1. Describe the scenario/example being explored. For the scenario selected you must utilize at least three different types of entities

    - Scenario: Owners of the three major banks First Citizens, Republic Bank, and Scotia Bank; have agreed upon having an application that would allow them to monitor their customer's transactions across the three major banks. The bank owners are also able to perform transactions on behalf of customers, hence the reason they can select the user associated with the transaction. This initiative is to provide a moment of transparency for the bank owners and for each of them to get an idea of their customers banking behaviour across other banks. For example, a customer can withdraw, or deposit money from or to Republic bank, but they may also be withdrawing money from Republic bank and then manually depositing it to Scotia Bank. The system would be able to capture potential customers that does such activity and would allow the Owners to investigate further into why a customer would do something like that.

    - The entities are: **User/Customer**, **Bank**, **Transaction**

2. Implement your Event Sourcing example by capturing events to an Azure Storage account (use Blob, Table or Queue). Justify why you've chosen the type of storage.

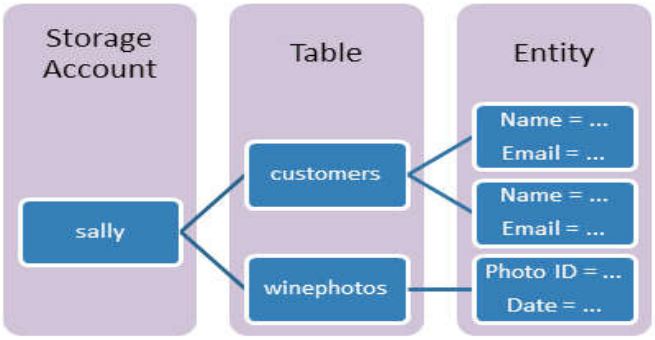
    The type of storage utilized to capture the events is azure Table storage. Reasons why Table Storage was selected is discussed below:

    Purpose of Azure Storage Tables:

    - Azure tables are ideal for storing structured, non-relational data. Such as Logging information or data similar. Non-relational means that there is no relation schema tied to the data. The event-store is required to store events, and my events are transactions, each transaction have user information, time of transaction and bank associated with the transaction; this kind of information is structured but a transaction object can be stored in a NoSQL store like Azure table store because there is no relational schema associated with the transactions being stored.
    - The Azure storage table was used because; in addition for allowing me to stored structured data without needing to a relational schema, it also is easy to query the data which allows for easy replay of events, to get to the final state. With azure table storage one can access an entity by specific details such as name or email as shown in the diagram below. For my purpose when creating the materialized view or just generating the view, specific details such as customer, Transaction, and Bank was needed.

Table storage concepts

Table storage contains the following components:

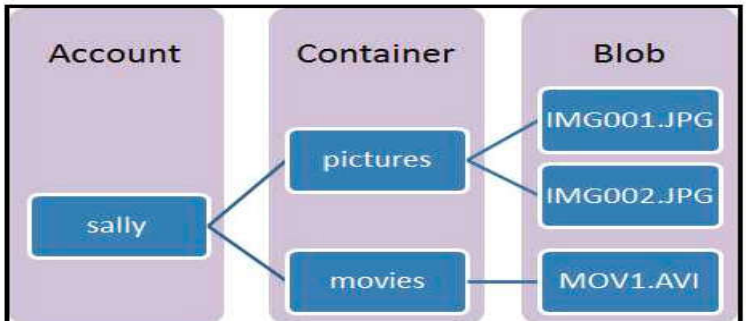


Purpose of Azure Storage Blobs

- As illustrated in the diagram bellow, though azure blob storage allows for the storage of objects, the objects stored is something call a blob, this means if we were to store a JSON object as a blob we would not get to easily query specific details in that JSON object as easily as we can with azure table storage. Hence the reason why I did not use azure table storage. Also blob storage is for storing large amounts of **unstructured** object data, such as text or binary data, however the data that I was storing still had some level of structure.

Blob service concepts

The Blob service contains the following components:



Purpose of Azure Queue Storage

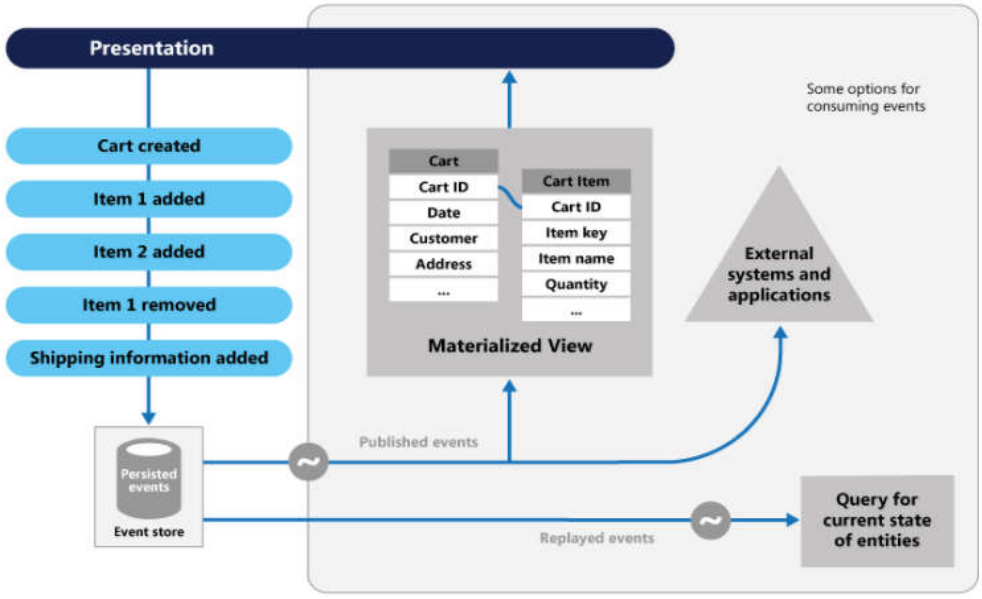
- Azure Queue storage is a service for storing large numbers of messages. Message implies some form of communication. Common use case for azure queue storage includes: creating a backlog of work to process asynchronously or passing messages from an Azure web role to an Azure worker role. None of the use cases resembled what I needed for implementing simple event sourcing and materialized view pattern, however a queue storage would be useful for handling eventual consistency where by updates that are to be made to a view gets stored on a queue, so that the clients would not have to wait on the server to apply the updates right away and the server can have some background process that takes stuff off of the queue to update the client views eventually. Queues are meant to be enqueued and dequeued and an event store main purpose is to append only, therefore I also chose azure table over queue, because queues aren't really suitable for storing events that would never leave it.

3. Create a user interface display for your data. Use the Materialized View pattern to inform how you display changes to your entities. Do justify the choices you made for the way your view presents the data and your user interface choices.

(Diagram 1)

"**Diagram 1**" shows the interface through which I take in the events (left hand side), as well as display the materialized view (right hand side).

(Diagram 2)



"Diagram 2" Gives an overview of eventsourcing and also illustrates the incorporation of the materialized view pattern. Event sourcing pattern defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store. In my solution, the append-only store that was used was an azure storage table. When an event (transaction) is sent to the server it gets added to my event store; after the event is inserted into the event store, the materialized view is updated by publishing the events from the Eventstore to the materialized view as illustrated in **"Diagram 2**". In the materialized view my balance is the main entity that is updated, the other entities customer and bank is used to identify which balance

should be updated, the updates that take place in the materialized view is done by first replaying the events in the event store and then performing an insert/replace function on the materialized view, what the insert/replace function does is that it inserts what is to be displayed on the client side into the materialized view if the record does not already exist, and if it does exist it replaces it with the most updated version of the record that is generated from the replay of the event store. In the materialized view the partition key is the userid and the rowkey is bank, therefore there cannot be two records with the same userid and bank. The UI on the client side is updated based on changes in the materialized view on the server side.

In **(Diagram 1)** on the left hand side, a form can be seen, the form uses drop downs to allow the user to specify details of the transaction and when the user have specified the necessary details, the submit button can then be pressed for the transaction to be sent to the backend/server, the server would then write to the event store based on a class called "WriteController" located inside of "jndcontrollers.py"'. In On the right hand side, a table is used to display the materialized view. The forms and the view is placed side by side to allow the user to see the updates when they submit the transaction, when the user clicks the submit button a loader appears for a brief moment (depending on network speed) and then the view gets updated accordingly with no need for page reload, it is also updated for other users that are viewing the same page in another browser. This ensures that each user sees the same data when one of them makes a change.

4. How is your view being updated? Justify your choice.

A library called socke.io is being used to update the view. When a user submits a transaction event, the json object that represents the event is sent to the server. The code that does this is in "routes.py", this is shown in "code snippet 1 below". Flask-SocketIO gives Flask applications access to low latency bi-directional communications between the clients and the server. Using something like SocketIO helps with the real time updates, and therefore each user would see the updates made by other users in their own browser/view.

Code Snippet 1:

```
@socketio.on( 'my event' )
def handle_my_custom_event( data ):
    data1 = json.dumps(data )
    data2= requests.post('http://localhost:8082/transaction', data1, headers=headers).content
    data3 = jnd.populateClientView()# gets the data from the azure table that stored the view to display on UI
    socketio.emit( 'my response', data3, callback=messageRecived )# broadcast the event, and updates all users UI
```

As shown in the above diagram data is sent to the "handle_my_custom_event" function, then it is put into a json format and post tohttp://localhost:8082/transaction, that transaction route is an api that is responsible for adding the transaction to the eventstore. It also updates the azure table storage that stores the materialized view data "table is called materializedview", immediately after the transaction is created.

Adding to the eventstore is done by "jnd.createTransaction(request)" and updating the materialized view is done by "jnd.updateView(request)" as shown in "code snippet 2" below.

Code Snippet 2:

```
@app.route('/transaction',methods=['POST'])
def add_transaction_to_eventstore():
    if request.method == 'POST':
        result=jnd.createTransaction(request)
        jnd.updateView(request)# update the materialized view stored in an azure table called materializedview
        return result
```

Azure table storage allows to query the tables quickly and flexible, therefore when updating the materialized view table, I don't have to search through every record, I only search for the records that matches the particular userid and bank. When replaying these records (withdrawing and depositing as necessary), because azure tables store the data in sorted order already this means that replaying is as simple as starting from the first record to the last record in the set of filtered records based on the restrictions (userid and bank), being processed at the time, the last record in the set would represent the last transaction made (for a particular userid and bank), and hence also represents the last event. The resulting userid, bank, and updated balance after replaying the events is what is used to update the view, based on the insert/replace function mentioned earlier

The data3 variable stores the response that is returned when "jnd.populateClientView()" is called see "**code snippet 1**". The data in data3 variable is then broadcasted to all the users that are using their UI, therefore allowing all each client interfaces that are open to have the same updated data displayed.

5. What is eventual consistency? What would you do to mitigate against some of the issues presented by eventual consistency in your example?

Eventual consistency is a consistency model used to allow for high availability that ensures that if no new updates are made to a given data item then eventually all access to that item will return the last update. In my own words eventual consistency simply means ensuring that even though the front facing part of the application (client) does not immediately get updated when a domain event is sent to the server, it would get updated eventually some time after (i.e. at a later time).

Note: A domain event is a full-fledged part of the domain model, a representation of something that happened in the domain.

Consider the diagrams below:

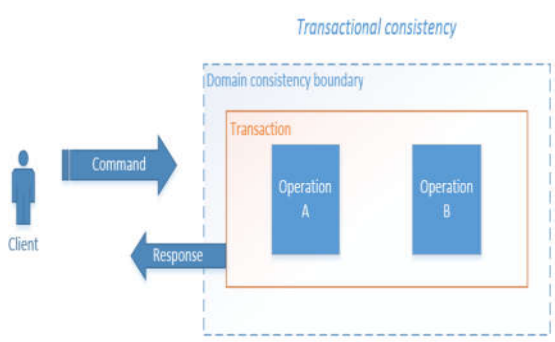


Diagram T

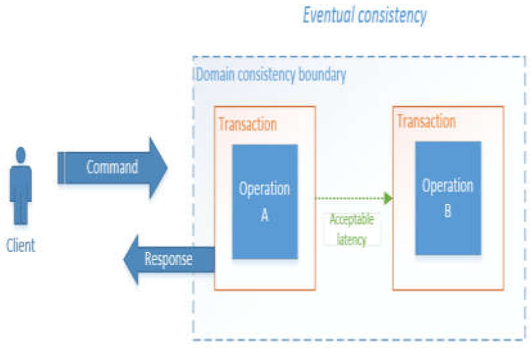


Diagram E

"Diagram T" Illustrates a typical transactional system, whereby a command is sent to the server which runs all operations (operation B and A) necessary to maintain the domain consistency inside a transaction. When the client gets the response one of the following would be true:

- Both Operation A and Operation B have succeeded or
- Both Operation A and Operation B have failed

However with eventual consistency approach as illustrated in "Diagram E", when the client gets the response it means that one of the following is true:

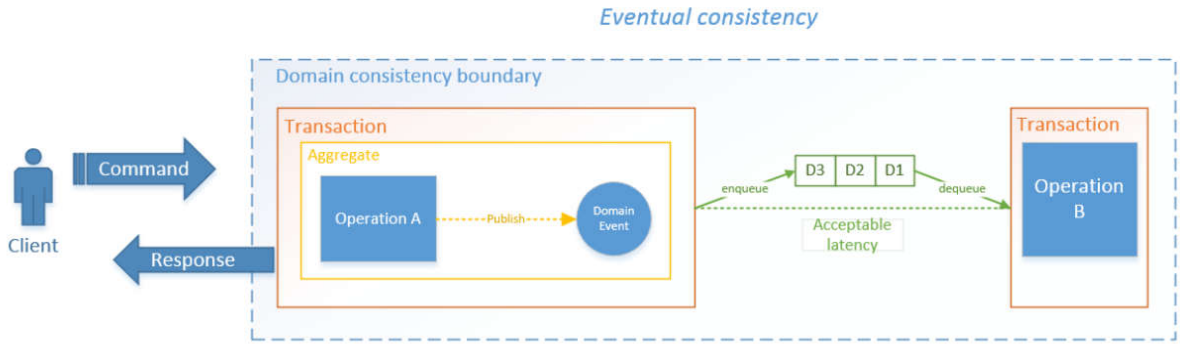
- Operation A has succeeded and Operation B is scheduled to run at a later time or
- Operation A has failed (and Operation B will not run)

Highlighting the differences between Eventual consistency and Transactional Consistency it is evident that when a client gets a 'success' response in an eventual consistency workflow, it is only guaranteed that a part of the necessary operations to maintain the domain consistency have been successfully executed and the rest are scheduled to run at a later time. Whereas for transactional consistency workflows if the client receives a 'success' response it is guaranteed that all the necessary operations to maintain the domain consistency have been successfully executed.

In my example some of the issues I face with eventual consistency is the response time from the server due to processing the transactions, i.e. the time it takes to write to the event store, then update the materialized view and then to broadcast the updates to all the client views causes the client to notice a load time (whether short or long depending on the network) that is indicated by the loader implemented in my solution. Dealing with eventual consistency in my solution would allow long running operations to run in the background in order to optimize the front-end performance.

A revision of Diagram E demonstrates how operation B would be triggered; this is shown below in Diagram E2:

Diagram E2



As can be seen by the utilization of a queue eventual consistency behavior can be obtained. Hence in my example to mitigate against the issues of eventual consistency instead of updating the view immediately, I would store the object/domain event that would act as a trigger for updating the materialized view, i.e. if a user submits a json structure with bank, userid, and transaction, then that object would be enqueue as well as added to the event store, and a message would be sent back to the client confirming that the transaction have been sent to the server for processing and clarifying that

the updates may not take effect immediately, this would allow the client to continue using the application without being concerned about wait time. On the server a separate process would be built that would have the responsibility of dequeueing the queue and based on the domain event that is dequeued the eventstore would be replayed to provide the update object that would either be inserted into the materialized view or replace the old record in the materialized view and then all the clients would get updated by this separate process, so instead of updates happening immediately after an event is sent, the updates occurs based on event read off of a queue.

Application Installation:

https://github.com/DavidDexterCharles/cloudAssignment2017

Application Usage:

- The Materialized view and Form to input transactions: http://localhost:8082/
- A simple view that shows the events in the azure table eventstore:http://localhost:8082/events
- Special Note: In the drop down list for trans ID, void is a valid selection, however it does not affect the balance and isn't used in the replay of the transactions to update the view, it can be used to add a user with an initial balance of $0.00 dollars.