**Assignment 5**

**Glen Correia**

**N01615526**

**Part 1)**

**Sample\_A**

A screenshot of a computer

Description automatically generated

**Sample\_B**

**A screenshot of a computer

Description automatically generated**

**Sample\_C**

**A screenshot of a computer

Description automatically generated**

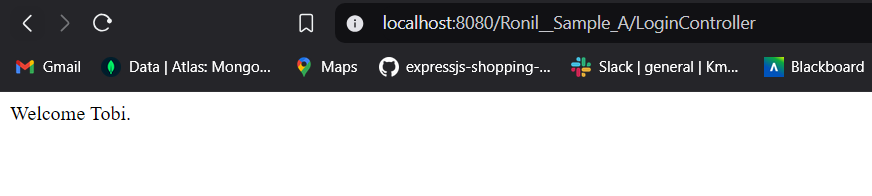
**Part 2)**

* **Error in Sample\_A**
* ***@WebServlet*("/LoginController") was missing in LoginController.**

**A screenshot of a computer

Description automatically generated**

**After the declaration**



**On a failed login attempt.**

A screenshot of a computer

Description automatically generated

* **Error in Sample\_B**
* **I faced the same error in sample\_B. After the declaration, error got solved.**
* A screenshot of a computer

  Description automatically generated
* A screen shot of a computer

  Description automatically generated
* **Error in Sample\_C**
* **In DB.java file jdbc driver was missing. Once I added it, code started running successfully.**
* **A screen shot of a computer

  Description automatically generated**
* **A screenshot of a computer

  Description automatically generated**

**Part-3**

**Sample\_A**

**1. Execution Flow:**

* User submits credentials via a login form (login.jsp).
* The LoginController processes the request and passes credentials to the Authenticator.
* Authenticator directly connects to the database to validate credentials.
* Based on the result, the controller forwards the user to either success.jsp or error.jsp.
* **Logic**: The application logic tightly integrates database operations within the Authenticator class, which validates user credentials by executing SQL queries.

**2. PROS**

* Simple structure; easy to understand for beginners.
* Minimal code; fewer layers of abstraction.
* Works well for small-scale applications.

**3. CONS**

* Tight coupling between Authenticator and the database logic.
* Hardcoded connection details reduce security and flexibility.
* Limited scalability due to direct SQL queries in the business logic.
* Error handling is minimal, making it prone to crashes.

**4. Improvements**

* Introduce a DAO (Data Access Object) layer to handle database interactions.
* Move database connection details to a configuration file.
* Add better exception handling.
* Use prepared statements with parameterized queries for enhanced security and maintainability.

**Sample\_B**

**1. Execution Flow:**

* Similar to Sample A, but the database connection logic is shifted to the LoginController.
* The controller creates a database connection and passes it to the Authenticator.
* The Authenticator uses the provided connection to validate credentials and returns a result.
* The controller forwards the user to the appropriate JSP based on the result.
* **Logic**: By separating database connection logic from the Authenticator, this approach introduces a slightly modular design.

**2. PROS**

* Better modularity: Authenticator does not manage database connections directly.
* Decouples business logic from database connection logic.
* Easier to adapt and test different connection strategies.

**3. CONS**

* Still tightly coupled between LoginController and database logic.
* Hardcoded connection details remain a security concern.
* Error handling is limited, leading to potential ungraceful failures.
* Lack of abstraction layers makes future scalability harder.

**4. Improvements**

* Use a DAO layer to abstract database operations.
* Externalize database connection configurations.
* Implement a connection pool for better database resource management.
* Add structured exception handling for robust error management.

**Sample\_C**

**1. Execution Flow:**

* User submits credentials via the login form.
* The LoginController receives the request and delegates database interaction to a new UserDao class.
* The UserDao uses the DB utility class to manage connections and queries the database.
* The Authenticator class validates credentials through the DAO.
* Based on the result, the user is directed to the appropriate JSP.
* **Logic**: This approach separates concerns by introducing a DAO (UserDao) for database operations and a DB utility for managing connections.

**2. PROS**

* High modularity: Clear separation between the controller, DAO, and utility classes.
* Connection management is centralized via the DB utility.
* More scalable and maintainable due to layered architecture.
* Reusable DAO logic for other operations beyond authentication.

**3. CONS**

* Slightly more complex structure, which could be challenging for beginners.
* Still dependent on hardcoded database connection details (though improved).
* Connection pooling or ORM tools are not implemented.

**4. Improvements**

* Use a connection pool (e.g., HikariCP) for efficient resource utilization.
* Externalize configuration (e.g., .properties or environment variables).
* Replace raw SQL with an ORM (e.g., Hibernate) for better query management and flexibility.
* Implement comprehensive exception handling across layers.