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.

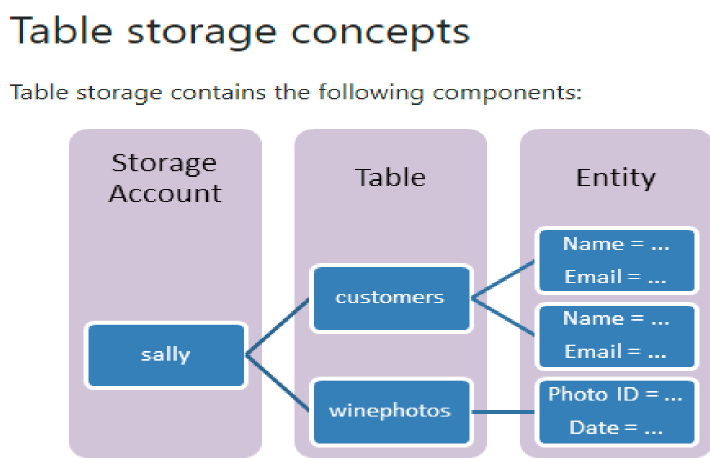
    - The entities are: **User/Customer**, **Bank**, **Transaction**, and **Time**

2. Implement your Event Sourcing example by capturing events to an Azure Storage account (use either 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:

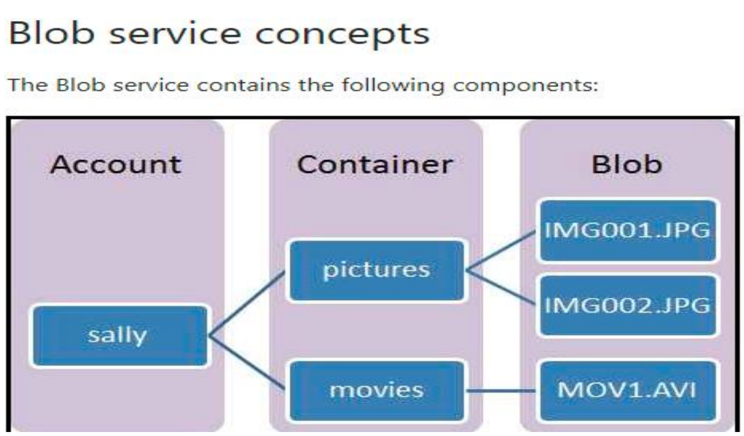
    Purpose of Azure Storage Tables:

    - 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.



Purpose of Azure Storage Blobs

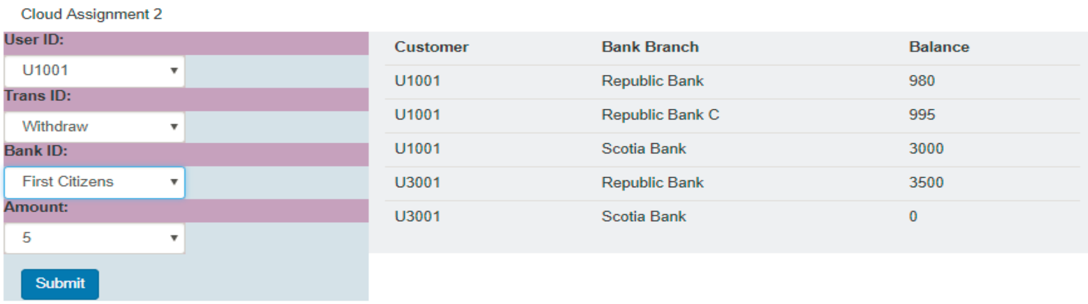
- As illustrated in the diagram bellow, though azure blob storage allows for the storage of objects, the objects stored is something call a blob, this means if we were to store a JSON object as a blob we would not get to easily query specific details in that JSON object as easily as we can with azure 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.



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 hence the reason why I did not use this service as my event store.

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.



Above is the interface through which I take in the events (left hand side), as well as display the materialized view (right hand side) which is a replay of all the events recorded so far in the event log. 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"'. 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 make 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 represent the event is sent to the server. The code that does this is in "routes.py", this is shown in "code snippet 1 below".

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 to http://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 diagram 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 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 the UI 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 scenario this item is the materialized view, all users that have the UI interface open is actually getting the data from the same azure table called “materializedview”, however if a user submits a transaction and the page in the browser updates that users view, other users would still see the same old data that may not be necessarily consistent with the “materializedview” table, this is the main issue presented, because unless resolved, transaction amounts would not reflect the true balance of money to everyone’s individual views. To mitigate against these issues, one approach would be to design the code to periodically update each person’s views, but there would still be possibility of stale data. The other option which I implemented, is to allow for real time live updates. 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.

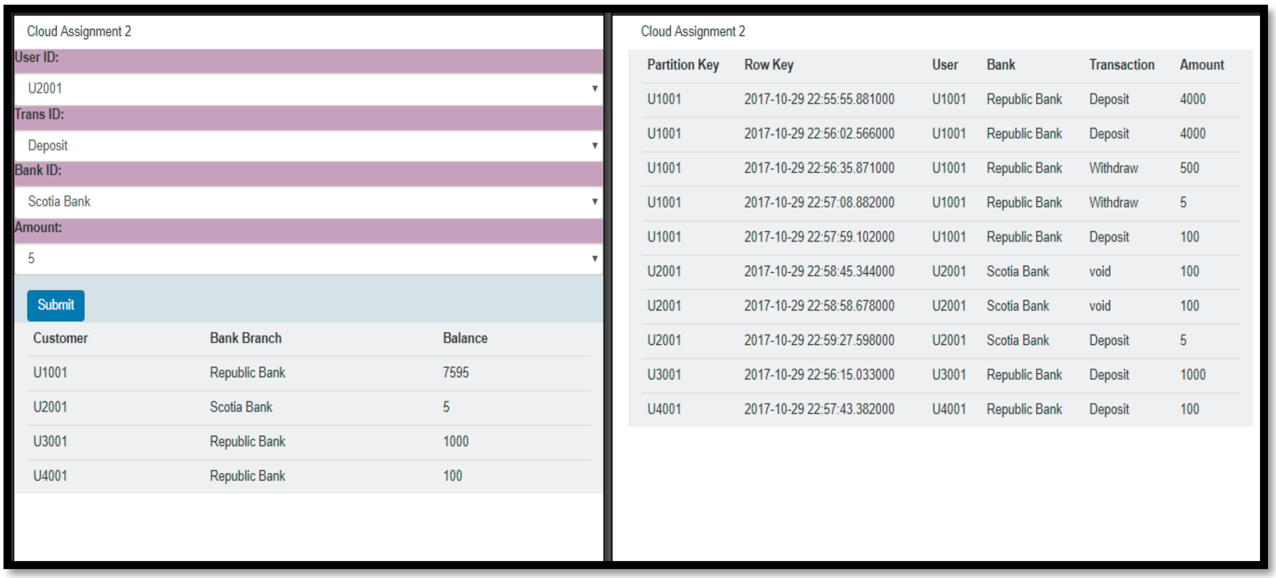
ID: 811000385   Name: David Charles   Course: COMP 6905   Course Title: Cloud Technologies

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.



Cloud Assignment 2

User ID:
U2001

Trans ID:
Deposit

Bank ID:
Scotia Bank

Amount:
5

Submit

Customer	Bank Branch	Balance
U1001	Republic Bank	7595
U2001	Scotia Bank	5
U3001	Republic Bank	1000
U4001	Republic Bank	100

Cloud Assignment 2

Partition Key	Row Key	User	Bank	Transaction	Amount
U1001	2017-10-29 22:55:55.881000	U1001	Republic Bank	Deposit	4000
U1001	2017-10-29 22:56:02.566000	U1001	Republic Bank	Deposit	4000
U1001	2017-10-29 22:56:35.871000	U1001	Republic Bank	Withdraw	500
U1001	2017-10-29 22:57:08.882000	U1001	Republic Bank	Withdraw	5
U1001	2017-10-29 22:57:59.102000	U1001	Republic Bank	Deposit	100
U2001	2017-10-29 22:58:45.344000	U2001	Scotia Bank	void	100
U2001	2017-10-29 22:58:58.678000	U2001	Scotia Bank	void	100
U2001	2017-10-29 22:59:27.598000	U2001	Scotia Bank	Deposit	5
U3001	2017-10-29 22:56:15.033000	U3001	Republic Bank	Deposit	1000
U4001	2017-10-29 22:57:43.382000	U4001	Republic Bank	Deposit	100