Exercise 4.1 – Nesting User Defined Types (UDTs)

This exercise is designed to familiarize you with the basic code required to write data to a UDT field in a DSE table.

In this exercise, you will:

- Understand how UDTs work with the DataStax drivers
- Learn how to work with nested UDTs by adding an address *User Defined Type* (UDT) to the list of users within the keyspace

This exercise uses the following files in the *Session 3* exercise project:

- *User.Java:* This file defines a Plain Old Java Object (POJO) class useful for storing and using user data at runtime. The format of this class matches that of the *user* table in the *killrvideo_test* keyspace.

- *CassandraUserDAO.java:* This file defines the class through which the Killrvideo application reads and writes data to DSE.

- *UserDAOAddressTest.java:* This file implements a class for adding a user to the *user* table and then updating the user's address.

Overview

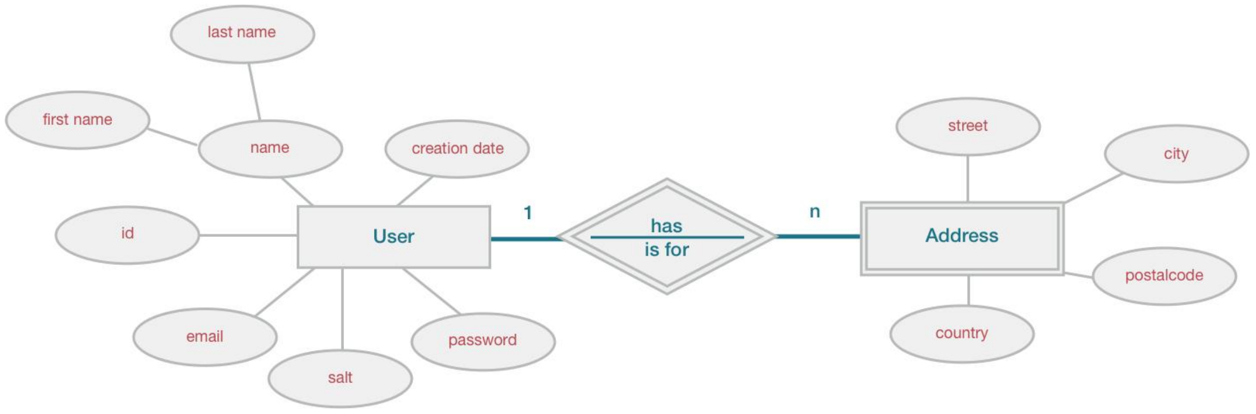
All steps will provide most of the required code. Your task is to place the additional code where needed and understand how the code works. Here is a summary of steps you should follow:

Step 1: Examine the exercise files
Step 2: Add functionality to manage addresses in the file *User.java*
Step 3: Add a Prepared Statement to the file *CassandraUserDAO.java* that will enable updating a user with new addresses
Step 4: Run the file *UserDAOAddressTest.java* and observe how data is read via the Java driver and the `getUser()` function

Step 1: Examine the exercise files

The purpose of this step is to understand which files will require modification and why.

1. This is a *1-n Diagram* that conceptually describes the problem space. Basically, each user has a 1-n relationship with his or her addresses. Take a moment to become familiar with this diagram.



2. Below are the scripts that implement the conceptual model shown above. They include the CREATE TABLE script for the address UDT and user. These scripts can also be found in the *session3/cql/create-schema.cql* file.

```
CREATE TYPE address (
    street text,
    city text,
    postalcode text,
    country text
);

CREATE TABLE user (
    email text PRIMARY KEY,
    addresses map<text,frozen<address>>,
    phone_numbers map<text, decimal>,
    created timestamp,
    fname text,
    lname text,
    password blob,
    salt blob,
    user_id uuid
);
```

3. Switch to *session3* of the killrvideo project and open the following files:

   a. ~/session3/src/main/java/com/datastax/training/killrvideo/model/Address.java

   b. ~/session3/src/main/java/com/datastax/training/killrvideo/model/User.java

   c. *~/session3/src/main/java/com/datastax/training/killrvideo/model/dao/cassandra/CassandraUserDAO.java*
   d. *~/session3/src/test/java/com/datastax/training/killrvideo/dao/cassandra/UserDAOAddressTest.java*

4. Take a moment to examine the files *Address.java* and *User.java*. They are both Plain Old Java Object (POJO) files, with private members, getters, setters and additional functions for processing, formatting or validating data.

5. Next, examine the file *CassandraUserDAO.java*. Note the code blocks in the constructor, `getUser()`, `addAddressToUser()`, `addressToUDT()`, and `UDTToAddress()` functions. In order to update a user in the user table with a set of addresses, these functions will need the appropriate code.

6. Examine the file *UserDAOAddressTest.java*. The method `testAddUserAndUpdateAddress()` has a few lines of code that add the user and then update the user's address via the code we provide to the file *CassandraUserDAO.java*.

Step 2: Manage addresses in the User.java file

The purpose of this step is to add a private property for the address as well as a *getter* and *setter* to manage its contents. This action completes the User class so that 1) one or more addresses can be associated with a user, and 2) it models the nesting of the address at the code level just as the addresses UDT is nested in the user table in the *killrvideo_test* keyspace.

1. Add the following property to the User class in the following file: *session3/src/main/java/com/datastax/training/killrvideo/model/User.java*:

```
//TODO: Begin custom code here
private Map<String, Address> addresses;
```

2. Directly below this line, add the following functions; `getAddresses()` and `setAddresses()`.

```
public Map<String, Address> getAddresses() { return addresses; }
public void setAddresses(Map<String, Address> addresses) {
        this.addresses = addresses; }
//TODO: End custom code here
```

3. Locate the `equals()` function. This function ensures the user object has been properly set up before it is used. Add the following line of code to include addresses in the verification process:

```
//TODO: Begin custom code here
if (addresses != null ? !mapEquals(addresses, user.addresses) :
        user.addresses != null) return false;
//TODO: End custom code here
```

4. Next, locate the `hashCode()` function. This method is used to hash the entire user, if required. Add the following code to include the addresses in the hashing:

```
//TODO: Add custom code here
result = 31 * result + (addresses != null ?
        User.getMapHashcode(addresses) : 0);
//TODO: End custom code here
```

5. Finally, find the `clone()` function. This function is used to clone the user if needed. Add the following line to the correct function to ensure that the address is included in the cloning:

```
//TODO: Add custom code here
clone.addresses = this.addresses;
//TODO: End custom code here
```

6. Note how `Address` is now a nested structure in the `User` class using a `Map<string, Address>` reflecting the same structure in our database. The `String` keys in this situation would be “home”, “work”, etc.

7. Go to “Build” on the menu bar and run the option “Recompile ‘User.java’”. It should recompile successfully. Confirm the message “Compilation completed successfully” in the lower-bottom margin. If not, try to rebuild the project and see if it successfully compiles at the end.

Step 3: Add a Prepared Statement to CassandraUserDAO.java

The purpose of this step is to enable the file *CassandraUserDAO.java* to process the addition of a new address to a user. This will be done via a Prepared Statement.

1. Open the following file in the IDE: *~/session3/src/main/java/com/datastax/training/killrvideo/model/dao/cassandra/CassandraUserDAO.java*

2. Add a private property for the `PreparedStatement` in the custom code block found at the start of the class where the other properties are listed. Also, add a property for the `addressType`:

```
//TODO: Begin custom code here
private PreparedStatement addAddressToUserStatement;
private UserType addressType;
//TODO: End custom code here
```

3. By adding a `UserType` we are setting up the class so it can write from Java code to DSE. In order to do that, it requires a `UserType` that matches the UDT in DSE. This particular `UserType` is used to perform this function.

4. In the constructor, add the code below to initialize our Prepared Statement property and our `UserType` property. This ensures the code is ready when it comes time to write addresses to the database:

```
//TODO: Begin custom code here
addAddressToUserStatement = session.prepare(
        "UPDATE user " +
        "SET addresses[:addressName] = :address " +
        "WHERE email = :email");

addressType = session.getCluster()
        .getMetadata()
        .getKeyspace(getCassandraSession().getLoggedKeyspace())
        .getUserType("address");
//TODO: End custom code ends here
```

5. Locate the functions `addressToUDT()` and `UDTToAddress()`. These use the address class's setters and getters to translate from an address to a UDT, or vice versa. Fill in the missing code in the appropriate TODO code blocks:

```
public UDTValue addressToUDT(Address address) {
        if (address != null) {

        //TODO: Add custom code here
        return addressType.newValue()
                .setString("street", address.getStreet())
                .setString("city", address.getCity())
                .setString("country", address.getCountry())
                .setString("postalcode", address.getPostalCode());
        //TODO: End custom code here

        }
        return null;
    }

    public Address UDTToAddress(UDTValue udtValue) {

        //TODO: Begin custom code here
        newAddress.setCity(udtValue.getString("city"));
        newAddress.setCountry(udtValue.getString("country"));
        newAddress.setStreet(udtValue.getString("street"));
        newAddress.setPostalCode(udtValue.getString("postalCode"));
        //TODO: End custom code here

    }
```

5. Find the method `addAddressToUser()`. Its purpose is to write a user's addresses to DSE. Add the following code:

```
//TODO: Add custom BoundStatement here
BoundStatement boundStatement =
       addAddressToUserStatement.bind()
               .setString("email", email)
               .setString("addressName", addressName)
               .setUDTValue("address", addressToUDT(newAddress));

session.execute(boundStatement);
//TODO: Your code BoundStatement here
```

Note: The `bind()` method of the Prepared Statement just implemented is used to create a Bound Statement in which the parameter values (email, address name and address) are bundled up for execution along with the prepared statement itself. We then pass the Bound Statement to `session.execute()` which in turn executes the write. Notice the `addressToUDT()` method, to which added additional code was just added, is used to convert the address into a format (basically a UDT) to which the Bound Statement can now write.

1. Navigate to the `getUser()` method. This method is used to fetch a specific user from the *user* table based on the primary key, which is the email address. Add the code needed to ensure that address is also fetched when `getUser()` is invoked. Add the following code to the TODO block at the end of the method:

```
//TODO: Add custom code here
Map<String, UDTValue> uDTAddresses = row.getMap("addresses",
        String.class, UDTValue.class);
Map<String, Address> addresses = new HashMap<>(uDTAddresses.size());
```

```java
for (Map.Entry<String, UDTValue> entry : uDTAddresses.entrySet()) {
        addresses.put(entry.getKey(), UDTToAddress(entry.getValue()));
}

newUser.setAddresses(addresses);
//TODO: End custom code here
```

Step 4: Run UserDAOAddressTest.java

This last step requires you to run the *UserDAOAddressTest.java* file and observe how data is read via the Java driver and the `getUser()` function.

1. Open a terminal window and truncate the *user* table.

```
cqlsh> TRUNCATE TABLE killrvideo_test.user;
```

2. Run a SELECT statement on the user table to ensure it is empty.

```
cqlsh> USE killrvideo_test;
cqlsh:killrvideo_test> SELECT* FROM killrvideo_test.user;

email | addresses | fname | joined | lname | password | phone_numbers |
        salt | user_id
-------+-----------+-------+--------+-------+----------+---------------
        +-------+---------

(0 rows)
```

3. Open the following file in the IDE: *session3/src/test/java/com/datastax/training/killrvideo/dao/cassandra/UserDAOAddressTest.java*. Locate the method named `testAddUserAndUpdateAddress()`.

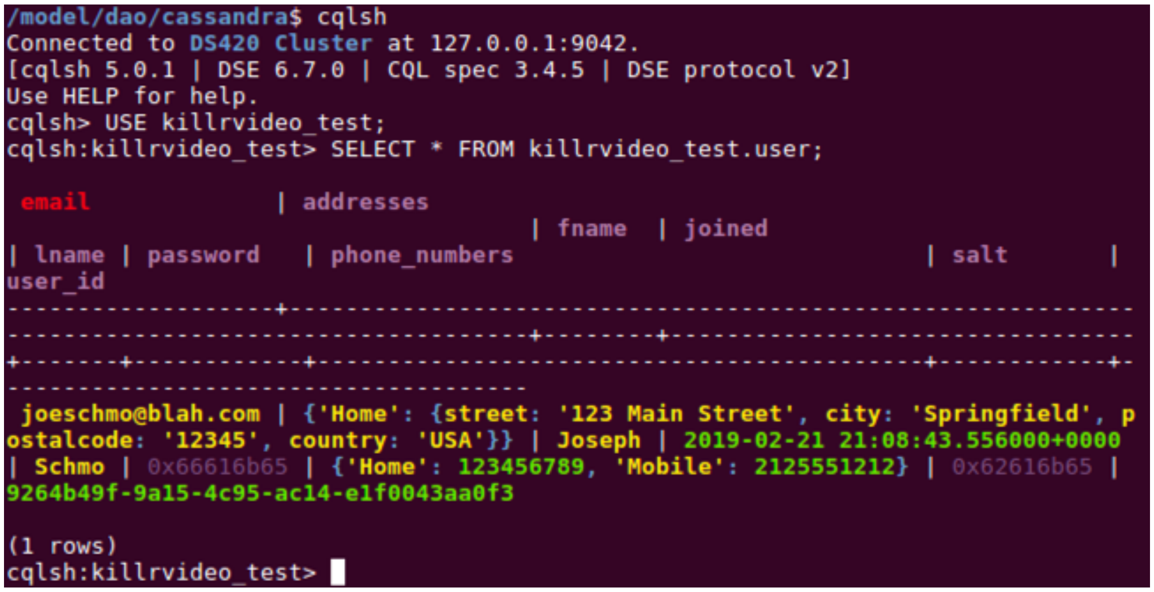
4. Add the following code to the TODO block:

```
        //TODO: Begin your code here

        CassandraUserDAO userDAO = new CassandraUserDAO();
        DseSession session = CassandraSession.getSession();
        userDAO.addUser(TestData.TEST_USER1);
        userDAO.addAddressToUser(TestData.TEST_USER1.getEmail(),
        "Home", TestData.TEST_ADDRESS1);

        //TODO: End your code here
```

Note the function simply adds *TEST_USER1* to the database and then updates that user's address.

5. Run the `TestAddUserAndUpdateAddress()` function by right-clicking the icon to the left of the function. Confirm it ran successfully.

6. Check the user table and confirm a user is present complete with addresses.



```
/model/dao/cassandra$ cqlsh
Connected to DS420 Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | DSE 6.7.0 | CQL spec 3.4.5 | DSE protocol v2]
Use HELP for help.
cqlsh> USE killrvideo_test;
cqlsh:killrvideo_test> SELECT * FROM killrvideo_test.user;

 email              | addresses
                                             | fname   | joined
| lname | password    | phone_numbers                                         | salt       |
user_id
-------------------+--------------------------------------------------------------------
-----------------------------------------+--------+----------------------------------
+-------+-----------+--------------------------------------------+-----------+-
-------------------------------------
 joeschmo@blah.com | {'Home': {street: '123 Main Street', city: 'Springfield', p
ostalcode: '12345', country: 'USA'}} | Joseph | 2019-02-21 21:08:43.556000+0000
| Schmo | 0x66616b65 | {'Home': 123456789, 'Mobile': 2125551212} | 0x62616b65 |
9264b49f-9a15-4c95-ac14-e1f0043aa0f3

(1 rows)
cqlsh:killrvideo_test>
```

7. Next, let's observe how data is read from DSE via the driver and the `getUser()` function. Comment out the following lines in `TestAddUserAndUpdateAddress()`:

```
// userDAO.addUser(TestData.TEST_USER1);
// userDAO.addAddressToUser(TestData.TEST_USER1.getEmail(), "Home",
        TestData.TEST_ADDRESS1);
```

8. Add the following line to read what you just wrote. It will fetch the email from the original test data of the user you just wrote to the database and pass it into the getUser() function.

```
User savedUser = userDAO.getUser(TestData.TEST_USER1.getEmail());
```

9. Put a break point on the closing bracket of the function and debug it. Explore the User object and note how an Address object is nested with the User object.