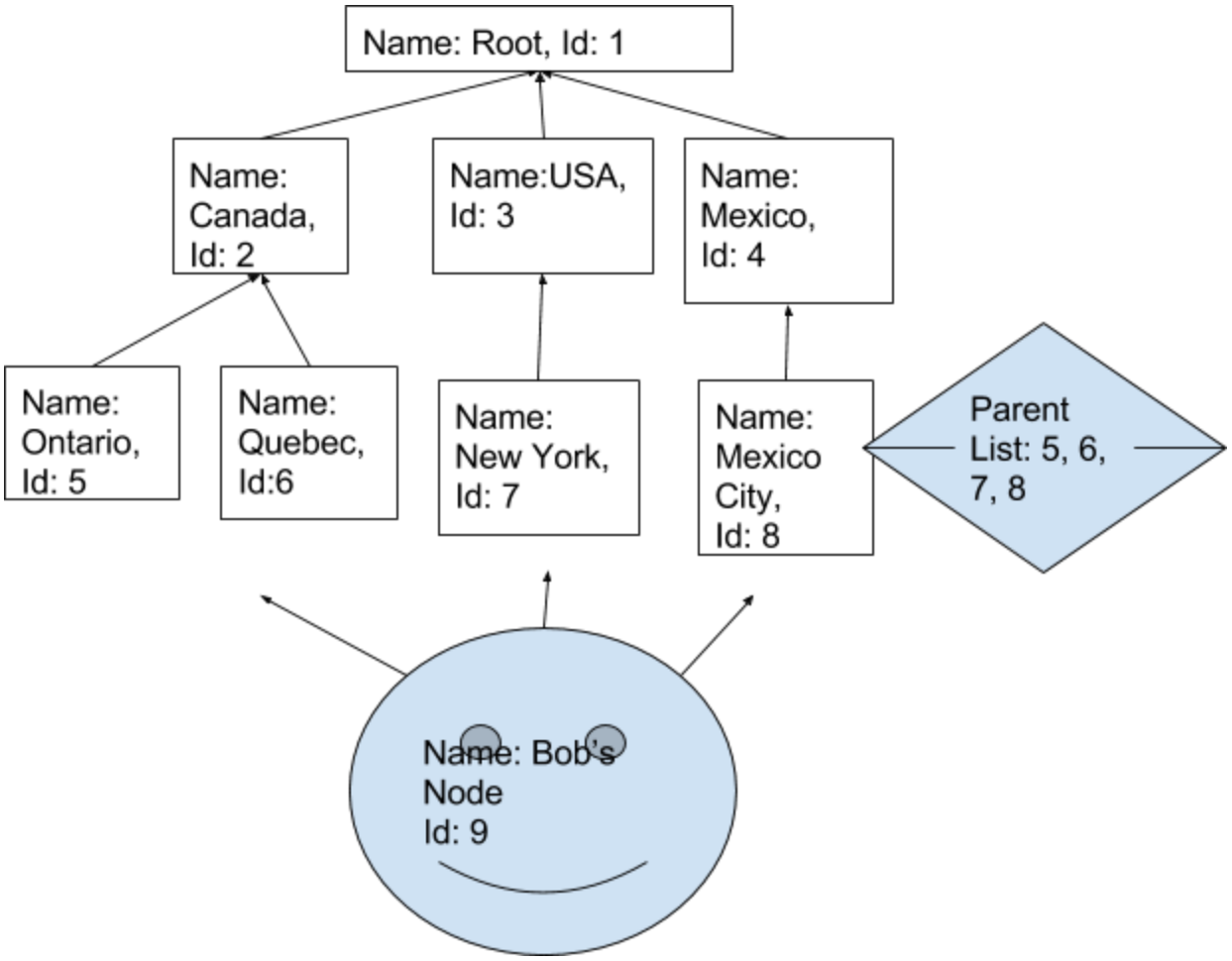


Image 1: a logical view of the system in action.

An HTTP POST web request can be sent by applications or the expert user to the service to update this information. (Berners-Lee, Fielding, & Frystyk, 1996) describe POST methods are useful for “[e]xtending a database through an append operation.” which covers most cases. ("Introducing JSON") describes JSON format. An example:

{"command":{"node":{"id":7,"name":"Bob’s Node","parent_id":5,"active":true},"method":"set","methodType":"checkout"}}

There may be multiple users or applications updating Bob's Node. Especially at political borders, multiple location updates can be occurring concurrently. As long as the transactions occur accurately, the latest location state posted is assumed as the correct location.

3 Requirements

Requirements will be given under standard headings. Stakeholders will be shown in brackets after a section. Engineering is concerned with learning requirements and providing systems to meet them.

2.1 Functional (Users)

What the software is supposed to do. The example describes a tree node storage. Methods are: Get, Add, Set, Delete.Do this accurately for concurrent updates.

2.2 Quantitative (Users)

The following two measures describe the external properties necessary to provide any service:

Reliability - number of errors / total requests
Response rate - time / request

Other quantitative measures are possible, like memory used are important, but because they are not externally-facing, they were not a priority.

2.3 Scalability (Users)

Ability to share access to modify the tree structure between many users. Scalability refers to this number of users in a given moment,at specific levels of sharing such as entity level or type level.

2.4 Experience (Users)

This is where regular user experience expertise goes. Describes different view on user experience. In summary, (Lallemand, Gronier, & Koenig, 2015, p. 41) shows that the definition given in (Hassenzahl & Tractinsky, 2006) is the most popular one:

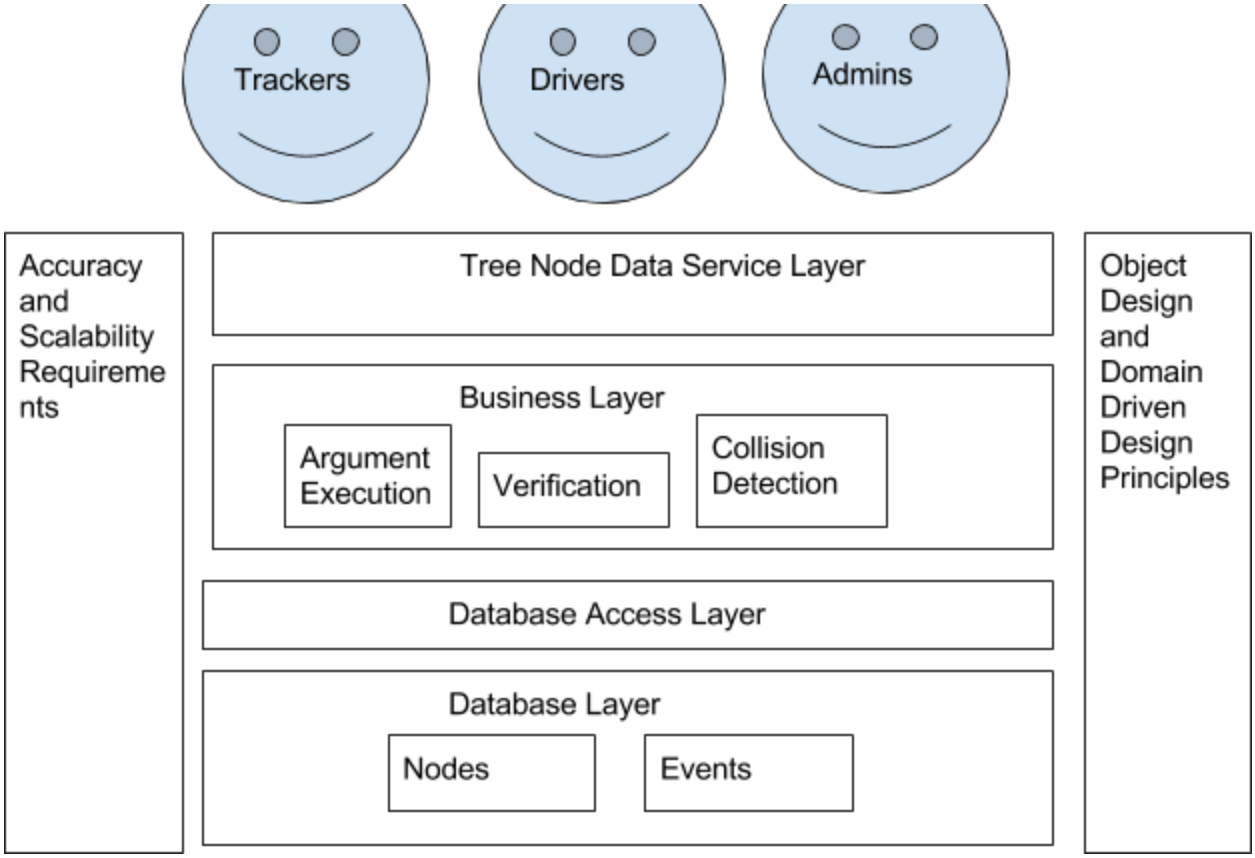
> “A consequence of a user’s internal state (predispositions, expectations, needs, motivation, mood, etc.) the characteristics of the designed system (e.g. complexity, purpose, usability, functionality, etc.) and the context (or the environment) within which the interaction occurs (e.g. organizational/social setting, meaningfulness of the activity, voluntariness of use, etc.)”

2.5 Maintenance (Owners)

How easy it is to keep the product up to date and add new versions. How much duplicated logic exists drives up maintenance.

3 Design

3.1 Overall Design



Picture 2: Architectural diagram of the application server.

3.2 Detailed Design

3.2.1 Interface

Node Data type provides the following information:

- Id
- active : whether the node is active

- Name: a name for the node

Get, Add, Set, Delete command:

- a search object that looks like a tree data node, but certain properties can be omitted and the name accepts wildcard characters for the Get method.

Command behaviours:

Node Data service supports the following commands interface:

- Get command: all properties allowed
- Add, set, delete - must have id
- Add, set - can have all non-id properties populated. Defaults are provided in add case. Set case provides existing consistent data.

The node data service comes with one pre-existing default root node.

HTTP POST is chosen as the request type the service will accept. JSON is used as the data format for the requests. These were chosen for the flexibility in size and composability respectively.

The user would receive their prepared transaction after performing a command:

{“command”:{
“method”: “set”,
“methodType”: “checkout”,
“node”: :{"id":7,"name":"Bob’s Node","parent_id":5,"active":true}
}, “checkout”: {
“get”: {
“nodes”: [{"id":7,"name":"Bob’s Original Node Name","parent_id":2,"active":true}],
“events”:[]
},
“result”: {
“nodes”: [],
“events”: [{"id":7, “active”: false, "node_id":7,"node_name":"Bob’s Node","node_parent_id":5,"node_active":true}]
}
}}

The user would change the methodType to “checkin” to tell the service to commit the transaction, and then perform the HTTP request. The result would be:

{“command”:{
“method”: “set”,
“methodType”: “check in”,
“node”: :{"id":7,"name":"Bob’s Node","parent_id":5,"active":true}
}, “checkout”: {
“get”: {
“nodes”: [{"id":7,"name":"Bob’s Original Node Name","parent_id":2,"active":true}],
“events”:[]
},
“result”: {
“nodes”: [],
“events”: [{"id":7, “active”: false, "node_id":7,"node_name":"Bob’s Node","node_parent_id":5,"node_active":true}]
}
}, “checkin”: {
“get”: {
“nodes”: [{"id":7,"name":"Bob’s Original Node Name","parent_id":2,"active":true}],
“events”:[{"id":7, “active”: false, "node_id":7,"node_name":"Bob’s Node","node_parent_id":5,"node_active":true}]
},
“result”: {
“nodes”: [{"id":7,"name":"Bob’s Node","parent_id":5,"active":true}],
“events”: [{"id":7, “active”: true, "node_id":7,"node_name":"Bob’s Node","node_parent_id":5,"node_active":true}]
}
}}

3.2.1 Tree Node Data Layer

This is a logical separator that allows for code reuse when serving a standard interface. This is repository architecture, achieved by overloading a back-end to the front-end implementation and making operations idempotent. Idempotency is necessary for independent client verification of server operations, and very important when performing updates to logic.

For example, if the service returns the following result:

```
{“command”:{
        “method”: “add”,
        “methodType”: “checkout”,
        “node”: {

        }
}, “checkout”: {
        “get”: {
                “nodes”: [{id = null}],
                “events”: [{id = 10}]
        },
        “result”: {
                “nodes”: [{id = 1}],
                “events”: [{id = 10}]
        }
}}
```

Then a client can run the service and logic layers on top of an HTTP request backed, achieving remote verification. The way to do this is to revert the last step of the logic postcondition that copies results to the checkout section. The server result would look like this after creating the idempotent argument:

```
{“command”:{
        “method”: “add”,
        “methodType”: “checkout”,
        “node”: {

        }
}, “read”: {
        “nodes”: [{id = null}],
        “events”: [{id = 10}]
},
“result”: {
        “nodes”: [{id = 1}],
        “events”: [{id = 10}]
}}
```

A debugging flag could be set to enable intermediary read values to be passed through the interface. Now, the read condition can also be run on the client and tested against client logic. The only parts that cannot verified by the user is what the domain returns to the read and save queries.

3.2.2 Repository Implementation Layer

Since current-day databases provide 2-step transactions as a means of achieving consistency, that model must be extended and exposed through the service. The first step in achieving this is recognizing that logic can wrap each back-end operation at the following stages: Pre - before the back-end transaction, read - at the stage where data is read, and post - after the write or change is executed. It also needs to have standard storage location ie.: read and result sections dedicated to read and written data after each of the last two stages.

This layer has its own special interface to the logic. Important examples:

- It runs its pre-read - logic - write-post operation in one transaction
- It takes the service input and save sections for arguments at the first two stages
- It updates read and result sections after the read and write operations.
- When performing get, set, or delete operations, process the argument and retrieve the nodes of interest.
- When adding a node, it must populate the event with the node_Id
- When saving events, update with transaction id

This layer produces deadlock errors if two events on the same entity are trying to activate at the same time as a result of concurrency.

3.2.4 Asynchronous Transaction Layer

Since the current web is asynchronous for many good reasons, commands are best kept asynchronous as well. This solves the issue of consistency checking by incorporating the asynchronicity of usage by domain users into the design. This repository architecture implies that transactions have two stages - checkout or prepare and check in or commit. Further, it implies some accounting for the order to the commands that dictate when deadlocks are triggered.

3.2.5 Event Sourcing Layer

In order to compute the order of transactions as they apply to entities in the domain, a separate event class is dealt with by the logic:
Node Data Event

- Id
- Active - whether the event is active. Events can only be activated once and they are not touched after that
- Method - command performed on this tree node
- Node_id: tree node id
- Node_active: tree node active
- Node_parent_id: tree node parent id
- Node_parent_name: tree node name

3.2.6 Logic Layer

At the top of the abstraction model sits the core logic of the application. It needs to incorporate all the architectural layers' concerns into code that only is called at the right time, it performs the correct computation, and satisfies the properties of the other layers in a scalable way.

The tree node example is below. You can assume the methodType, Method, and Stage checks:

3.2.6.1 Checkout: methodType == "checkout"

Applies to all methods stages:

- Pre: Stage == "pre"
- Read: Stage == "read"
    - If there were any nodes or events read, they all must have Id
    - All node events read have an id
    - Command entity is created if method is not "get", if command is forced, or there is a command id provided.
- Post: Stage == "post"
    - All written nodes must have id
    - All written events must have id, and active=false
    - Put the read and result section in the checkout section

3.2.6.1.1 Get: method == “get”

Stages:

- Pre: None
- Read: If there is a command provided in the input, create an event in the save location for every retrieved node in the read location.
- Post: If there is a command provided in the input, check there is an event in the result for every node in the result.

3.2.6.1.2 Add

Stages:

- Pre: The input node has no id and is not active.
- Read: put the input node in the save location, and also put a method=“add” node event in the save location
- Post: The input node is included in the result, and its method=add event.

3.2.6.1.3 Set

Stages:

- Pre: The input node has an id. The input parent id is provided . The input parent id is not equal to the node id. The node id cannot be equal to the root node id.
- Read: The read must return an active node and its events. Add a method=”set” event for that node. Add the node to the result.
- Post: An event that includes the input node.

3.2.6.1.4 Delete

Stages:

- Pre: The input node has an id. The node id cannot be equal to the root node id.
- Read: The read must return an active node and its events. Update the save location with the read node, then set the active property to false. Add a method=”delete” event for that node into the save location.
- Post: An event that includes the input node id and node.active=false

3.2.6.2 Checkin

Stages:

- Pre: checkout section is populated

- Read: Checkout result active events must be equal to checkin read section. Also check nodes. Otherwise, logic deadlock is detected. Copy all checkout section events to the save section and set them to be active.
- Post: All updated nodes and events in the result section have an id. All updated events are active. Read and result section is put in the checkin section.

3.2.6.2.1 Get

Stages:

- Pre: None
- Read:If a transaction is provided, copy read nodes to the save location.
- Post: None

3.2.6.2.2 Add

Stages:

- Pre: None
- Read: The first read node is made active. Its corresponding method=”Add” event is also activated. Both are put in the save section.
- Post: The first result node is active. The first node “add” event is also active.

3.2.6.2.3 Set

Stages:

- Pre: None
- Read: The first read node is put in the save section, then the input node information is updated. The method=”set” event is activated.
- Post: The result.node includes the input.node, corresponding event is active

3.2.6.2.4 Delete

Stages:

- Pre: None
- Read: The first read node is put in the save section, then made inactive. The method=”delete” event is activated.
- Post: the result.node is not active, corresponding event is active

3.3 Example Integration

The example problem is implemented by the design in a demonstrably transparent fashion. First of all, a node id list is provided for all the locations that the vehicle can be reported in. The design provided, it is not possible to prevent cycles in the tree if there are arbitrary node to parent moves. The example does not need this, provided an external process for such an error is followed. An example would be a positive result in a VLOOKUP function on the prospective parent location node in Microsoft Excel.

Second, the user is able to add and move the vehicle node around by running commands against the tree node data service. Third, current vehicle parent location node can be queried by supervisors. Thus, the example problem can use the given design to perform its basic requirements. The example is used to show how different concurrency scenarios engage different layers of the architecture to achieve the desired command collision detection and logic verification.

Last, current status of Bob's location can be retrieved using a set of get commands to the system, specifically requesting the node pointed to by parent_id until the root node is reached.

3.4 Similar Solutions

According to (Castro, Melnik, & Adya, p. 1070), "Interaction with the data [in ADO.NET] can take place using a SQL- based data manipulation language and iterator APIs, or through an object-based domain model in the spirit of object-to-relational mappers."

The highest-level framework for data are simple database socket to HTTP socket converters or object-to-relational mappers. Otherwise, abstract logic frameworks were found such as (Bonner,A. J., & Kife, 1995) that deal with the core abstract concepts. There were also examples of domain driven design frameworks in (H., 2017). In contrast, the framework presented here minimizes the concepts of the design in order to analyze tradeoffs to the architectural modules and provide alternatives and arguments for scalability and extensibility. It doesn't prescribe the how necessarily, it uses the concepts presented in current literature to build the basic measurements for a trusted, maintainable logic fabric based on information.

3.5 Requirements matrix

The framework for evaluating the requirements is set up and expectations are stated. First, it is important to say the experimental setup will compare the given design to a bare-bones SQL transaction only implementation. In the SQL transaction only implementation, locks are used to manage data only in the database, even though usually memory would also be locked for caching purposes -transparent or not- and verification logic would run after the read stage of the transaction. The state management architecture described also suffers from having to provide cache and database-specific business logic, while the architecture described here relies only on query caching. This difference is not measured here, and neither is the testability between this architecture and the non-transparent cache architecture alternative that is mentioned. In conclusion, the SQL transaction only implementation shows a baseline that approximates a best case of a full cache-managed domain-implementing architecture as well as the domain driven design shown here.

3.5.1 Accuracy

Good functional requirements are tested continuously, as the commands are run. The test code is inspected manually, by tracing through its execution. The end accuracy errors are aggregated for each test. There are no accuracy errors expected.

Twenty target nodes are used to reduce the probability of a false positive command to 5%. Ie.: that moves node B from parent node A to parent node A.

3.5.2 Quantitative

Reliability of the testing is measured by aggregating client-side errors. There are no expected reliability issues. Reliability is tested by running multiple concurrent modifications on the same nodes. Deadlocks are measured at this time.

Response time is recorded as total time taken for a test that serves a specific number of requests. Response time is expected to be between two and five times slower than just SQL transaction only.

3.5.3 Scalability

Scalability is tested by running concurrent updates over same or different nodes. The scalability of the system will come from a few factors:

First, there is an invariant that under contention, the work on one node will be linearized, and this should still be visible in the baseline. Second, scalability of the SQL-only baseline should be the scalability of this system.Third, as long as for each checkout, check in command there are only two SQL transactions, a finite event-chain length to analyze, and no other resources used between the checkout and check in phases, the system should scale constantly compared to a SQL-only baseline.

3.5.4 Experience

In order to simulate a passable user experience, the design must be improved with event cleanup logic that updates entities' event chains based on usage. In lieu of this, the current design is tested with limited event chains, by creating a new node when such an event limit is reached.

3.5.5 Maintenance

Object design and domain driven design principles are great for maintenance, isolating layer concerns to specific classes. As well, the event sourcing logic helps with updating logic. Duplication and mis-categorizing of logic is reported.

4 Implementation

Once the design is understood, the implementation becomes a task of serving up the application interface. Since javascript is the language of the web and very interoperable, all the code implementing this example is in Javascript. The hosting technology is docker.
The database is PostgreSQL, the basics of which are described by (Momjian, 2001). The testing is done locally through NodeJs, which is acting as the application server as well. (Cantelon, et. al., 2014) describes the basics of Node Js.

4.1 Results

4.1.1 Accuracy

The test code was verified manually. Every time a request runs, it is checked that the desired outcome is achieved. There were no errors during the test runs, thus no data races were detected where the result was different than expected at the beginning of the operation.

4.1.2 Reliability

The reliability of the test results was monitored, and no errors occurred.

4.1.3 Response Time and Ineffectiveness Under Contention

Distributes events for the same node across the number of threads.

Nodes	**Events**	**Total Events**	**Threads**	**Sleep Ms**	**Time**	**Ineffectiveness = Failed events / total events**
100	10	1000	1		49.5 sec	
100	10	1000	2	200	2.3 min	21 %
100	10	1000	5	~1000	5.4 min	43.0 %
100	100	10000	1		8 min	
100	100	10000	2	200	28.6 min	23.2 %
100	100	10000	5	~1000	38.4 min	40.0 %

The sleep time argument is determined by evaluating an upper limit of 50% for ratio of failed events to successful events. A +-500 millisecond random value is used to avoid starvation and other contention issues.

4.1.3.1 Response Time and Ineffectiveness Under Contention Baseline

Nodes	Events	Total Events	Threads	Sleep Ms	Time	Ineffectiveness = Failed events / total events
100	10	1000	1		5.1 sec	
100	10	1000	2	200	16.3 sec	0
100	10	1000	5	200	32.7 sec	0
100	100	10000	1		10.1 sec	
100	100	10000	2	200	25.8 sec	0
100	100	10000	5	200	38.9 sec	0

4.1.4 Sleep Time Versus Response Time and Ineffectiveness

Nodes	Events	Total Events	Threads	Sleep Ms	Sleep Variance Ms	Time	Ineffectiveness = Failed events / total events
100	10	1000	2	~25	12	2.1 min	41.2 %
100	10	1000	2	~50	25	2.4 min	37.0 %
100	10	1000	2	~100	50	2.0 min	21.8 %
100	10	1000	2	~200	50	2.0 min	25 %
100	10	1000	2	200	0	2.3 min	21 %
100	10	1000	5	~200	50	Inf	Inf
100	10	1000	5	~500	50	6.1 min	61.9 %
100	10	1000	5	~700	50	4.7 min	45.7 %
100	10	1000	5	~1000	500	5.4 min	43.0 %

The infinite value occurs as a result of a race condition that pushes the event chains to unbounded length.

4.1.5 Response Time, no Contention

Distributes nodes to threads and then set events are done in isolation.

Events	**Nodes**	**Total Sets**	**Threads**	**Time**
10	100	1000	10	16 sec
10	100	1000	20	16.8 sec
10	200	2000	10	13.6 sec
10	200	2000	20	13.5 sec
10	500	5000	10	11.8 sec
10	500	5000	20	20.1 sec

4.1.5.1 Response Time, no Contention Baseline

Events	**Nodes**	**Total Events**	**Threads**	**Time**
10	100	1000	10	10.6 sec
10	100	1000	20	15.7 sec
10	200	2000	10	10.4 sec
10	200	2000	20	14.9 sec
10	500	5000	10	10.8 sec
10	500	5000	20	15.8 Sec

4.1.6 Experience

If a post-condition fails, the transaction is committed, so corrective steps must be further be taken. An improvement would be when automatically detecting the postcondition failure, to not just issue an error but create a new checkout result for restoring the node to its initial state.

Events associated with nodes can grow and slow down transaction times. The testing procedure tries to alleviate this problem by creating an upper limit to the number of events that occur on any one tree node. An improvement would be to clean up the events table regularly or using a table partition to separate current and old events. Another improvement would be to issue a new checkout when checking in if a deadlock is detected. Last, inactive events should be investigated to be omitted from the interface, which will definitely help optimize the communication.

Logic needs to be added to the current implementation to verify the assumed implementation properties - such as external environmental argument refinement to internal arguments. Further, the “read prepare” and “save” section, for running read logic verification on the client was found to be missing, ie.: server runs: prepare logic, get from storage, read logic, write to storage, after logic, client then runs on the result from the server: prepare verify logic on arguments and “read prepare” section, read verify logic on “read” and “save” sections, and post-transaction verification on the “result” section.

4.1.7 Maintenance

Code has minimal duplication and well defined boundaries. This allowed for straightforward debugging, and errors are propagated to the client for further analysis.

4.2 Discussion

4.2.1 Response Time, Scalability, and Ineffectiveness Under Contention

Distributes events for the same node across the number of threads.

Nodes	**Events**	**Total Events**	**Threads**	**Sleep Ms**	**Time**	**Ineffectiveness = failed events / total events**	**Events per Thread**	**Response Time per event**
100	10	1000	1		49.5 sec		1000	5.1 ms
100	10	1000	2	200	2.3 min	21 %	500	138 ms
100	10	1000	5	~1000	5.4 min	43.0 %	200	324 ms
100	100	10000	1		8 min		10000	48 ms
100	100	10000	2	200	28.6 min	23.2 %	5000	172 ms
100	100	10000	5	~1000	38.4 min	40.0 %	2000	230 ms

4.2.1.1 Response Time, Scalability, and Ineffectiveness Under Contention Baseline

Nodes	**Events**	**Total Events**	**Threads**	**Sleep Ms**	**Time**	**Ineffectiveness = Failed events / total events**	**Events per Thread**	**Response Time per event**
100	10	1000	1		5.1 sec		1000	50 ms
100	10	1000	2	200	16.3 sec	0	500	16 ms
100	10	1000	5	200	32.7 sec	0	200	33 ms
100	100	10000	1		10.1 sec		10000	1 ms
100	100	10000	2	200	25.8 sec	0	5000	3 ms
100	100	10000	5	200	38.9 sec	0	2000	4 ms

First, as a result of separate events both trying to lock the parent entity for writing when being activated, this is used to accurately detect command collisions. In addition, active and inactive events are used to detect concurrent event activations. The fact that all postconditions were satisfies shows this was successfully accomplished.

Second, the response time and scalability results show that there is a finite overhead to avoiding stateful algorithms in the service logic. The reliability and maintainability observed is a testament to the appropriateness of the design presented here. Compared to only SQL transaction only, it is shown how contention is serialized and thus a large difference is the time taken between the two systems. Non-contention is the usual case, and it is shown here a consistent scalability as with the database transactions only baseline. Given the asynchronous nature of the system, it should be possible to create an out-of transaction merge service that merges conflicts between temporary separate values observed by the system. See section 5.2 Asynchronous Complete Set Logic next

for more details. This technique is exemplified by (Vernon, 2013, p. 297), yet it wouldn't fix the issue, just shift it to other parts of the architecture. Better, scheduling theory could also be used to avoid contention before it occurs and allow for prioritization of changes in the case of resource-free locks where instead of locking resources, any operation interrupts previously unfinished commands. This linearization timing optimization change should bring the performance in line with a SQL-only implementation, which also does not scale under contention conditions on the same node.

Last, reordering by a synchronized start time would further solve the race issue suffered by the policy taken in the testing of saving the last command's outcome to the domain.

4.2.2 Scalability, no Contention

Distributes nodes to threads and then set events are done in isolation.

Events	**Nodes**	**Total Events**	**Threads**	**Time**	**Nodes / Thread**	**Response Time per Events**
10	100	1000	10	16.0 sec	10	16 ms
10	100	1000	20	16.8 sec	5	17 ms
10	200	2000	10	13.6 sec	20	7 ms
10	200	2000	20	13.5 sec	10	7 ms
10	500	5000	10	11.8 sec	50	2 ms
10	500	5000	20	20.1 sec	25	4 ms

4.2.2.1 Scalability, no Contention Baseline

Events	Nodes	Total Events	Threads	Time	Nodes / Thread	Response TIme per Events
10	100	1000	10	10.6 sec	10	11 ms
10	100	1000	20	15.7 sec	5	16 ms
10	200	2000	10	10.4 sec	20	5 ms
10	200	2000	20	14.9 sec	10	7ms
10	500	5000	10	10.8 sec	50	2 ms
10	500	5000	20	15.8 sec	25	3ms

From this perspective, it would seem that while locking alone is much more scalable in contention on single entities, measures need to be taken to avoid the issue. Ultimately, it seems that the contention case and the expectation that a command may be conflicting. Assuming that commands finish quickly may yield better results for the contention pitfall as well as the event chain length pitfall, shifting the onus of the programmer to cleaning up uncharacteristically long time between a checkout and a check in.

5 Extensions

5.1 Logic Testing Options

Testing has come a long way from test driven development that is described by (Beck, 2014, p. ix)

- “Write new code only if tan automated test has failed”
- “Eliminate duplication”

These simple rules are stated to create complex behaviour, but they put the onus on the team instead of achieving the systematic elimination of bugs. If there is no test created for some edge case behaviour, that requirement would go untested.

(McFarland, 2017) shows how to use the JsVerify framework to perform property-testing on Javascript code. This is an extension of the test driven development discovers edge cases by testing properties of code and giving the minimum counter-example to break that code property. This seems like the most appropriate testing based on complexity.

(Sen, 2007, p.571) describes that concolic testing can use symbolic execution of the logic to dynamically augment this counter-example search.

Statically, logic development could also use the help of dependent types. Developing with ranges of values and their properties promises to create a much more well-verified environment. For example, (Brady & Hammond, 2010) shows how to use the Idris language to create a concurrent resource sharing protocol.

5.2 Asynchronous Complete Set Logic

Since commands can be replayed using events, another commands check logic may be implemented separately that looks at results from the first service, negates preconditions, and makes sure that the sets of those negated preconditions and the nodes that were not returned are the exact same set. The complete set can be used with other queries like asynchronous aggregate updating, where the commands replay is used to update aggregates as the commands are replayed.

5.3 Superclass Membership

Obviously this small example of distributed logic design only deals with the data. What if there are logical constraints on the nodes, like a node and its parent cannot end up with a cycle in the parent_id field? In this case, another type of logic service - Tree Node Logic is created using the previous Node Data Logic. Using the idempotency of pre, read, and post now that the database dependency is abstracted away by Node Data Logic, these stages can be made recursive.

For example, if there is a node event on any node, then the nodes from the updated node to the leafs of the tree should also receive an event. This could be implemented on the Tree Node logic Checkout Get for example:

Pre: Starting at the target node recurse until no more nodes with parents that currently exist in the read section. Add the intended parent to the read section. Create a new Get request for all the subnodes of the read section.
Read: None
Post: None

This works because for every Set or Add operation, the parent and all its new children nodes have to be locked, meaning that another transaction cannot change the parent's parents(closer to the root) to belong to any of its existing descendants. The nodes use this minimal locking structure to maintain this non-cyclic invariant across all nodes.

The same could be done for growing other parts of the command domain. The read logic can then recursively transform the read data into write or range logic interface calls to Node Data Logic objects. The post-condition can follow the same recursive pattern if necessary.

Implementation-wise, there are two possible configurations for the Tree Node Service. The Tree Node service can be served from the server or serve Node Data Service from the server and deliver the Tree Node service only from the client. The possibilities grow again if the double-check interface is enabled or not between service and client, or whether transactional back-ends are necessary.

It's interesting that a tree is the perfect example for domain driven design because the union of the nodes' forests represents the aggregate root that is necessary for tree structure modifications operations between the two nodes. The above extension shows how this is achieved at the Tree Node logic level.

5.4 Storage Trust

The fact that the data is stored by one entity could in some circumstances not provide enough trust to the data, meaning that unilateral changes can be made by the business domain service host. A blockchain back-end in the database layer could be the answer to this.

Conclusion

At it is shown here, there is a minimal, well defined set of patterns that are helpful in developing scalable software services. It is rare that software developers receive the right software to work on, and a redesign for usability will not always mean an easy refactor. Open closed principle shows that sometimes it’s better to design a separate new system to replace an SQL transaction only workflow. By abstracting away the time and data constraints, it is possible to build up very lean logic, meaning that testing is not only encouraged by the stateless model’s interface, it’s also applied to much less complex code. By focusing on standard and scalable logic, services can be created, maintained, and reused with ease in today’s Internet of Things. The long-term goal is to establish a software community standard where logic is treated as a contract and published alongside implementations with a commonly shared business domain stored on a blockchain.

References

Beck, K. (2014). Test-driven development by example. Boston: Addison-Wesley.

Berners-Lee, T., Fielding, R., & Frystyk, H. (1996). *Hypertext transfer protocol--HTTP/1.0* (No. RFC 1945).

Bonner, A. J., & Kifer, M. (1995). Transaction logic programming (or, a logic of procedural and declarative knowledge).

Brady, E., & Hammond, K. (2010). Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. Fundamenta Informaticae, 102(2), 145-176.

Cantelon, M., Harter, M., Holowaychuk, T. J., & Rajlich, N. (2014). *Node. js in Action* (pp. 17-20). Manning.

Castro, P., Melnik, S., & Adya, A. (2007, June). ADO. NET entity framework: raising the level of abstraction in data programming. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (pp. 1070-1072). ACM.

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

H. (2017, September 01). Heynickc/awesome-ddd. Retrieved September 14, 2017, from https://github.com/heynickc/awesome-ddd#libraries-and-frameworks

Hassenzahl, M., & Tractinsky, N. (2006). User experience - a research agenda. Behaviour & Information Technology,25(2), 91-97. doi:10.1080/01449290500330331

Introducing JSON. (n.d.). Retrieved September 13, 2017, from http://json.org/

Lallemand, C., Gronier, G., & Koenig, V. (2015). User experience: A concept without consensus? Exploring practitioners' perspectives through an international survey. Computers in Human Behavior,43, 35-48. doi:10.1016/j.chb.2014.10.048

McFarland, M. (2017, March 14). An introduction to property based testing with JSVerfiy. Retrieved September 13, 2017, from https://medium.com/front-end-hacking/an-introduction-to-property-based-testing-with-js-verfiy-c194d60222f8

Momjian, B. (2001). *PostgreSQL: introduction and concepts* (Vol. 192). New York: Addison-Wesley.

Riel, A. J. (1996). Object-oriented design heuristics (Vol. 335). Reading: Addison-Wesley.

Sen, K. (2007, November). Concolic testing. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (pp. 571-572). ACM.

Vernon, V. (2013). *Implementing domain-driven design*. Addison-Wesley.

Wiegers, K. E., & Beatty, J. (2013). Software requirements(3rd ed.). Redmond, WA: Microsoft Press.