Lesson 11-Stored Procedures

Lesson 10 Concepts
1. Gain working knowledge of how to create and use stored procedure in MySQL
2. Understand the limitations of stored procedures
3. Understand the difference stored procedure and functions

Stored procedures are essentially pre-written SQL code that can be saved for repeated use. They are particularly useful when you frequently need to write certain queries. The process is simplified by stored procedures as they allow you to group one or more SQL statements under a single, identifiable name. This method effectively encapsulates common business logic within the database itself. An application that accesses the database can then call upon these procedures to retrieve or manipulate data in a consistent manner. This ensures uniformity and efficiency in data handling.

Advantages of Stored Procedures
Stored procedures offer several advantages that make them a vital tool in database management. Firstly, they simplify complex queries or computations by providing a platform where you can easily input, select, update, or delete data. This simplification makes handling complex data tasks more manageable.
Secondly, stored procedures provide a secure interface to data. This feature is particularly useful in sensitive sectors like banking, where data security is paramount. Banks often lock down data and only expose operations through stored procedures, thereby enhancing data security.
Lastly, stored procedures encapsulate business logic. For instance, a bank might use a stored procedure to calculate the interest on a loan. This procedure encapsulates the business logic for interest calculation, ensuring consistency in the calculation process.
The concept of **encapsulation**, which involves bundling data variables and methods that process and manipulate data together, is akin to a capsule pill. Much like the inner ingredients of a pill are hidden and only the results are visible, encapsulation hides the inner workings of data processing and manipulation, providing only the results. This approach ensures consistency and reliability in data handling.

BASIC SYNTAX:

```
DELIMITER //
CREATE PROCEDURE procedure_name(parameters)
BEGIN
  SQL statement(s) -----
END; //
DELIMITER ;
```

- CREATE PROCEDURE : statement used to create a new stored procedure.
- procedure_name: name of your stored procedure.

- parameter_name : name(s) of parameters that the stored procedure will use.
- SQL statements: SQL commands that you want the stored procedure to execute.
- BEGIN and END enclose the SQL statements
- DELIMITER //, DELIMITER ; - changes the default delimiter - semicolon (;) - to 2 forward slashes (or double dollar signs $$). This allows the semicolons (i.e. multiple statements ending in semi-colons) to be used within routines without any error.

To call the stored procedure, use the following syntax:

```
CALL procedures_name(parameter);
```

Difference between stored procedure and function:
While both stored procedures and stored functions are created in PHPMyAdmin under ROUTINES and store blocks of code, they have several key differences:

1. **Invocation**: Stored functions are invoked in SELECT statements, whereas stored procedures are invoked using the CALL command.
2. **Return Value**: Functions always return a value, whereas this is not always the case for stored procedures.
3. **Parameters**: Stored functions have parameters equivalent to the IN or input type. In contrast, stored procedures allow different types of parameters: IN, OUT, and INOUT.
4. **DML Commands**: Functions cannot perform any Data Manipulation Language (DML) commands such as INSERT, UPDATE, and DELETE. However, stored procedures can execute these commands.

IN parameter

A simple stored procedure to illustrate the use of IN parameter.

```
CREATE PROCEDURE get_employee(IN p_id INT)
    SELECT *
    FROM employees
    WHERE id = p_id;
```

This stored procedure is named get_employee. Here's a breakdown of what it does:

- CREATE PROCEDURE get_employee: This line creates a new stored procedure named get_employee.
- (IN p_id INT): The procedure takes one input parameter, p_id, which is of type INT.
- SELECT * FROM employees WHERE id = p_id;: This is the SQL statement that the procedure executes when it is called. It selects all columns (*) from the employees table where the id matches the input parameter p_id.

So, when you call this stored procedure with an integer argument, it will return all the column values from the employees table where the id is equal to the input argument. This is useful for retrieving the details of an employee with a specific id.

This example procedure is simple and doesn't require the use of DELIMITER, BEGIN, and END because it contains only a single SQL statement.

To call the procedure and pass in the parameter:

```
CALL get_employee(111);
```

CALL will pull the information of employee of id = 111.

Multiple parameters:

Stored procedures are able to take multiple parameters. Here is an example of a simple stored procedure taking 2 parameters.

```
CREATE PROCEDURE get_employees (
                          IN p_first_name VARCHAR(30),
                          IN p_last_name VARCHAR(30)
                          )
SELECT * FROM employees
WHERE first_name = p_first_name
AND last_name = p_last_name;
```

- `**CREATE PROCEDURE** get_employees: This line creates a new stored procedure named `get_employees`.
- **(IN p_first_name VARCHAR(30), IN p_last_name VARCHAR(30))** : `The stored procedure takes two input parameters, `p_first_name` and `p_last_name`, both of which are of type `VARCHAR(30)`.
- `**SELECT * FROM employees WHERE first_name = p_first_name AND last_name = p_last_name;**`: SQL statement that the procedure executes when it is called. It selects all columns (`*`) from the `employees` table where the `first_name` and `last_name` match the input parameters `p_first_name` and `p_last_name`.

To call the stored procedure called get_employees to retrieve the details of an employee with a specific name.

```
CALL get_employees("Jas", "Kaur");
```

Stored procedures and Data Manipulation Language (DML)
One of the advantages of stored procedures is the ability to use Data Manipulation Language (DML). This means that stored procedures can perform UPDATE, INSERT, or DELETE operations, providing greater flexibility in managing data.

Example:
A procedure using DML, INSERT for a stored procedure. This stored procedure will increment the ID by 1 and add a new row in the stock_items table of the petstore database.

```
DELIMITER //

   CREATE PROCEDURE add_new_stock  (
                              p_item VARCHAR(30),
                              p_price DECIMAL(10,2),
                              p_inventory SMALLINT(4),
                              p_category VARCHAR(7))

BEGIN
   DECLARE var_stock_id INT;
   SELECT MAX(id) + 1 INTO var_stock_id FROM stock_items;
   INSERT INTO stock_items (id, item, price, inventory, category)
      VALUES (var_stock_id, p_item, p_price, p_inventory, p_category);
END; //

DELIMITER ;
```

Here is a breakdown of each part of the stored procedure:

- DELIMITER //: This line changes the default delimiter from a semicolon (;) to //. This allows the semicolon to be used within the stored procedure.
- CREATE PROCEDURE add_new_stock (p_item VARCHAR(30), p_price DECIMAL(10,2), p_inventory SMALLINT(4), p_category VARCHAR(7)): This line creates a new stored procedure named add_new_stock. It takes four input parameters: p_item of type VARCHAR(30), p_price of type DECIMAL(10,2), p_inventory of type SMALLINT(4), and p_category of type VARCHAR(7).
- BEGIN ... END;: These keywords define the start and end of the procedure block.
- DECLARE var_stock_id INT;: This line declares a variable named var_stock_id of type INT.
- SELECT MAX(id) + 1 INTO var_stock_id FROM stock_items;: This line selects the maximum id from the stock_items table, adds 1 to it, and stores the result into the var_stock_id variable. This is done to generate a new unique id for the new stock item.
- INSERT INTO stock_items (id, item, price, inventory, category) VALUES (var_stock_id, p_item, p_price, p_inventory, p_category); This line inserts a new row into the stock_items table. The values for the row are taken from the var_stock_id variable and the input parameters.
- DELIMITER ;: This line changes the delimiter back to the default semicolon (;).

When this stored procedure is called, a new row will be added to the stock_items table with the provided item details and a unique id:

```
CALL add_new_stock("modern cat tower", 249.99, 5,
"feline");
```

OUT parameter

Stored procedures differ from stored functions in that they do not return a value in the same way. The use of RETURNS or RETURN commands is not permitted in a stored procedure, and a stored procedure cannot be called within a SELECT statement.
So, how can we make a value available for use in other statements? The answer is using an OUT parameter.

Before we delve into that, it's important to understand the different types of variables:

1. Variable scope
2. User-defined variable / Session variable

These concepts will help us better understand how to effectively use the OUT parameter in stored procedures.

1. **Variable scope**:

Simply put, the "scope" is the context in which a variable is accessible. Using a stored function as an example to illustrate scope:

Example:

```
CREATE FUNCTION plus_two( input INT )
RETURNS INT
RETURN input + 2;
```

In this example, we cannot access the parameter, "input" outside of that function. For example,

```
SELECT input(2);
```

This query would result in error. The scope of the variable, "input" is limited to use only within that function. The same logic also applies to stored procedures.

2.User-defined variable – session variables.

There is a type of variable called user-defined variable or otherwise called "session variable".

These variables are accessible by any functions, procedures, SELECT statements, DML etc that are in the same session. A session is defined as a successful connection to the database. When you start MAMP/XAMPP and open PHPMyAdmin , you have successfully connected to the database - that is referred to as a session. If, after a while, the database disconnects, then the session has ended.

Now, understanding what a session is, we can better understand the use of a session variable. The user-defined (i.e. session variable) are defined with the @ symbol at the beginning of the variable name. Session variables also store only a limited number of data types which includes: integers, decimals, strings, NULL.

Unlike stored functions, session variables in stored procedures DO NOT get declared with the DECLARE command. Instead, session variables rely on the SET command.

Syntax for session variables:

```
SET @v1 = 1;
SET @v2 = 4;

SELECT (@v1 + @v2) * 3 AS "result";
```

SELECT INTO multiple variables:
The SELECT INTO statement in SQL is used to retrieve data from a database and store it directly into variables. This is useful when you want to manipulate the data and use it later in your code.

SYNTAX:

```
SELECT column1, column2, ...
INTO variable1, variable2, ...
FROM table_name
WHERE condition;
```

SELECT INTO:

- column1, column2, … are the names of the columns in the table from which you want to select data.
- variable1, variable2, … are the variables where you want to store the selected data.
- table_name is the name of the table from which you want to select data.
- condition is the condition that must be satisfied for the rows to be selected.

Example:

```
SELECT first_name, last_name
INTO @fname, @lname
FROM employees
WHERE id = 112;
```

In the above example, the first_name and last_name of the employee with id = 112 are selected from the employees table and stored into the variables @fname and @lname, respectively.

In the next example shows a stored procedure using the OUT parameters and calling the session variables to use in a SQL statement :

Example:

```
DELIMITER //
CREATE PROCEDURE GetEmployeeDetails (
  IN p_employee_id SMALLINT(4),
  OUT p_first_name VARCHAR(30),
  OUT p_last_name VARCHAR(30)
)
BEGIN
  SELECT first_name, last_name
    INTO p_first_name, p_last_name
    FROM employees
    WHERE id = p_employee_id;
END; //
DELIMITER ;
```

Here, p_first_name and p_last_name are OUT parameters. The procedure retrieves the first name and last name of the employee with the given ID and stores the results into the p_first_name and p_last_name variables. After the procedure is called, you can use the values of p_first_name and p_last_name in your code.

```
CALL GetEmployeeDetails(111, @p_first_name,
@p_last_name);
SELECT @p_first_name, @p_last_name;
```

1.CALL GetEmployeeDetails(111, @p_first_name, @p_last_name);
This line is calling a stored procedure named GetEmployeeDetails. The procedure is being called with three parameters:

- 111 is an input parameter, presumably representing an employee ID.

- @p_first_name and @p_last_name are output parameters. These are variables that the procedure will fill with data. The @ symbol indicates that these are user-defined variables.

2. SELECT @p_first_name, @p_last_name;

This line is retrieving the values of the @p_first_name and @p_last_name variables. After the GetEmployeeDetails procedure is called, these variables should hold the first name and last name of the employee with ID 111.

MODIFY or DROP existing stored procedure.

Once created, it is not possible to directly modify a stored procedure. Instead, drop the function and then recreate it with a new definition.

```
DROP PROCEDURE procedure_name;
```

or

```
DROP PROCEDURE IF EXISTS procedure_name;
```

IF EXISTS: This optional clause ensures that the database will not throw an error if the specified function does not exist. If the procedure exists, it will be dropped; if it doesn't, no error will be raised.