1. User Authentication & Session Management

What is User Authentication?

It means checking **who the user is** before allowing them inside the system.

Example:
When you enter **email + password** to login into Instagram → Instagram checks if you are real → this is **authentication**.

What is Session Management?

After login, we don't want the user to login again and again on every page.
So the server gives the user a **session (temporary identity)** and remembers them until logout.

We store this session using:

- **Cookies**
- **Session ID**
- **JWT tokens - JSON (optional alternative method)**

Simple Example (Express Login + Session)

```
const express = require("express");
const session = require("express-session");

const app = express();

app.use(session({
   secret: "mySecretKey",
   resave: false,
   saveUninitialized: true,
}));

app.get("/login", (req, res) => {
   req.session.user = "Ronak";   // Session created
   res.send("Login successful, session saved!");
});

app.get("/dashboard", (req, res) => {
   if (req.session.user) {
      res.send("Welcome " + req.session.user);
   } else {
      res.send("Please login first!");
   }
});

app.listen(3000);
```

After login, the user can visit `dashboard` without logging in again.

2. Basic Security Practices

Common Attacks:

Attack	Meaning
SQL Injection	Hacker enters SQL code inside input fields to steal database
XSS (Cross Site Scripting)	Hacker injects JavaScript code in website forms to steal cookies/data

How to Prevent Them?

1. Avoid SQL Injection by not trusting user input

Use **prepared statements** or ORM like Sequelize/Mongoose.

Dangerous code (vulnerable):

```
db.query("SELECT * FROM users WHERE email = '" + email + "'");
```

Safe code:

```
db.query("SELECT * FROM users WHERE email = ?", [email]);
```

2. Prevent XSS by sanitizing input (don't allow scripts)

User enters:

```
<script>alert('Hacked')</script>
```

Never show this directly. Instead sanitize using libraries like:

```
npm install xss-clean
const xss = require("xss-clean");
app.use(xss());
```

3. Use Helmet.js to secure HTTP headers

Helmet adds security like:

- prevents clickjacking
- hides server info
- blocks malicious code

Install:

```
npm install helmet
```

Use:

```
const helmet = require("helmet");
app.use(helmet());
```

3. Password Hashing using Bcrypt

What is Password Hashing?

Never store passwords in plain text like: `password123`

Instead convert it into a secret hashed format:

✅ `$2b$10$...randomhashedstring...`

This ensures even hackers can't read real passwords.

Example using Bcrypt

Install:

```
npm install bcrypt
```

Hashing password while signup:

```
const bcrypt = require("bcrypt");

const password = "MySecretPass";
const hashedPassword = bcrypt.hashSync(password, 10);

console.log(hashedPassword);
```

Checking password during login:

```
const isMatch = bcrypt.compareSync("MySecretPass", hashedPassword);

console.log(isMatch);  // true
```

Original password is never stored
We only compare the hash version

Password Hashing with Bcrypt – Step-by-Step

When building login systems in **Node.js + Express**, storing raw passwords is dangerous. Instead, we use **hashing**—a one-way process that converts the password into a scrambled string.
bcrypt is the most popular and secure library for hashing passwords.

1. Install bcrypt
npm install bcrypt

2. Import bcrypt in your code
const bcrypt = require("bcrypt");

3. Hashing a new user's password
Step-by-step process

1. **User enters a password**
2. **Your backend hashes it**
3. **Only the hash (not the original password) is stored in the database**

Example code

```
const saltRounds = 10;   // cost factor — higher means slower but more secure

const hashedPassword = await bcrypt.hash(userPassword, saltRounds);

// Save hashedPassword into MongoDB
```

Here:

- saltRounds = 10 means bcrypt will run the hashing algorithm 2^{10} times.
- This makes it computationally expensive for attackers.

4. Comparing password during login
When user logs in:

1. You fetch the hashed password from the database.
2. Compare the entered password with the stored hash.

Code

```
const match = await bcrypt.compare(enteredPassword, storedHashedPassword);

if (match) {
   console.log("Login successful");
} else {
   console.log("Invalid credentials");
}
```

bcrypt internally re-hashes the entered password with the same salt and checks if both hashes match.

🔐 **What is "Salting" a Password?**
A **salt** is random data added to the password before hashing.
Example:
Without salt:
password → hash123
password → hash123 (same hash for same password)
With salt:

password + randomSaltA → 9fj39j3...
password + randomSaltB → asd8d09...
Even if two users use **same password**, their hash becomes **different**.

🎯 Purpose of Salting

✔ 1. Prevents Rainbow Table Attacks

Hackers cannot use pre-computed hash tables because every password gets a **unique salted hash**.

✔ 2. Protects Users with Same Password

Even if 100 users choose "123456", all hashes will be different.

✔ 3. Makes brute-force attacks slower

Salt increases the complexity of guessing the correct hash.

📌 In short:

- **Hashing** = one-way encryption for storing passwords.
- **Salting** = adding unique random data before hashing.
- **bcrypt** handles both hashing and salting automatically.
- bcrypt stores: salt + cost factor + hashed password in the final hash string.

✅ Summary Table

Topic	Purpose	Technology
Authentication	Verify user identity	Email/Password, Sessions, JWT
Session Management	Keep user logged in	express-session, cookies
SQL Injection Protection	Prevent database hacking	Prepared statements / ORM
XSS Protection	Block harmful scripts	xss-clean, input sanitization
Security Headers	Secure HTTP headers	Helmet.js
Password Protection	Store safe passwords	Bcrypt hashing

In simple words:

Authentication tells who you are, session remembers you, bcrypt protects your password, and helmet + sanitization protects your app from hackers.