CST-339 Activity 4 Guide

Contents

Part 1: Creating Data Services using Spring JDBC

Overview

Goal and Directions:

In this activity, you will code a data service using the Data Access Object design pattern to persist data to a relational MySQL database using Spring JDBC. We will use the standard JDBC template that is built into Spring JDBC and use SQL to persist data to and from the database.

Execution

Execute this assignment according to the following guidelines:

1. Make a copy of the *topic3-2* project by right clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace, and then selecting the Paste menu option. Name the project *topic4-1*. Rename the Spring application file from *Topic32Application.java* to *Topic42Application.java*. Fixup the *artifactId* and *name* tags in the POM file (pom.xml) so the value is set to *topic4-1*. Run the project to ensure that everything is working properly.
2. Add the following entries in the POM file (*pom.xml*) to add libraries for the MySQL Database and Spring JDBC.

```
<!--  For Spring JDBC and the MySQL Database -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

3. Initialize the MySQL Database using the DDL script *MySQL-ddl.sql* from the course materials. You will need to install MAMP and MySQL Workbench for this activity. See Appendix A for instructions.
    a. To initialize the MySQL Database, open MySQL Workbench, click on the Administration tab, select the Data Import/Restore icon, select the Import from

Self-Contained File option, browse to the *ddl.sql* script, and click the Start Import button.

b. Click on the Schema tab in MySQL Workbench and click on the refresh icon. The database named *cst339* should be listed that contains a single *ORDERS* table populated with some test data.

4. Open up the *application.properties* file located in the *src/main/resources* directory. Add the following database configuration properties. The property names **MUST** match the following in order for Spring JDBC to work properly with Spring Boot. Note, if you are on Windows, your MySQL database port will not be 8889 but 3306 (unless you have changed this default value). Also, see Appendix B to learn how to encrypt your database credentials in Spring Boot.

```
# Local MAMP Database
spring.datasource.url=jdbc:mysql://localhost:8889/cst339
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

5. Create a new Java Interface named *DataAccessInterface* (DAO design pattern) by selecting the New→Interface menu options in the *com.gcu.data* package. Add the following CRUD methods and use Generics in the design of this interface.

```
public interface DataAccessInterface <T>
{
    public List<T> findAll();
    public T findById(int id);
    public boolean create(T t);
    public boolean update(T t);
    public boolean delete(T t);
}
```

6. Create a data service class by right-clicking on the project and selecting the New→Class menu options. Create the class in the *com.gcu.data* package with a class name of *OrdersDataService* that implements the *DataAccessInterface* interface with a Generic type of *OrderModel* specified on the interface. The class should be marked with the *@Service* annotation at the class level. Implement the DAO design pattern methods in the *OrdersDataService* class as follows:

a. Add a private class member named *datasource* of type *DataSource* that is autowired.

b. Add a private class member named *jdbcTemplateObject* of type *JdbcTemplate* that is autowired.

c. Add a non-default constructor named *OrdersDataService(DataSource dataSource)* that sets the class member variable *dataSource* from the constructor argument, and then create an instance of a *JdbcTemplate(dataSource)* and set the class member variable *jdbcTemplateObject*. Spring will use constructor injection to initialize this class.

```
@Autowired
private DataSource dataSource;
private JdbcTemplate jdbcTemplateObject;

/**
 * Non-Default constructor for constructor injection.
 */
public OrdersDataService(DataSource dataSource)
{
    this.dataSource = dataSource;
    this.jdbcTemplateObject = new JdbcTemplate(dataSource);
}
```

d. *public List<OrderModel> findAll()*:
   i. Assign a variable *sql* of type String to a value of *SELECT * FROM ORDERS*
   ii. Assign a variable *orders* of type *List<OrderModel>* to an empty list.
   iii. Assign a variable *srs* of type *SqlRowSet* from a call to *jdbcTemplateObject.queryForRowSet(sql)* passing the method the SQL query statement to execute
   iv. Loop over *srs.next()* until false and, in the loop, create an instance of an *OrderModel* using the appropriate getter method for each column on the *srs* variable to populate each default constructor method argument. Add the instance of an *OrderModel* to the *orders* list.
   v. Surround the database logic with a try catch block that catches *Exception* and for now just prints the stack track to the console.
   vi. Return the list of *orders*.

```
/**
 * CRUD: finder to return all entities
 */
public List<OrderModel> findAll()
{
    String sql = "SELECT * FROM ORDERS";
    List<OrderModel> orders = new ArrayList<OrderModel>();
    try
    {
        // Execute SQL Query and loop over result set
        SqlRowSet srs = jdbcTemplateObject.queryForRowSet(sql);
        while(srs.next())
        {
            orders.add(new OrderModel(srs.getLong("ID"),
                                      srs.getString("ORDER_NO"),
                                      srs.getString("PRODUCT_NAME"),
                                      srs.getFloat("PRICE"),
                                      srs.getInt("QUANTITY")));
        }
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    return orders;
}
```

e. *public OrderModel findById(int id)*: For now, return null.
f. *public boolean create(OrderModel order)*:

i. Assign a variable *sql* of type String to a value of *INSERT INTO ORDERS(ORDER_NO, PRODUCT_NAME, PRICE, QUANTITY) VALUES(?, ?, ?, ?)*
ii. Assign a variable *rows* of type *int* from *jdbcTemplateObject.update(sql, order.getOrderNo(), order.getProductName(), order.getPrice(),* and *order.getQuantity())*, passing the SQL insert statement to execute along with each of the query parameters from the *OrdersModel* data
iii. Surround the database logic with a try catch block that catches *Exception* and for now just prints the stack track to the console.
iv. Return the *boolean* value of true if the *rows* variable is equal to 1, otherwise return false.

```
/**
 * CRUD: create an entity
 */
public boolean create(OrderModel order)
{
    String sql = "INSERT INTO ORDERS(ORDER_NO, PRODUCT_NAME, PRICE, QUANTITY) VALUES(?, ?, ?, ?)";
    try
    {
        // Execute SQL Insert
        int rows = jdbcTemplateObject.update(sql,
                                             order.getOrderNo(),
                                             order.getProductName(),
                                             order.getPrice(),
                                             order.getQuantity());

        // Return result of Insert
        return rows == 1 ? true : false;
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    return false;
}
```

g. *public boolean update(OrderModel order)*: For now, return true.
h. *public boolean delete(OrderModel order)*: For now, return true.

7. Update the *OrdersBusinessService* class by adding a class member variable named *service* of type *DataAccessInterface<OrderModel>* that is autowired.
8. Remove the code in the getOrders() method of the *OrdersBusinessService* class and replace the code with a call to the *findAll()* method of the *service* variable and returns the list from this method call.
9. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form to display the Orders page. Verify that the Orders are displayed correctly from the database. Take a screenshot of the Orders page.
10. For extra practice (not graded), update the implementation to use a Spring JDBC Row Mapper versus looping over the Result Set in the *findAll()* method.
11. For extra practice (not graded), complete the implementation of the *findById()*, *update()* and *delete()* methods of the *OrdersDataService* and test them within your application.

Deliverables:

The following needs to be submitted as this part of the activity:

a. Screenshot of the Orders page.

Part 2: Creating Data Services using Spring Data JDBC

Overview

Goal and Directions:

In this activity, you will code a data service using the Repository design pattern to persist data to a relational MySQL database using Spring Data JDBC. We will use the standard CRUD Repository that is built into Spring Data JDBC, which provides an ORM like interface that allows you to easily persist object to and from the database without having to write SQL and is mapped to an existing database schema. The proper design and separation of both application object models and entity object models will also be demonstrated, such that the presentation layer is not polluted with data persistence technologies.

Execution

Execute this assignment according to the following guidelines:

1. Make a copy of the *topic3-2* project by right-clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace, and then selecting the Paste menu option. Name the project *topic4-2*. Rename the Spring application file from *Topic32Application.java* to *Topic42Application.java*. Fixup the *artifactId* and *name* tags in the POM file (pom.xml) so the value is set to *topic4-2*. Run the project to ensure that everything is working properly.
2. Add the following entries in the POM file (*pom.xml*) to add libraries for the MySQL Database and Spring JDBC.

```
<!--  For Spring JDBC and the MySQL Database -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

3. Open up the *application.properties* file located in the *src/main/resources* directory. Add the following database configuration properties. The property names **MUST** match the following in order for Spring JDBC to work properly with Spring Boot. Note, if you are on Windows, your MySQL database port will not be 8889 but 3306 (unless you have changed this default value). Also, see Appendix B, to learn how to encrypt your database credentials in Spring Boot.

```
# Local MAMP Database
spring.datasource.url=jdbc:mysql://localhost:8889/cst339
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

4. Create a new *OrderEntity* class in the *com.gcu.data.entity* package. Add five private class member variables: *id* of type Long, *orderNo* and *productName* both of type String, *price* of type float, and *quantity* of type int, all with getter and setter methods, and with a default constructor and a non-default constructor that initialize all class member variables. The class should be marked with the *@Table("ORDERS")* annotation at the class level. The private *id* class member variable should be marked with the *@Id* annotation. The remaining class member variables should be marked with the *@Column("COLUMN_NAME")* annotation with the COLUMN_NAME replaced with the corresponding database column name.

```
@Table("ORDERS")
public class OrderEntity
{
    @Id
    Long id;

    @Column("ORDER_NO")
    String orderNo;

    @Column("PRODUCT_NAME")
    String productName;

    @Column("PRICE")
    float price;

    @Column("QUANTITY")
    int quantity;
```

5. Create a new *OrderRowMapper* class in the *com.gcu.data.mapper* package that implements the *RowMapper* interface from the *org.springframework.jdbc.core* package with a Generic class type of <*OrderEntity*>. Implement the *public mapRow(Result rs, int rowNumber)* method that returns an instance of an *OrderEntity*. The implementation should create an instance of an *OrderEntity*, using the appropriate getter method for each column on the *rs* method argument to populate each default constructor method argument. The new instance of an *OrderEntity* should be returned.

```
public class OrderRowMapper implements RowMapper<OrderEntity>
{
    @Override
    public OrderEntity mapRow(ResultSet rs, int rowNumber) throws SQLException
    {
        return new OrderEntity(rs.getLong("ID"), rs.getString("ORDER_NO"), rs.getString("PRODUCT_NAME"), rs.getFloat("PRICE"), rs.getInt("QUANTITY"));
    }
}
```

6. Create an *OrdersRepository* interface class in the *com.gcu.data.repository* package that extends from the *CrudRepository* in the *org.springframework.data.jdbc.repository* package with a Generic class type of <*OrderEntity, Long*>. None of the methods will be overridden and we will use the default CRUD methods in the super class.

```
public interface OrdersRepository extends CrudRepository<OrderEntity, Long>
{
}
```

7. Create a new interface class named *DataAccessInterface* in the *com.gcu.data* package using a Generic T at the interface level. Define the following public CRUD methods to support data persistence.
    a. public List<T> findAll();
    b. public T findById(int id);
    c. public boolean create(T t);
    d. public boolean update(T t);
    e. public boolean delete(T t);

```
public interface DataAccessInterface <T>
{
    public List<T> findAll();
    public T findById(int id);
    public boolean create(T t);
    public boolean update(T t);
    public boolean delete(T t);
}
```

8. Create new *OrderDataSevice* class in the *com.gcu.data* package that implements the *DataAccessInterface* interface. The following data access methods should be implemented as follows:
    a. Add a *@Service* annotation to the class level.
    b. A class-scoped instance variable *ordersRepository* of type *OrdersRepository* that is autowired.
    c. Non-default constructor that takes an instance of *OrdersRepository* as a method argument that initializes the class-scoped instance variable. This is required to support constructor injection of the Repository in this service class.
    d. *findById()*: return null for now.
    e. *findAll()*:
        i. Create a variable name *orders* of type *List<OrderEntity>* and initialize it to an empty *ArrayList<OrderEntity>*.
        ii. Call the *findAll()* method on the *ordersRepository* class scoped variable and assign the return value to a variable named *orderIterable* of type *Iterable<OrderEntity)* name.
        iii. Convert the *Iterable<OrderEntity>* to a *List<OrderEntity>* by calling the *forEach(orders::add)* method on the *orderIterable.*
        iv. Return the *orders* List.
        v. Surround all logic with a try catch block that for now just prints the stack trace to the console.
    f. create():

i. Call the *save(order)* method on the *ordersRepository* class-scoped variable and return a boolean true from the method.
ii. Surround all logic with a try catch block that for now just prints the stack trace to the console that returns a boolean true.

g. *update()*: return a boolean true for now.
h. *delete()*: return a boolean true for now.

```java
@Service
public class OrdersDataService implements DataAccessInterface<OrderEntity>
{
    @Autowired
    private OrdersRepository ordersRepository;

    /**
     * Non-Default constructor for constructor injection.
     */
    public OrdersDataService(OrdersRepository ordersRepository)
    {
        this.ordersRepository = ordersRepository;
    }

    /**
     * CRUD: finder to return a single entity
     */
    public OrderEntity findById(int id)
    {
        return null;
    }

    /**
     * CRUD: finder to return all entities
     */
    public List<OrderEntity> findAll()
    {
        List<OrderEntity> orders = new ArrayList<OrderEntity>();

        try
        {
            // Get all all the Entity Orders
            Iterable<OrderEntity> ordersIterable = ordersRepository.findAll();

            // Convert to a List and return the List
            orders = new ArrayList<OrderEntity>();
            ordersIterable.forEach(orders::add);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

        // Return the List
        return orders;
    }

    /**
     * CRUD: create an entity
     */
    public boolean create(OrderEntity order)
    {
        try
        {
            this.ordersRepository.save(order);
        }
        catch (Exception e)
        {
            e.printStackTrace();
            return false;
        }
        return true;
    }
```

9. Update the *OrdersBusinessService* as follows:
    a. Add a class-scoped variable named *service* of type *OrdersDataService* that is autowired.
    b. Remove the existing code from the *getOrders()* method.
    c. Call the *findAll()* method on the *service* class-scoped variable and save the return value in a variable *ordersEntity* of type *List<OrderEntity>*.
    d. Create an empty *List<OrderModel>* in a variable named *ordersDomain*.
    e. Iterate over the *ordersEntity* List and create a new instance of *OrderModel* from the *OrderEntity* objects and add the instances of the *OrderModel* to the or *ordersDomain* List. This is required so the persistence technology dependencies from the *OrderEntity* do not get propagated to the presentation layer.
    f. Return the *ordersDomain* List.

```
@Override
public List<OrderModel> getOrders()
{
    // Get all all the Entity Orders
    List<OrderEntity> ordersEntity = service.findAll();

    // Iterate over the Entity Orders and create a list of Domain Orders
    List<OrderModel> ordersDomain = new ArrayList<OrderModel>();
    for(OrderEntity entity : ordersEntity)
    {
        ordersDomain.add(new OrderModel(entity.getId(), entity.getOrderNo(), entity.getProductName(), entity.getPrice(), entity.getQuantity()));
    }

    // Return list of Domain Orders
    return ordersDomain;
}
```

10. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form to display the Orders page. Verify that the Orders are displayed correctly from the database. Take a screenshot of the Orders page.
11. For extra practice (not graded), complete the implementation for a *findById()*, *update()*, and *delete()* methods for the *OrdersDataService* using the default methods available from the *CrudRepository* and test them within your application.

Deliverables:

The following needs to be submitted as this part of the activity:

1. Screenshot of the Orders page.

Part 3: Creating Data Services using Spring Data JDBC Native Queries

Overview

Goal and Directions:

In this activity, you will code a data service using the Repository design pattern to persist data to a relational MySQL database using Spring Data JDBC and custom native SQL queries. We will again use the standard CRUD Repository that is built into Spring Data JDBC and override some of the methods. This would be useful if you ever needed to override any of the methods in the Repository and provide your own high-performance handwritten SQL.

Execution

Execute this assignment according to the following guidelines:

1. Make a copy of the *topic4-2* project by right-clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace, and then selecting the Paste menu option. Name the project *topic4-3*. Rename the Spring application file from *Topic42Application.java* to *Topic43Application.java*. Fix up the *artifactId* and *name* tags in the POM file (pom.xml) so the value is set to *topic4-3*. Run the project to ensure that everything is working properly.
2. Update the *OrdersRepository* interface class in the *com.gcu.data.repository* package and override the following method. The following data access method will be overridden and re-implemented using customized SQL as follows:
    a. *public List<OrderEntity> findAll()*:
        i. Method declaration only with the method marked with the *@Query* annotation with the *value* attribute set to a value of *SELECT * FROM ORDERS*.

```
public interface OrdersRepository extends CrudRepository<OrderEntity, Long>
{
    // Example of truly overriding a method from the CrudRepository and using our own customized SQL
    @Override
    @Query(value = "SELECT * FROM ORDERS")
    public List<OrderEntity> findAll();
}
```

3. Update the *OrdersDataService* by using Spring JDBC queries to "override" the default *save()* method from the *CrudRepository*.
    a. Add a private class member named *datasource* of type *DataSource* that is autowired.
    b. Add a private class member named *jdbcTemplateObject* of type *JdbcTemplate* that is autowired.
    c. Update the non-default constructor to add a second argument *DataSource dataSource)* and in the implementation set the class member variable *dataSource* from the constructor argument and then create an instance of a

JdbcTemplate(dataSource) and sets the class member variable *jdbcTemplateObject*. Spring will use constructor injection to initialize this class.

```
@Autowired
private OrdersRepository ordersRepository;
@SuppressWarnings("unused")
private DataSource dataSource;
private JdbcTemplate jdbcTemplateObject;

/**
 * Non-Default constructor for constructor injection.
 */
public OrdersDataService(OrdersRepository ordersRepository, DataSource dataSource)
{
    this.ordersRepository = ordersRepository;
    this.dataSource = dataSource;
    this.jdbcTemplateObject = new JdbcTemplate(dataSource);
}
```

d. Remove the existing code from the *create()* method that is using the *CrudRepository*. Update the implementation of the create() method to use the JDBC Template and SQL statements similar to the code written in Part 1. This is another example of how to override the default implementation provided by the CrudRepository and implement your own SQL code.

```
// Example of "overriding" the CrudRepository save() because it simply is never called
//  You can inject a dataSource and use the jdbcTemplate to provide a customized implementation of a save() method
String sql = "INSERT INTO ORDERS(ORDER_NO, PRODUCT_NAME, PRICE, QUANTITY) VALUES(?, ?, ?, ?)";
try
{
    // Execute SQL Insert
    jdbcTemplateObject.update(sql,
                              order.getOrderNo(),
                              order.getProductName(),
                              order.getPrice(),
                              order.getQuantity());
}
catch (Exception e)
{
    e.printStackTrace();
    return false;
}
return true;
```

4. Remove the code in the getOrders() method of the *OrdersBusinessService* class and replace the code with a call to the *findAll()* method of the *service* variable and returns the list from this method call.
5. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form to display the Orders page. Verify that the Orders are displayed correctly from the database. Take a screenshot of the Orders page.
6. Update the POM file to name the output JAR file *cst339activity*. Run a Maven build and test the JAR file out from a terminal window. Reference the steps as needed from the Activity 1 Guide.
7. For extra practice (not graded), complete the implementation for a *findById()*, *update()* and *delete()* methods for the *OrdersDataService* and *CrudRepository* and test them within your application.

Deliverables:

The following needs to be submitted as this part of the activity:

1. Screenshot of the Orders page.

Research Questions

1. Research Questions: For traditional ground students, in a Microsoft Word document, answer the following questions:
    a. How does Spring Data JDBC differ from standard Java JDBC programming?
    b. How does Spring Data JDBC support transaction management and the ACID principle?

Final Activity Submission

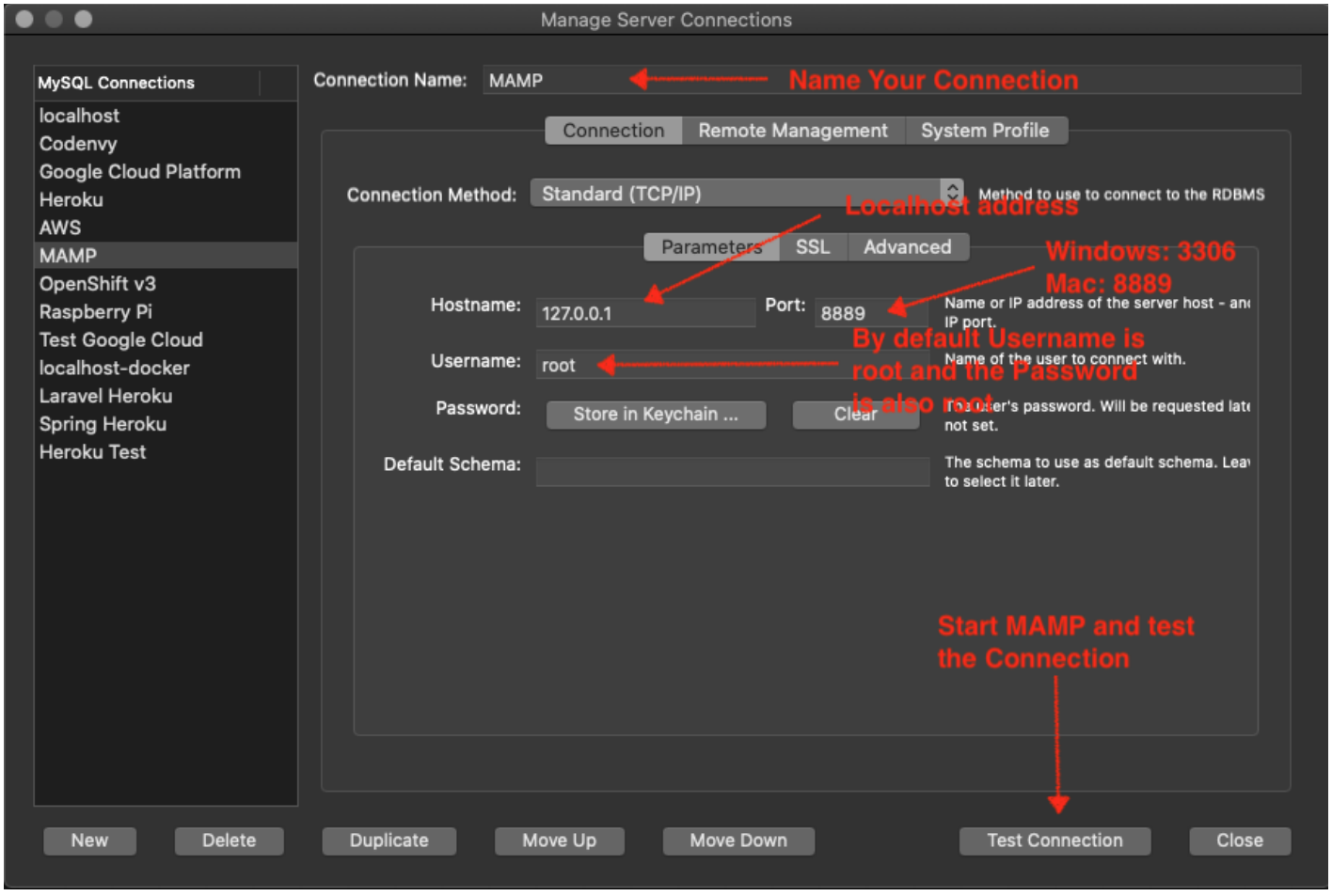
1. In a Microsoft Word document, complete the following for the Activity Report:
    a. Cover Sheet with the name of this assignment, date, and your name.
    b. Section with a title that contains all the screenshots for each part of the activity.
    c. Section with a title that contains the answers to the research questions (traditional ground students only).
2. Submit the Activity Report as directed by your instructor.

Appendix A: MAMP and MySQL Workbench Install Instructions

It is recommended that both Windows and Mac users install MAMP for this class, as this has proven to work a bit more reliably and offers a simpler distribution to use than other distributions like WAMP or XAMP.

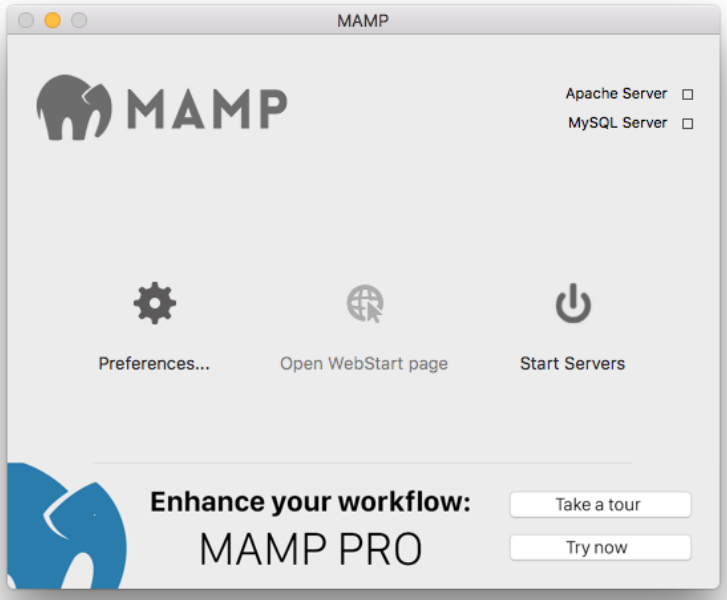
MAMP can be downloaded from the topic materials. Install MAMP using the default instructions. DO NOT enable the PRO version of MAMP.

MySQL Workbench is a powerful MySQL database client that can be used to manage your MySQL databases, export data, create ER diagrams, and much more. MySQL Workbench can be downloaded from the topic materials. Install MySQL Workbench using the default instructions. You can add a MySQL database connection to MAMP by adding a new MySQL Connection and using the following settings:

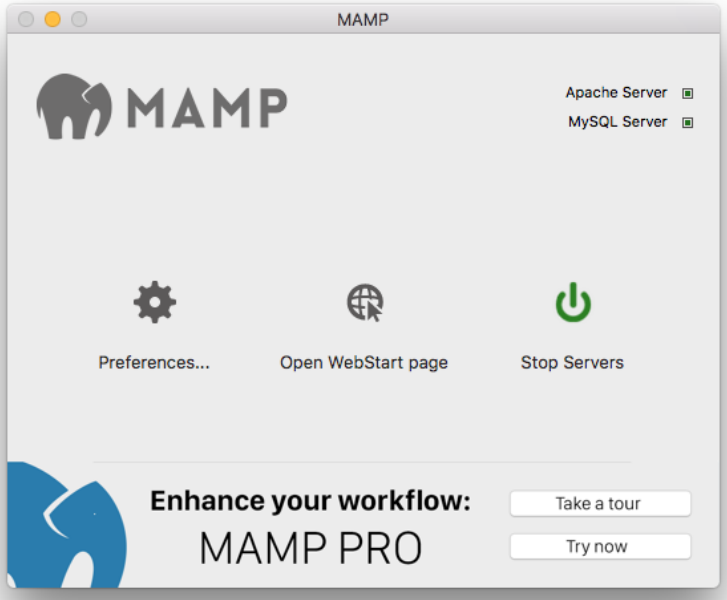


Once you have MAMP installed, the following instructions can be used to set up, validate, and use MAMP:

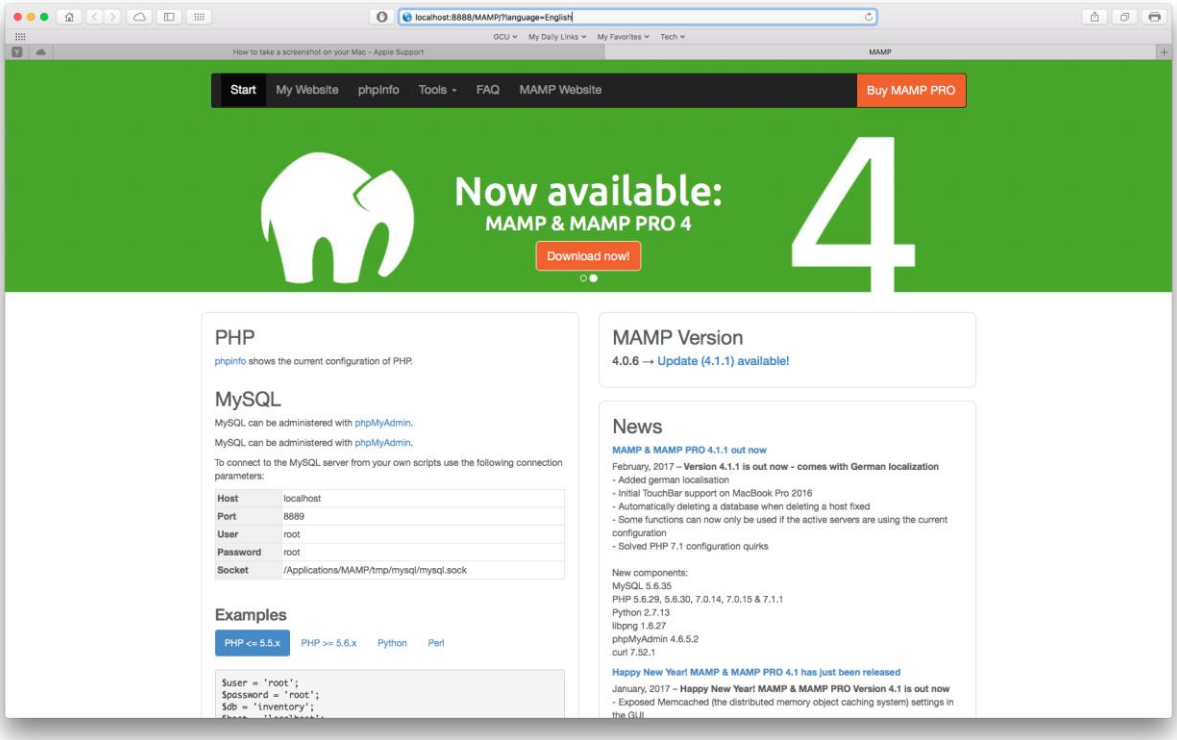
1. Start MAMP by double-clicking on the MAMP icon in your MAMP start menu for Windows or in your Applications folder for the Mac. The MAMP Control Panel should be displayed as shown in the figure below.



2. To start MAMP click the "Start Servers" button in the MAMP Control Panel. Note the status of the Apache Server and MySQL Server checkboxes, as they should light up green if all was started without error, as shown in the figure below.



3. If either of the Servers did not start, refer to the following troubleshooting guide:
    a. If the MySQL Server did not start, verify that you do not have another instance of MySQL running or that the default port of 3306 is not being used by another application.
    b. If the Apache Server did not start, chances are the default port is already being used. To change the Apache Server port, click the "Preferences…" button from the MAMP Control Panel, then click the Ports tab in the MAMP Preferences dialog, and change the Apache Port to 8888, and click OK to close the dialog. Your Servers should automatically restart. Again, make sure that both the Apache Server and MySQL Server checkboxes light up green if all was started without error as shown in the previous figure.
4. As a final step to verify that MAMP is running properly click the "Open WebStart page" button from the MAMP Control Panel. This should display the MAMP Main Application screen in your default browser as shown in the figure below.



NOTE: The MAMP Main Application is the application that can be used to access your deployed applications, display your PHP environment, and access the phpMyAdmin application. These functions are available from the top menu bar as outlined below.

1. **Start** – Displays the MAMP Main Application start page.
2. **My Website** – Displays a page that lists all of your deployed applications; click on the hyperlink for a desired application to access the application.
3. **Tools→phpinfo** – Displays all of the configuration settings and plugins that are set up in PHP; use this screen to validate the version of PHP you are running and what plugins are enabled.
4. **Tools→phpMyAdmin** – Displays the web-based MySQL administration pages; use this to manage your MySQL database to add databases, tables, and table columns to your MySQL database.

Other Resources Located in the Topic Materials:

1. Software documentation
2. 12 - PHP and MySQL, Using MAMP as Local Development Environment

Appendix B: How to Encrypt your Database Credentials

You can encrypt your database credentials that are configured in your application property file by using the following steps:

1) Add the Jasypt library to your Maven project.

```
<!-- For Jasypt to Encrypt Passwords-->
<dependency>
    <groupId>com.github.ulisesbocchio</groupId>
    <artifactId>jasypt-spring-boot-starter</artifactId>
<version>2.1.0</version>
```

2) Refer to "Jasypt Online Encryption and Decryption," located in the topic materials, to encrypt your credentials.

3) Then update your application property file, enclosing encrypted text in ENC() as shown below.

```
spring.datasource.username=ENC(2ZanVNv7l8UWFWDgnAcSIjdvxkyAPUAd+8CcYzBQwHk=)
spring.datasource.password=ENC(tcZg2HhX9r3bXXcqsRMz2pfa0C+JWc3RihXG+ZVIr8E=)
```

4) Enable property encryption in your application by adding the following annotation to the Spring Application class:

```
@EnableEncryptableProperties
```

5) Set an environment variable JASYPT_ENCRYPTOR_PASSWORD to your secret key. FOR TESTING ONLY, you can also set the property *jasypt.encryptor.password* to the value of your secret key.