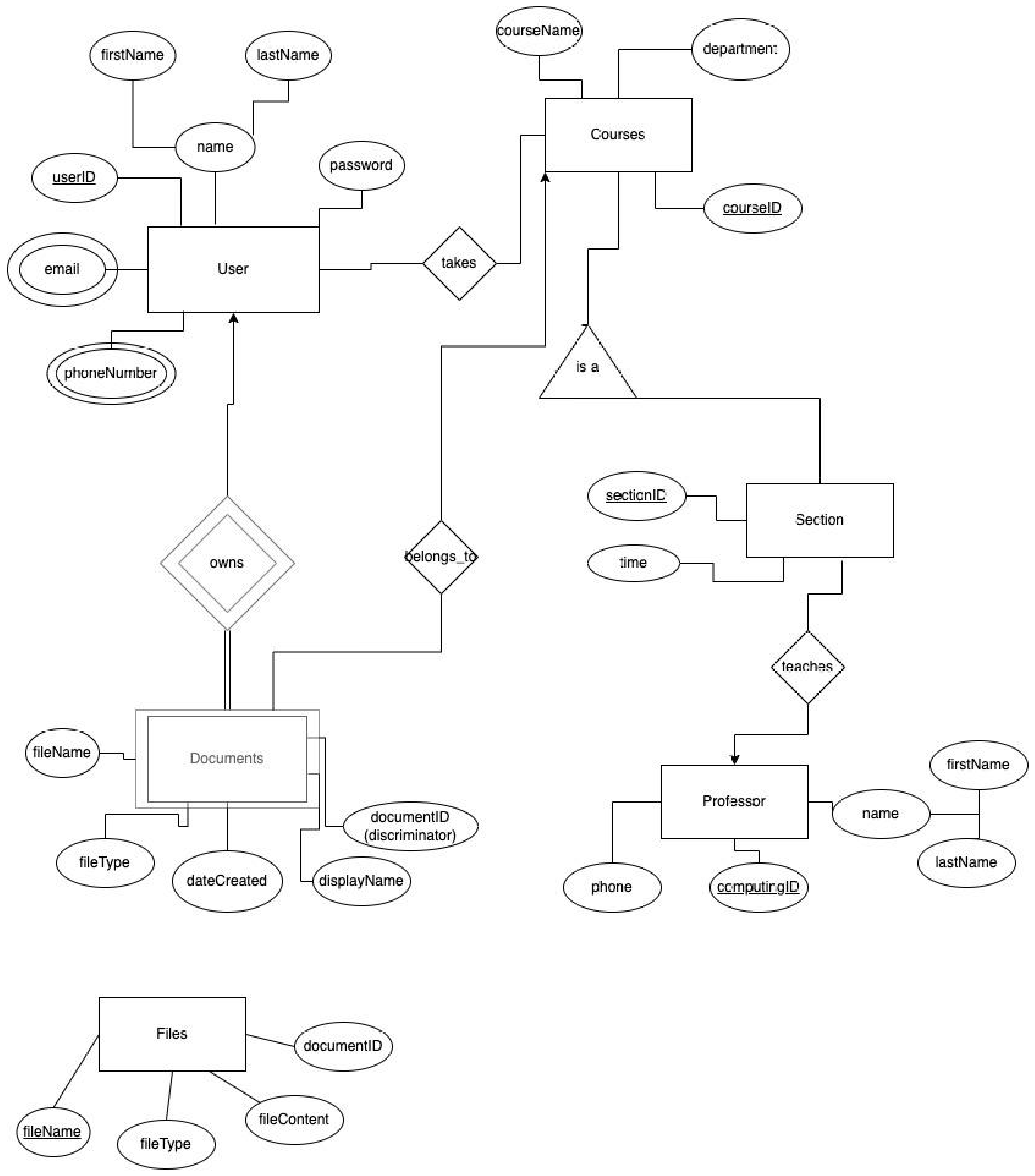
By: DB DADDIES
Names: Cameron Mukerjee (crm6zg), David Tran (dqt5vt), Sid Chauhan (ssc9yd), Kabir Menghrajani (km5qte), Ruthvik Gajjala (rrg5kq)

Final Deliverable - Report

Database Design

Final ER Diagram:



Final Tables:
User(userID, password, firstName, lastName)
Courses(courseID, courseName, department)
Professor(computingID, firstName, lastName, phone)
Document(documentID, userID, displayName, fileName, dateCreated)
File(fileName, fileType, fileContents, documentID)
User_email(user_id, email)
Section(courseID, sectionID, time)
takes(userID, courseID)
teaches(sectionID, computingID)
belongs_to(userID, documentID, courseID)
User_phone(user_id, phone)

Database Programming
Database host: **XAMPP, phpMyAdmin**
App host: **localhost**

Steps to deploy and run project:

1) **Clone the project repository located at https://github.com/trandavidq/4750-f21**
   a) **Make sure repository is cloned into htdocs directory within XAMPP**
   b) **Source code is also zipped up in collab submission**
2) **Set up the database using the Notemates.sql file found in the repository**
3) **Open up XAMPP and navigate to "Manage Servers"**
4) **Click "Start All" to start MySQL Database, ProFTPD, and Apache Web Server**
5) **Open a web browser and navigate to "http://localhost/4750-f21/"**
   a) **Log-in from this page, and use the application**

Advanced SQL commands discussion:

In our implementation, we have incorporated three advanced SQL commands to manage the control and ease of development for our application. In particular, we have chosen to use a trigger, stored procedure, and check in our database, each serving a particular purpose specific to the requirement of our app.

(1) First, we have included a trigger that communicates with changes to the "takes" table we have included. The courses table holds all the courses that are offered at UVA, and enables end users of our application to create their own course list from this table, which is then stored in "takes". In addition, they are also able to upload relevant documents associated with specific courses from their course list. The purpose of the trigger is to remove documents that are no longer needed by the end user. When a user removes a course from the "takes" table, all documents associated with that course are then deleted from the Documents table. This enables the user to have quick deletion of Documents, and mimics the real world scenario of when a student is done taking a course. Thus, the features that are heavily reliant on this trigger are the feature that allows for the tracking of the list of courses that a student is taking, as well as the feature

that allows for document upload/download, which is the centerpoint of our application. As mentioned above, the trigger reflects the database by communicating with multiple related tables and executing commands to mimic the real world; in particular, the tables that are altered are the "takes" table and the Document table on execution of this trigger.

(2) Originally, we were going to add a check to our application. After writing the necessary advanced SQL command, we realized that MySQL does not support checks. However, we have decided to include our code as well as the intention behind the check we would have incorporated. The check command that limits the possible user input when creating a new user account on the login/user creation page. The command checks that the user's input has more than 1 character for the first and last name. It also checks to make sure the password has a length greater than 6. The change reflected in the database ensures a more secure application, since the password isn't too short, and the names are legitimate (there are no real names that are less than 2 characters long).

(3) The final advanced SQL command we included in our app was a Stored Procedure command. This command is used to update the User"s first name and last name. It acts as a function that takes in a first name and last name for a particular user with a designated UserID. This procedure command is particularly useful because it allows for the common command used in our application to be stored and reused pretty easily within our database.

(1)

```
CREATE PROCEDURE `updateUser`(IN `inputUserID` INT(11), IN `inputFirstName`
VARCHAR(15), IN `inputLastName` VARCHAR(15)) NOT DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER UPDATE User SET firstName = inputFirstName, lastName =
inputLastName WHERE userID = inputUserID;
```

(2)

```
INSERT INTO User (password, firstName, lastName)
VALUES (hashedPassword, inputFirstName, inputLastName)
CHECK LEN(inputFirstName)>1 and LEN(inputLastName)>1 and LEN(password>=6);
```

(3)

```
CREATE TRIGGER `removeDocs` AFTER DELETE ON `takes`
FOR EACH ROW BEGIN
DELETE FROM belongs_to WHERE userID = old.userID and courseID = old.courseID;
DELETE FROM Document WHERE documentID NOT IN (SELECT documentID FROM
belongs_to);
END
```

Database Level Security
Security set for: **End User**

Access control discussion:

Access controls in our application are set via privileges for users. Upon creation of a new user their privileges are set to allow for specific data manipulations within the database, such as searching, inserting, and deleting certain data based on the table. College student's make up the majority of our desired user base, as such they should all have the same privileges. Thus, implementing a Mandatory Access Control model is adequate as all privileges are constant and don't change for users. Privileges that include manipulating tables and granting access controls to other users are not given to users of the application. This form of access control of our application keeps users from unauthorized access and prevents tampering with the database.

SQL commands for privileges:

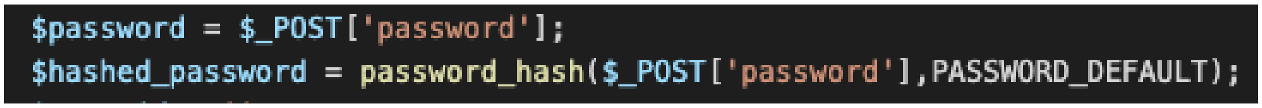
```
grant all on belongs_to to Student;
grant all on Courses to Student;
grant all on Document to Student;
grant all on File to Student;
grant all on Professor to Student;
grant all on Section to Student;
grant all on takes to Student;
grant all on teaches to Student;
grant all on User_phone to Student;
grant all on User_email to Student;
grant all on User to Student;
REVOKE DELETE,INSERT ON Courses FROM Student;
REVOKE DELETE ON User FROM Student;
REVOKE ALL ON Professor FROM Student;
REVOKE ALL ON teaches FROM Student;
```

Application Level Security
Discussion:

On the application level we have implemented password hashing to protect the passwords of users. We used php's built-in password_hash() method with the PASSWORD_DEFAULT option, which uses the bcrypt algorithm to encrypt the inputted password. We also used prepared statements for more efficient execution of queries, but this also had the important security benefit of preventing SQL injection attacks. Prepared statements treat inputted data as a parameter and not part of the actual SQL statement, which defends against those kinds of attacks. Finally, we used Session statements to allow for passage of user data between pages so that the logged-in user could remain in their own "session" with consistent data. The security advantage in using Session

statements is that they can be used to check if a user is logged in, so the database cannot be changed or accessed without a logged-in user. This is accomplished by sending the user back to the login page if it is detected that the session tracking the logged-in user is empty.

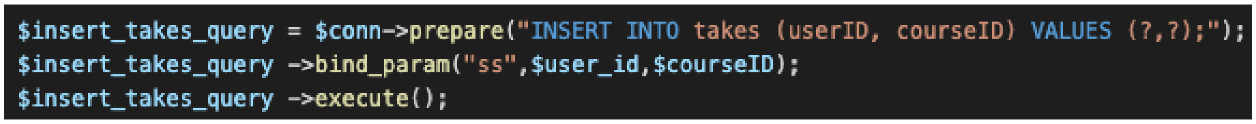
Code snippets:
Password Hashing



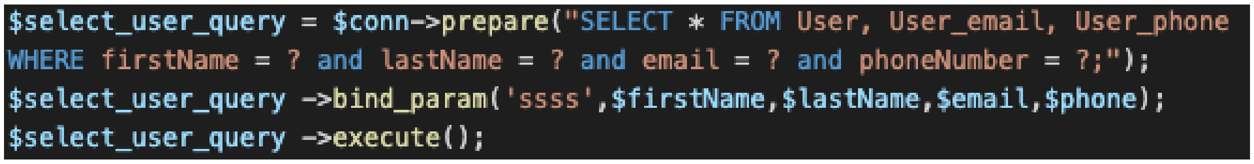
```
$password = $_POST['password'];
$hashed_password = password_hash($_POST['password'],PASSWORD_DEFAULT);
```

password_hash method used to hash password from form and store it in hashed_password

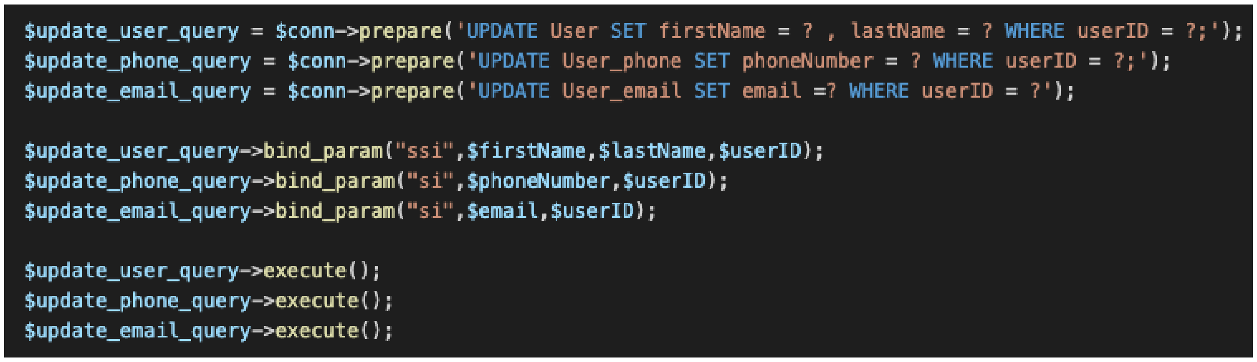
Prepared Statements



```
$insert_takes_query = $conn->prepare("INSERT INTO takes (userID, courseID) VALUES (?,?);");
$insert_takes_query ->bind_param("ss",$user_id,$courseID);
$insert_takes_query ->execute();
```



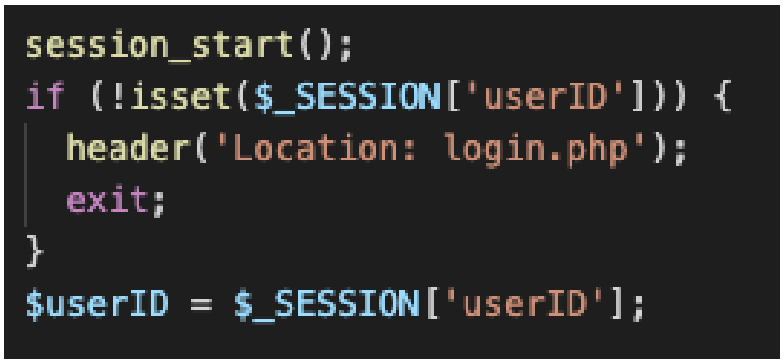
```
$select_user_query = $conn->prepare("SELECT * FROM User, User_email, User_phone
WHERE firstName = ? and lastName = ? and email = ? and phoneNumber = ?;");
$select_user_query ->bind_param('ssss',$firstName,$lastName,$email,$phone);
$select_user_query ->execute();
```



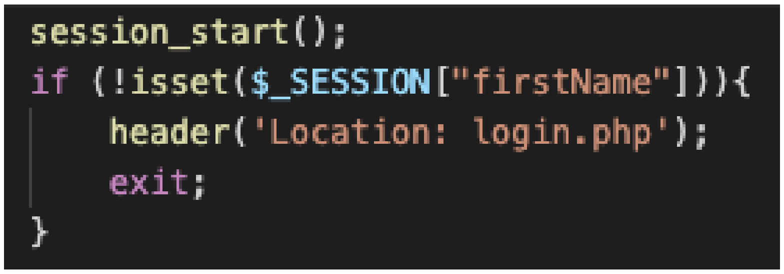
```
$update_user_query = $conn->prepare('UPDATE User SET firstName = ? , lastName = ? WHERE userID = ?;');
$update_phone_query = $conn->prepare('UPDATE User_phone SET phoneNumber = ? WHERE userID = ?;');
$update_email_query = $conn->prepare('UPDATE User_email SET email =? WHERE userID = ?');

$update_user_query->bind_param("ssi",$firstName,$lastName,$userID);
$update_phone_query->bind_param("si",$phoneNumber,$userID);
$update_email_query->bind_param("si",$email,$userID);

$update_user_query->execute();
$update_phone_query->execute();
$update_email_query->execute();
```

Examples of prepared statements done in 3 separate steps: 1) prepare query, 2) bind parameters to query, 3) execute statement

Sessions



```php
session_start();
if (!isset($_SESSION['userID'])) {
  header('Location: login.php');
  exit;
}
$userID = $_SESSION['userID'];
```



```php
session_start();
if (!isset($_SESSION["firstName"])){
    header('Location: login.php');
    exit;
}
```

If the SESSION's userID/firstName is not set, the user is sent back to the login page. This prevents the user from altering or accessing any part of the database while not logged in.