Task 1

- **How did you use connection pooling?**
  We updated our servlets as per the instructions included in project 5 task 1. JDBC connection pooling is enabled in much the same way as the example which is by modifying the content.xml and web.xml files as well as the appropriate Java files. The Java servlets are found in the src folder and only files that connect to the database were modified to accommodate connection pooling. For the most part, the process of switching from a direct connection to pooling is the same across the servlets in that we commented out our variables storing the login credentials, creation of an instance of the JDBC driver, and establishment of a connection.

  - **File name, line numbers as in Github**
    An example of the changes can be found in src/Search.java in lines 48 to 74. This shows how we commented out the parts that performed a direct connection and added the code displayed above and found in the instructions to use pooling. Similar changes can be found throughout the src folder containing the Java files and servlets. The link is https://github.com/UCI-Chenli-teaching/cs122b-winter18-team-35/commit/c0256c738cf34900527f7125e3cc8e671c7b512c.

  - **Snapshots**

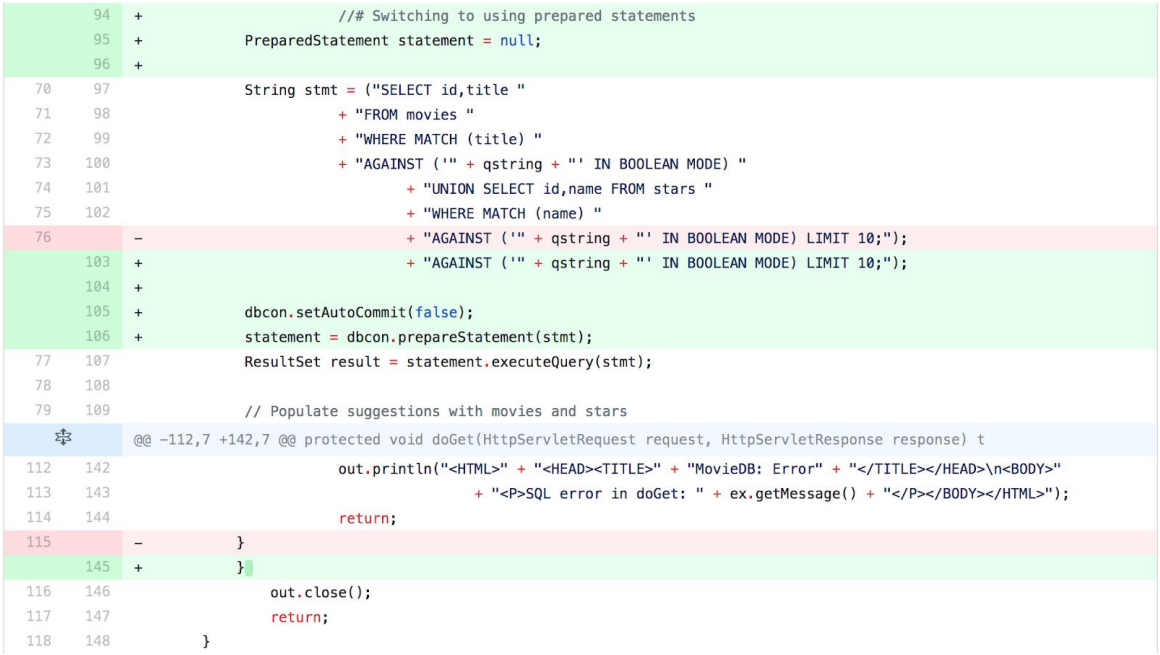
```
+               PrintWriter out = response.getWriter();
+               response.setContentType("application/json");
+
+               /*
                String loginUser = "testuser";
                String loginPasswd = "password";
                String loginUrl = "jdbc:mysql://localhost:3306/moviedb";
+               */

-               response.setContentType("application/json");
-
-               PrintWriter out = response.getWriter();
                try {
-           Class.forName("com.mysql.jdbc.Driver").newInstance();
-           Connection dbcon = DriverManager.getConnection(loginUrl, loginUser, loginPasswd);
-           Statement statement = dbcon.createStatement();
+                       Context initCtx = new InitialContext();
+                       if(initCtx == null)
+                               out.println("initCtx is NULL");
+
+                       Context envCtx = (Context) initCtx.lookup("java:comp/env");
+                       if(envCtx == null)
+                               out.println("envCtx is NULL");
+
+                       DataSource ds = (DataSource) envCtx.lookup("jdbc/MovieDB");
+
+                       //# Removing direct connections to use pooling
+           //Class.forName("com.mysql.jdbc.Driver").newInstance();
+           //Connection dbcon = DriverManager.getConnection(loginUrl, loginUser, loginPasswd);
+                       //Statement statement = dbcon.createStatement();
+
+                       if(ds == null)
+                               out.println("ds is NULL.");
+
+                       Connection dbcon = ds.getConnection();
+                       if(dbcon == null)
+                               out.println("dbcon is NULL.");
```

- **How did you use Prepared Statements?**
  For implementing prepared statements, we modified the servlets involved in the search. While we attempted to use the prepared statement format of parameterization, we encountered issues with the MATCH AGAINST statements due to the syntax which generated SQL syntax errors. Since the statements do not change during the time of the query and only at the beginning, we defined the statement first without the use of prepared statement parameters and then created it into a prepared statement. The strings were completed prior to becoming prepared statement objects. This prevented the error while satisfying the requirement of making the queries into prepared statements.

  - **File name, line numbers as in Github**
    Lines 94 to 106 in file Search.java show the transition to using prepared statements and much like connection pooling, similar changes can be found in other files. In particular, ModifiedShowSearch.java and ShowSearch.java use prepared statements.
    (Search.java; lines 94-106, ModifiedShowSearch.java; lines 102-126, ShowSearch.java; lines 93-151)

  - **Snapshots**



```
94  +                       //# Switching to using prepared statements
95  +           PreparedStatement statement = null;
96  +
97              String stmt = ("SELECT id,title "
98                          + "FROM movies "
99                          + "WHERE MATCH (title) "
100                         + "AGAINST ('" + qstring + "' IN BOOLEAN MODE) "
101                                 + "UNION SELECT id,name FROM stars "
102                                 + "WHERE MATCH (name) "
    -                               + "AGAINST ('" + qstring + "' IN BOOLEAN MODE) LIMIT 10;");
103 +                               + "AGAINST ('" + qstring + "' IN BOOLEAN MODE) LIMIT 10;");
104 +
105 +           dbcon.setAutoCommit(false);
106 +           statement = dbcon.prepareStatement(stmt);
107             ResultSet result = statement.executeQuery(stmt);
108
109             // Populate suggestions with movies and stars
@@ -112,7 +142,7 @@ protected void doGet(HttpServletRequest request, HttpServletResponse response) t
142                         out.println("<HTML>" + "<HEAD><TITLE>" + "MovieDB: Error" + "</TITLE></HEAD>\n<BODY>"
143                                 + "<P>SQL error in doGet: " + ex.getMessage() + "</P></BODY></HTML>");
144                         return;
    -           }
145 +           }
146                 out.close();
147                 return;
148         }
```

Task 2

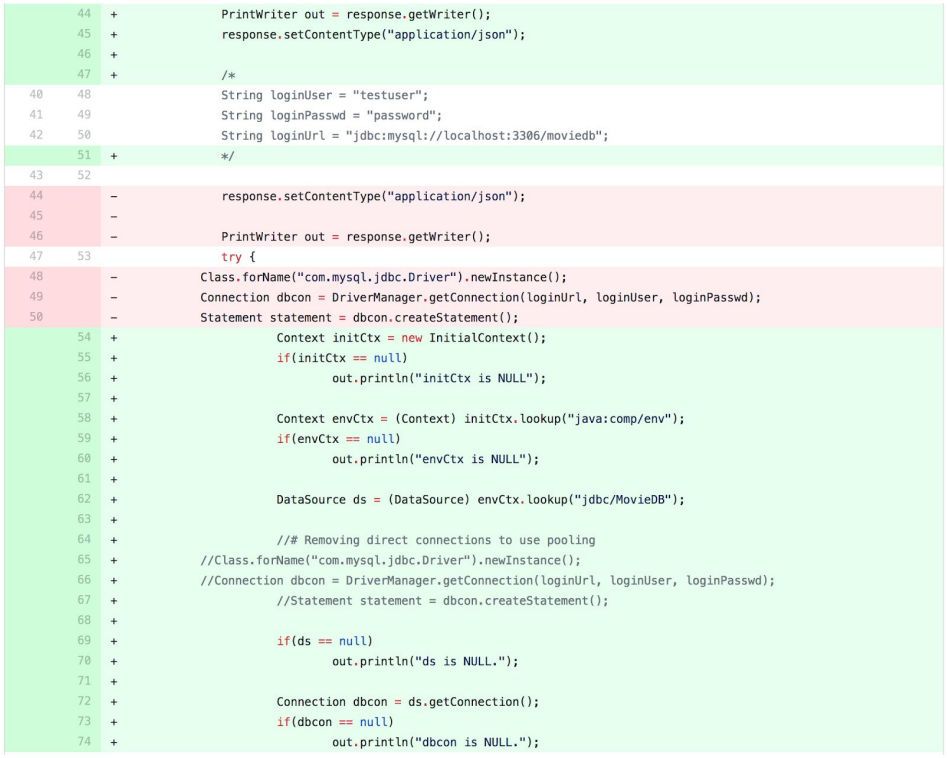
- **Address of AWS and Google instances**
  Load Balancer: 18.222.13.85
  Master Instance: 18.218.173.114
  Slave Instance: 18.188.88.199
  Google Instance: 104.198.103.206

- **Have you verified that they are accessible? Does Fablix site get opened both on Google's 80 port and AWS' 8080 port?**
  Yes, we have verified and the Fablix site does work on both Google and AWS ports.

- **How connection pooling works with two backend SQL?**
  Connection pooling with two backend servers divides the work by distributing the connections across the different servers. The effective load for reads in our case is doubled because both master and slave instances can handle them while the writes remain with the master instance.

- **File name, line numbers as in Github**
  This may be visible in src/Search in lines 48 to 74 as well because that file implements connection pooling and observing the logs will show that the search is divided between the instances per connection. Almost all the other files found in src also implement connection pooling because they use JDBC.

- **Snapshots**



```
+               PrintWriter out = response.getWriter();
+               response.setContentType("application/json");
+
+               /*
                String loginUser = "testuser";
                String loginPasswd = "password";
                String loginUrl = "jdbc:mysql://localhost:3306/moviedb";
+               */

-               response.setContentType("application/json");
-
-               PrintWriter out = response.getWriter();
                try {
-           Class.forName("com.mysql.jdbc.Driver").newInstance();
-           Connection dbcon = DriverManager.getConnection(loginUrl, loginUser, loginPasswd);
-           Statement statement = dbcon.createStatement();
+                       Context initCtx = new InitialContext();
+                       if(initCtx == null)
+                               out.println("initCtx is NULL");
+
+                       Context envCtx = (Context) initCtx.lookup("java:comp/env");
+                       if(envCtx == null)
+                               out.println("envCtx is NULL");
+
+                       DataSource ds = (DataSource) envCtx.lookup("jdbc/MovieDB");
+
+                       //# Removing direct connections to use pooling
+           //Class.forName("com.mysql.jdbc.Driver").newInstance();
+           //Connection dbcon = DriverManager.getConnection(loginUrl, loginUser, loginPasswd);
+                       //Statement statement = dbcon.createStatement();
+
+                       if(ds == null)
+                               out.println("ds is NULL.");
+
+                       Connection dbcon = ds.getConnection();
+                       if(dbcon == null)
+                               out.println("dbcon is NULL.");
```

- **How read/write requests were routed?**
  The read/write requests were routed using the resource tags in context.xml and using it accordingly in the files that perform write requests. The files that only perform reads have a separate resource which allows the files to route them to their appropriate instances.

  - **File name, line numbers as in Github**
    AddMovie.java; line 60-68, AddStar.java; line 72-78, Verify.java; line 70-78

  - **Snapshots**

```
Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e; path=/" env=BALANCER_ROUTE_CHANGED

<Proxy "balancer://TomcatTest_balancer">
    BalancerMember "http://172.31.44.68:9999/TomcatTest/"
    BalancerMember "http://172.31.45.181:9999/TomcatTest/"
</Proxy>

<Proxy "balancer://Session_balancer">
    BalancerMember "http://172.31.44.68:9999/Session" route=1
    BalancerMember "http://172.31.45.181:9999/Session" route=2
ProxySet stickysession=ROUTEID
</Proxy>

<Proxy "balancer://CS122B_balancer">
    BalancerMember "http://172.31.44.68:9999/CS122B" route=1
    BalancerMember "http://172.31.45.181:9999/CS122B" route=2
ProxySet stickysession=ROUTEID
</Proxy>
```

```
ProxyPass /TomcatTest balancer://TomcatTest_balancer
ProxyPassReverse /TomcatTest balancer://TomcatTest_balancer

ProxyPass /Session balancer://Session_balancer
ProxyPassReverse /Session balancer://Session_balancer

ProxyPass /CS122B balancer://CS122B_balancer
ProxyPassReverse /CS122B balancer://CS122B_balancer
```

```
1  <Context path="/CS122B">
2
3      <Resource name="jdbc/MovieDB" auth="Container" type="javax.sql.DataSource"
4          maxTotal="100" maxIdle="30" maxWaitMillis="10000" username="testuser"
5          password="password" driverClassName="com.mysql.jdbc.Driver"
6          url="jdbc:mysql://localhost:3306/moviedb?autoReconnect=true&amp;useSSL=false&amp;cachePrepStmts=true"
7
8      <Resource name="read-write" auth="Container" type="javax.sql.DataSource"
9              maxTotal="100" maxIdle="30" maxWaitMillis="10000" username="testuser"
10             password="password" driverClassName="com.mysql.jdbc.Driver"
11             url="jdbc:mysql://172.31.44.68:3306/moviedb?autoReconnect=true&amp;useSSL=false&amp;cachePrepStmt
12
13 </Context>
14
```

Task 3

- Have you uploaded the log file to Github? Where is it located?

- Have you uploaded the HTML file to Github? Where is it located?

- Have you uploaded the script to Github? Where is it located?

- Have you uploaded the WAR file and README to Github? Where is it located?