SMART PASSWORD VERIFICATION SYSTEM

A project submitted to the Bharathidasan University
in partial fulfillment of the requirements
for the award of the Degree of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

Submitted by

KARISHMA R
Register Number: 235114243

UNDER THE GUIDANCE OF
Dr. M. JAYAKKUMAR M.Sc.,M.Phil.,MCA.,Ph.D
Associate Professor

PG DEPARTMENT OF COMPUTER SCIENCE (S.F)
BISHOP HEBER COLLEGE (AUTONOMOUS)
(Nationally Reaccredited by NAAC at 'A++' Grade with a CGPA of 3.69 out of 4)
(Recognized by UGC as "College of Excellence")
(Affiliated to Bharathidasan University)
TIRUCHIRAPPALLI-620 017

OCTOBER – 2025

DECLARATION

I hereby declare that the project work presented is originally done by me under the guidance of **Dr. M. JAYAKKUMAR M.Sc.,M.Phil.,MCA.,Ph.D Associate Professor, PG Department of Computer Science (S.F), Bishop Heber College (Autonomous), Tiruchirappalli-620 017** and has not been included in any other thesis/project submitted for any other degree.

Name of the Candidate : KARISHMA R

Register Number : 235114243

Batch : 2023-2026

Signature of the Candidate

Dr. M. JAYAKKUMAR M.Sc.,M.Phil.,MCA.,Ph.D

Associate Professor,

PG Department of Computer Science (S.F),

Bishop Heber College (Autonomous),

Tiruchirappalli – 620017

Date:

CERTIFICATE

This is to certify that the project work entitled **"SMART PASSWORD VERIFICATION SYSTEM"** is a bonafide record work done by **KARISHMA R, Roll No: 235114243** in partial fulfillment of the requirements for the award of the degree of **BACHELOR OF SCIENCE IN COMPUTER SCIENCE** during the period **2023 – 2026.**

Place: Trichy. **Signature of the Guide**

PG DEPARTMENT OF COMPUTER SCIENCE (S.F)

BISHOP HEBER COLLEGE (AUTONOMOUS),
(Nationally Reaccredited by NAAC at 'A++' Grade with a CGPA of 3.69 out of 4)
(Recognized by UGC as "College of Excellence")
(Affiliated to Bharathidasan University)
TIRUCHIRAPPALLI - 620017

Date:

Course Title: Project **Course Code: U21CS6PJ**

CERTIFICATE

The Viva-Voce examination for the candidate **KARISHMA R, 235114243** was held on _____.

Signature of the Guide **Signature of the HOD**

Examiners:

1.

2.

ACKNOWLEDGMENT

I want to thank the Principal **Dr. J. Princy Merlin, M.Sc., SET**, **B.Ed., M.Phil., Ph.D., PGDLC, Principal, Bishop Heber College (Autonomous),** sincerely. He offered real encouragement along the way. Gave us steady support too. And that infrastructure was just excellent. All of it helped get the project done right. The one called Smart Password Verification System. His motivation never let up. Leadership like that builds a place for learning. For innovation even. Right through the whole development.

I would like to thank our Head of the Department, **Dr. G. Sobers Smiles David, M.C.A., M.Phil., NET. Assistant Professor, PG Department of Computer Science, Bishop Heber College (Autonomous), Tiruchirappalli** from the heart. Guidance came from him. Kept us on track. Monitoring was constant. Technical advice helped big time. Suggestions came when needed. Boosted the quality of everything, Accuracy too. In the work we did.

A special thanks to the Guide, **Dr. M. Jayakkumar M.Sc.**, **M.Phil., M.C.A., Ph.D., Associate Professor, PG Department of Computer Science, Bishop Heber College(Autonomous) Tiruchirappalli**. Dedication in guiding us. Expert supervision in every step. Encouragement that stuck around. Inputs were valuable. Expertise professional. Feedback constructive always. Built the base for hitting objectives. We succeeded because of that.

KARISHMA R

1% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography

Match Groups

- 5 Not Cited or Quoted 1%
  Matches with neither in-text citation nor quotation marks
- 0 Missing Quotations 0%
  Matches that are still very similar to source material
- 0 Missing Citation 0%
  Matches that have quotation marks, but no in-text citation
- 0 Cited and Quoted 0%
  Matches with in-text citation present, but no quotation marks

Top Sources

1% Internet sources
1% Publications
1% Submitted works (Student Papers)

Integrity Flags

0 Integrity Flags for Review

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

SYNOPSIS

People still talk about how tough it is to keep digital accounts safe these days. The “Smart Password Verification System” aims to fix that. It offers a secure way to check user logins on various platforms. Traditional passwords just do not cut it anymore. They get weak or reused a lot. Plus, attacks are easily. This project steps in with smarter checks and extra layers to handle those issues. The whole thing builds better password security in a few key ways. It checks strength right away for things like length and complexity. Users have to make unique ones too. Then, everything gets hashed for storage. That means even if someone grabs the database, they cannot pull out the real passwords. It is pretty solid against breaches. There are two main parts to the setup. The User Module lets people sign up and log in with those secure passwords. The Admin Module keeps an eye on logs and the database. Admins can spot misuse that way. The Interface stays simple. No one needs tech skills to use it. Organizations or just regular folks can pick it up quick. This system works great for schools, businesses, or anywhere needing safe logins. It beats older methods hands down. Risks drop for unauthorized entry. Trust in the whole process goes up too. Thing is, it combines smart rules for passwords with strong back-end protection. That tackles one of the biggest headaches in tech, which is authenticating users properly. The smart password verification system aims to make online logins a bit safer. It also makes them easier to handle day to day. Thing is, it tackles those usual issues like weak passwords or ones people reuse all the time. It does this by throwing in some smart checks. Plus extra layers of protection. Passwords get hashed securely right away. Then they are stored that way. So even if someone gets Into the database, the actual passwords remain hidden. The system watches login patterns too. There are two key parts to the project. One is the user module. That lets folks sign up and log in without worries. The other is the admin module. Admins can keep an eye on accounts

there. They manage records. And they spot potential problems like misuse. The whole Interface stays pretty straightforward. You do not need tech skills to get around it. Schools could use this system. Companies too. Or really any group handling logins. It cuts down on unauthorized access risks. And it helps build trust in online setups. In the end, the smart password verification system provides solid protection. It is Intelligent. And simple enough for everyday use. It boosts security for user accounts overall.

CONTENTS

CHAPTER-I
PROJECT DESCRIPTION

1.1 INTRODUCTION

These days, with everything online, keeping systems and personal info safe is a big deal. Old school passwords get hit hard by stuff like guessing games, phishing tricks, or just brute forcing through them. A smart setup for checking passwords tries to fix that, offering way better protection that you can actually count on. This whole project is about building something that goes beyond just saying Yes or No to a password. It looks at how strong the password really is, spots weird login tries, and blocks out anyone who shouldn't be there. You could add in things like one time codes, some encryption, or even watching how the user acts to make it all tighter for folks. Putting this smart password checker in place means companies and people keep their important stuff locked down. Plus, logins get quicker and less risky for everyone. The project really shows how mixing easy to use features with solid tech builds a login you can trus. The smart password verification system has a few layers of protection. It mixes old school password checks with some newer smart analysis. A user tries to log in. The system starts by matching the password they type against what is stored. It uses secure hashing for that. Then it looks at how strong the password really is. Things like length, how complex it gets, if it is unpredictable, and whether it showed up in any data breaches before. The password makes it through those checks. The system keeps going. It looks at the login behavior now. Stuff like what time it is, where the IP is from, how fast they type, and even the device fingerprint. Something seems off. Like a login from a whole new country or a bunch of failed tries that look suspicious. Then it kicks in extra steps. Maybe sends a one time password or asks for biometrics. All this process happens behind the screen with strong encryption. Algorithms like AES and SHA

hashing keep everything safe. No one can peek at the password or the behavior data. Even if they break Into the database. The system runs on machine learning too. It picks up on how each user normally acts over time. Gets better and smarter the more it runs. The workflow goes like this. User makes a login request. They enter the password and send it off securely. Next comes password verification. It checks with hashing and runs a strength analysis. The decision engine steps in. It accepts or rejects based on risk. Or ask for more verification if needed. Finally the response gets sent. Secure access if all good. Denied otherwise. And alerts go out if there is trouble.

1.2 EXISTING SYSTEM

Most password checks these days stick with the old-school text way of doing things. Users pick a username and password, and that gets saved in some database, usually with just basic encryption if any. Then when someone tries to log in, the system compares what they type against what's stored there. Matches up, and they get in. Doesn't match, tough luck, no access. A lot of setups just depend on that one layer, ignoring how strong the password really is or watching for weird login attempts. People tend to go for easy or reused passwords, which leave accounts wide, open. Sure, some places make you use a certain length or mix in special characters, but that's it. They miss out on better stuff like checking password strength right then and there, or sending one-time codes, or even tracking how users normally act. Most systems out there handle password checks in a pretty basic way. They just hash or encrypt the password once and store it. Then when you log in, they compare that with stored thing. Nothing smart about spotting weird behavior or checking if the password is actually safe. It accepts it and moves on, without caring if its weak or out there somewhere. One real issue comes with how these old setups keep all the user info in one big central spot. Hackers love that kind of target. Break in once, grab the database, and suddenly tons of usernames and passwords are out in the open. Plus they rarely watch for too many wrong tries or quick guesses. Brute force stuff can go on forever, and no one notices till damage is done. These systems fall short on phishing too. Password matches fine, but it has no clue if its the real person or some thief using stolen details. So everything depends on users picking good passwords and remembering them. The system does not really step in to help keep things secure**.**

1.2.1DISADVANTAGES:

1. Weak Security: Traditional security setups just aren't that solid. They rely on basic password matches alone. So attackers can crack them pretty quick using brute force stuff or dictionary tricks.

2. Lack of Monitoring: People make it worse on their end too. They go for weak passwords all the time. Or ones that are way too easy to guess. And yeah they often reuse the exact same one across different sites. That really ramps up the odds of getting hacked.

3. Easy to Bypass: You don't get any live monitoring in these systems either. Things like a bunch of failed login tries or logins at odd hours just slip by. Nothing tracks that kind of activity right.

4. High Maintenance: Resetting passwords feels shaky too. The whole recovery process has holes. Hackers exploit that without much trouble.

5. Phishing Attacks: All kinds of attacks hit these setups hard. Phishing scams snag your login details easy.

1.3 PROPOSED SYSTEM:

People still talk about how regular password systems fall short sometimes. This new smart verification setup aims to fix that. It checks if the password you type is right. But it goes further. It looks at how strong the password is too. Plus it spots weird login tries. And it layers on extra security in a few ways. One part handles password strength right as you enter it. It checks for complexity and if it's unique enough. Then there's multi-factor stuff. Like sending an OTP or confirming by email. Or even using biometrics for another check. Passwords get stored encrypted. Real secure methods to keep hackers out. It watches for odd patterns in logins. Blocks anything that seems off. This smart password setup is meant to be pretty smart. It adapts to things and stays easy for people to use. Rather than sticking to one kind of protection. It pulls together a bunch of solid methods to keep users safe. And data too. The thing is. It does not sit around waiting for trouble to show up. It pushes hard to stop threats and spot them early. Before any real damage gets done. When someone goes to sign in. The system looks beyond just matching the password. It watches how the whole attempt plays out. Stuff like the location of the request. The device involved. Even the way the typing happens. If anything seems off. It kicks in another check right away. Maybe an OTP sent to the phone on file. Or some biometric scan. Credentials for users get locked down tight. With encryption like AES for storage. And SHA for the checks. So even if a bad guy gets Into the storage. Nothing makes sense to them. Plus. Machine learning runs in the background. It picks up on user habits as time goes on. Each login helps it get better. Figuring out normal from weird. Admins have this dashboard thing built right in. It tracks logins as they happen. Shows the fails. The blocks patterns that look like attacks. Helps the team to jump on issues fast. In the end the whole idea here mixes tough security with no hassle. Users skip the extra steps most days. The

system does the hard part quietly. That mix of safe and simple. It really works for protecting stuff online these days.

1.3.1 ADVANTAGES:

1. Enhanced Security through Multi-Factor Authentication: This proposed system comes with some solid benefits. It really ramps up security in a big way. User accounts stay safe thanks to strong encryption all around. Multi-factor authentication adds another layer too. Activity monitoring watches over everything pretty closely.

2. Real Time Threat Detection: It handles unauthorized access without too many slip-ups. Suspicious logins pop up and get spotted immediately. Those potential Intruders, they just get blocked off before causing real trouble.

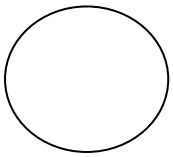
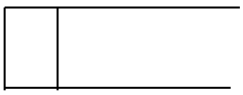
3. Creating strong and unbreakable passwords: The setup pushes users to pick strong passwords. It kind of guides them to ones that are hard to guess or crack open. You know the tough kind.

4. Secure and strong password: Risks from data breaches go down a lot with that secure storage in place. Verification steps help cut back on any password leaks that might happen.

CHAPTER-2
LOGICAL DEVELOPMENT

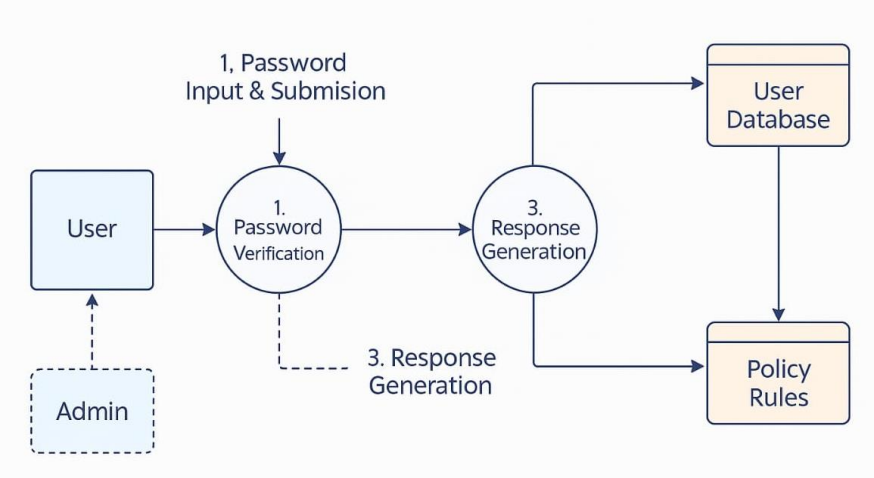
2.1 Data flow Diagram:

People still use this two-dimensional diagram to show how data gets processed and moved around in a system. It lays out each data source pretty clearly. Then it shows how those sources connect and work together toward one main output. When someone puts together a data flow diagram, they have to spot the external inputs and outputs first. They figure out the connections between them. Graphics help to explain how all that leads to the final result. Design teams find this kind of diagram useful. It lets them see data movement in action. Plus it poInts out spots that could use some fixing up. Data flow diagrams tend to have a few levels to them. Level 0 is basically the context diagram. It keeps things simple with just the overall view. You see the whole system as one big process there. It Interacts with stuff outside, like users or databases and other systems. Level 1 takes that and breaks it apart Into smaller bits. Those show the inner workings kind of. Data comes in, gets processed, then goes back out. You can go further with Level 2 or 3 if you want. They drill down Into even more steps for the details.

SYMBOLS	DESCRIPTION
	• It sends data through the system or takes data back from it is known as an **Entity.**
	• A **process** or task that is performed by the system.
	• A **data store,** it is place where data is been stored.
	• It indicates the movements of data between entities, processes and data stores.

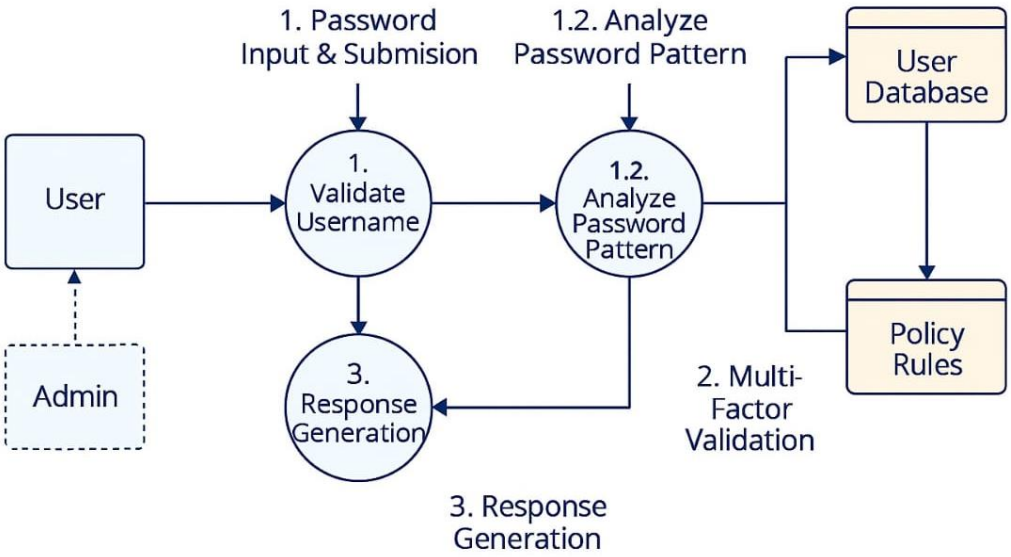
LEVEL 0:

At level 0, the SPVS, shows up as one big process. It connects with a few outside things. The user is the main one here. They put in their login stuff, like username and password, through some web app or mobile thing. SPVS takes that and checks it against the user database. It pulls up the stored info to see if it matches for authentication. At the same time, it sends the password over to a breach check API. That figures out if the password popped up in any data breaches people know about. SPVS also grabs some context details. Things like IP address, device type, relocation. It passes all that to a risk analysis engine or threat detection part. They work out a real-time risk score for the login try. From mixing the password check, the breach status, and that risk score, SPVS picks what to do next. It could let them in. Or ask for more verification, like multi-factor authentication. Or just block access completely. The result comes back to the user. There's the system admin too. They deal with SPVS to set up security policies. They look at logs, monitor activities through an admin Interface. So, this level zero diagram gives a broad view. SPVS acts as a central smart authentication setup. It links with users, admins, outside services, Internal databases. All to handle secure, adaptive password checks.


1, Password
Input & Submision
User
1.
Password
Verification
3.
Response
Generation
User
Database
Policy
Rules
3. Response
Generation
Admin

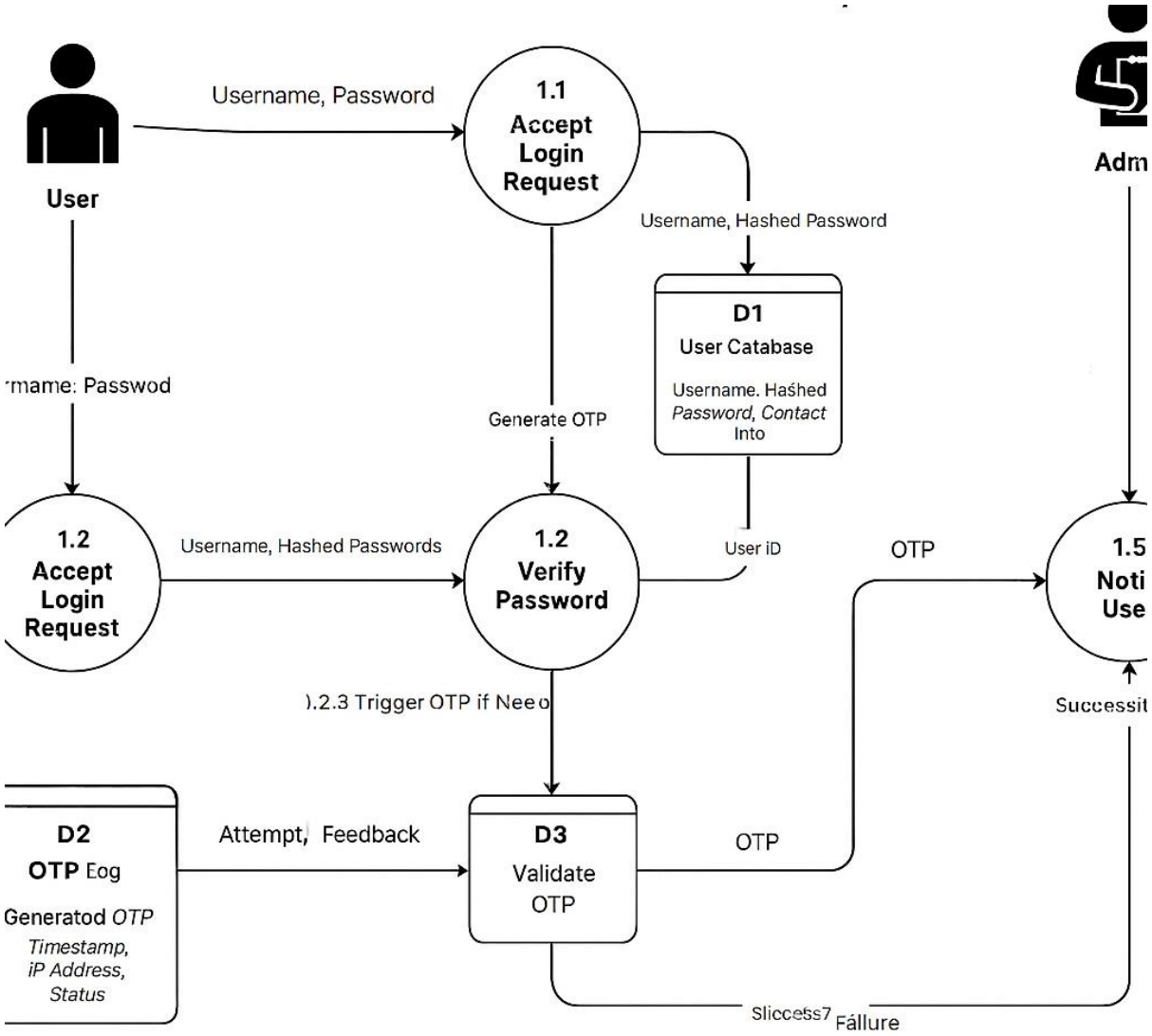
LEVEL 1:

At its core, the SPVS breaks down Into these main pieces. They all team up to manage user logins pretty smoothly. Someone tries to log in. The system routes that request right over to the Password Validation Module. This part looks over the password for basic stuff like format and how strong it is. It sticks to guidelines on length, character mixes, all that. If it clears those checks, the request heads to the Breach Detection Module. That one runs the password against known leaks from places like the Have I Been Owned API. You know, it's this outside service that tracks breaches. Meanwhile, the system pulls in some additional details behind the scenes. Things like the IP address, device info, even location data show up. All of it feeds Into the Contextual Risk Analysis Module. This module sorts out whether the login attempt fits the user's normal habits. It compares against old patterns, say typical spots or login times. Device traits play a role too. Right around then, the Behavioral Biometrics Module starts up. It digs Into typing habits and keystroke speeds. The poInt is to catch any odd inputs that might signal an impersonator. From there, the outputs from those modules go straight to the Risk Scoring Engine. The engine mixes it all together and spits out a risk score for the login try. Next, the Decision Engine grabs that score and decides what to do. It might just let access through no problem. Or it could demand multi-factor authentication. Sometimes it shuts the whole thing down. The Audit and Logging Module keeps track of everything that happens. Admins can pull up a dashboard to check on things. They watch failed logins, adjust rules as needed. This whole arrangement links the SPVS components in a solid way. It boosts authentication security quite a bit.


1. Password
Input & Submision
1.2. Analyze
Password Pattern
User Database
User
1.
Validate
Username
1.2.
Analyze
Password
Pattern
Policy
Rules
Admin
3.
Response
Generation
2. Multi-
Factor
Validation
3. Response
Generation

LEVEL 2:

A level two smart password setup really steps things up from that basic level one stuff. It boosts security a bit more, and kind keeps users from getting frustrated too. The first level just looks at simple things, like how long the password is or if it mixes in uppercase letters, numbers, special characters. This second level gets cleverer though. It throws in a strength meter that pops up live while you type, showing exactly how solid your password looks right then. That way, you can tweak it on the spot without all the trial and error. It even checks your choice against a big list of lousy ones, you know, those super common passwords or the ones leaked in hacks that too many folks still pick. So yeah, it blocks the weak ones before they stick. Then you got brute force guards, like temporary lockouts or slowdowns after a bunch of failed attempts. It handles two factor stuff as well, sending codes to your email or phone for one more layer. Passwords stay protected through hashing methods, say BCrypt with salt, so hackers can’t just crack them quick. Some versions force passwords to expire after time passes, and they stop you from recycling old ones. Overall, it balances being easy on users with really tight security. Pretty good fit for apps handling sensitive data.

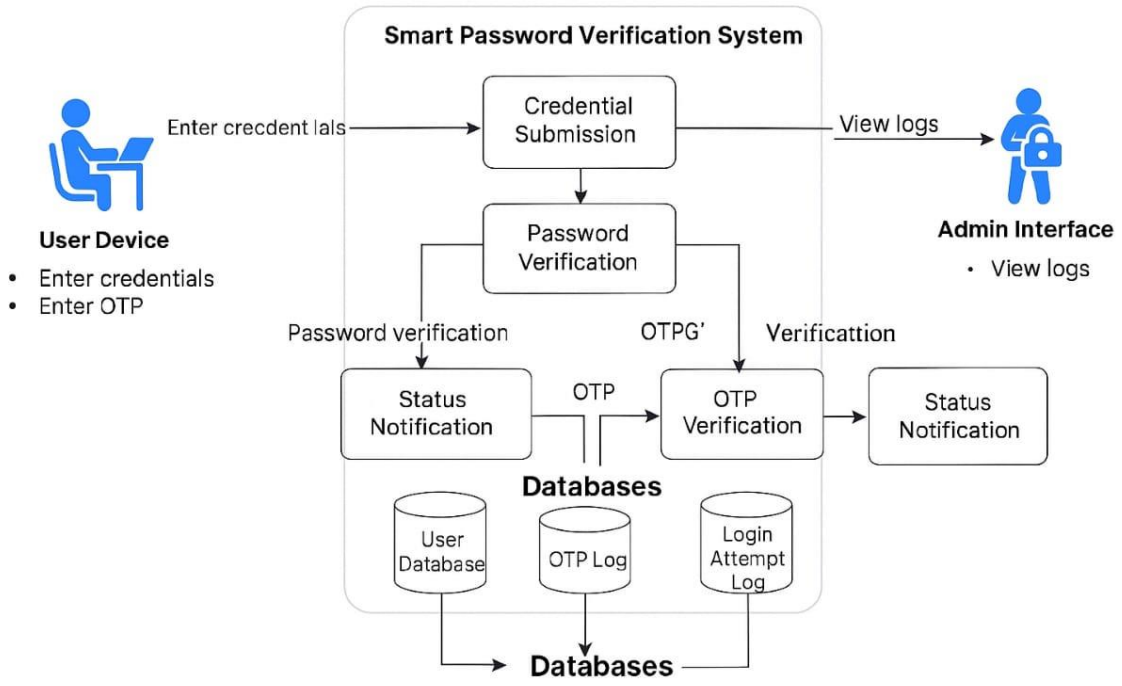

User
Username, Password
1.1
Accept
Login
Request
Username, Hashed Password
D1
User Catabase
Username. Haśhed
Password, Contact
Into
Adm
rmame: Passwod
Generate OTP
1.2
Accept
Login
Request
Username, Hashed Passwords
1.2
Verify
Password
User iD
OTP
1.5
Noti
Use
).2.3 Trigger OTP if Nee o
Successit
D2
OTP Eog
Generatod OTP
Timestamp,
iP Address,
Status
Attempt, Feedback
D3
Validate
OTP
OTP
Slicccss7
Fállure

2.2 ARCHITECTURAL DESIGN:

The smart password verification system has this architecture that follows a Multi - layered client-server setup. It keeps everything modular, Scales up pretty easy, and stays secure that way. There are three main parts to it, the user Interface layer, the application logic layer, and the data storage layer. Users Interact with the user Interface layer via a secure web app or mobile app. It manages the input forms for login or signup stuff. It does real-time checks on password strength too. And message feedback if the password meets the security rules or not. The application logic layer handles the business side. Detects brute-force attempts, and runs two-factor authentication when it's set up. So when a user submits a password, this layer checks it against a blacklist of common weak ones. It evaluates the complexity. Sees if it's reused from recent logins. Then it hashes the passwords using something solid like Bcrypt or Argon2. Adds a unique salt to it. Before storing it in the database. If two-factor is enabled, it generates only one at a time passwords sent via email or apps, and verifies them later. It stores user credentials all hashed and salted. Plus metadata like password change history, Login attempt counts, Account lock details, and two-factor tokens. They apply encryption for rest. And set up access controls to safeguard the sensitive data. Oh and logging for suspicious activity is in there. It helps detect Intrusions, and supports audits. The setup for this smart password verification system is all about getting security, scalability, and performance to work together nicely. It will be having different layers, and each one will handle its own job. They talk to each other without any hitches. The whole thing is modular, so you can tweak or add to one part. It won’t mess up everything else. Things move between layers using secure APIs. Data stays encrypted the whole way. Protocols like HTTPS and SSL over TLS keep it safe from anyone trying to grab or change stuff in transit. There’s also this session manager. It keeps user sessions going securely. Inactive ones get logged out automatically. That

cuts down on risks. To keep things running fast, they use caching for stuff that comes up a lot. Load balancing spreads out the work when a bunch of users hit it all at once. You can scale it up vertically or horizontally. Works for small operations or big companies. Error handling is built in solid. It logs failures, login problems, and database glitches. Sensitive info stays hidden though. Audit trails let admins track every key move. Helps with keeping things transparent and accountable. Maintenance wise, its straightforward. Update password rules, security methods, or the Interface. Existing stuff keeps working fine. The system stays flexible. It can shift with new standards as they come. In the end, this architecture pulls together layers, safe data flow, and efficient resource use. You get a reliable setup that’s easy for users.

Smart Password Verification System



CHAPTER- 3

DATABASE DESIGN

3.1 Data Dictionary

When it comes to software setups, managing data properly really matters for smooth operations, strong security, and overall reliability. The data dictionary gathers all the data elements from the system into one place. It explains their meanings, formats, types, constraints, and relationships with each other. In the Smart Password Verification System, this dictionary serves as a main reference for developers, testers, and other team members. It ensures consistent data usage throughout, avoiding any confusion or errors.

This dictionary helps people understand how user information, login credentials, and various system components are structured and managed. By describing each data element clearly and directly, the system follows its guidelines more effectively. It validates inputs accurately, maintains tight security, and speeds up data storage and retrieval. It also provides thorough documentation for future maintenance or enhancements. The Smart Password Verification System relies on robust data management to remain secure and precise.

The data dictionary offers several key benefits in this area. Clarity stands out as another strong point. Developers, testers, and even users can easily see what each data item represent. With fewer errors, the entire system performs more reliably. For validation and data integrity, it establishes clear guidelines. Constraints, formats, and validation rules for each element ensure only valid and secure information enters the system. Consider passwords as an example. Strict requirements prevent weak passwords that could be easily compromised.

In terms of security, it provides significant improvements. For a system handling sensitive items like passwords and security question responses, the

dictionary specifies encryption methods, proper storage practices, and access controls. This enhances overall protection and safeguards user privacy.

Field Name	Data type	Size	Description
User-Id	Integer	10	Unique identification number of each user.
User_Name	Varchar	50	Name chosen by user for login.
Email	Varchar	100	Registered email address of the user.
Password_Hash	Varchar	255	Encrypted password stored securely.
Salt_values	Varchar	255	Unique random values added to each password before hashing.
PasswordStrength	Varchar	20	Indicates strength of password.
Password_Change	DateTime	-	Records date and time of last password update.
Attempt_count	Integer	5	Tracks the number of failed login attempts
Account_Status	Varchar	20	Shows whether account is active, locked or suspended
Twofactor_Enable	Boolean	1	Indicates if it is active
OTP_Code	Varchar	10	Generate temporary one time password.
LastLogin_Time	DateTime	-	Record the last successful login
Role	Varchar	20	Defines user type
Activity_log_id	Integer	10	Reference Id linking to user activity log
IP_Address	Varchar	45	Stores IP address of last login attempts
Created_At	DateTime	-	Timestamp when user account was created

3.2 Table Design:

1. Secure Storage of Credentials:

- Passwords stay away from plain text storage every time. The data table holds hashed and salted versions of those passwords. This setup keeps things safe from theft or any unauthorized peeks.
- Take a hacker who somehow gets into the table. They still cannot reverse that hash to pull out the real password without a ton of work.

2. Password Strength and Policy Enforcement:

- The table keeps track of password strength ratings. Those might show up as weak, medium, or strong. This helps the whole system push security rules. Think minimum length requirements, special characters, or steering clear of common passwords people tend to pick.
- It can also log password history. That way user cannot just cycle back to old ones they have used before.

3. Authentication and Verification:

- When someone logs in, the system pulls credentials from the table. It uses them to check identity and make sure everything matches up.
- The setup tracks login attempts along with timestamps. Something like last login helps spot unusual patterns in activity.
- It even supports multi-layer authentication options. Those could include security questions or one time passwords for extra checks.

4. Account Security Monitoring:

- Fields such as failed attempts or last login give the system tools to act. It can lock accounts if too many failed tries pile up.
- The system notifies users about any suspicious login moves. Cool down periods come into play too. They stop brute force attacks from overwhelming things.

5. Support for Recovery and Multi Factor Authentication:

- Security questions and their answers sit right in the table. This makes password recovery straightforward when needed. Extra fields handle OTP secrets or tokens. Those support two factor authentication setups.
- The whole thing gets smarter than basic username and password checks. It builds in layers for better protection.

6. Data Tracking and Analytics:

- The system digs into the table for analysis. It measures users with strong passwords against those with weak ones.
- Patterns show up in failed logins. That reveals potential issues early. It spots users who have not bothered to update passwords lately. This prompts reminders or flags.

3.1.1 User table:

Field Name	Data Type	Description
User_id	Int (PK)	Unique id for user
User_name	Varchar (50)	Name must be unique
Password	Varchar (20)	Unique password
Email	Varchar (50)	Original email
Phone	Int	Valid phone no (optional)
Role	Varchar (15)	Role of the user
Created_at	DateTime	Account creation
Updated_at	DateTime	Latest update

3.1.2 Admin Table:

Field Name	Data Type	Description
Admin_id	Int (PK)	Unique admin id
Username	Varchar (20)	Unique login name
Password	Varchar (20)	Encrypted password
Email	Varchar (40)	Admin contact email
Created_at	DateTime	Account creation
Updated_at	DateTime	Latest update

3.1.3 Password Policy Table

Field Name	Data Type	Description
Policy_id	Int (PK)	Unique id each policy
min_length	Int	Minimum password length
Max_length	Int	Maximum password length
Require_uppercase	Boolean	Require uppercase letter
Require_number	Boolean	Require number
Require_special	Boolean	Require special character
Created_at	DateTime	Policy creation
Updated_at	DateTime	Latest update

3.1.4 Login Attempts

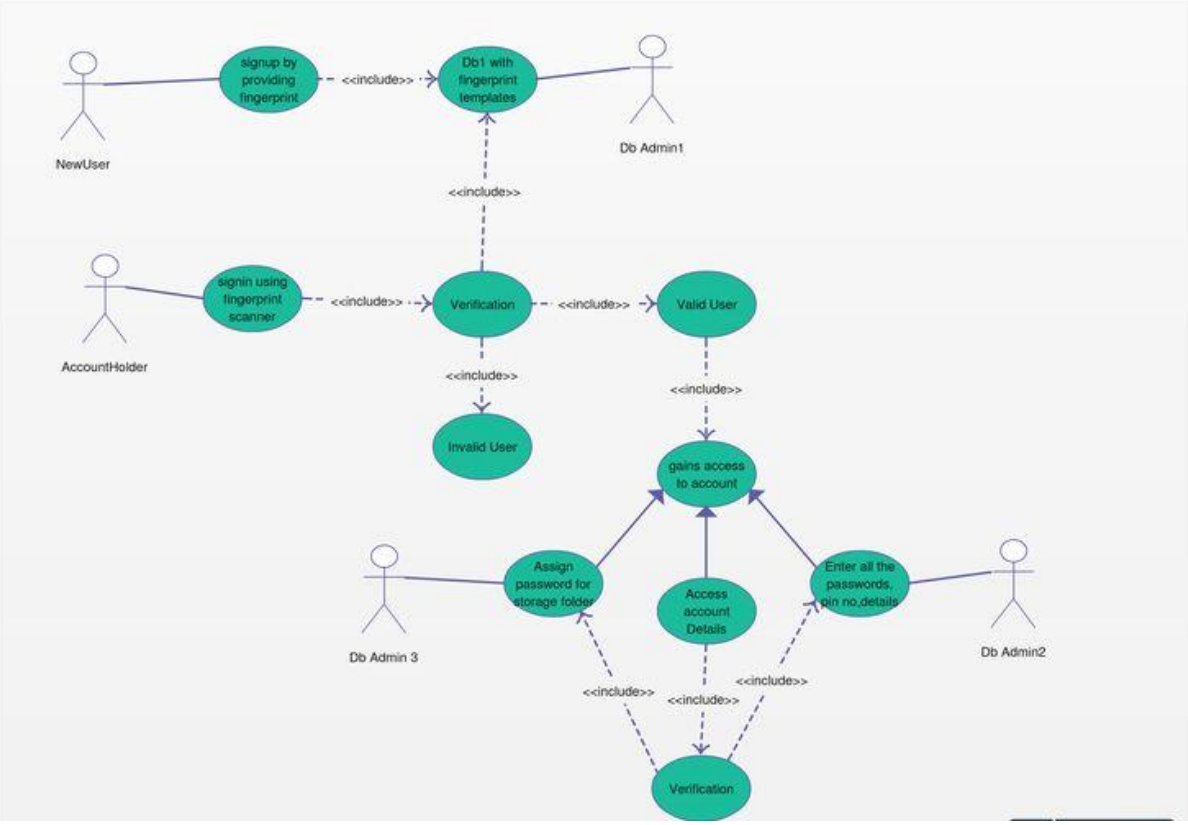
Field Name	Data Type	Description
Attempt_id	Int	Unique id for each login
User_id	Int	Reference user or admin
Attempt_Time	DateTime	Timestamp of login attempts
Success	Boolean	Login successful or failed
IP_Address	Varchar (50)	IP address or login attempts

3.1.5 Security Question

Field Name	Data Type	Description
Question_id	Int	Unique id for each question
User_id	Int	Reference users table
Question	Varchar (30)	Security question
Answer	Varchar (200)	Encrypted answer

3.3 Relationship Diagram:

The smart password verification system works through a few main parts that link up to handle user stuff safely, like managing accounts and logins. Users sit at the center of it all. They set up accounts, sign in, swap out passwords when needed. One user ends up with a bunch of login records tied to them, all kept in the login log part. That way, you can follow every try at getting in, whether it worked or not. Users connect to password history too. It holds onto old passwords to stop anyone from circling back to the same one. Thing is, its a one to many setup, since folks build up a list of past ones over time for better protection. Two factor authentication hooks right Into each user on a one to one basis. Nobody gets more than one going at once, which layers on that extra check to keep things tight. Then there’s the activity log catching everything users do, from login tries to password switches or account tweaks. It ties back to users in a one to many way, because one person racks up plenty of those entries. Admins oversee the whole crowd of users and what they're up to. A single admin handles lots of users, so again one to many there. They get eyes on all the logs too, controlling access as needed. In the end, user info, those logs, histories, details, all land secure in the data storage layer. It’s the core database pulling everything together; making sure data stays safe, whole, and easy to reach when it counts.


signup by providing fingerprint
<<include>>
Db1 with fingerprint templates
Db Admin1
NewUser
<<include>>
signin using fingerprint scanner
<<include>>
Verification
<<include>>
Valid User
AccountHolder
<<include>>
<<include>>
Invalid User
gains access to account
Assign password for storage folder
Access account Details
Enter all the passwords, pin no,details
Db Admin 3
Db Admin2
<<include>>
<<include>>
<<include>>
Verification

CHAPTER 4
PROGRAM DESIGN

The Smart Password Verification System gets built with a modular programming setup. Thing is, each piece of the program handles its own job. That setup boosts security a bit, adds some flexibility, and makes maintenance simpler overall. At its core, the system has two main modules. One is the User Module. The other is the Admin Module. They get backed by a solid backend and a database that keeps things secure.

User Module: It comes first. This part deals with everything users do on their end. It covers registration, where you put in details like username, email, password. The system checks the password right away for length and how complex it is. It also makes sure it is not already taken.

Password checking: Then there is the password strength checker. It looks at what you type in real time. If it seems weak, it suggests tweaks to make it better. Password hashing happens next. The system turns the password into a hashed version. It uses something like Bcrypt or Argon2, along with a unique salt. That way, it stays safe in the database. For login, it verifies what you enter against the stored hash. If you mess up too many times, it blocks you out. Two-factor authentication adds another layer. It sends an OTP to your email or app. You need that for extra checks. Password recovery lets you reset if you forget. It uses secure links to verify you.

Admin Module: It handles the oversight side. Admins use it to monitor and control user actions. They can view accounts, edit them, or deactivate if something seems off. Login monitoring tracks all attempts. Both the ones that work and the failures from every user. Activity logging keeps records of events. Like password changes, failed logins, lockouts. Everything gets noted.

Security Management: Security policy management lets admins update rules. They can change password requirements, set login limits, adjust encryption as the organization needs. The database and security layer form the backend. It stores user data in a secure way. Encryption covers sensitive information. Access controls apply to user and admin tasks alike. Logs stay around for audits. They help spot Intrusions too. Program flow goes like this in simple steps. First, a user opens the login page. Then the system checks password strength. It also sees if it follows policies. Then the password gets hashed. It goes Into the database. Access grants once everything checks out. Admins review those logs. They keep the whole system running smooth.

Design approach: It picks languages like Python, Java and PHP. Depends on what you go with.

- **Frontend:** HTML, CSS, JavaScript.
- **Backend**: Flask.
- **Database**: MySQL

CHAPTER 5
TESTING

Testing a smart password verification system is really key in the whole development thing. It makes sure everything works right, stays secure, and runs smooth before going live with actual users. You go through all the features and parts one by one. Each one does what it is supposed to do. No glitches, no weak spots, no slowdowns get in the way It handles logins okay. It pushes for strong passwords too. And it keeps user information safe from anyone. Only real users with the right details get through. Security gets a lot of focus during all this. That is the big goal here, protecting private stuff. They try out common attacks on it. Like brute force, where someone hammers guesses. Or dictionary stuff, using word lists. Even phishing tricks to fool people. The system spots them and shuts them down quick. They check the lockout too. After a few wrong tries, it freezes the account. Hackers cannot keep pounding away at passwords. Password resets get tested hard. Recovery options as well. Users get back in safe. Functionality is not all. Performance matters a bunch, usability too. Does verification fly fast even with tons of users piling on. It does not lag. For usability, the setup feels easy. Navigation is simple. Feedback comes clear when making passwords or signing in. Reliability tests hit real life scenarios like Network lags, Lots of logins at once, Server reboots. The system keeps going. No lost data. No crashes. Developers run unit tests, Then Integration ones, System level next, Acceptance at the end. All pieces fit together nice, Bugs pop up, they squash them fast, weaknesses too, Quality goes up, Strength improves.

1. UNIT TESTING:

Unit testing is all about checking the tiniest pieces of a system on their own. You make sure each one does what it is supposed to do. In something like a smart password setup, that means looking at stuff like the password check function or the hashing part. First off, you figure out what those small units are. Take the password validation, which handles length and if it is complex enough or avoids bad patterns. Then there is the hashing or encryption function for the password. Do not forget the comparison function that matches things up. And the multi-factor authentication check fits in there too. All of these count as units. Next, you come up with test cases for each one. That includes the good scenarios where passwords follow all the rules just right. On the flip side, you have the bad ones, like passwords that are way too short or missing that complexity kicks. Throw in edge cases too, you know, empty ones or super long strings or weird characters that nobody expects. Setting up the testing spot comes after that. You keep the module away from everything else in the system. If there are outside things like databases or APIs, you mock them out so they do not mess with the test. Now you run the tests. Go through each case one by one on the module. Keep track if it passes or fails. See to it that the validation accepts good passwords and kicks out the bad ones. The hashing should give the same output every time. And the comparison needs to match those hashed versions correctly. Look over the results when it is done. If everything passes, great, the module is solid. But if some fail, hunt down the problem, patch it up, and run the test again to confirm.

2. Integration testing:

Integration testing in a smart password verification system involves pulling together separate modules. Things like password validation, Hashing, Encryption, User authentication. You test them all as one unit to see if they function properly or not. The main goal here is to confirm that the modules Interact without hitches. Data moves smoothly from one part to another. No errors pop up. No security weaknesses either. Testers run through various scenarios that feel like everyday use. For instance, logging in with a correct password. Trying several failed login attempts in a row. Or going through password reset and recovery steps. All this helps check how well the modules manage inputs and outputs. Dependencies come Into play too, databases, APIs, External authentication services. You verify that whether it is connected without issues to the overall system. When problems show up, inconsistencies or failures or odd behaviors, you identify them. Fix what needs fixing. Then test again to make sure its sorted. This testing also covers security features. Password encryption stays solid. Access controls will be hold up during module Interactions. In the end, Integration testing confirms the systems full functionality. Its reliability and security too. All before deployment in a live setting. So components operate just as they should.

3. System testing:

It looks at the whole Smart Password Verification System all at once. You run through the full process to confirm it does what it's supposed to. That covers successful logins with good info. It also blocks bad attempts immediately. After several failed tries, account lockout kicks in. Password resets work as planned. This checks all the key requirements. The system acts right in real situations users deal with. End to end testing shows developers it's ready and functional. They can pass it on for actual use now. They handle password resets too. And they test different security setups. Performance tests show how the system deals with lots of users at once and many login tries happening together. Security reviews make sure passwords get encrypted right. They check for protection against unauthorized data. Data transmission stays safe as well. Usability checks prove the Interface is straight forward. Users can get around it without getting lost. Thing is, by the end of this testing, developers get a full picture of how the system stacks up against its requirements in actual use. Any leftover problems get spotted and fixed. Then it moves to acceptance testing. There, end users validate it for final okay and rollout.

4. Acceptance testing:

In acceptance testing, they look for what the end user sees. The idea is to make sure it hits all the requirements and works just like it should in everyday situations. This comes right before they roll it out for real. The main point stays on checking if its user friendly, dependable, and handles password stuff and logins without a hitch. Real people or folks from the client side jump in during this part. They try things out, you know, like setting up new accounts, signing in with the right info or messing it up on purpose, asking for password resets. They also see what happens with a bunch of failed tries at logging in. Does it lock the account after too many goes. Does it shoot off those verification emails quick enough. And all the while, the data stays secure, no leaks or weak spots. They poke at the Interface too. Is it clear, easy to get around, quick to respond when you click around. This testing covers everything for business needs and security. Safe ways to deal with data, modules talking to each other smoothly, performance that holds up steady. If the users and big shots give it the thumbs up, then the system gets the green light for going live in the real world. But if stuff crops up, it heads back to the devs for those last fixes before anything goes out.

CHAPTER 6
CONCLUSION

The smart password verification system appears to offer a robust method for user authentication in current digital setting Cyber threats continue to grow in number and Sophistication. Traditional password often lack the strength needed to counter them effectively. It aims to protect data integrity alongside user privacy in a reliable way. Design and development of the system emphasize efficient identity checks while upholding high security measures, Password strength gets assessed regularly, Login details undergo validation, and unauthorized entries face intelligent blocks. Testing covered multiple phases to verify smooth operation. Unit tests checked individual parts. Integration efforts combined components. System-wide reviews followed, along with acceptance trials. Each step confirmed that the whole functions as intended. Users encounter a straightforward interface. Messages stay clear. Navigation proves simple. Responses come quickly during checks. Developers benefit from a solid base for expansion. Options like two-factor methods, biometrics, or temporary codes could integrate easily. Applications span areas such as finance, learning environments, medical services, and large-scale businesses. Secure entry remains critical there. In the end, the Smart Password Verification System meets its aim of bolstering login safeguards without complicating daily use. It highlights how smart tech paired with sound design yields practical and dependable results. Fewer breaches and intrusions follow from this approach. The effort aids in fostering more secure online spaces overall.

KEY STRENGTH

1. Secure Credential Management:
   - Passwords stay safe because the system never keeps them in plain text form. It relies on hashing and salting methods to protect against any unauthorized access or theft attempts.

2. Password Strength Enforcement:
   - The setup helps users build strong passwords right from the start. This approach lowers the chances of accounts getting compromised in some way. Clear policies block out weak choices or ones that anyone could guess easily. They also push against reusing passwords from the past.

3. Multi-Layer Authentication:
   - This feature adds extra layers like security questions or OTP Verification. It steps up the protection level. Even if a password somehow leaks out, the system holds strong.

4. Monitoring and Account Protection:
   - This tool keeps an eye on login tries and times along with any failed attempts.

5. Centralized Data Management:
   - A main data table serves as the core for holding all user details in a secure spot.

6. User Convenience and System Intelligence:

- Security stays tight, but the system keeps things easy for users with recovery paths and tips on passwords. Admins get tools to review data and decide on ways to boost security overall.

CHAPTER 7

REFERENCE

7.1 Books

1. **"Security Engineering: A Guide to Building Dependable Distributed Systems" – Ross J. Anderson**.

2. **"Applied Cryptography: Protocols, Algorithms, and Source Code in C" – Bruce Schneier**.

3. **"Information Security: Principles and Practice" – Mark Stamp**

4. **"Introduction to Computer Security" – Matt Bishop**.

5. **"Computer Security: Principles and Practice" – William Stallings and Lawrie Brown (Latest Edition)**

6. **"Cybersecurity Essentials" – Charles J. Brooks, Christopher Grow, Philip Craig, Donald Short**.

7. **"The Art of Deception" – Kevin D. Mitnick & William L. Simon**

7.2 Websites

1. **OWASP (Open Web Application Security Project)** – https://owasp.org

   ➔ Offers guides on password security, authentication, and best practices.

2. **GeeksforGeeks –** https://geeksforgeeks.org

   ➔ Simple explanation and examples for password verification systems.

3. **Tutorials Point –** https://tutorialspoint.com

   ➔ Covers encryption, hashing, and secure login techniques.

4. **IEEE Xplore** – https://www.researchgate.net, https://ieeexplore.ieee.org

   ➔ Academic papers on smart password verification systems, multi-factor authentication, and security testing.

CHAPTER 8

APPENDIX

8.1 Source code:

App.py

```
from flask import Flask, render_template, request, redirect, url_for, session,
flash
import json, random, re, os
app = Flask(__name__)
app.secret_key = 'secretkey123'

# Load user data
def load_users():
if os.path.exists('users.json'):
with open('users.json', 'r') as f:
return json.load(f)
return {}

# Save user data
def save_users(users):
with open('users.json', 'w') as f:
json.dump(users, f, indent=4)



# Password strength checker
def password_strength(password):
if len(password) < 8:
return "Weak (Too short)"
if not re.search("[a-z]", password):
```

```
return "Weak (Add lowercase letter)"
if not re.search("[A-Z]", password):
return "Weak (Add uppercase letter)"
if not re.search("[0-9]", password):
return "Weak (Add a digit)"
if not re.search("[!@#$%^&*(),.?\":{}|<>]", password):
return "Weak (Add a special character)"
return "Strong"

@app.route('/')
def index():
return render_template('index.html')

@app.route('/register', methods=['GET', 'POST'])
def register():
users = load_users()
if request.method == 'POST':
username = request.form['username']
password = request.form['password']
color = request.form['color'].lower()



if username in users:
flash("Username already exists!", "danger")
flash(f"Hi {username}, your registration was successfully registered! Password
strength: {strength}", "success")
return redirect(url_for('register'))
strength = password_strength(password)
flash(f"Password strength: {strength}", "info")
```

```
users[username] = {"password": password, "color": color}
save_users(users)
flash("Registration successful!", "success")
return redirect(url_for('login'))

return render_template('register.html')

@app.route('/login', methods=['GET', 'POST'])
def login():
users = load_users()
if request.method == 'POST':
username = request.form['username']
password = request.form['password']

if username not in users or users[username]["password"] != password:
flash("Invalid username or password!", "danger")
return redirect(url_for('login'))



# Level 1 passed
session['username'] = username
otp = str(random.randint(1000, 9999))
session['otp'] = otp
flash (f"Your OTP is: {otp}", "info")
return redirect(url_for('otp_verify'))

return render_template('login.html')
```

```
@app.route('/otp', methods=['GET', 'POST'])
def otp_verify():
if 'otp' not in session:
return redirect(url_for('login'))

if request.method == 'POST':
user_otp = request.form['otp']
if user_otp == session['otp']:
flash("OTP verified successfully!", "success")
return redirect(url_for('security_verify'))
else:
flash ("Incorrect OTP!", "danger")
return redirect(url_for('otp_verify'))

return render_template('otp.html')



@app.route('/security', methods=['GET', 'POST'])
def security_verify():
users = load_users()
username = session.get('username')
```

```
if not username:
return redirect(url_for('login'))

if request.method == 'POST':
color = request.form['color'].lower()
if color == users[username]["color"]:
flash ("✓Access Granted! Welcome " + username, "success")
session.clear()
return redirect(url_for('index'))
else:
flash("✗Incorrect answer!", "danger")
return redirect(url_for('security_verify'))

return render_template('security.html')

if __name__ == '__main__':
app.run  (debug=True)
```

INDEX.HTML

```
<!DOCTYPE html>
<html>
<head>
<title> Triple Level Authentication </title>
</head>
<body style="text-align:center; font-family:Arial;">
    <h1>Welcome to Smart Password Verification System</h1>
    <a href="{{ url_for('register') }}">Register</a> |
    <a href="{{ url_for('login') }}">Login</a>
</body>
</html>
```

REGISTER.HTML

```
<!DOCTYPE html>
<html>
<head>
<title> Register </title>
<style>
body { font-family: Arial; text-align: center; background: #f2f2f2; }
input { padding: 10px; margin: 5px; width: 200px; }
button { padding: 10px 20px; background: #4CAF50; color: white; border:
none; border-radius: 5px; }
button:hover { background: #45a049; }
p.message  { color: green; font-weight: bold; }
p.error { color: red; font-weight: bold; }
p.info { color: blue; font-weight: bold; }
</style>
</head>
<body>
<h2>Register User</h2>

<!-- Flash messages -->
{% with messages = get_flashed_messages(with_categories=true) %}
{% if messages %}
{% for category, message in messages %}
<p class="{{ category }}">{{ message }}</p>
{% endfor %}
{% endif %}
{% endwith %}
```

```
<form method="POST">
<input type="text" name="username" placeholder="Username" required><br>
<input type="password" name="password" placeholder="Password"
required><br>
<input type="text" name="color" placeholder="Favorite Color"
required><br><br>
<button type="submit">Register</button>
</form>

<p><a href="{{ url_for('login') }}">Already have an account? Login</a></p>
</body>
</html>
```

LOGIN.HTML

```
<!DOCTYPE html>
<html>
<head>
   <title>Login</title>
   <style>
      body { font-family: Arial; text-align: center; background: #f2f2f2; }
      input { padding: 10px; margin: 5px; width: 200px; }
      button { padding: 10px 20px; background: #4CAF50; color: white; border:
none; border-radius: 5px; }
      button:hover { background: #45a049; }
      p.message { color: green; font-weight: bold; }
      p.error { color: red; font-weight: bold; }
      p.info { color: blue; font-weight: bold; }
   </style>
</head>
<body>
   <h2>Login</h2>

   <!-- Flash messages -->
   {% with messages = get_flashed_messages(with_categories=true) %}
     {% if messages %}
       {% for category, message in messages %}
         <p class="{{ category }}">{{ message }}</p>
       {% endfor %}
     {% endif %}
   {% endwith %}
```

```
    <form method="POST">
        <input type="text" name="username" placeholder="Username"
required><br>
        <input type="password" name="password" placeholder="Password"
required><br><br>
        <button type="submit">Login</button>
    </form>

    <p><a href="{{ url_for('register') }}">Don't have an account?
Register</a></p>
</body>
</html>
```

OTP.HTML

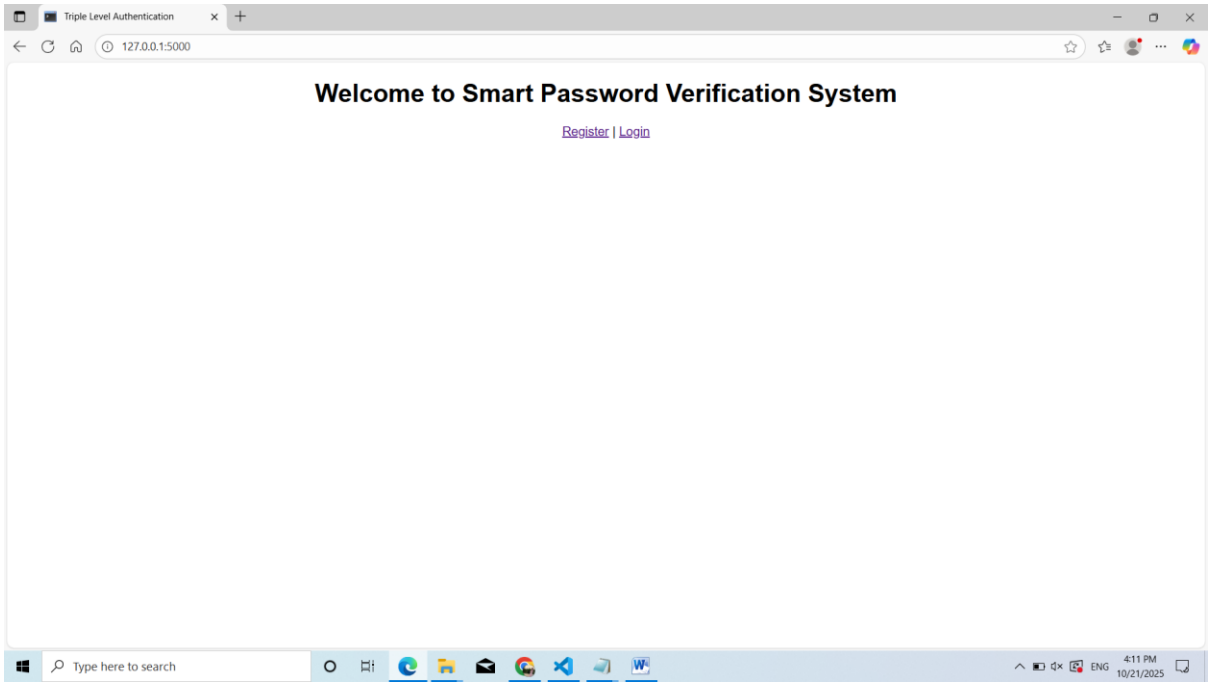
```
<!DOCTYPE html>
<html>
<head>
   <title>OTP Verification</title>
   <style>
      body { font-family: Arial; text-align: center; background: #f2f2f2; }
      input { padding: 10px; margin: 5px; width: 150px; }
      button { padding: 10px 20px; background: #4CAF50; color: white; border:
none; border-radius: 5px; }
      button:hover { background: #45a049; }
      p.message { color: green; font-weight: bold; }
      p.error { color: red; font-weight: bold; }
      p.info { color: blue; font-weight: bold; }
   </style>
</head>
<body>
   <h2>Enter OTP</h2>

   <!-- Flash messages: OTP and password strength -->
   {% with messages = get_flashed_messages(with_categories=true) %}
     {% if messages %}
       {% for category, message in messages %}
         <p class="{{ category }}">{{ message }}</p>
       {% endfor %}
     {% endif %}
   {% endwith %}
```

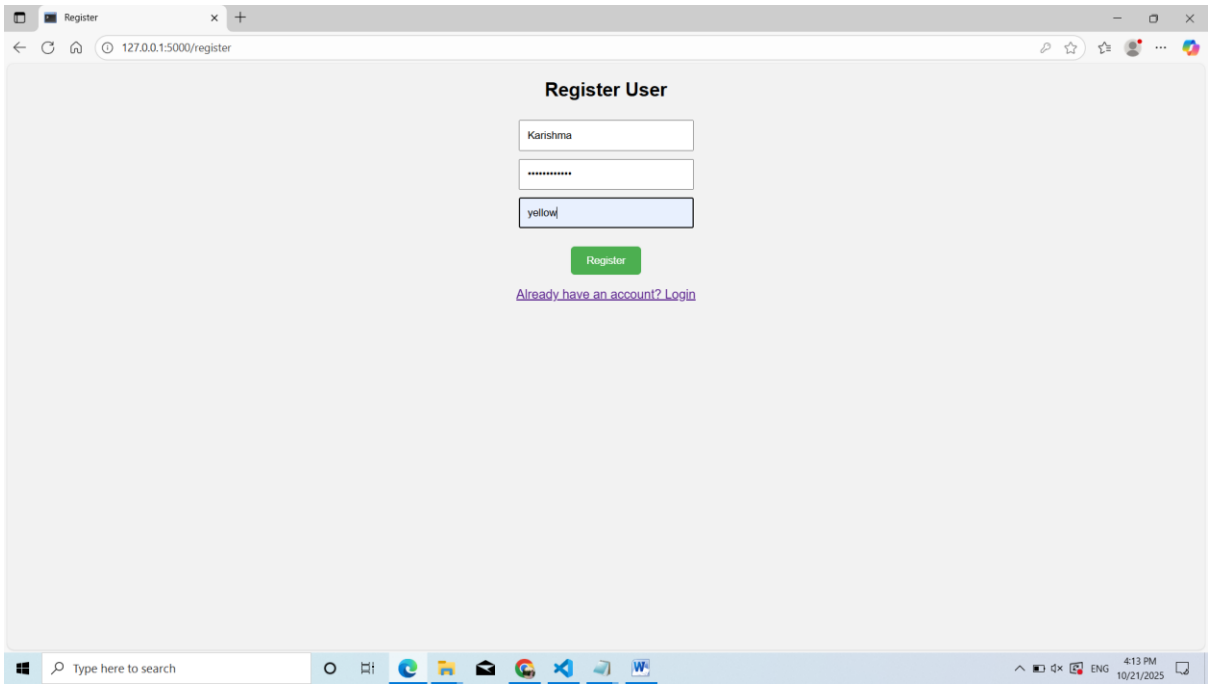
```
    <form method="POST">
        <input type="text" name="otp" placeholder="Enter OTP"
required><br><br>
        <button type="submit">Verify OTP</button>
    </form>
</body>
</html>
```

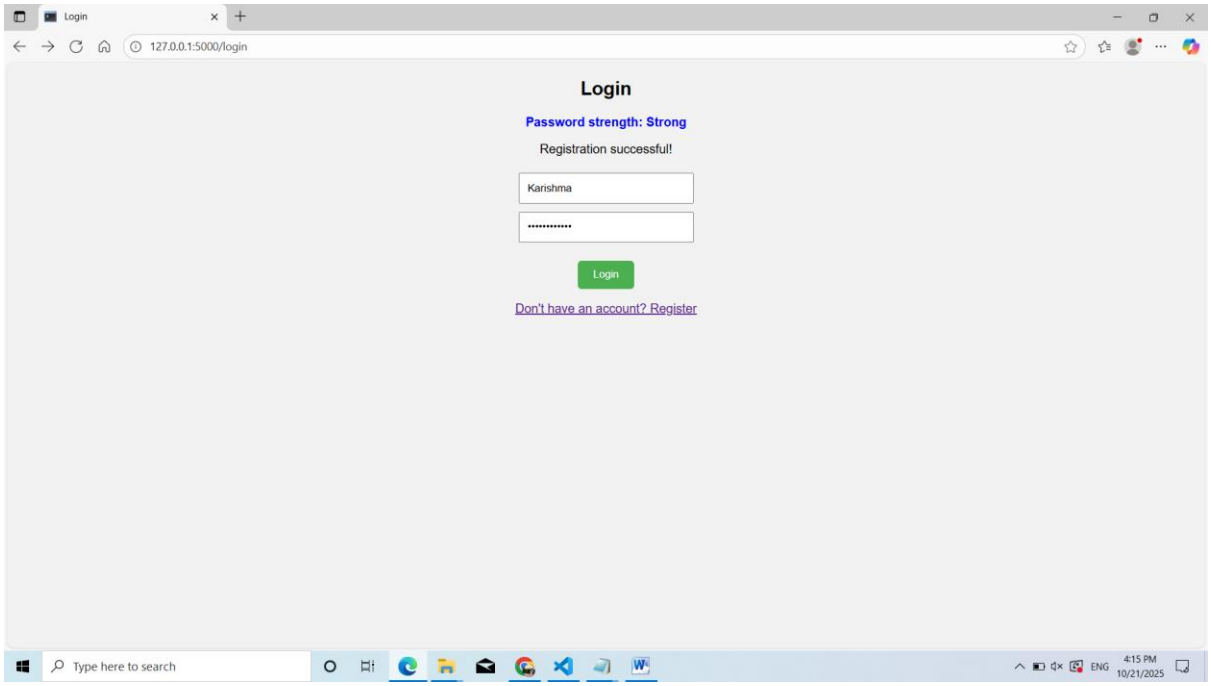
SECURITY.HTML

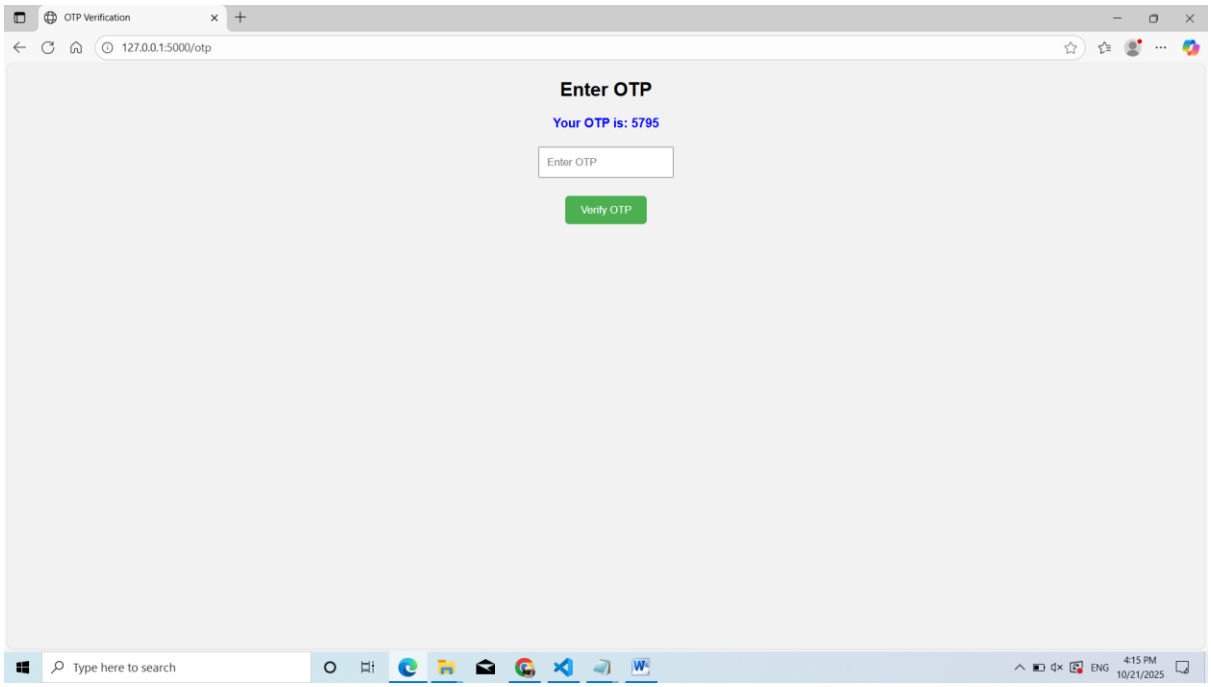
```
<!DOCTYPE html>
<html>
<head>
   <title>Security Question</title>
   <style>
      body { font-family: Arial; text-align: center; background: #f2f2f2; }
      input { padding: 10px; margin: 5px; width: 200px; }
      button { padding: 10px 20px; background: #4CAF50; color: white; border:
none; border-radius: 5px; }
      button:hover { background: #45a049; }
      p.message { color: green; font-weight: bold; }
      p.error { color: red; font-weight: bold; }
      p.info { color: blue; font-weight: bold; }
   </style>
</head>
<body>
   <h2>Security Question</h2>
   <p>What is your favorite color?</p>

   <!-- Flash messages -->
   {% with messages = get_flashed_messages(with_categories=true) %}
     {% if messages %}
       {% for category, message in messages %}
         <p class="{{ category }}">{{ message }}</p>
       {% endfor %}
     {% endif %}
   {% endwith %}
```

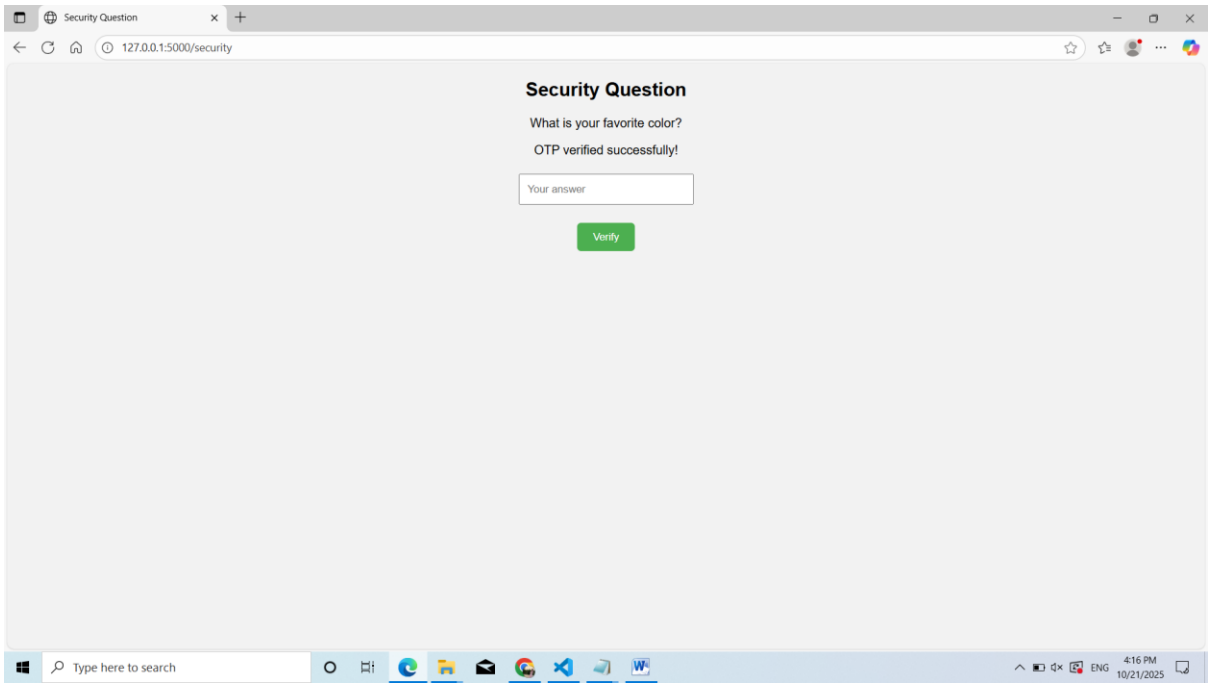
```
    <form method="POST">
        <input type="text" name="color" placeholder="Your answer"
required><br><br>
        <button type="submit">Verify</button>
    </form>
</body>
</html>
```


Triple Level Authentication
127.0.0.1:5000
Welcome to Smart Password Verification System
Register | Login
Type here to search
ENG
4:11 PM
10/21/2025

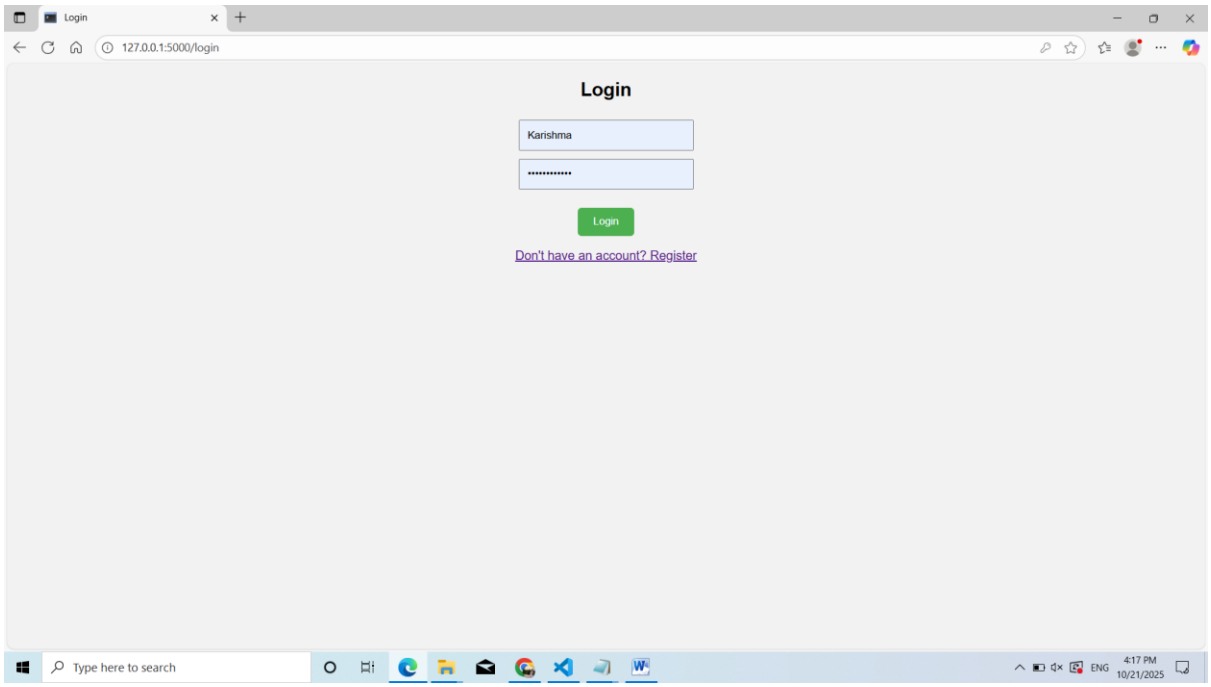

Register
127.0.0.1:5000/register
Register User
Karishma
yellow
Register
Already have an account? Login
Type here to search
ENG
4:13 PM
10/21/2025

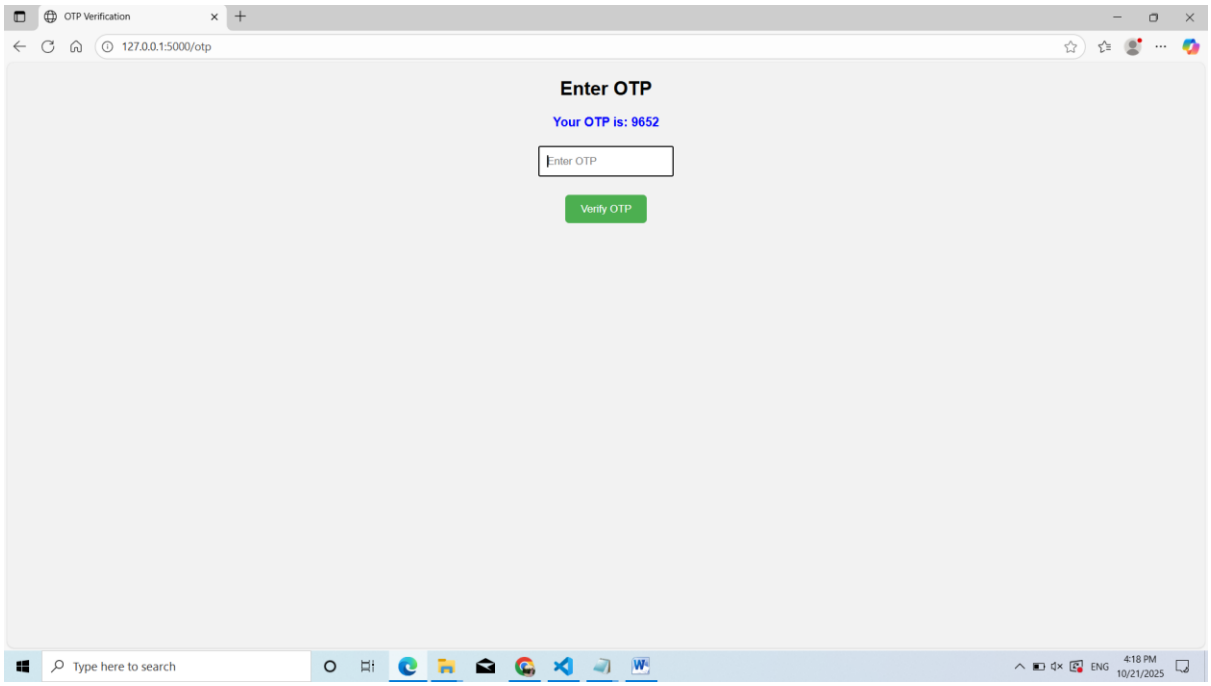

Login
127.0.0.1:5000/login
Login
Password strength: Strong
Registration successful!
Karishma
Login
Don't have an account? Register
Type here to search
ENG
4:15 PM
10/21/2025


OTP Verification
127.0.0.1:5000/otp
Enter OTP
Your OTP is: 5795
Enter OTP
Verify OTP
Type here to search
ENG
4:15 PM
10/21/2025


Security Question
127.0.0.1:5000/security
Security Question
What is your favorite color?
OTP verified successfully!
Your answer
Verify
Type here to search
ENG
4:16 PM
10/21/2025

AFTER REGISTRATION




OTP Verification
127.0.0.1:5000/otp
Enter OTP
Your OTP is: 9652
Enter OTP
Verify OTP
Type here to search
ENG
4:18 PM
10/21/2025