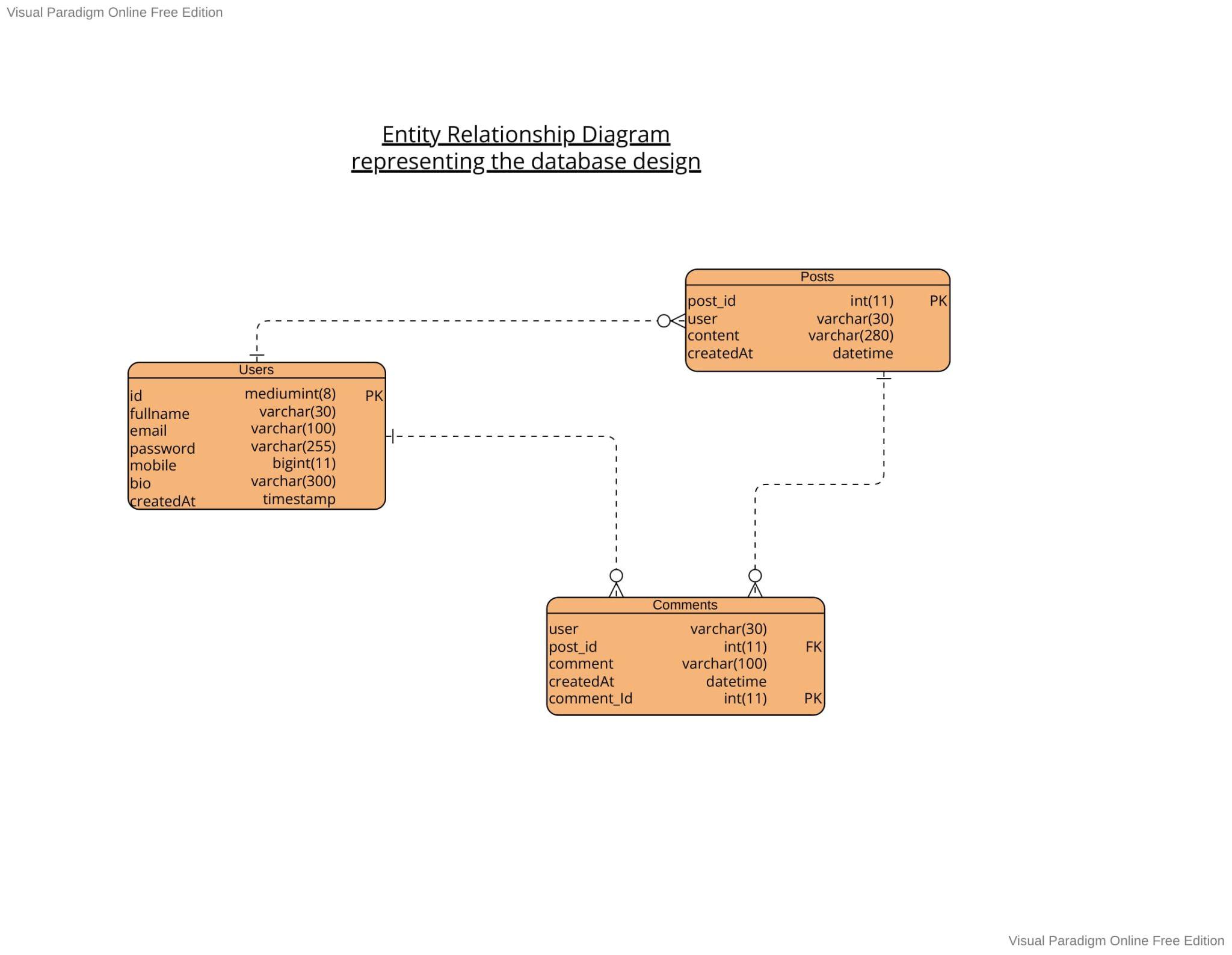
**Friendzone Design**



Sitemap

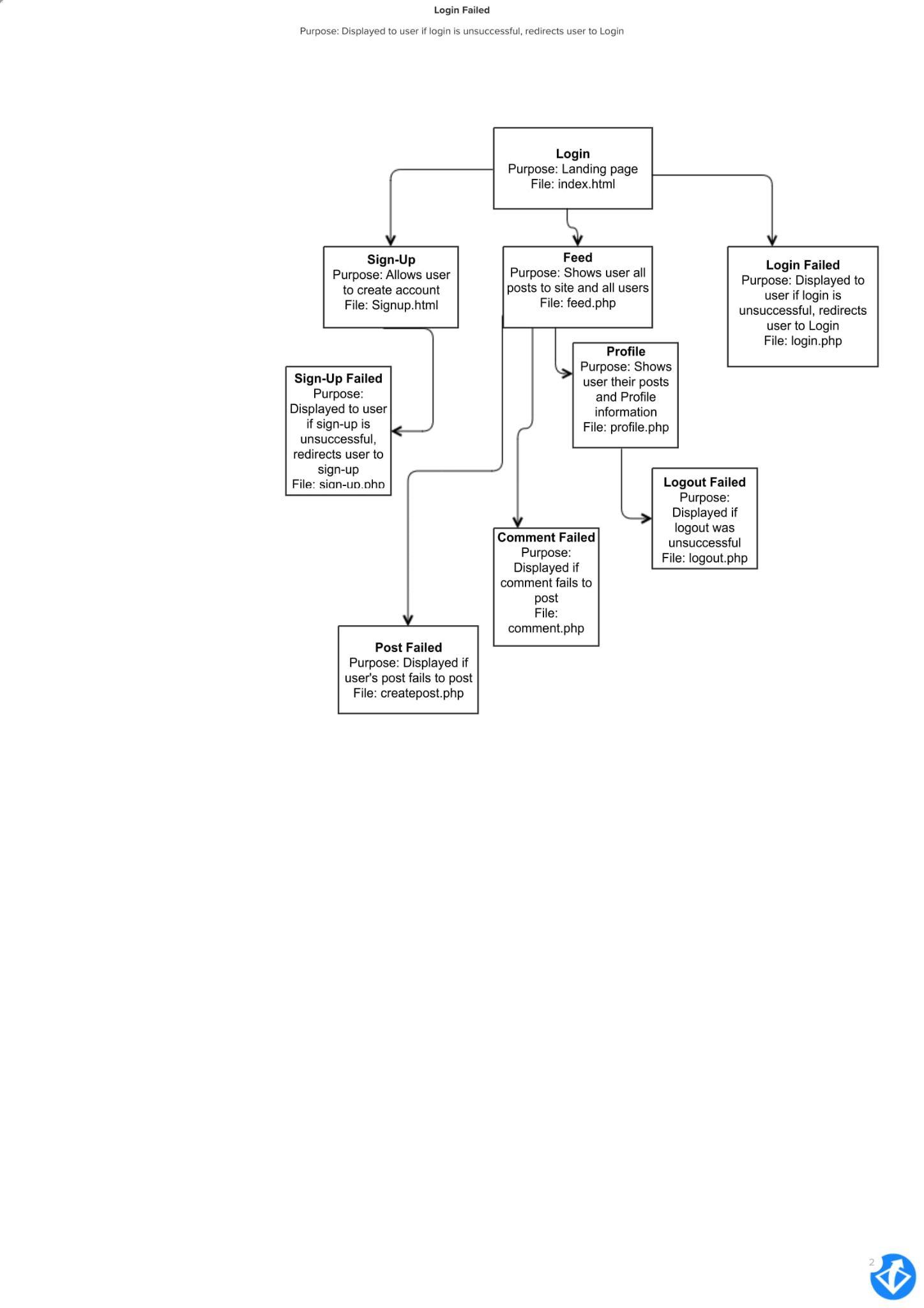


Table statements

CREATE TABLE ‘users’ (

id mediumint(8) NOT NULL AUTO\_INCREMENT PRIMARY KEY,

fullName varchar(30) NOT NULL,

email varchar(100) NOT NULL UNIQUE KEY,

password varchar(255) NOT NULL,

mobile varchar(12) DEFAULT NULL UNIQUE KEY,

bio varchar(300) DEFAULT NULL,

createdAt timestamp NOT NULL DEFAULT current\_timestamp())

CREATE TABLE posts (

post\_id int(11) NOT NULL AUTO\_INCREMENT PRIMARY KEY,

user varchar(30) NOT NULL,

content varchar(280) NOT NULL,

createdAt datetime NOT NULL DEFAULT current\_timestamp())

CREATE TABLE comments (

user varchar(30) NOT NULL,

post\_id int(11) NOT NULL,

comment varchar(100) NOT NULL,

createdAt datetime NOT NULL DEFAULT current\_timestamp(),

comment\_Id int(11) NOT NULL AUTO\_INCREMENT PRIMARY KEY,

FOREIGN KEY (post\_id)

REFERENCES posts (post\_id)

ON DELETE CASCADE)

Security Considerations

Attackers of a website are looking to gain private information that is stored in the database. They will do this through various methods but the main concern for this site is SQL attacks such as SQL injections.

The database holds the private information that needs to be protected from these attacks, the most important and easiest way to do this is to ensure that database dumps are not included in the website files on the server, this prevents the attackers from knowing the structure of the database. SQL injections are used by attackers to gain information from the database (Imperva, 2021), this is done putting SQL statements input fields on the page that access the database. If SQL queries are not carefully crafted more a concatenated query will be run and will return to the attacker the information that they are trying to find. Therefore SQL queries need to be arranged so that only one query can be run and any extra SQL queries inputted will return nothing. The simplest and most effective way to do this is by using parameterised queries. This is done by preparing the SQL statement without the parameters required for execution of the statement, and then binding the parameters with the statement when it is called to be run (SQL Server Team, 2022). Doing this ensures that the SQL injection is only viewed as a string in the statement.

An example of a non-parameterised query:

$statement = “INSERT INTO ‘users’(‘fullName’, ‘email’, ‘password’) VALUES($name, $email, $password)

$result = mysqli\_query($conn, $statement)

An example of the same query but changed into a parameterised query:

$statement = $conn->prepare("INSERT INTO `users`(`fullName`, `email`, `password`) VALUES (?,?,?)");

$statement->bind\_param("sss", $name, $email, $password);

$statement->execute();

$result = $statement->get\_result();

Another method of protection that can be used against SQL injections is data validation. Implementing data validation on all input forms can help to prevent SQL injections from reaching the database by cheking the data that has been typed in the input (Simplilearn, 2021). This method protects the database from any malicious or poor quality data. This is easy to implement, with the built in php functions:

is\_numeric

is\_array

is\_string

is\_bool

is\_scalar

Used when checking the data before returning a result.

The most important information to protect is the user’s password as most people, although its ill-advised, use the same passwords across multiple accounts and sites.Therefore password security is incredibly important. The front end of a site needs to be able to force a user to use a strong password to reduce the likelihood of the password being weak, and it also needs to be checked by the backend as attackers can disable the javascript forcing the strong password on the front end. Additionally, a password cannot be stored in a database in its raw form as if any attackers did gain access to the database all passwords would be compromised. The current standard for storing passwords is using password\_hash. password\_hash transforms the password into a string of numbers letters and symbols, previous methods that are no longer safe have also done this, but password\_hash is safe due to the fact it hashes and salts the password, and that every time a password is transformed using password\_hash it produces a different string (PHP). This can then be stored in the database safely and decoded with password\_verify.

An example of how password hash and password verify are used:

password\_hash($\_POST['pwd'], PASSWORD\_DEFAULT);

password\_verify($password, $row['password'];

References:

Imperva (2021) *What is SQL injection: SQLI attack Example & Prevention Methods: Imperva*, *Learning Center*. Imperva. Available at: https://www.imperva.com/learn/application-security/sql-injection-sqli/#:~:text=SQL%20injection%2C%20also%20known%20as,lists%20or%20private%20customer%20details (Accessed: January 8, 2023).

PHP (no date) *Password\_hash*, *php*. Available at: https://www.php.net/manual/en/function.password-hash.php (Accessed: January 8, 2023).

Simplilearn (2021) *PHP form validation: An in-depth guide to form validation in PHP*, *Simplilearn.com*. Simplilearn. Available at: https://www.simplilearn.com/tutorials/php-tutorial/php-form-validation#:~:text=Form%20Validation%20is%20a%20necessary,does%20not%20meet%20the%20requirements (Accessed: January 8, 2023).

SQL Server Team (2022) *How and why to use parameterized queries*, *TECHCOMMUNITY.MICROSOFT.COM*. Available at: https://techcommunity.microsoft.com/t5/sql-server-blog/how-and-why-to-use-parameterized-queries/ba-p/383483#:~:text=A%20parameterized%20query%20is%20a,to%20avoid%20SQL%20injection%20attacks (Accessed: January 8, 2023).