**ONLINE MARKETPLACE**

**Introduction**

The client desires an online marketplace where they can sell goods (and possibly services) to customers geographically dispersed around the world. Think Amazon but on a smaller scale and budget. Their desire is to have a system that is constructed in a portable language (Java) and makes use of their existing network. The system itself should present a view for the customer to interact with as well as a view for the employees or administrators of the company to interface with. For the customer there is a need for them to be able to browse available products – this should present the customer with the type, description and price of the item with the options to add to their shopping cart. If the customer attempts to add a quantity of the item more than the current supply the system should prevent the customer from adding these and prompt them with a message on the availability of the item. The customer should be able to also purchase their items from the shopping cart. The administrators should be able to update an item’s description within the system, update its price, and update its quantity. The administrator should also be able to remove items from the system if so desired. Administrators should be able to add other administrators as well as add/remove customer accounts. A customer should be able to initially register for their account by themselves. The system should handle any faults or unexpected scenarios gracefully. It should be reliable and should allow for multiple customer requests during the course of execution.

**Requirements**

The project is to create an online marketplace using the MVC design pattern, Front Controller Pattern, Authorization Pattern and remote method invocation so that the marketplace can be accessed from different systems using the Java language. The online marketplace should have different views for customers and administrators. The system should be able to handle faults gracefully. It should also be reliable and allow multiple customer requests during the course of execution.

**Understanding**

I have identified following users from the description:-

-Customer

-Administrator/Employees

I have identified the following entities for the domain model from the description:-

-Marketplace

-Login View

-Marketplace Model

-Marketplace Server

-Marketplace Client

-Front Controller

-Dispatcher

-Admin View

-Customer View

-Item

-Shopping cart

-Customer

-Administrator

The customer will be able to perform following actions:-

-register account

-browse items

-add item to cart

-purchase item

-pay for the item

The administrator will be able to perform following actions:-

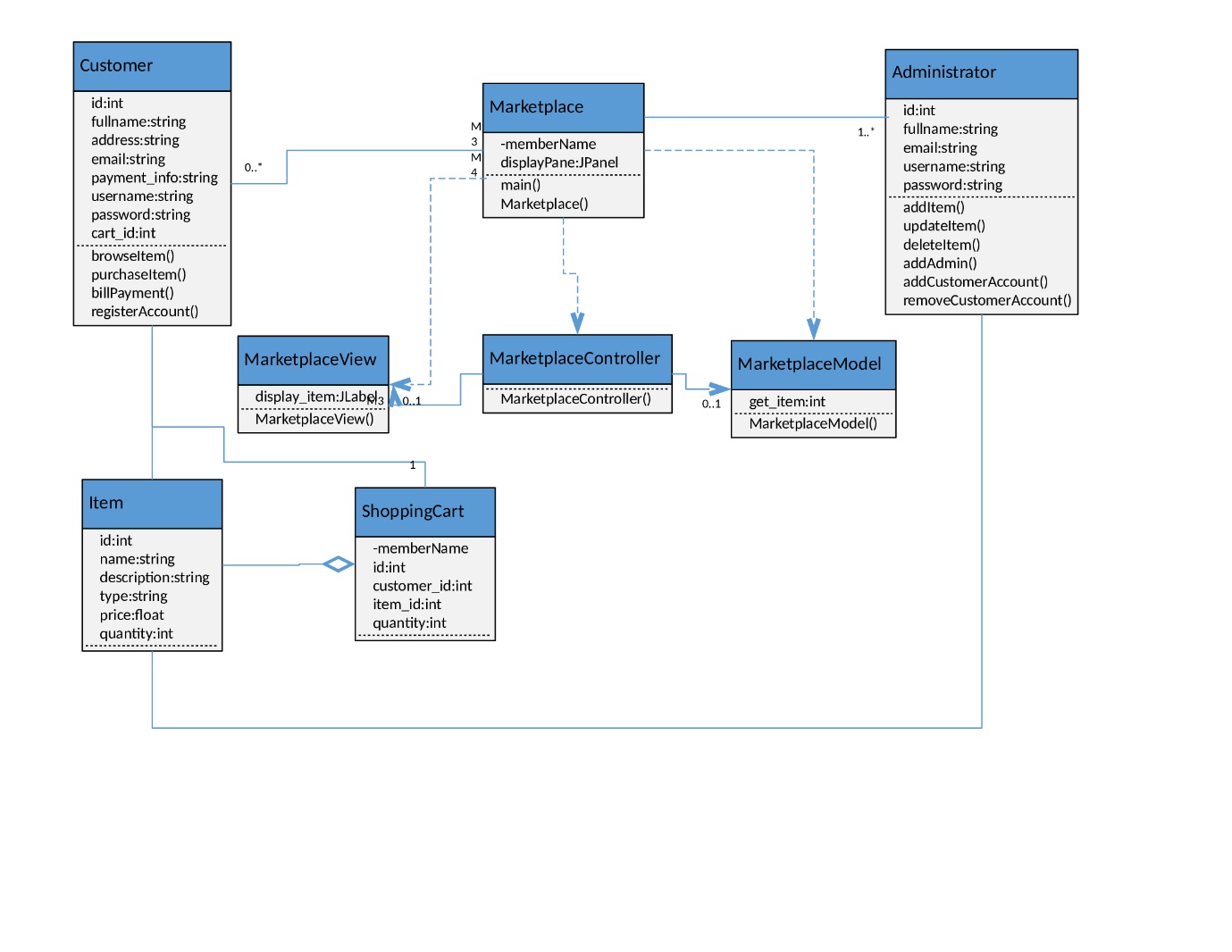
-add other administrator

-add/delete customer account

-add item

-update item attributes

-remove item



**Implementation of Java RMI and MVC**

The marketplace is implemented using MVC architecture pattern and Java RMI to connect from various systems.

Java RMI is implemented using the client-server model. To incorporate MVC architecture into that I have considered view on client side and model on server side. The RMI client is considered as client controller and RMI server as server controller. Also an interface is implemented for the RMI. So in total there are five classes to implement the RMI with MVC architecture. This will keep the model, view and controller separate from each other so that any change in one does not affect another.

To implement RMI, some libraries need to be imported. Below is a code snippet for the libraries imported in the server.

We need to bind the location of the server to the application using the rebind method. For this we import the Naming method of RMI. We do this using the following:

import java.rmi.Naming;

and using the rebind method to connect server location with the application. This is done as follows:

Naming.rebind(name, marketplace);

Where name is the location of the server and marketplace is the instance of MarketplaceServer class.

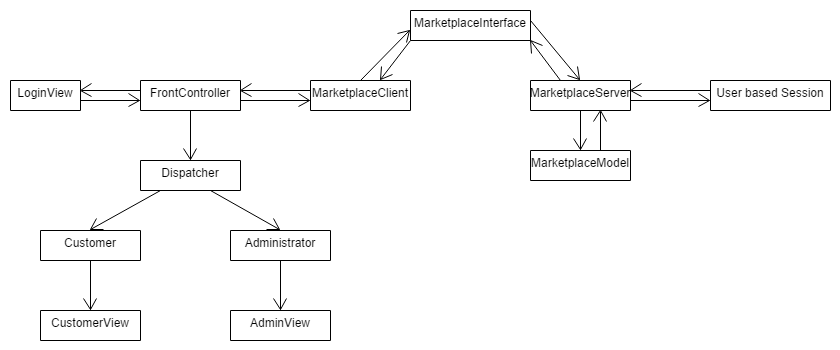
**Implementation of Front Controller and Authorization Pattern**

The users enter their username and password from Login view. This information is passed to front controller from where it is passed to client controller. The client controller establishes the remote connection to server controller and the information is passed to the model to confirm authorization and get role and name. If the authorization is confirmed a session is created which is sent over the network to reach front controller via client controller. The front controller then sends the details to dispatcher which then dispatches the request to appropriate view.

We want the session object to be serialized so we have implanted serializable in Session class as:

public class Session implements Serializable

The session will be created only if the user is authorized else null will be passed instead of session object over the network.



**Implementation of Role Based Authorization**

Role based authorization is a type of authorization pattern which uses reflection pattern and proxy pattern for implementation. In role based authorization the user access to a method is determined based on the user’s role. Since we are using Java RMI the RMI interface will work as the proxy for role based authorization. We create a server implementation class which will implement all the methods that are present in the interface that were previously present in the server itself. We also create authorization invocation handler which checks whether the user role allows access to a specific method.

We also need to create one annotation class which contains the annotation that has role type as an argument which will be used to match against the user role to provide access.

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

**public** **@interface** RequiresRole {

String value();

}

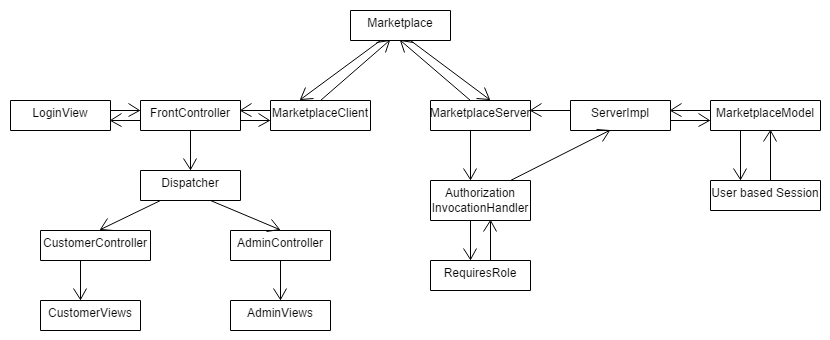
So all the methods that need to have access control will have annotation like:

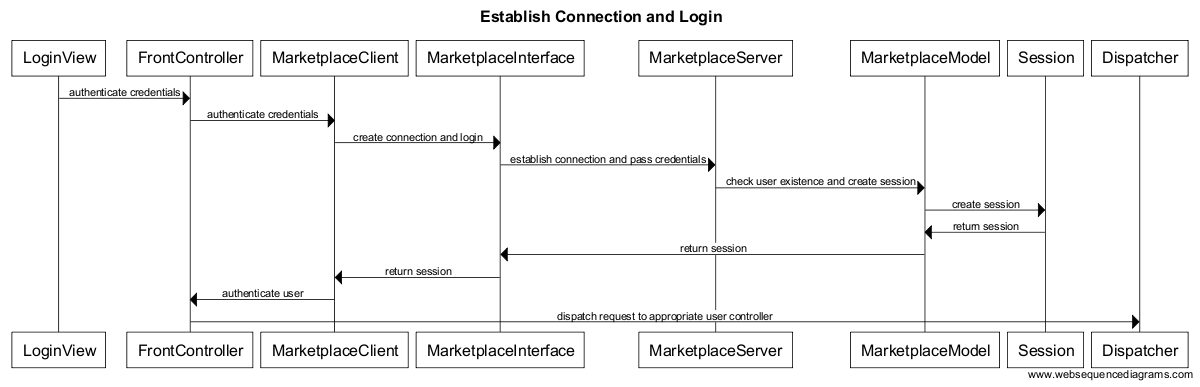
@RequiresRole("Customer")

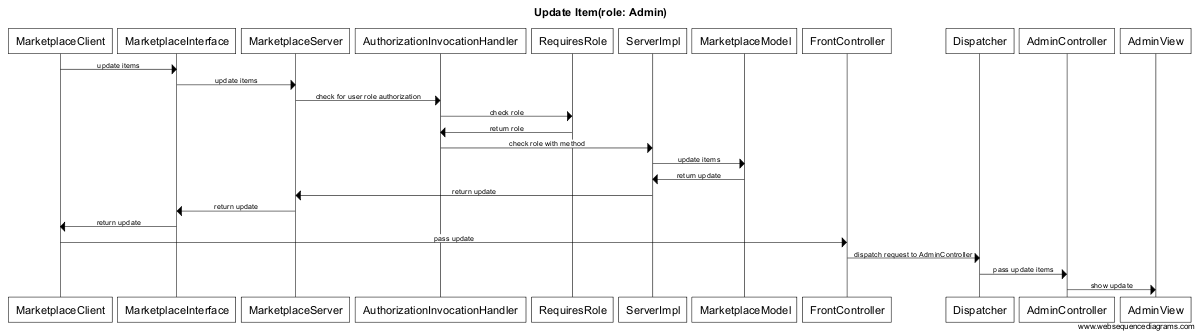
**public** String browseItem(String role);

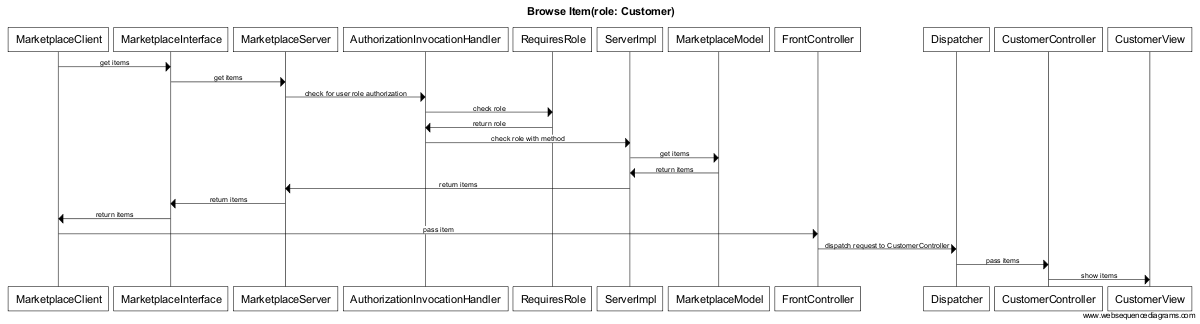
Previously where the server was calling the RMI interface it will now call the authorization invocation handler which will check if the user should be provided access to the method or not.

Marketplace assignment = (Marketplace) Proxy.newProxyInstance(Marketplace.**class**.getClassLoader(), **new** Class<?>[] {Marketplace.**class**},**new** AuthorizationInvocationHandler(**new** ServerImpl()));



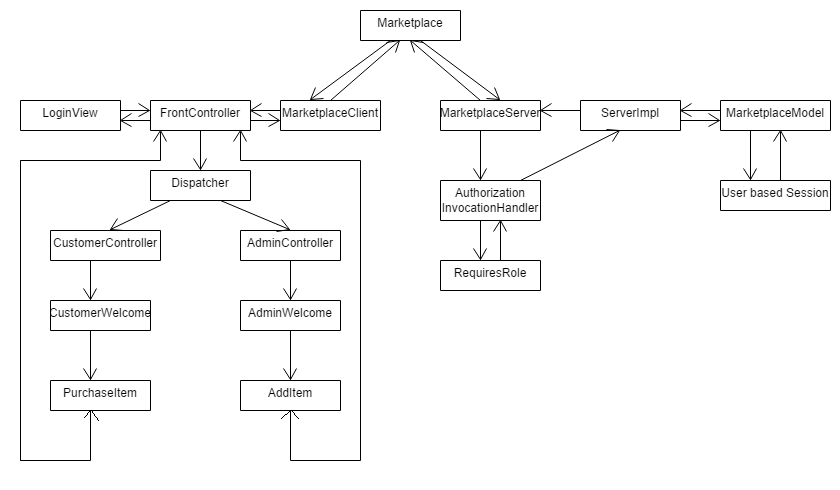


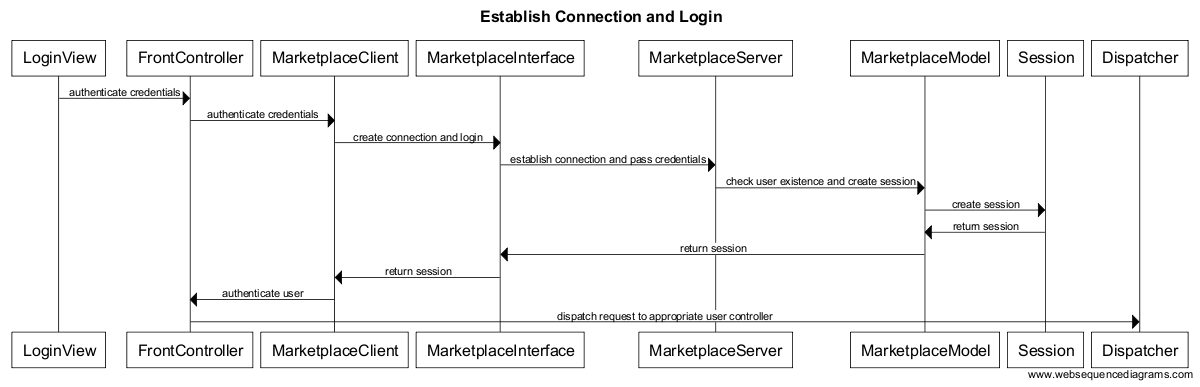


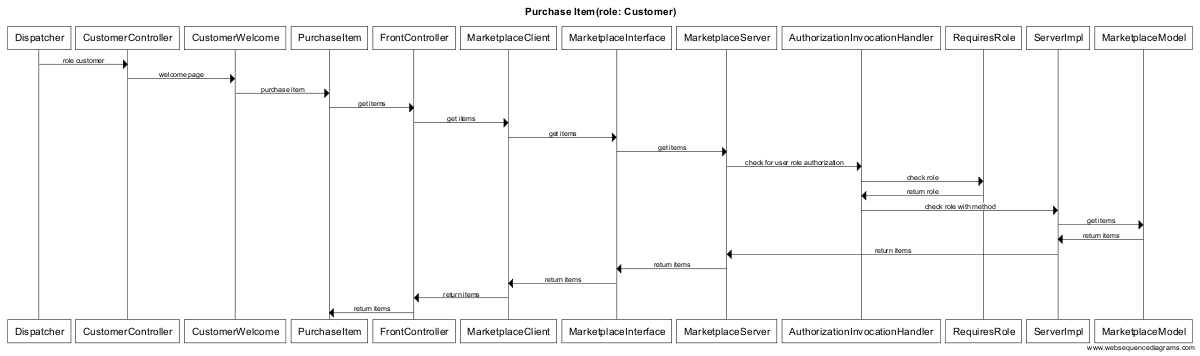


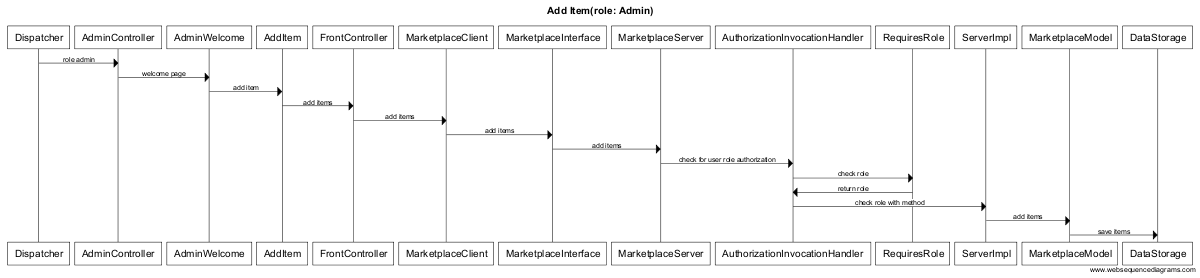
**Concurrency**

To implement addItem and purchaseItem methods I removed the previously added test methods of updateitem and browseItem. Since database is not available to keep track of items a text file is used.









RMI does not provide a guarantee for concurrency. It is assumed that concurrency is provided in RMI. To make it concurrent one solution can be to make all the methods synchronized. But this may lead to some unnecessary bottlenecks or deadlock.

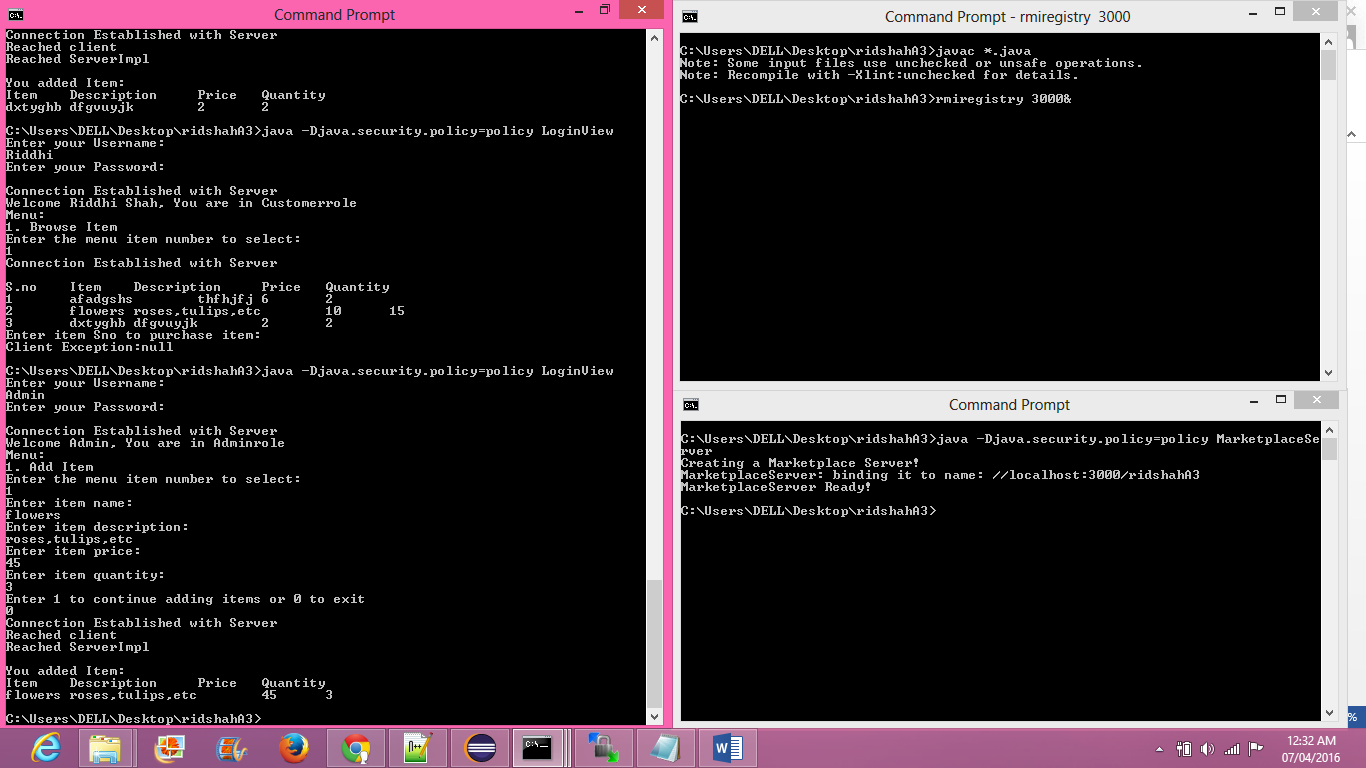
One problem that I found was that in a distributed system there is a possibility that different machines may have different versions of java installed. So if the java is compiled on a higher version and the other systems have lower version then there is a problem. Solution to this is making sure that java version is compatible in all machines.

Another potential problem is when only a single quantity of a product is left and two users are trying to access it at the same time. Currently my system allows both the users to purchase the product which is wrong. One solution can be to allow only one user to access the method while the other user needs to wait.

One other problem is when two admins try to add the same product with the same price and description at the same time then two different entries will be created instead of one. Solution can be to allow only one admin to go ahead and then check the other entry if the product already exists then prompt for confirmation. Another solution can be to allow only one admin login at any time.

We need to use synchronization on the methods to let only one client access the read or write in the file. This way multiple client writing to the same file can be prevented and we may not have the above mentioned issues.

**Sample Run**

****

**Using Synchronization Patterns**

As identified in the last assignment there are several problems when using a concurrent system where multiple clients are trying to access the same method. To solve these problems we can use the synchronization patterns. There are mainly 5 synchronization patterns:

-Monitor Object

-Guarded Suspension

-Scoped Locking

-Future

-Thread-Safe Interface

**Monitor Object**

Monitor Object pattern synchronized the execution of method to ensure that only one method runs within an object at a time. This allows scheduling execution sequence. There is no polling involved in this method. This method is implemented using the synchronized keyword in java. The synchronized method takes care of this pattern by itself. This can be implemented either at method level or statement level.

Method level use of synchronization:

public synchronized void removeItem(List<String[]> items){

List<String[]> catalog = new ArrayList <String[]>();

catalog.addAll(items);

removeProducts(catalog);

catalog.clear();

}

As can be seen in the above example, we are using the synchronized keyword at the method level. This will put the whole method in the critical section. It automatically establishes a “happens-before” relationship with any subsequent invocation of synchronized method for the same object. This guarantees the change to the state of the object are visible to all threads. It is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block execution until the first thread is done with the object. This may sometimes create unnecessary overhead in the situations where there is no need for the other thread to wait for the completion of execution of the other thread. One solution to this is using the statement level synchronization.

Statement level use of synchronization:

public void removeItem(List<String[]> items){

List<String[]> catalog = new ArrayList <String[]>();

catalog.addAll(items);

synchronized(this){

removeProducts(catalog);

}

catalog.clear();

}

Unlike method level synchronization we have to specify the object that provides the lock. This provides finer synchronization than the synchronized method. This helps synchronizing only the potion of the method that actually needs to be synchronized leaving the other portion out of the critical section.

If the entire method is part of the critical section then there is no difference between using the method level or statement level synchronization. If the method has parts that don’t need to be in the synchronized block then statement level synchronization is more effective to use. The disadvantage to this is that the more the synchronized blocks, the less overall parallelism we get, so it is better to keep the synchronized statement level to the minimum.

Monitor object pattern uses two other patterns: guarded suspension and scoped locking.

Guarded suspension:

We often put conditions on when to execute a particular method i.e. we put preconditions to apply on component of method. We don’t want to abort the method execution if condition is not met right away. So we need a mechanism to wait until the condition is met. Blocking queue is used to ensure that access is provided to the method only if the condition is met otherwise it was blocked. We don’t want to right away block the method execution so the solution is to use guarded suspension. In guarded suspension instead of aborting the method, its client thread is suspended so that the other client threads can still access the shared component safely and change the state of the method’s guard condition. When the state will change for the suspended client thread it will be resumed so that it can continue to execute the execution of the interrupted method. The called method encapsulates the policy and mechanism for access, leaving the calling code free of clutter and potentially duplicated mechanism. While using this pattern it is important to consider the time period of waiting for the thread it is important that the waiting period is finite and reasonable otherwise the thread will keep waiting infinitely which creates overhead for the system. Mostly this pattern is used when we want the thread to sleep for a particular amount of time until the condition. This is also used in the future pattern which is discussed later.

Scoped locking:

Scoped locking ensures that the lock is acquired when the control enters the scope and the lock is released automatically when control leaves the scope. Sometimes after the lock is acquired on a section it is not released after the execution either because the release statement for forgotten or the execution was terminated due to an exception and the lock was not released. This is a major leak issue. So it is important to make sure that the lock is released after the execution is completed. To ensure this a guard class is defined that acquires and releases lock automatically in the method or blocked scope. The constructor of the guard class stores a pointer or reference to the lock and then acquires the lock before the critical section is entered. The destructor of this class uses the pointer or reference stored by the constructor to release the lock automatically when leaving the scope of the critical section. All this handled by the “synchronized” feature in java. The advantage to this is increased robustness and less maintenance effort. One thing to take care in using this deadlock prevention. Care should be taken such that the recursive synchronized blocks do not create deadlock.

**Future Pattern**

When we call a method we need to wait until we get the return of the method to continue execution. There are times when we make the method call but we don’t immediately need the return, we want to continue execution of other statements until we need to use the return object of the method to move forward. In such cases time is wasted in waiting for the call to return while we could have continued the execution of other statements that are not using the return of the method. In such scenarios we can use the future pattern.

private static final ExecutorService threadpool =Executors.newFixedThreadPool(1);

DatabaseConnection task = new DatabaseConnection ();

Future<Connection> future = threadpool.submit(task);

while (!future.isDone()) {

System.out.println("Waiting for database connection....");

Thread.sleep(5);

}

Connection conn = (Connection) future.get();

@Override

public Connection call()

{

Connection conn = null;

try {

  conn =  dbconnect();

}

catch (InterruptedException ex)

{

System.out.println(ex);

  }

return conn;

 }

public Connection dbConnect() throws InterruptedException {

Connection conn = null;

String hostname = "in-csci-rrpc01.cs.iupui.edu:3306";

String dbName = "[tablename]\_db";

String url = "jdbc:mysql://" + hostname + "/" + dbName;

String username = "[username]";

String password = "[password]";

System.out.println("Connecting database...");

try

{

conn = (Connection) DriverManager.getConnection(url, username, password);

System.out.println("Database connected!");

}

catch (SQLException e)

{

throw new IllegalStateException("Cannot connect the database!",e);

}

return conn;

}

Connecting to the database takes time. Until the connection is obtained we may want to continue with the execution of other statement that do not need the connection object.

We submit a future task to the threadpool using the

Future<Connection> future = threadpool.submit(task);

Task is an instance of the connection class.

We have the call method which implements the database connection and returns the connection when it is established.

public Connection call()

{

Connection conn = null;

try {

  conn =  dbconnect();

}

catch (InterruptedException ex)

{

System.out.println(ex);

  }

return conn;

 }

Until the connection is made we have the future object that was returned immediately when we submitted the task so we can continue the execution.

We keep check if the actual connection is ready using the while loop. When the connection is ready we use the future.get() to get the result. This is similar to guarded suspension discussed before. Thread is put to sleep for some time instead of aborting the thread execution.

while (!future.isDone()) {

System.out.println("Waiting for database connection....");

Thread.sleep(5);

}

When the connection is ready we use the get method to get the connection.

Connection conn = (Connection) future.get();

We throw the Interrupted Exception if there is an exception in the thread execution. Thus we do not have to wait until the connection is established to continue with the execution.

This pattern is not useful if the return value is needed immediately because the method will have to wait until the actual value is returned. This pattern is useful only if there is something to execute between the time the call is made and the value is returned. In my implementation of marketplace I need the connection immediately to execute the query so using future pattern is not useful so I have not used the future pattern.

**Thread-Safe Interface Pattern**

As discussed in monitor object pattern there is a problem of self-deadlocking. This can be prevented using thread-safe interface pattern. Components in a concurrent program need to be thread-safe. Methods acquire locks to protect against concurrent access. This results in self-deadlock. As a solution to this problem we create one public method which has the private methods implementation. The interface will acquire the lock calling its implementation methods and releases the lock. The private method will not worry about the lock will do the implementation. The interface will release the lock after the implementation method is returned and will give the control over to the caller. This will ensure that the intra component method calls have no self-deadlocking. Also the locks will not be acquired unnecessarily. All this will simplify the implementation process increasing the robustness and performance but it will also cause additional indirection and extra methods which will lead to overhead.

public synchronized String purchaseItem(int item\_index, int item\_quan){

String message = null;

try{

Connection conn = dbConnect();

int item\_quantity = selectProduct(conn, item\_index);

item\_quantity=item\_quantity-item\_quan;

if(item\_quantity >= 0){

updateQuantity(conn, item\_quantity, item\_index);

message = "\nProduct purchase successful!!";

}

else{

message = "\nSorry not enough product left!!";

}

conn.close();

}

catch(SQLException e)

{

System.out.println(e);

}

return message;

}

private int selectProduct(Connection conn, int item\_index){

DatabaseConnect db = new DatabaseConnect();

int item\_quantity = db.selectProduct(conn, item\_index);

return item\_quantity;

}

private void updateQuantity(Connection conn, int item\_quantity, int item\_index){

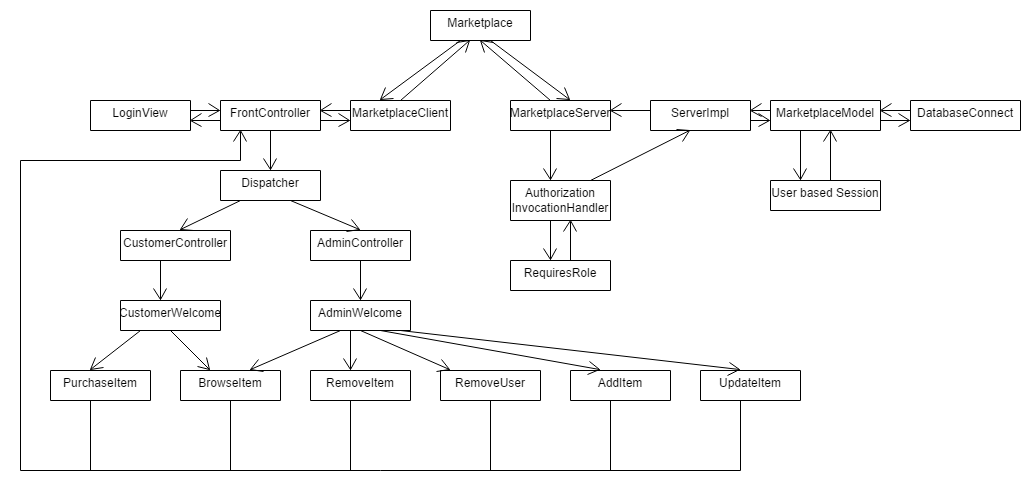
DatabaseConnect db = new DatabaseConnect();

db.updateQuantity(conn, item\_quantity, item\_index);

}

The purchaseItem is a public method which calls two private methods selectProduct and updateQuantity. The synchronization is put on the purchaseItem. It will acquire the lock and implement the private methods. The private methods will not worry about the acquiring locks and perform the execution. The lock will be released by the purchaseItem when both the private methods have completed their execution.

**All Function Implementation**



All the functions from the product requirement are now implemented.

Customer Role:

-Login

-BrowseItem

-PurchaseItem

Admin Role

-Login

-BrowseItem

-AddItem

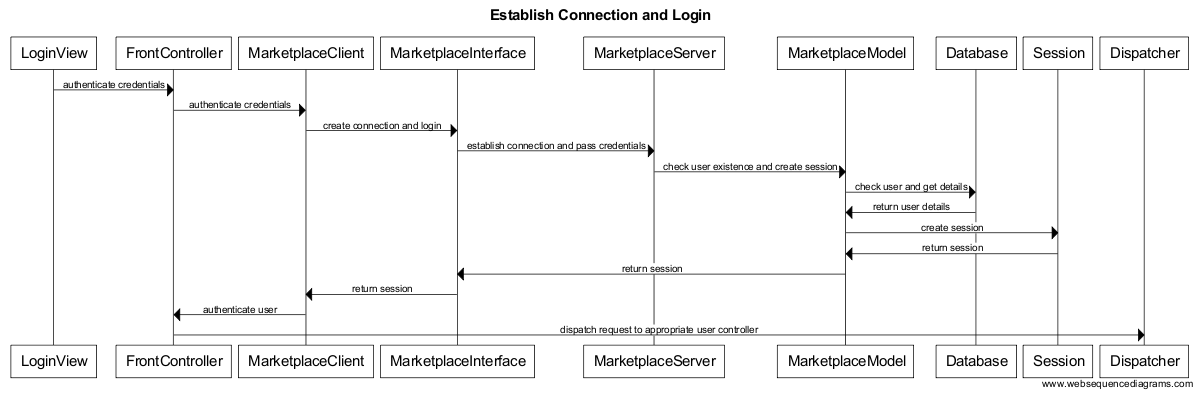
-UpdateItem

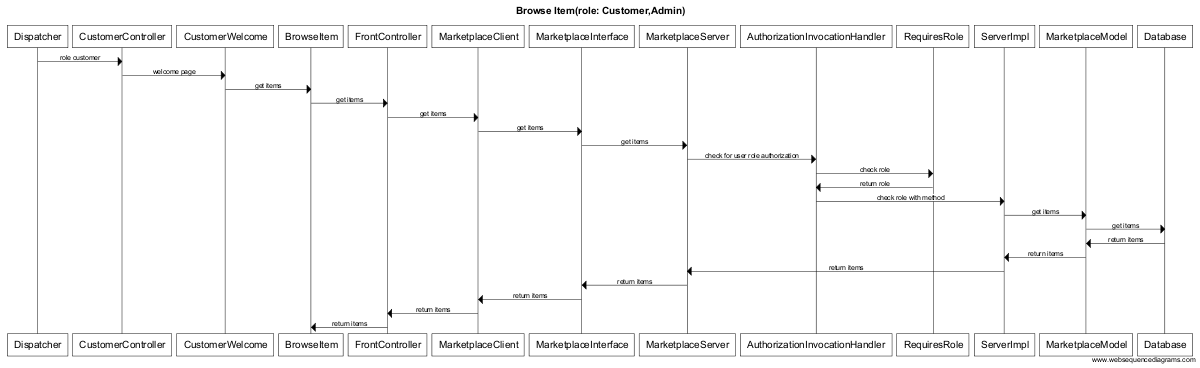
-RemoveItem

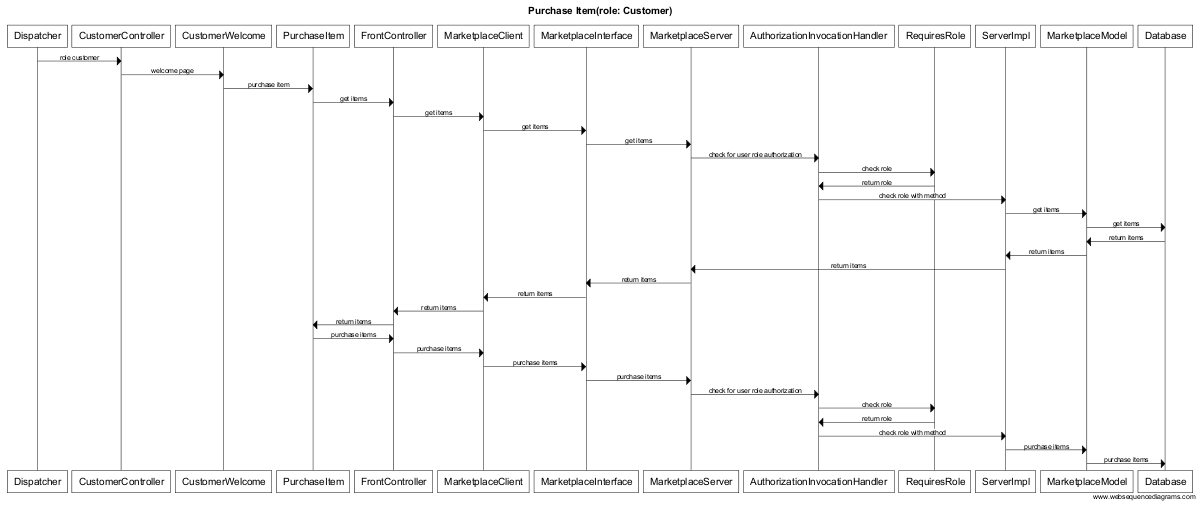
-RemoveUser

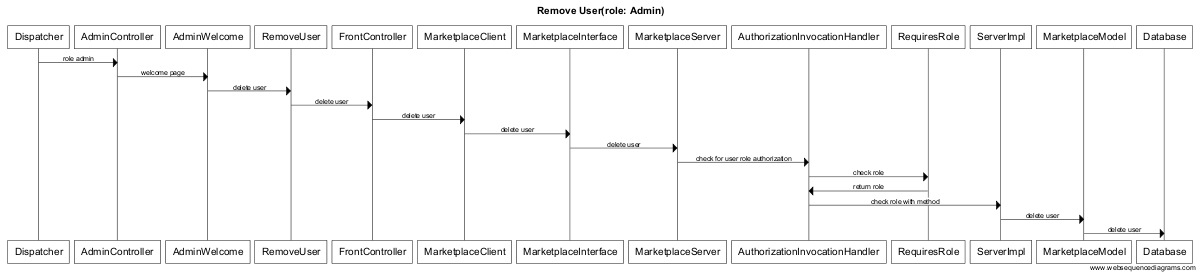
DatabaseConnect is the class used to implement all the queries required to implement the functions.

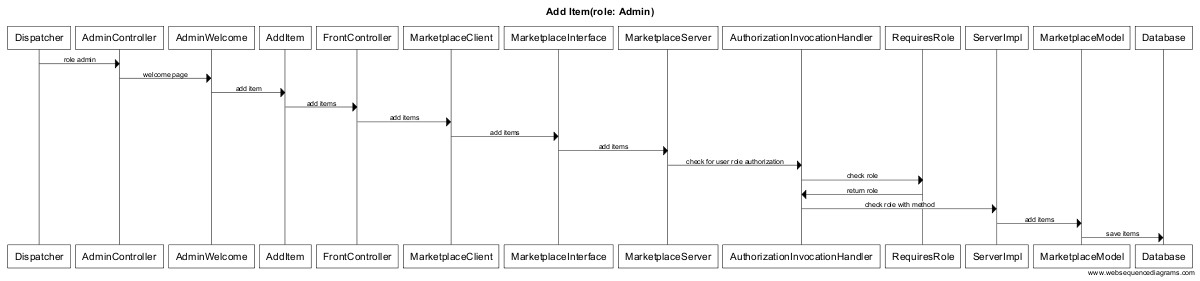
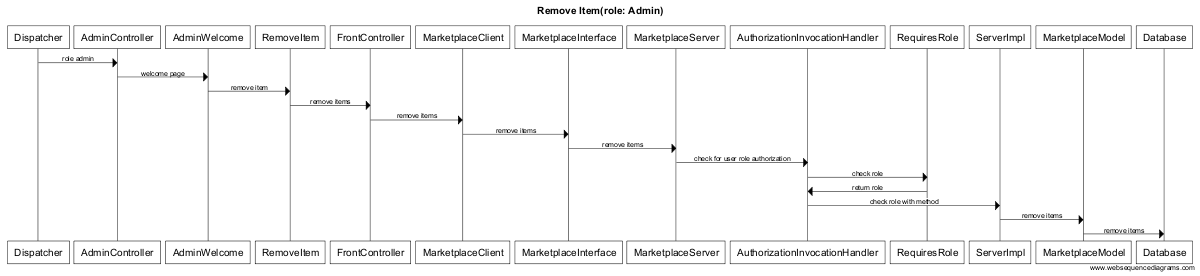
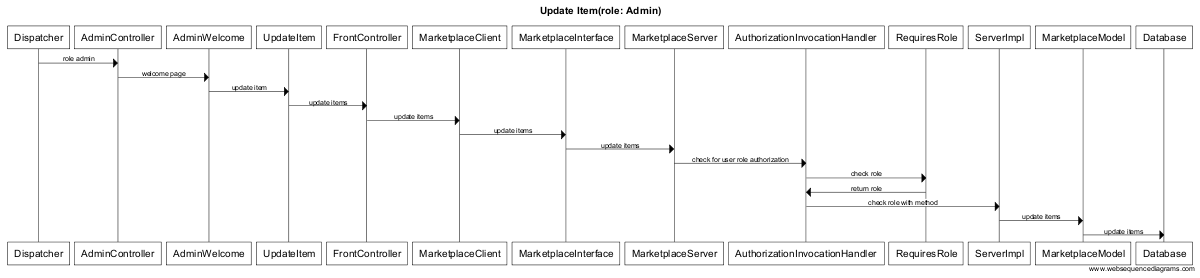
**Sequence Diagrams**



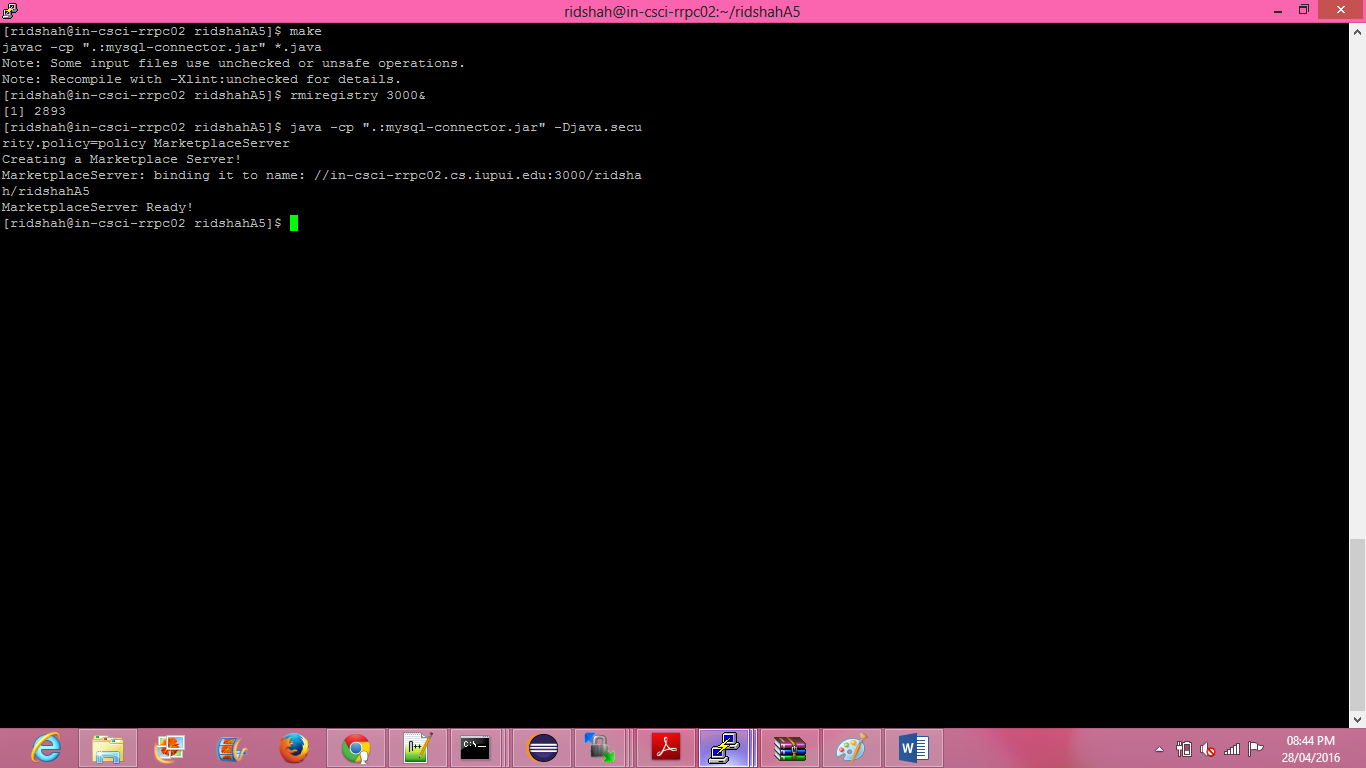


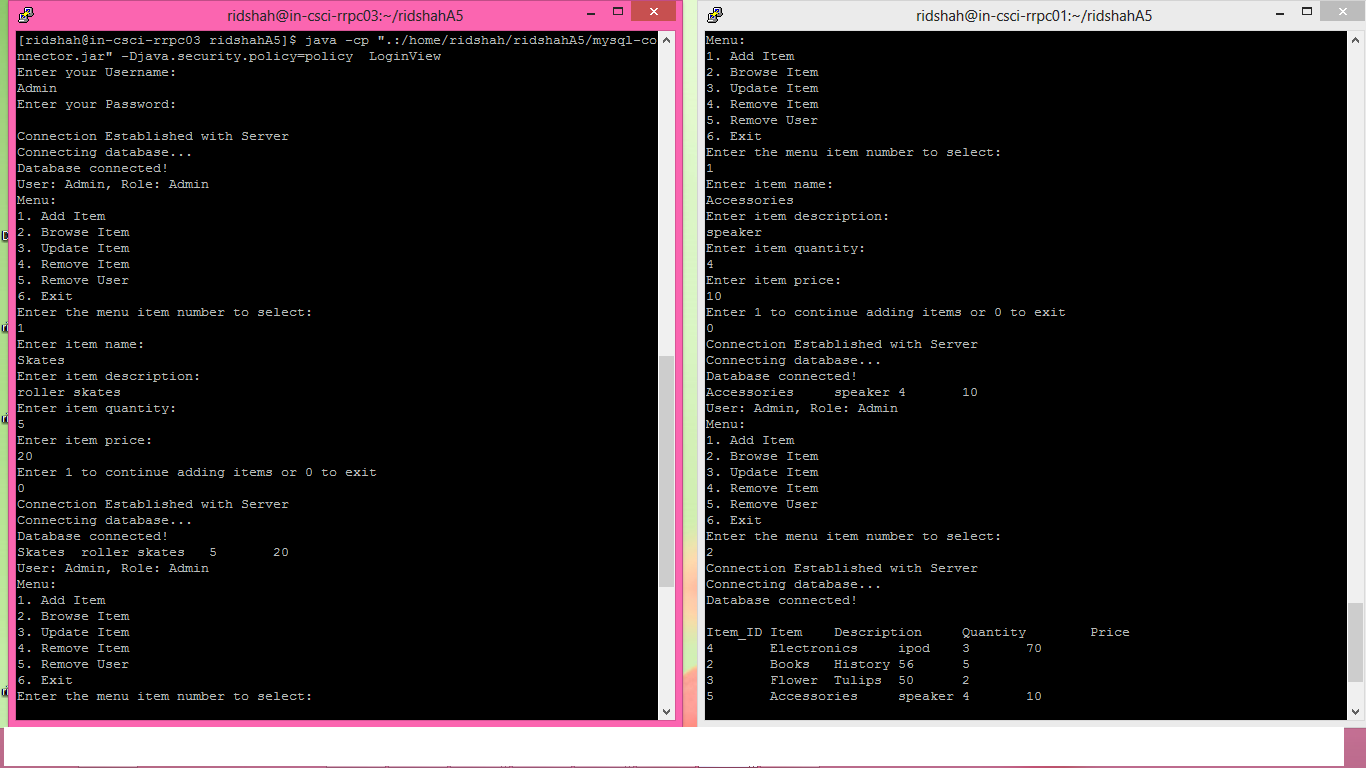


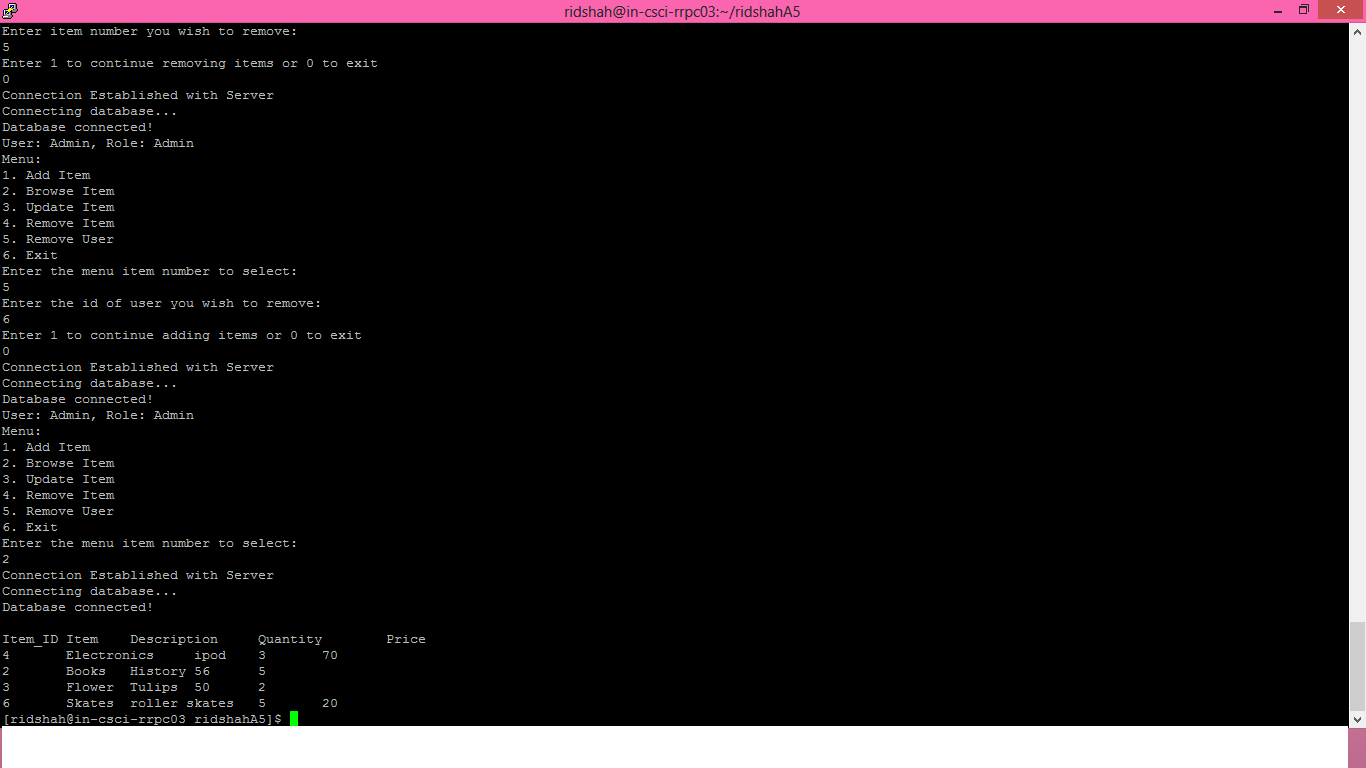


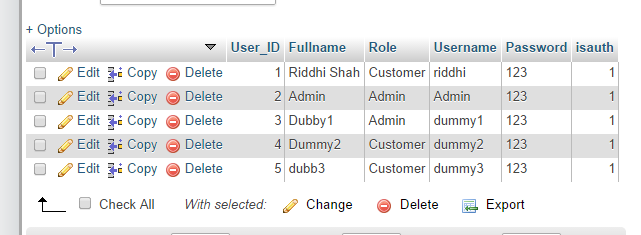


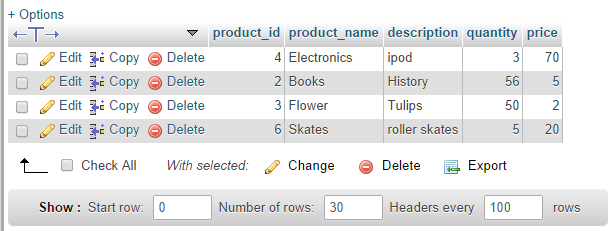
**Sample Run**

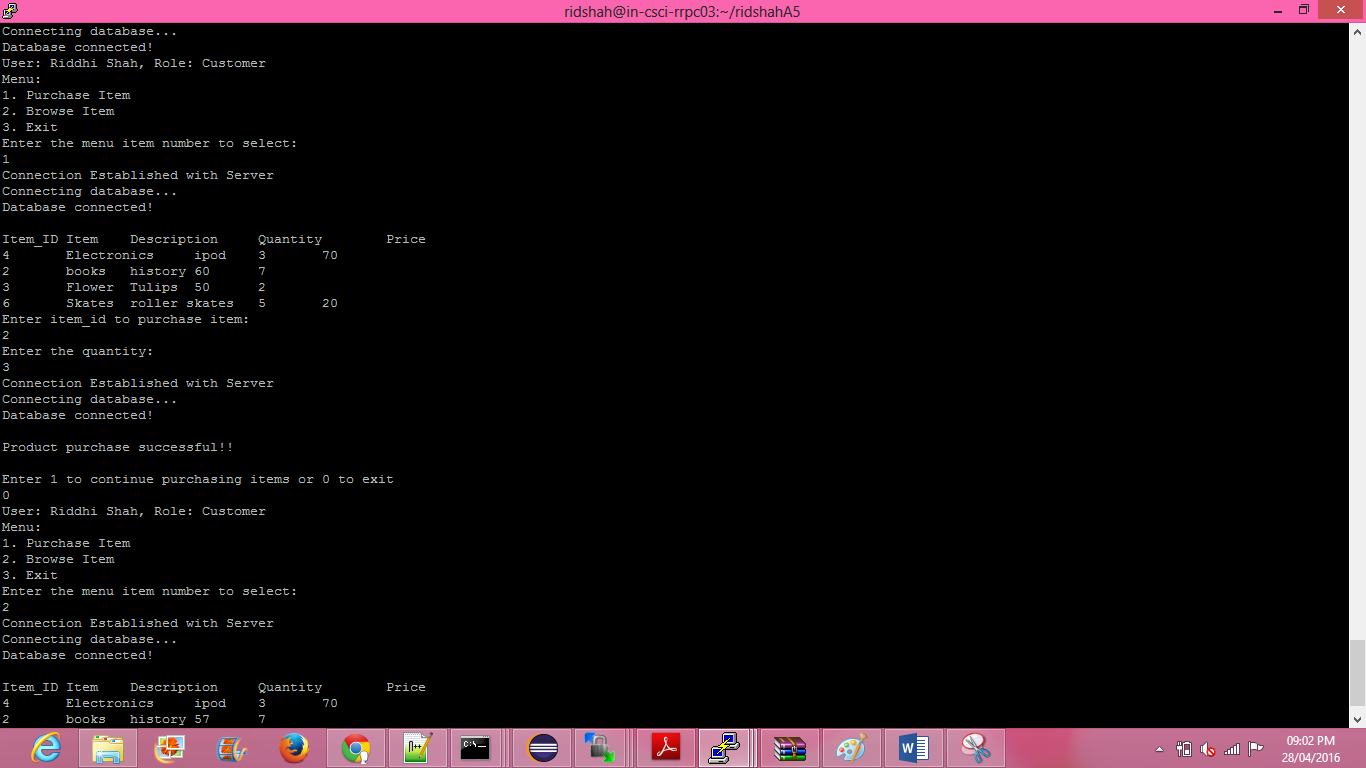
****

****

****

****

****

****

**Summary**

In this application we used different design patterns to make the application design scalable. For the client server architecture we used Java RMI (Remote method invocation). RMI provided an interface to connect the client to the server.

Next we decided to use the MVC architecture for the application. MVC stands for Model View Controller architecture. This helps us maintain a decoupled system. Separating the page views from the method implementation helps keep the client side light weight. To combine MVC with the RMI, I have considered view on client side and model on server side. The RMI client is considered as client controller and RMI server as server controller. Also an interface is implemented for the RMI. This ensures that if any changes are made in the view will not affect the implementation of that method. This is similar to keeping the business logic separate from the presentation layer.

Next we implemented front controller pattern. Front controller pattern provides a centralized entry point for handling server request calls. We implemented dispatcher as an interface between front controller and other admin/customer controller. All the requests from the views will go to client controller only through the front controller. Front controller will receive the return from the server and pass to dispatcher which will then dispatch the request to appropriate user controller. Using front controller it becomes easy to control the navigation of the system.

Next we implemented authorization pattern- role based authorization. Role based authorization guarantees that a method will be accessed by only the user who has the permission to access that method. No other user can access it. This provides security against malicious users. Authorization invocation handler is used for this purpose. Authorization invocation handler which checks whether the user role allows access to a specific method. RMI will be used as an interface for this.

Next we implemented synchronization patterns. Marketplace application will be accessed by multiple clients on a single server. This makes the necessity to make sure that all the calls performed by different clients do not override each other. All the clients’ request should be performed such that there is only that client. To ensure that it is important to implement synchronization patterns. In my application as per my design I have used thread-safe synchronization and monitor object synchronization. There was no need for future pattern in my application so it was not implemented.

The implementation of these patterns have helped make the system easily scalable and easy to test. Changes in view will not affect the implementation. Even in a distributed environment where there are multiple client calls the application will be easily able to handle it. The implementation on client side is light weight. All the processing is done on the server side making it easy at the presentation layer. The system has less coupling and a fair amount of cohesion. From the security point of view also the application will not provide access to unauthorized users and will not let user access methods they are not allowed to access. This case study has also help helped understand when and how to use design pattern and most importantly when not to use what pattern.