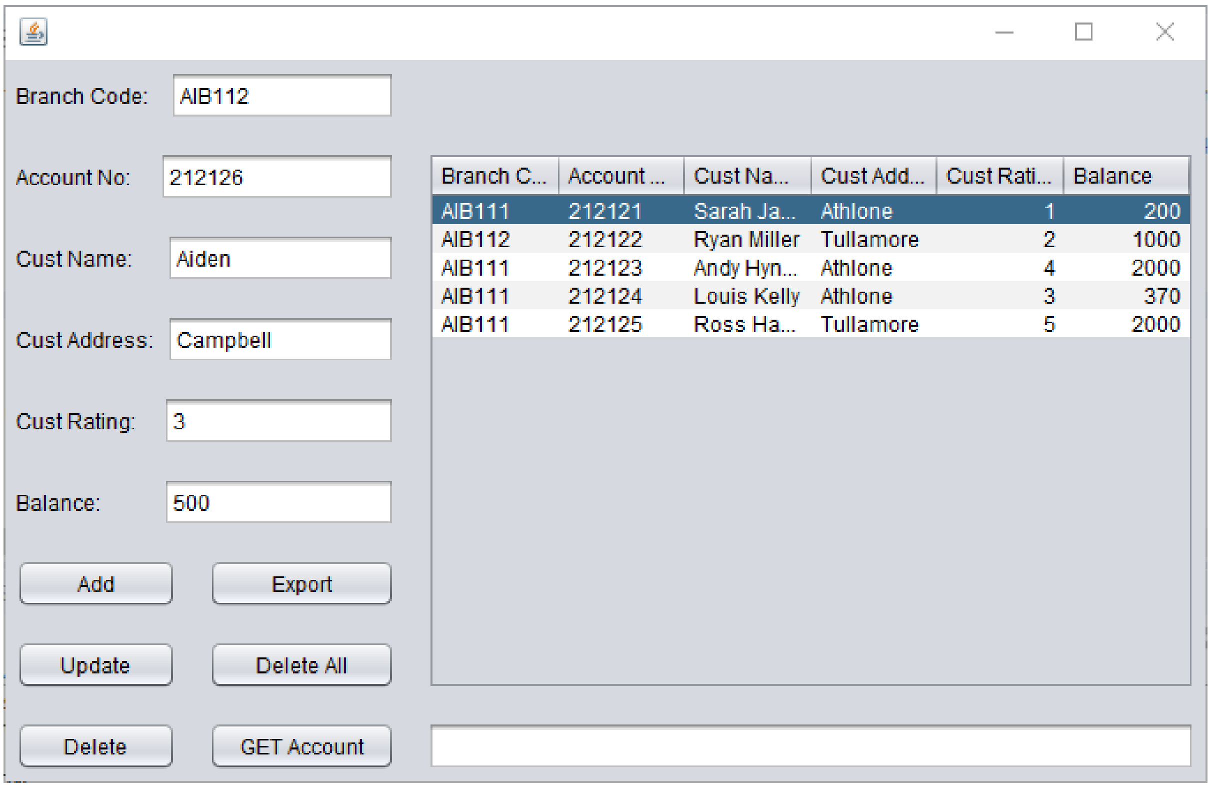
Distributed Systems Assignment 2020

Ross Hayden – A00240194

Project Overview

Main Window



Branch Code: AIB112

Account No: 212126

Cust Name: Aiden

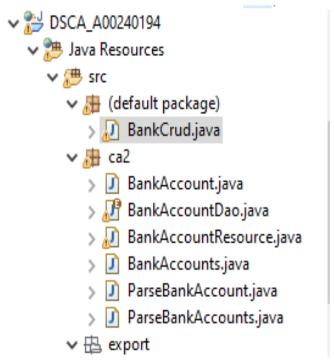
Cust Address: Campbell

Cust Rating: 3

Balance: 500

Branch C...	Account ...	Cust Na...	Cust Add...	Cust Rati...	Balance
AIB111	212121	Sarah Ja...	Athlone	1	200
AIB112	212122	Ryan Miller	Tullamore	2	1000
AIB111	212123	Andy Hyn...	Athlone	4	2000
AIB111	212124	Louis Kelly	Athlone	3	370
AIB111	212125	Ross Ha...	Tullamore	5	2000

Files



```
DSCA_A00240194
  Java Resources
    src
      (default package)
        BankCrud.java
      ca2
        BankAccount.java
        BankAccountDao.java
        BankAccountResource.java
        BankAccounts.java
        ParseBankAccount.java
        ParseBankAccounts.java
      export
```

This application contains a table for displaying and manipulating data about bank accounts and runs off a Tomcat 8.5 server in Eclipse. This data is stored in a HSQLDB table, to keep record of the customers branch, account number, name, address, rating, and balance. An XMLPullParser is then used to process and organize the Information. The Tomcat server uses the Javax ws rest api to retrieve and show the data.

Main Functions:

Get - Return the information of single row by entering an Account Number and Branch Code.

Delete - Enter a Branch Code and Account Number to delete the entry from the table.

Put – Enter the Branch Code and Account Number of an already existing account and the table is updated.

Post – Enter the details for a new account and create it on the server.

Get All – Returns all data from the table

Delete All – Removes all data from the tables.

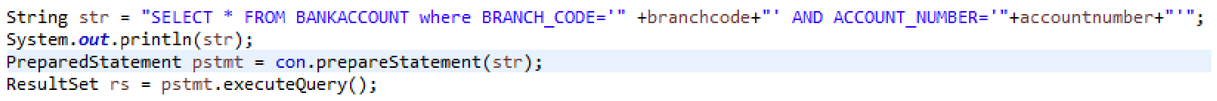
Export to Excel – Copies the data from the table into an Excel Sheet

1. Requirement

" Build a client application that sends all of the HTTP requests GET/PUT/POST/DELETE. "

Completion

The select, update, delete and insert functions are called from the BankAccountDao class (data access object) and connects to the HSQLDB from here using DriverManager. A DAO class allows the client to interact with the database using SQL commands in the background. This is an example of the select command for getting a single bank account by its branch code and account number:



```
String str = "SELECT * FROM BANKACCOUNT where BRANCH_CODE='" +branchcode+"' AND ACCOUNT_NUMBER='"+accountnumber+"'";
System.out.println(str);
PreparedStatement pstmt = con.prepareStatement(str);
ResultSet rs = pstmt.executeQuery();
```

Please see the BankAccountDao java class for more implementations of the SQL commands.

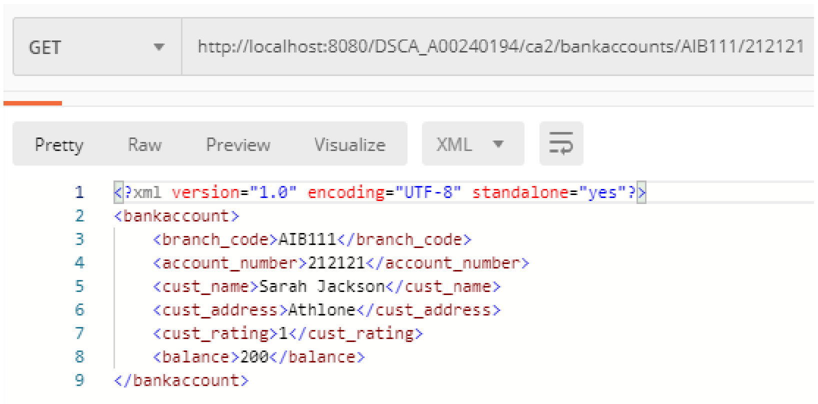
2. Requirement

" Build a server application using tomcat server, that responds to all of the HTTP requests GET/PUT/POST/DELETE "

Completion

Get: 2 different @GET JAX:RS commands are used, one which returns all of the accounts in the database and one for returning a specific bank account by branch code and account number. The single get works by supplying this information as part of the URI.

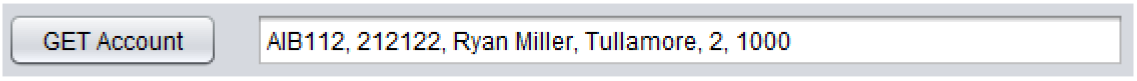
Example of a single bar returned:


GET
http://localhost:8080/DSCA_A00240194/ca2/bankaccounts/AIB111/212121
Pretty
Raw
Preview
Visualize
XML

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bankaccount>
    <branch_code>AIB111</branch_code>
    <account_number>212121</account_number>
    <cust_name>Sarah Jackson</cust_name>
    <cust_address>Athlone</cust_address>
    <cust_rating>1</cust_rating>
    <balance>200</balance>
</bankaccount>
```

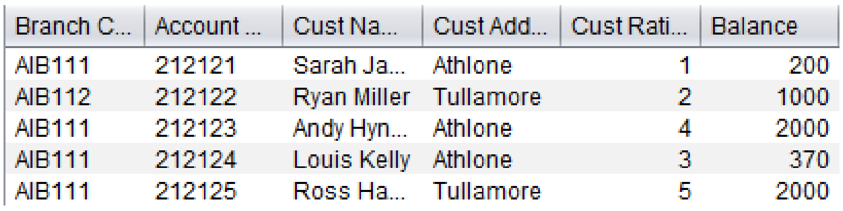
As part of the GET the XMLPullParser is used to organize the data and take out what is needed from between the XML tags, as can be seen in ParseBankAccount. When the information is returned it is pulled apart using if statements to determine which of the XML tags the parser is currently in. For example, as the parser loops through the specific bank account it will encounter the start of a tag such like <branch_code>. Then it sets the text between the end of this tag and the end of the closing </ branch_code > as the Branch Code for the current bank account. The bank account will be returned and when the </bankaccount> closing tag is found, the end of the information about that bank account is determined.

The information returned is the displayed in the Single Get Area:



AIB112, 212122, Ryan Miller, Tullamore, 2, 1000

The getAll works similarly but it uses the ParseBankAccounts class instead. This returns a list of all the bank accounts and is used to write to the table in the same way that the single get was.



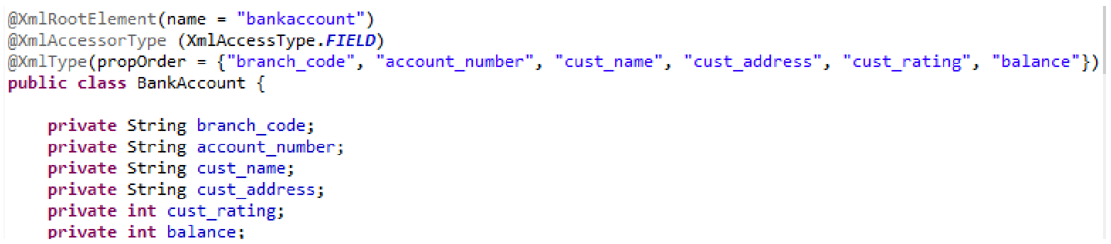
Branch C...	Account ...	Cust Na...	Cust Add...	Cust Rati...	Balance
AIB111	212121	Sarah Ja...	Athlone	1	200
AIB112	212122	Ryan Miller	Tullamore	2	1000
AIB111	212123	Andy Hyn...	Athlone	4	2000
AIB111	212124	Louis Kelly	Athlone	3	370
AIB111	212125	Ross Ha...	Tullamore	5	2000

Put: The @PUT command is used to update the details of a bank account. It takes in the values in the Branch Code and Account Number as parameters and the new details are taken from the other text fields. These values are passed into an inputXML string placing the values into a block of XML that is pushed to the server. These are parsed in the BankAccountsResource class as a bankAccount object, as the bankAccount object class is used to detect XML tags also, so they may be passed into an object which is then sent to the DAO. The update method in the DAO uses the getter methods to get the values and put them into an SQL query.

The branch code and account number being read in as @PathParam so they can be then sent as parameters to the BankAccountDao:

```
putBankAccount(BankAccount bankAccount, @PathParam("branch_code") String branch_code, @PathParam("account_number")
```

The BankAccount object class and its ability to read tags:

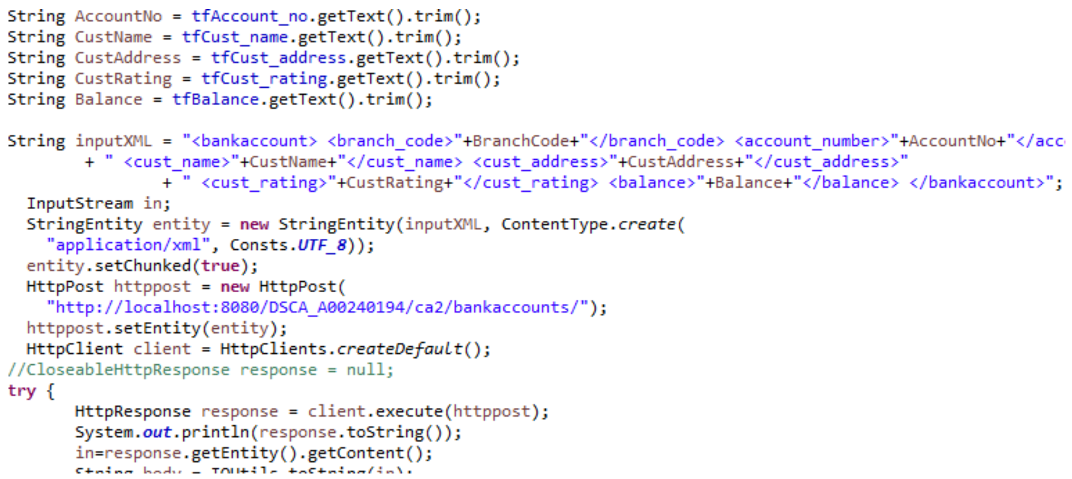


```
@XmlRootElement(name = "bankaccount")
@XmlAccessorType (XmlAccessType.FIELD)
@XmlType(propOrder = {"branch_code", "account_number", "cust_name", "cust_address", "cust_rating", "balance"})
public class BankAccount {

    private String branch_code;
    private String account_number;
    private String cust_name;
    private String cust_address;
    private int cust_rating;
    private int balance;
```

<u>Post</u>: Takes in six values from the user for a new bank account and then passes the data to the server and onto the BankAcountResource class in a similar fashion to the PUT (I.e. with parameters and into a block of XML) and then on the BarDao to create the object in SQL.

The GUI class taking in the values and putting them into XML:



```
        String AccountNo = tfAccount_no.getText().trim();
        String CustName = tfCust_name.getText().trim();
        String CustAddress = tfCust_address.getText().trim();
        String CustRating = tfCust_rating.getText().trim();
        String Balance = tfBalance.getText().trim();

        String inputXML = "<bankaccount> <branch_code>"+BranchCode+"</branch_code> <account_number>"+AccountNo+"</acc
                + " <cust_name>"+CustName+"</cust_name> <cust_address>"+CustAddress+"</cust_address>"
                        + " <cust_rating>"+CustRating+"</cust_rating> <balance>"+Balance+"</balance> </bankaccount>";
          InputStream in;
          StringEntity entity = new StringEntity(inputXML, ContentType.create(
            "application/xml", Consts.UTF_8));
          entity.setChunked(true);
          HttpPost httppost = new HttpPost(
            "http://localhost:8080/DSCA_A00240194/ca2/bankaccounts/");
          httppost.setEntity(entity);
          HttpClient client = HttpClients.createDefault();
        //CloseableHttpResponse response = null;
        try {
                HttpResponse response = client.execute(httppost);
                System.out.println(response.toString());
                in=response.getEntity().getContent();
                String body = IOUtils.toString(in);
```

<u>DELETE</u>: There are two versions of the @DELETE command used, one that takes a Branch Code and Account number given by the user as parameters and one that simply deletes everything in the table. The single delete works very similarly to the @GET, except it calls a delete command in SQL based on the URI provided

Every time a command is called (except for the single get) then the fetch() function is called to display the newly updated table on the JTable. This was done to make sure the user can see their changes without having to click a refresh button.

3. Requirement

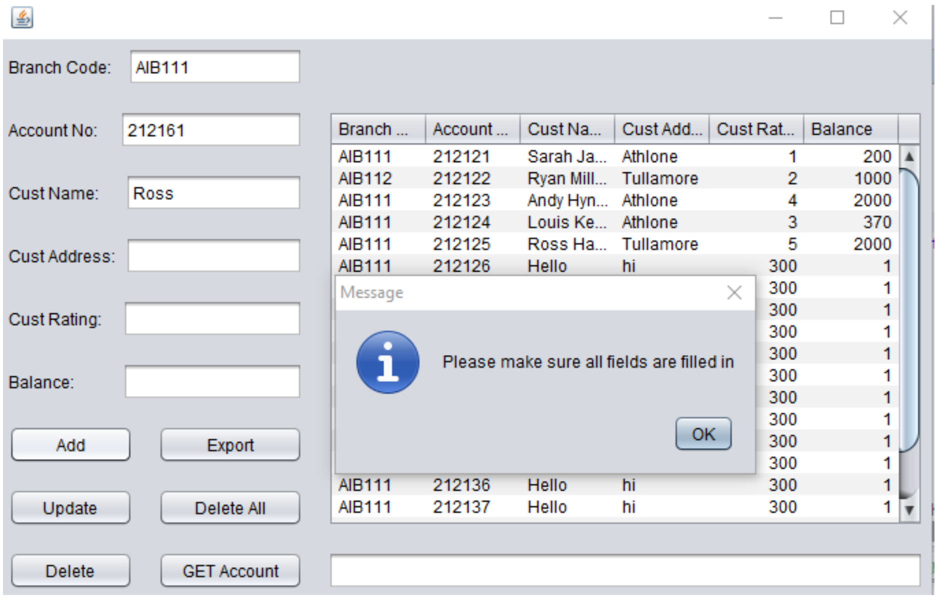
" The client application will parses the response using XMLPullParser and outputs to the GUI" + "A tomcat server that responds to all of the HTTP requests GET/PUT/POST/DELETE""

Completion

Most screenshots of the GUI and XML parsing have already been included but there are some extra features that will be highlighted here.

Alerts

If a user tries to call a command without entering all the required fields for that method a popup box is displayed and no further code is executed:



Branch Code:	AIB111
Account No:	212161
Cust Name:	Ross
Cust Address:	
Cust Rating:	
Balance:	

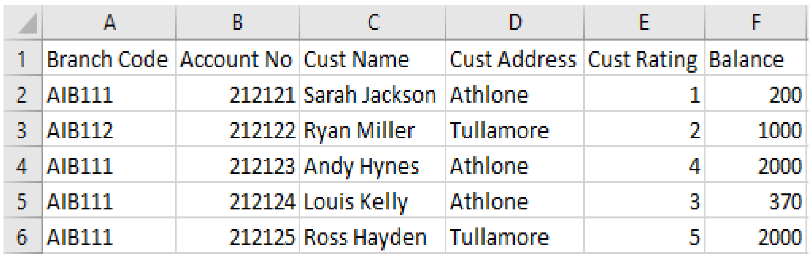
Branch ...	Account ...	Cust Na...	Cust Add...	Cust Rat...	Balance
AIB111	212121	Sarah Ja...	Athlone	1	200
AIB112	212122	Ryan Mill...	Tullamore	2	1000
AIB111	212123	Andy Hyn...	Athlone	4	2000
AIB111	212124	Louis Ke...	Athlone	3	370
AIB111	212125	Ross Ha...	Tullamore	5	2000
AIB111	212126	Hello	hi	300	1
				300	1
				300	1
				300	1
				300	1
				300	1
				300	1
				300	1
				300	1
				300	1
AIB111	212136	Hello	hi	300	1
AIB111	212137	Hello	hi	300	1

Message

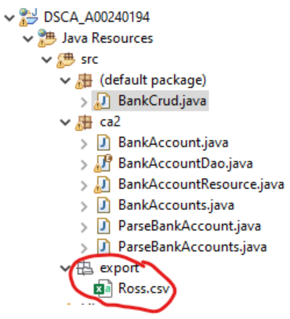
Please make sure all fields are filled in

Export

All data from the table is printed to an excel file.



	A	B	C	D	E	F
1	Branch Code	Account No	Cust Name	Cust Address	Cust Rating	Balance
2	AIB111	212121	Sarah Jackson	Athlone	1	200
3	AIB112	212122	Ryan Miller	Tullamore	2	1000
4	AIB111	212123	Andy Hynes	Athlone	4	2000
5	AIB111	212124	Louis Kelly	Athlone	3	370
6	AIB111	212125	Ross Hayden	Tullamore	5	2000

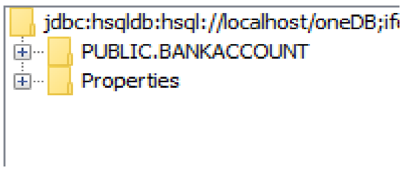


```
DSCA_A00240194
  Java Resources
    src
      (default package)
        BankCrud.java
      ca2
        BankAccount.java
        BankAccountDao.java
        BankAccountResource.java
        BankAccounts.java
        ParseBankAccount.java
        ParseBankAccounts.java
      export
        Ross.csv
```

4. Requirement

" The data in the response will be taken from an HSQLDB database."

Completion

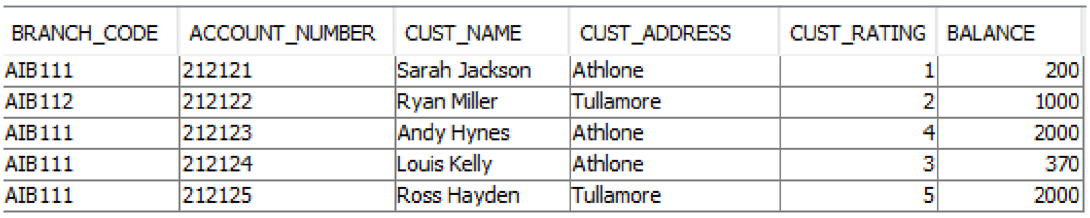


jdbc:hsqldb:hsql://localhost/oneDB;ifı

PUBLIC.BANKACCOUNT

Properties

SELECT * FROM BANKACCOUNT



BRANCH_CODE	ACCOUNT_NUMBER	CUST_NAME	CUST_ADDRESS	CUST_RATING	BALANCE
AIB111	212121	Sarah Jackson	Athlone	1	200
AIB112	212122	Ryan Miller	Tullamore	2	1000
AIB111	212123	Andy Hynes	Athlone	4	2000
AIB111	212124	Louis Kelly	Athlone	3	370
AIB111	212125	Ross Hayden	Tullamore	5	2000

Creation SQL:

CREATE TABLE PUBLIC.BANKACCOUNT(BRANCH_CODE VARCHAR(10), ACCOUNT_NUMBER VARCHAR(10), CUST_NAME VARCHAR(20), CUST_ADDRESS VARCHAR(50), CUST_RATING DECIMAL(3), BALANCE DECIMAL(7));

Steps for First Time Use:

1. **Run Tomcat Server.**
2. **Drag the db/managedDB.xml file into Ant**
3. **Double click it, and when the database has set up, click start, then manage.**
4. **If there is no data in the table, paste the SQL from the table.txt file into the HSQLDB Interface.**

The first time the project is run the user must populate the table their self. Once the table is set up this does not need to be done a second time. This isn't a very practical way of creating the table because the user has to launch the HSQLDB interface, and it requires some work.