Nolan Evans, NTE0004

Project 3 Video: <https://youtu.be/I2Sr9ugwEc0>

Project 3 Objectives and Implementation:

1. RESTful Webserver:
   1. This webserver is extremely simple and can be created by running the WebServer application. The webpage can be opened on any browser with the url “localhost:8500”.
   2. On this page the user will see description on how to search for users, products, and orders.
      1. Graphical user interface, text, application, email

         Description automatically generated
   3. The user can query the database as described on the main page, an example of each is given.
      1. Graphical user interface, text, application

         Description automatically generated
         1. Clearly, showing a user’s password given their username is not a good idea, this part can be removed by commenting out lines 115 of the WebServer file.
      2. Graphical user interface, text, application, email

         Description automatically generated
         1. additional feature: “localhost:8500/products/all” shows all products in the database.
      3. Graphical user interface, text, application

         Description automatically generated
   4. Searching for data on the webpage:
      1. The queries work by taking two inputs that are added onto the main url, localhost:8500/input1/input2. Input1 tells the servers if it is a user, product, or order query, and input2 specifies what exactly to search for. It uses this data to make a query via the dataAccess interface. The dataAccess interface will use SQLiteDataAdapter to query the database. Input1 determines the table that will be searched and input2 will be the value to find in the table and return that objects information. This information will be formatted to html and the webpage will be generated.
2. Microservices Overview:
   1. Each microservice provides a unique service:
      1. Registration: When a microservice server is ran it will register itself with the service registry, essentially telling the registry its contact information (address and port) and the service that it provides.
      2. Deregistration: Once the microservice is called to fulfill a request from the service registry it will notify the registry that it needs to be de-registered, again sending its contact information and service type. This way the service won’t be called to fulfill a service while it actively serving another user.
      3. Service: The microservice’s service is unique, however after the service is performed it will send the requested data back to the service registry and then request to register with the service registry again since it is able to accept request again.
   2. The ServiceRegistry application keeps track of available services, and dispatches request for a service from users. This way, all available resources can be managed.
      1. Inflow of data: The user and microservices interact with the service registry via ServiceMessage objects (see ServiceMessageModel). The messages tell the registry whether the request is for the discovery of a server or publishing of a service (service registration) along with and data needed by the registry to fulfill the request. The data inside the message is a ServiceInfo object (see ServiceInfoModel). If it is a publish request the service info tells the registry the service it is providing and how to contact and find it later. If it is a discovery request the service info data tells the registry which microservice it needs to request information from and additional information that the microservice will need to use.
      2. Outflow of data:
         1. After the serviceRegistry has completed what it was requested it will respond to its client with a ServiceMessage object. This will tell the client if the service was/wasn’t successful along relevant information that depends on the request.
   3. The TestClient application, which acts as a user, makes a series of a request through the serviceRegistry that will test every microservice at least once.
      1. Functions called in TestClient:
         1. *updateProductPrice*(int productID, double newPrice);
            1. void, test ProductPriceUpdateService
         2. *updateProductQuantity*(int productID, double newQuantity);
            1. void, test ProductQuantityUpdateService
         3. *requestProduct*(int productID);
            1. Returns ProductModel object, test ProductInfoService
         4. *requestOrder*(orderID);
            1. Returns OrderModele object, test OrderInfoService
         5. *checkUserExist*(username, password);
            1. Returns boolean, test UserExistService
         6. *checkCanCancelOrder*(customer, orderID);
            1. Returns Boolean, test CanCancelOrderService
      2. Using the test client:
         1. The main function has 7 different parameters that are used in the test.
            1. Int productID is the productID that will be used by upDateProductPrice, updateProductQuantity, and requestProduct.
            2. Int orderID is the orderID used by requestOrder and checkCanCancelOrder.
            3. Double newPrice is the price used by updateProductPrice
            4. Double newQuantity is the quantity used by updateProductQuantity
            5. String username and String password are used by checkUserExist
            6. String customer is used by checkCanCancelOrder.
         2. Each of these parameters can be changed and the TestClient rerun to show different outcomes from the microservices.
3. Microservices – The majority of each microservice file is the same except for the serve method, which distinguishes the role of the service so that is the only part of the code I will focus on. The other distinguishing factors are in the register method where each microservice’s service code depends on what service it is, the same concept applies to deregistering. In all cases, data discovered by the microservice is returned to the service registry, which is then sent by the registry to the client.
   1. CanCancelOrderService
      1. This service receives a username and orderID from the service registry and uses a data access adapter to call the doOrdererMatch(username, orderID) function from SQLiteDataAdapter. This will query the database to check the order table for an entry with both the orderID and username. If one exist, it returns true; if not, false.
   2. OrderInfoService
      1. This service receives an orderID from the service registry and uses the data access adapter to call the loadOrder(orderID) function form SQLiteDataAdapter. This will query the database for an entry with the orderID and return an orderModele object. The microservice will then return this to the service registry as a json string.
   3. ProductInfoService
      1. Is the same as the OrderInfoService, except it uses a ProductID and uses the loadProduct(productID) function to return a ProductModel object.
   4. ProductPriceUpdateService
      1. This service receives productID and newPrice, it this uses the DataAccess interface to load a productModel object from the database using the productID. Then it changes that productModels price value to the new price, and again uses the DataAccess interface to save the updated ProductModel using the saveProduct(ProductModel product) function. Once completed it sends a message back to the service registry saying it has updated the product.
   5. ProductQuantityUpdateService
      1. This service acts the same as the ProductPriceUpdateService except with a newQuantity value instead of newPrice.
   6. UserExistService
      1. This service receives a username and password to check if it exists in the database. It uses the dataAccess interface to call the doUsernamePasswordExist(username, password) method which will query the database users table for an entry with the username and password. If one exists it will return true; if not, false.
4. Database Schema:
   1. CREATE TABLE "Order" (OrderID INTEGER PRIMARY KEY, OrderDate date, Customer CHAR(30), TotalCost DOUBLE, TotalTax DOUBLE)
   2. CREATE TABLE OrderLine
   3. (
   4. OrderID INTEGER NOT NULL,
   5. ProductID INTEGER NOT NULL,
   6. Quantity DOUBLE,
   7. Cost DOUBLE,
   8. FOREIGN KEY (OrderID) REFERENCES "Order"(OrderID),
   9. FOREIGN KEY (ProductID) REFERENCES Product(ProductID),
   10. PRIMARY KEY (ProductID, OrderID)
   11. )
   12. CREATE TABLE "Product"
   13. (
   14. ProductID INT NOT NULL,
   15. Name CHAR(30) NOT NULL,
   16. Price DOUBLE NOT NULL,
   17. Quantity DOUBLE
   18. )
   19. CREATE TABLE User
   20. (
   21. UserID INT PRIMARY KEY NOT NULL,
   22. UserName CHAR(30) NOT NULL,
   23. Password CHAR(30) NOT NULL,
   24. DisplayName CHAR(30),
   25. IsManager BOOL DEFAULT FALSE
   26. )
5. Testing:
   1. Testing was done through the TestClient, by simulating user’s request with specific data, I can see if the serviceRegistry, microservices, and SQLiteDataAdapter are working correctly by cross referencing their outputs with the actual data in the DB Browser for SQlite. I have also designed the outputs of each application such that is more clear of what point the application is at in its running (i.e. messages for when a server has been registered, is deregistering, accessing the database, etc.). The WebServer was tested with a similar concept, via running searches and making sure the data is correct.
6. Potential Problems:
   1. Throughout the testing of this project, the biggest issue has been with accessing the database in such a way that it will not be busy or closed when attempting to access. Along with successfully implementing a registration and de-registration strategy, closing each connection in the SQLiteDataAdapter has fixed all apparent issues with database access. Under the circumstances that my TestClient application runs there seem to be no issues, and is able to be re-ran without restarting all the microservices and service registry. However, I suspect that if multiple TestClients were ran at the same time, which would simulate a real-world environment, unforeseen issues may arise with accessing the database. But I have assumed that multi-threading is outside of the scope of this project and also just wanted to save myself the headache.
7. Running the Project:
   1. In order to run the microservices portion of the project, first run the ServiceRegistry, then all the microservices, and finally the TestClient.
   2. To use the webserver, only the webserver application should be ran.