**CST-256 Activity 3 Guide**

Contents

[Part 1: Data Validation 2](#_Toc528659359)

[Part 2: ACID Database Transactions 3](#_Toc528659360)

This activity has multiple parts. All parts must be completed prior to documentation submission.

## Part 1: Data Validation

**Overview**

In this activity, students will use the Laravel Data Validation framework to implement form validation to the Login Page created in Activity 2, Part 3: Laravel Blade Templates. Note: You will need functional code from Activity 2, Part 3: Laravel Blade Templates.

**Execution**

Execute this activity according to the following guidelines:

1. The following steps are optional and assumes you want to make copies of code from Activity 2, Part 3: Laravel Blade Templates. To update the code:
   1. Make a copy of LoginController.php to Login3Controller.php.
   2. Make a copy of login2.blade.php to login3.blade.php and update the form action to dologin3.
   3. Create a new route for /login3 to return a view login3.
   4. Create a new route for /dologin3 to return a Login3Controller@index.
2. Update Login Controller (or Login3Controller if you are using copies) to add Data Validation Rules to the request:
   1. Add a private function in the Controller to encapsulate running the Data Validation Rules:

**private function** validateForm(Request $request)  
 {  
 *// Setup Data Validation Rules for Login Form* $rules = [**'username'** => **'Required | Between:4,10 | Alpha'**,  
 **'password'** => **'Required | Between:4,10'**];  
  
 *// Run Data Validation Rules* $this->validate($request, $rules);  
 }

* 1. Call your validateform() method after your get your posted form variables. Note, this method automatically handles the redirection back to the Login View if the validation rules fail.

*// Validate the Form Data (note will automatically redirect back to Login View if errors)* $this->validateForm($request);

1. Update the Login View (or login3.blade.php if you are using copies) to display the data validation messages:
   1. Add the following PHP code next to each form element to display any validation error message:

**<?php echo** $errors->first(**'username'**)**?>**

**<?php echo** $errors->first(**'password'**)**?>**

* 1. Optionally you can display all errors at the top or bottom of the page:

*<!-- Display all the Data Validation Rule Errors -->***<?php  
 if**($errors->count() != 0)  
 {  
 **echo "<h5>List of Errors</h5>"**;  
 **foreach**($errors->all() **as** $message)  
 {  
 **echo** $message . **"<br/>"**;  
 }  
 }  
**?>**

* 1. Take screenshots showing Data Validation rules failing for both the Username and Password fields in the Login Form.

**Documentation**

All documentation will be submitted at the end of the activity to the learning management system. Ensure documentation of the following:

1. Create a Project Report using a GCU standard Project Header Page/Information containing your name, course, assignment name, and date. Make sure to include any screenshots demonstrating working code and write-ups as instructed.
2. Add the screenshots for the following in your Project Report:
   1. Login Form with Username validation errors
   2. Login Form with Password validation errors

## Part 2: ACID Database Transactions

**Overview**

In this activity, students will learn how to support ACID transactions in PHP.

**Execution**

Execute this assignment according to the following guidelines:

1. Create the following support Tables in MySQL:
   1. Create a Customer Table with the following columns:
      1. ID that is auto generated.
      2. First\_Name that is a varchar(100).
      3. Last\_Name that is a varchar(100).
   2. Create an Order Table with the following columns:
      1. ID that is auto generated.
      2. Product that is a varchar(100).
      3. Customer\_Id that is a foreign key to the ID column in the Customer Table.
2. Create a Customer DAO class:
   1. Create a class in Eclipse PHP called CustomerDao.
   2. Add a public method called addCustomer($firstName, $lastName).
   3. Add SQL Code that inserts the $firstName and $lastName arguments in the Customer table.
   4. Test your DAO class (with auto commit transactions enabled).
3. Create an Order DAO:
   1. Create a class in Eclipse PHP called OrderDao.
   2. Add a public method called addOrder($product).
   3. Add SQL Code that inserts the $order argument and Customer ID in the Order table.
   4. Test your DAO class (with auto commit transactions enabled).
4. Create an Order Business Service class:
   1. Create a class in Eclipse PHP called OrderService.
   2. Add a public method called createOrder(($firstName, $lastName, $product).
   3. Refactor DAOs: Move any mysqli construction code that returns a DB connection from the DOAs to the Order Business Service.
   4. Refactor DAOs: Add a constructor to each DAO that takes a DB connection variable.
   5. Implement the following workflow in the createOrder() method:
      1. Turn off Autocommit using mysqli:: autocommit(FALSE).
      2. Begin a transaction using mysqli::begin\_transaction().
      3. Call the CustomerDao->addCustomer() method.
      4. Call the OrderDa->addOrder() method.
      5. Commit your transaction using mysqli::commit().
         1. OR rollback your transaction using using mysqli::rollback().
      6. Test your Business Service class.

**Submission**

Submit the following to the learning management system:

A Project Report to include:

1. Screenshots for the following:
   1. Login Form with Username validation errors
   2. Login Form with Password validation errors
2. Screenshots of the database tables (before and after creating an order).
3. A block diagram for each of the components (Customer DAO, Order DAO, and Order Service) and how the components are related to each other.
4. A 50- to 100-word explanation on how this design supports ACID, providing rationale for why this design is better than running with auto commit enabled.
5. The Bitbucket URL after uploading your code to Bitbucket.