Task 1

- How did you use connection pooling?

We are using a connection pool to cache our connection to our database, which will cut down on the overhead needed to connect, we did this by setting our username and password for our mysql database connection in the context.xml file in the web meta with a maxTotal of 100 a maxIdle of 30, a maxWaitMillis of 10000, and cachePrepStmts set to true. We then set up the resource-ref in the web.xml for the project. With all of these set up we are able to create a connection to our database (in the same way shown in the tomcat-pooling demo we were provided on the git) from this pool in our java servlets. This was done near the top of every working servelt of our project, as seen below in things like the the MovieServlet where the connection pooling is used to query the database for movie information faster.

- File name, line numbers as in Github

AddMovie.java	lines 67-79
AddMovieFilter.java	lines 66-76
AddStar.java	lines 67-79
AddStarFilter.java	lines 70-82
AutoJS.java	lines 69-81
BrowseG.java	lines 77-89
BrowseT.java	lines 77-89
ConfirmationServlet.java	lines 78 -90
EmployeeLoginFilter.java	lines 71-83
EncryptEmployeePassword.java	lines 35-41
LoginFilterServlet.java	lines 77-89
LoginServlet.java	lines 65-77
MainPage.java	lines 80-92
MovieServlet.java	lines 177-189
PaymentFilterServlet.java	lines 68-80
PaymentServlet.java	lines 77-89
Search.java	lines 78-90
ShoppingCart.java	lines 81-93
ShowMetadata.java	lines 74-86
SingleMovieServlet.java	lines 91-103
SingleStarServlet.java	lines 72-84
_dashboard.java	lines 75-87

Auto.java	lines 70-82
Browse.java	lines 66-78

- Snapshots showing use in your code

MovieServlet.java - connecting to the pooled connection

```
Context initCtx = new InitialContext();

Context envCtx = (Context) initCtx.lookup("java:comp/env");
if (envCtx == null)
    out.println("envCtx is NULL");

DataSource ds = (DataSource) envCtx.lookup("jdbc/moviedb");


if (ds == null)
    out.println("ds is null.");

Connection connection= ds.getConnection();
```

AutoJS.java - connecting to pooled connection and creating a prepared statement using that connection

```
Context initCtx = new InitialContext();

Context envCtx = (Context) initCtx.lookup("java:comp/env");
if (envCtx == null)
    out.println("envCtx is NULL");

DataSource ds = (DataSource) envCtx.lookup("jdbc/moviedb");


if (ds == null)
    out.println("ds is null.");

Connection connection= ds.getConnection();

        String query =  "SELECT id, title from movies WHERE MATCH (title) AGAINST (? IN BOOLEAN MODE)";

        PreparedStatement stmt = connection.prepareStatement(query);
```

LoginFilterServlet.java - connecting to pooled connection and creating a prepared statement using that connection

```
     Context initCtx = new InitialContext();

Context envCtx = (Context) initCtx.lookup("java:comp/env");
if (envCtx == null)
    out.println("envCtx is NULL");

DataSource ds = (DataSource) envCtx.lookup("jdbc/moviedb");


if (ds == null)
    out.println("ds is null.");

Connection connection= ds.getConnection();
    String query = "SELECT * from customers where email = ? ";
    PreparedStatement stmt = connection.prepareStatement(query);
```

BrowseG.java - connecting to pooled connection and creating a prepared statement using that connection

```
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    // create database connection
     Context initCtx = new InitialContext();

Context envCtx = (Context) initCtx.lookup("java:comp/env");
if (envCtx == null)
    out.println("envCtx is NULL");

DataSource ds = (DataSource) envCtx.lookup("jdbc/moviedb");


if (ds == null)
    out.println("ds is null.");

Connection connection= ds.getConnection();


    String query = "SELECT name FROM genres";

    PreparedStatement stmt = connection.prepareStatement(query);
    ResultSet resultSet = stmt.executeQuery();
```

- How did you use Prepared Statements?

As in project 3, every search related servlet is set up properly with prepared statements as required in every servlet that requires them, which is primarily the . First a string containing “?” for the place of every user defined value is made and put into a prepared statement, the users values are then set into the prepared statements, the prepared statement is then executed and sent to the server which returns results if necessary. Using prepared statements like this is a advantageous because it alllows us to avoid SQL injection from the users side, and we can cache prepared statements in the connection pool, which saves us time and increases effectiveness.

- File name, line numbers as in Github

MovieServlet.java	lines 207-380
SingleMovieServlet.java	lines 106 -125
SingleStarServlet.java	lines 86-95

- Snapshots showing use in your code

MovieServlet - setting up the string for genre search that will be put in the prepared statement

```
qry2+="SELECT * FROM movies as m Left JOIN ratings as r ON r.movieId = m.id join ( select movieId, title, group_concat(name) as"
        + " genres from genres_in_movies join genres on genres_in_movies.genreId = genres.id join movies on genres_in_movies.movieId"
        + " = movies.id Group by movies.id HAVING FIND_IN_SET( ? , genres) > 0 ) as gm ON gm.movieId = "
        + "m.id join ( select movieId, title, group_concat(name) as stars, group_concat(starId) as starID from stars_in_movies"
        + " join stars on stars_in_movies.starId = stars.id join movies on stars_in_movies.movieId = movies.id"
        + " Group by movies.id) as sm ON sm.movieId = m.id";
```

MovieServlet - setting up the string for fulltext search that will be put in the prepared statement

```
qry2 = "SELECT * FROM movies as m Left JOIN  ratings as r ON r.movieId = m.id left join "
        + "(select movieId, title, group_concat(name) as genres from genres_in_movies left join genres "
        + "on genres_in_movies.genreId = genres.id left join movies on genres_in_movies.movieId = movies.id "
        + "Group by movies.id ) as gm ON gm.movieId = m.id left join ( select movieId, title, group_concat(name)"
        + " as stars, group_concat(starId) as starID from stars_in_movies left join stars on stars_in_movies.starId "
        + "= stars.id left join movies on stars_in_movies.movieId = movies.id Group by movies.id ) "
        + "as sm ON sm.movieId = m.id WHERE MATCH (m.title) AGAINST (? IN BOOLEAN MODE)"        ;
```

MovieServlet - Inserting values into the prepared statement for the fulltext search

```
 qry = connection.prepareStatement(qry2);

qry.setString(1, parsedSearch);
qry.setInt(2, currentPage);
qry.setInt(3, pCount);
```

MovieServlet - the executing the prepared statement to get the query info

```
ResultSet resultSet = qry.executeQuery();
long endJD = System.nanoTime();

//set up body
out.println("<body>");
out.println("<button onclick=\"window.location.href = \'/project1/MainPage\';
out.println("<button onclick=\"window.location.href = \'/project1/ShoppingCar
```

MovieServlet.java - the insertion of variable search parameters for the variable search using prepared statements

```
n=1;
qry = connection.prepareStatement(qry2);
if(titleSearch!=null) {qry.setString(n, "%"+titleSearch+"%");n++;}
if(directorSearch!=null) {qry.setString(n, "%"+directorSearch+"%");n++;}
if(starSearch!=null) {qry.setString(n, "%"+starSearch+"%");n++;}
if(yearSearch!=null) {qry.setString(n, yearSearch);n++;}
```

Task 2

- Address of AWS and Google instances

AWS instance: 3.16.214.9 update
Google Cloud Instance: 35.235.66.178

- Have you verified that they are accessible? Does Fablix site get opened both on Google's 80 port and AWS' 8080 port?

Yes, Fabflix is accessible on both instances and through the specified ports.

- Explain how connection pooling works with two backend SQL (in your code)?

With two backend SQLs, there are two different database connection types in the connection pool. One type connects to the master instance's SQL, and one type connects to the localhost's SQL. The connection to the master is used for all write requests, and the connection to the localhost is used only for read requests. Depending on whether a request needs to read or write to the database, it will lease the appropriate connection type from the pool. If at any point all connections are being using when an operation needs a connection, a new connection of the appropriate type is made and added to the pool. Afterwards, the connections can be reused when future requests are made. To implement this in our code, we defined a new data source in the context.xml and web.xml files. The URLs of each point to different backend SQLs. In the individual servlets, the code is the same as task 1.

- File name, line numbers as in Github

(Same files as in task 1, included configuration files below)

context.xml	Lines 17-26
web.xml	Lines 25-35

- Snapshots

(Same snapshots as in task 1, included configuration files below)

context.xml - new resource pointing to master SQL

```
<Resource name="jdbc/write_moviedb"
                        auth="Container"
                        type="javax.sql.DataSource"
               maxTotal="100"
               maxIdle="30"
                maxWaitMillis="10000"
               username="mytestuser"
               password="mypassword"
               driverClassName="com.mysql.jdbc.Driver"
               url="jdbc:mysql://172.31.42.140:3306/moviedb?
autoReconnect=true&amp;useSSL=false&amp;cachePrepStmts=true"/>
```

web.xml - new resource ref

```
  <resource-ref>
        <description>
            Resource reference to a factory for java.sql.Connection
            instances that may be used for talking to a particular
            database that
            is configured in the server.xml file.
        </description>
        <res-ref-name>jdbc/write_moviedb</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
```

MovieServlet.java - connecting to the pooled connection

```
            Context initCtx = new InitialContext();

        Context envCtx = (Context) initCtx.lookup("java:comp/env");
        if (envCtx == null)
            out.println("envCtx is NULL");

        DataSource ds = (DataSource) envCtx.lookup("jdbc/moviedb");


        if (ds == null)
            out.println("ds is null.");

        Connection connection= ds.getConnection();
```

AddMovieFilter.java - connecting to pooled connection

```
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    // create database connection
    // create database connection
Context initCtx = new InitialContext();

    Context envCtx = (Context) initCtx.lookup("java:comp/env");
    if (envCtx == null)
        out.println("envCtx is NULL");

    DataSource ds = (DataSource) envCtx.lookup("jdbc/write_moviedb");

    if (ds == null)
        out.println("ds is null.");

    Connection connection= ds.getConnection();
    CallableStatement callStmt = connection.prepareCall("{call add_movie(?, ?, ?, ?, ?, ?)}");
```

- How read/write requests were routed?

On the main instance the load balancer works through port 80. When the fabflix application is accessed through this port on the main instance, the load balancer will redirect the request to either the master instance's or slave instance's 8080 port. From there, the servlet that is accessed determines whether to route the database request to either the master's mysql or slave's mysql on port 3306, depending on if it is a read or write. If it is a read, the request will be handled on whichever instance it is already on, but if it is a write it will be sent to the master's mysql. To handle this routing to the proper instance, we added a new resource to the context.xml file which defines another data source. The first resource's URL points to the localhost's mysql which is used for reading. The second resource's URL points to the master instance's mysql. This way the behavior of the application follows the previous explanation.
In each of the fabflix servlets, we see if the requests require either a reads or write. Depending on this, we load the correct resource. Specifically, if it only sends reads to moviedb, we use the resource that accesses the localhost's mysql. But if it sends writes to moviedb, we use the resourcee that accesses the master instance's mysql. The parts of the website that make writes to the database include the checkout confirmation and the add movie and add star actions of the employee dashboard.

- File name, line numbers as in Github

context.xml	Lines 17-26
web.xml	Lines 25-35
AddMovieFilter.java	Lines 63-78
AddStarFilter.java	Lines 67-82
ConfirmationServlet.java	Lines 76-90, 112-120
EncryptEmployeePassword.java	Lines 33-41, 45-47, 72-75, 84

- Snapshots

context.xml - new resource, pointing to master SQL

```
<Resource name="jdbc/write_moviedb"
                          auth="Container"
                          type="javax.sql.DataSource"
               maxTotal="100"
               maxIdle="30"
                maxWaitMillis="10000"
               username="mytestuser"
               password="mypassword"
               driverClassName="com.mysql.jdbc.Driver"
               url="jdbc:mysql://172.31.42.140:3306/moviedb?
autoReconnect=true&amp;useSSL=false&amp;cachePrepStmts=true"/>
```

web.xml - new resource ref

```
  <resource-ref>
        <description>
            Resource reference to a factory for java.sql.Connection
            instances that may be used for talking to a particular
            database that
            is configured in the server.xml file.
        </description>
        <res-ref-name>jdbc/write_moviedb</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
```

AddMovieFilter.java - setting up connection to write resource and making write request

```
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    // create database connection
    // create database connection
Context initCtx = new InitialContext();

    Context envCtx = (Context) initCtx.lookup("java:comp/env");
    if (envCtx == null)
        out.println("envCtx is NULL");

    DataSource ds = (DataSource) envCtx.lookup("jdbc/write_moviedb");

    if (ds == null)
        out.println("ds is null.");

    Connection connection= ds.getConnection();
    CallableStatement callStmt = connection.prepareCall("{call add_movie(?, ?, ?, ?, ?, ?)}");
```

ConfirmationServlet.java - setting connection to write resource and making write request

```
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    // create database connection
Context initCtx = new InitialContext();

    Context envCtx = (Context) initCtx.lookup("java:comp/env");
    if (envCtx == null)
        out.println("envCtx is NULL");

    DataSource ds = (DataSource) envCtx.lookup("jdbc/write_moviedb");


    if (ds == null)
        out.println("ds is null.");

    Connection connection= ds.getConnection();
```

```
String update = "INSERT INTO sales VALUES (NULL, ?, ?, ?, ?)";
stmt = connection.prepareStatement(update);

stmt.setInt(1, customer_id);
stmt.setString(2, movie_id);
stmt.setString(3, date);
stmt.setInt(4, quantity);

stmt.executeUpdate();
```

Task 3

- Have you uploaded the log files to Github? Where is it located?
  Yes. All log files are located inside Project5/log info folder, they are all called TimeLogs_(description of test).txt

- Have you uploaded the HTML file (with all sections including analysis, written up) to Github? Where is it located?
  Yes. the HTML file is located inside the Project5/log info folder as well, it is called jmeter_report.html

- Have you uploaded the script to Github? Where is it located?
  It is in the root of the folder on the landing page /cs122b-winter19-team-12, it is called log parser.py, see the README on the landing page for instructions on how to operate it.

- Have you uploaded the WAR file and README to Github? Where is it located?
  Yes they are located in the first folder of the repo on the landing page of the git repository, /cs122b-winter19-team-12