

Chapter 18: Authentication & Authorization

18.1 Authentication vs Authorization

18.1.1 What is Authentication?

- * Authentication is the process of **verifying the identity** of a user.
- * It answers the question "**Who are you?**"
- * The system checks if the user is valid by using **credentials** like username, password, OTP, etc.
- * Once authenticated, the user gets access to the system.



Authentication
(Who are you?)

18.1.2 What is Authorization?

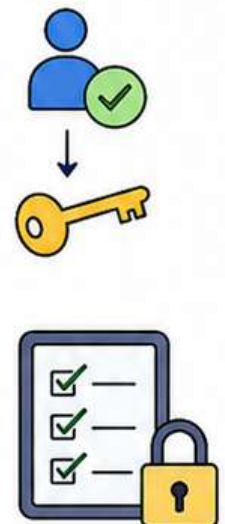
- * Authorization is the process of determining **what** an authenticated user is **allowed to do**.
- * It answers the question "**What are you allowed to do?**"
- * It defines the **permissions** and **access levels** for the user.
- * It ensures that users can access only the resources they are permitted to use.



Authorization
(What are you allowed to do?)

18.1.3 Authentication vs Authorization – Key Differences

Feature	Authentication	Authorization
Definition	Verifies the identity of a user.	Determines what the user can do.
Question	"Who are you?"	"What are you allowed to do?"
Purpose	To confirm the user is who they claim to be.	To control access to resources and actions.
Example	Login with username and password.	Allow user to view dashboard but not delete data.
Occurs	First – before accessing the system.	After authentication – to decide access level.
Methods	Password, OTP, Biometric, Token, etc.	Roles, Permissions, Policies, Access Levels.
Result	Grants access to the system after verifying identity.	Grants specific permissions to perform actions.



Key Points:

- ✓ Authentication verifies the **identity** of the user.
- ✓ Authorization decides what **actions** the user can perform.
- ✓ Both are essential for building **secure applications**.
- ✓ Authentication comes **first**, followed by Authorization.



Key Takeaway:



Authentication is about **verifying** the user, while Authorization is about **controlling** what the user can access and do.

18.2 User Registration

18.2.1 What is User Registration?

- * User registration is the process of creating a new account for a user in an application.
- * It collects **user information** and stores it in the **database**.
- * After registration, the user can **log in** to access the application.
- * It is the **first step** before authentication and authorization.



18.2.2 User Registration Flow



18.2.3 Common Registration Fields

Field	Description
Username	Unique name chosen by the user.
Email	User's email address, used for communication and verification.
Password	Secret key used to authenticate the user.
Confirm Password	Re-enter password to ensure correctness.
Full Name	User's full name.
Phone Number	User's contact number (optional).
Date of Birth	User's birth date (optional).



Key Points:

- ✓ User registration **collects** user details and **stores** them securely.
- ✓ Input **validation** is important to prevent invalid or malicious data.
- ✓ Passwords should be **hashed** before storing in the database.
- ✓ Email **verification** can be used to confirm the user's identity.



Key Takeaway:

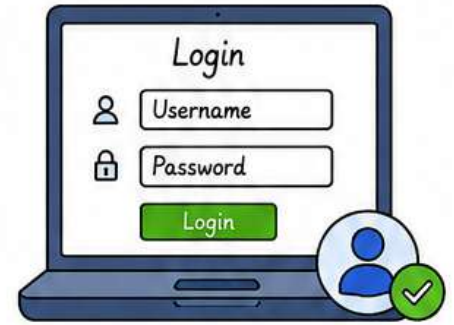


User registration is the **foundation** of any secure application. It ensures that only **valid users** can access the system.

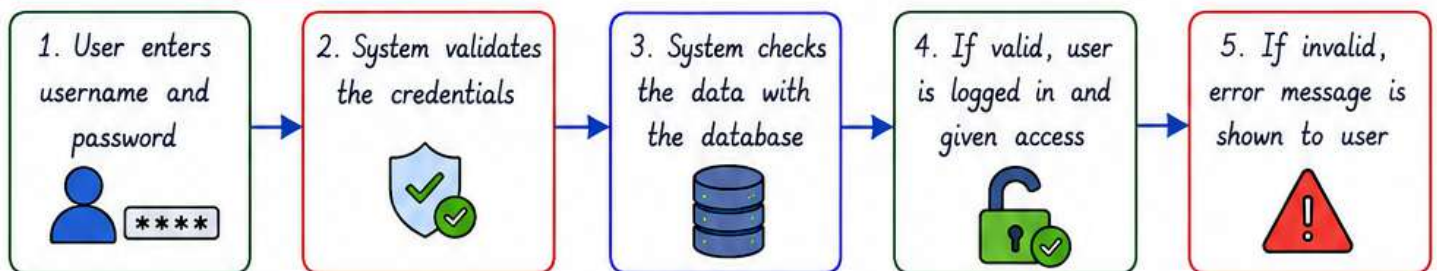
18.3 Login System

18.3.1 What is Login System?

- * Login system is the process of **verifying a user's identity** before allowing access to an application.
- * It checks the **credentials** (username and password) provided by the user.
- * If the credentials are **valid**, the user is **granted access**.
- * If not valid, access is **denied** and an error message is shown.



18.3.2 Login Process Flow



18.3.3 Components of Login System

Component	Description
Login Page	The page where user enters username and password.
Validation	Checks if the input is in the correct format and not empty.
Authentication	Verifies the username and password with stored data.
Session Management	Creates a session for logged-in user.
Error Handling	Displays appropriate messages for invalid attempts.
Logout	Ends the session and removes user access.



18.3.4 Common Features

- * Secure password storage (**hashed**).
- * Password **reset** / forgot password.
- * Remember me functionality.
- * Active **session** management.
- * Multi-factor authentication (**MFA**).
- * Role-based **redirection** after login.
- * Account lock after multiple failed attempts.



Key Points :

- ✓ Login system ensures that only **authorized users** can access the application.
- ✓ Credentials are **validated** and checked against the database.
- ✓ Secure session management **protects** user data and privacy.
- ✓ Always use **strong password** policies and secure authentication methods.



Key Takeaway:

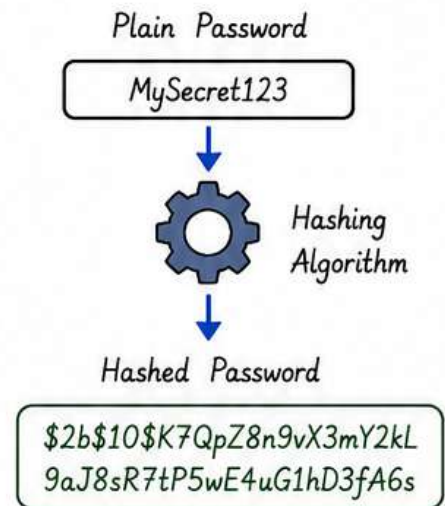


A good login system is the **gatekeeper** of any application. It verifies the user and **protects** the system from unauthorized access.

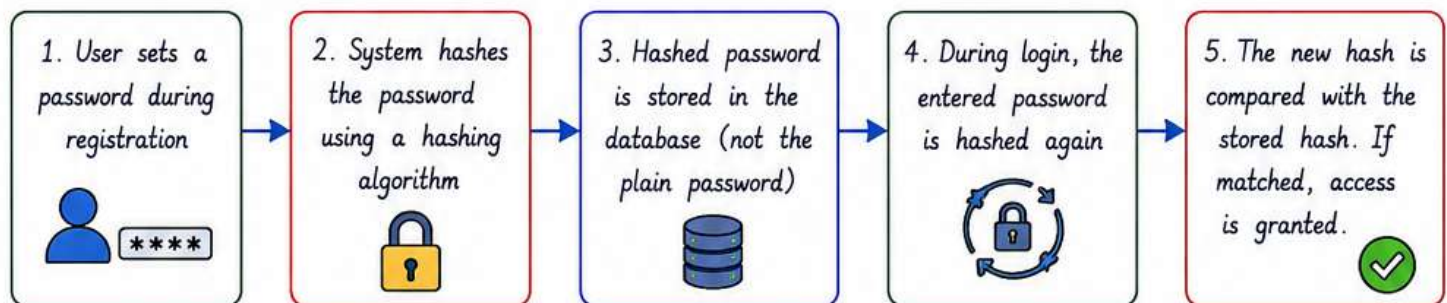
18.4 Password Hashing

18.4.1 What is Password Hashing?

- * Password hashing is the process of **converting** a password into a **fixed-length string** of characters (**hash**).
- * It is a **one-way process**, which means the original password **cannot be recovered** from the hash.
- * Instead of storing passwords in **plain text**, applications store only the **hash** in the database.
- * During login, the entered password is **hashed** and compared with the **stored hash**.



18.4.2 How Password Hashing Works?



18.4.3 Common Hashing Algorithms

Algorithm	Description	Features
MD5	Produces a 128-bit (32 characters) hash.	Fast but not secure, vulnerable to collisions.
SHA-1	Produces a 160-bit (40 characters) hash.	More secure than MD5 but now deprecated.
SHA-256	Produces a 256-bit (64 characters) hash.	Widely used and more secure.
bcrypt	Password-based hashing function.	Includes salt, slow by design, very secure.
Argon2	Modern password hashing algorithm.	Highly secure, resistant to brute-force attacks.



18.4.4 Key Points

- ✓ Never store passwords in **plain text**.
- ✓ Always use **strong** hashing algorithms with **salt**.
- ✓ Hashing **protects** users' passwords even if the **database is leaked**.
- ✓ Use **slow hashing** algorithms (like bcrypt, Argon2) to prevent **brute-force attacks**.
- ✓ Always **compare hashes**, never compare plain passwords.



Key Takeaway:

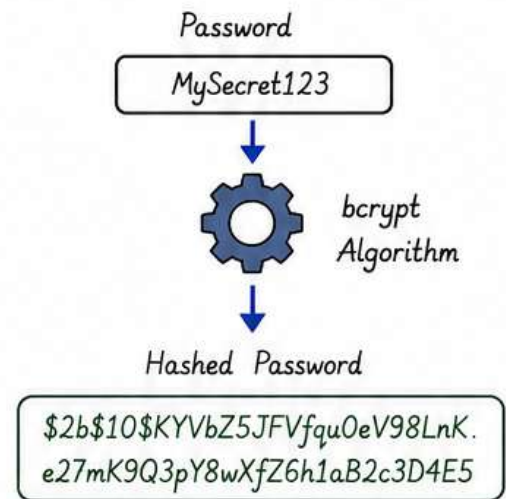


Password hashing is a critical **security** measure that ensures user passwords are stored securely and cannot be read even if the database is compromised. Always use **strong, modern** hashing algorithms to keep your application **safe**.

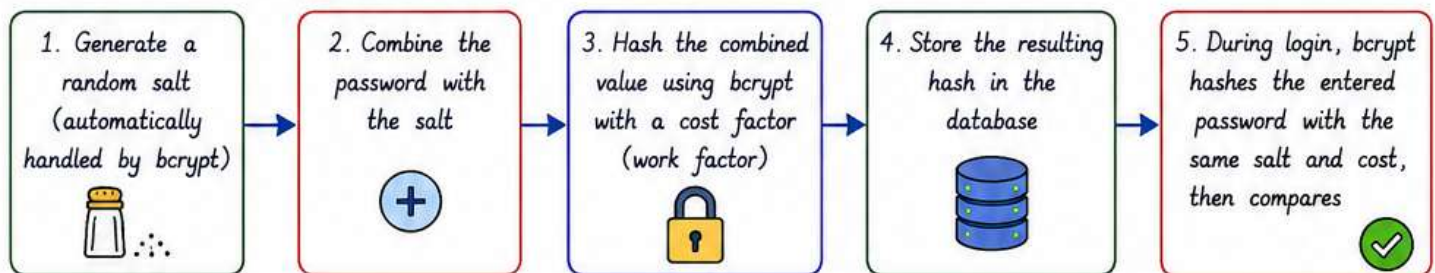
18.5 bcrypt

18.5.1 What is bcrypt?

- * bcrypt is a password hashing function based on the **Blowfish** encryption algorithm.
- * It is designed to be **slow** and computationally expensive to resist **brute-force attacks**.
- * It automatically handles **salting** and allows you to set the **cost** (work factor).
- * It is widely used for **securely storing** passwords.

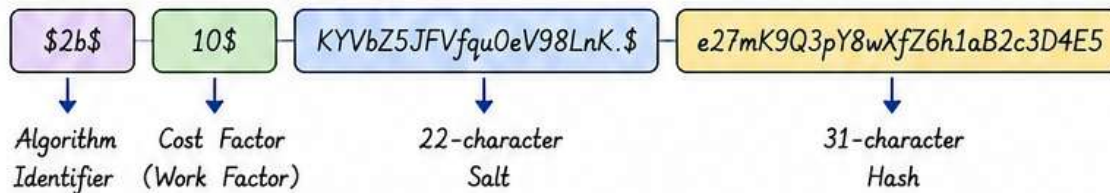


18.5.2 How bcrypt Works?



18.5.3 bcrypt Hash Format

A bcrypt hash has a specific format:



18.5.4 Key Features of bcrypt

- * Includes a unique **salt** for each password.
- * Adjustable **cost factor** to increase security.
- * **Slow** and computationally expensive.
- * Resistant to **brute-force** and rainbow table attacks.
- * Automatically handles **salt generation**.
- * Produces a **fixed-length hash** (60 characters).
- * **Secure** and widely recommended for password hashing.
- * Supported by many programming **libraries** and frameworks.



18.5.5 Advantages of bcrypt

- ✓ Makes password **cracking** extremely difficult and time-consuming.
- ✓ Each password has a **unique salt**, even if the same password is used.
- ✓ **Cost factor** can be increased over time as hardware becomes faster.
- ✓ **Simple** to use and available in most programming languages.
- ✓ Industry standard for **secure password storage**.



Key Takeaway:



bcrypt is a strong and reliable password hashing algorithm that adds **salt** and a configurable **cost factor** to protect passwords from **brute-force attacks**. Always use **bcrypt** to store and verify user passwords securely.

18.6 JWT

18.6.1 What is JWT?

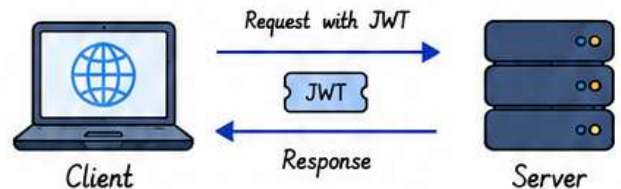
- * JWT (JSON Web Token) is a compact and self-contained way to **securely transmit information** between parties as a JSON object.
- * It is **digitally signed**, so the data cannot be altered.
- * It is commonly used for **authentication** and **authorization** in web applications and APIs.
- * JWT is **stateless**, which means the server does not need to store session data.

JWT Structure

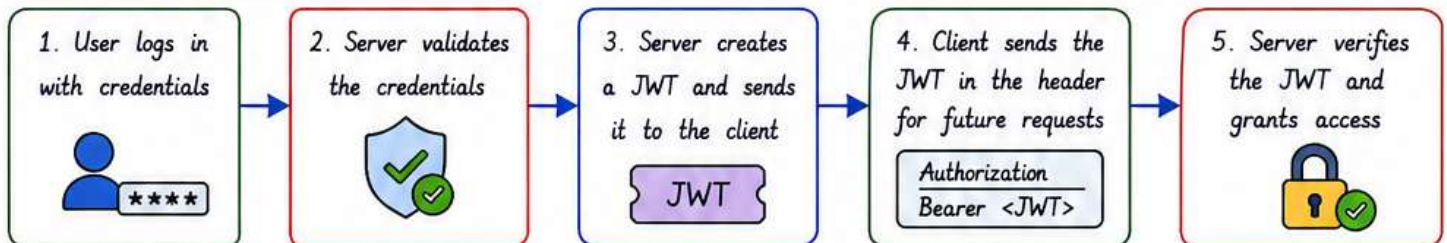


Example JWT

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM1LjI1IjoiaWoiQmF2aXRoYSIs
ImldCI6MTcxNTYxNzU4N30.
SflKxwRjSMekKF2QT4fwpMeJf36P0k6yJV_adQssw5c

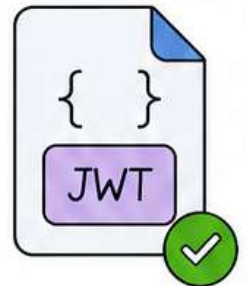


18.6.2 How JWT Works?



18.6.3 JWT Structure

Part	Description	Example
Header	Contains the type of token and signing algorithm used.	<pre>{ "alg": "HS256", "typ": "JWT" }</pre>
Payload	Contains the claims (user information and metadata).	<pre>{ "sub": "123", "name": "Bhavitha", "role": "user", "iat": 1715617583 }</pre>
Signature	Created by combining header, payload and secret key. It ensures integrity.	<pre>HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)</pre>



Key Points:

- ✓ JWT is compact, URL-safe and can be used across domains.
- ✓ It is stateless, so no session storage is required on the server.
- ✓ It can be used for both authentication and authorization.
- ✓ Always use HTTPS to transmit JWT.
- ✓ Set an expiration time (exp claim) for security.



Key Takeaway:



JWT is a **secure and efficient** way to transmit information between parties. It is widely used in modern web applications and APIs for **authentication and authorization**.



18.7 Access Tokens

18.7.1 What is an Access Token?

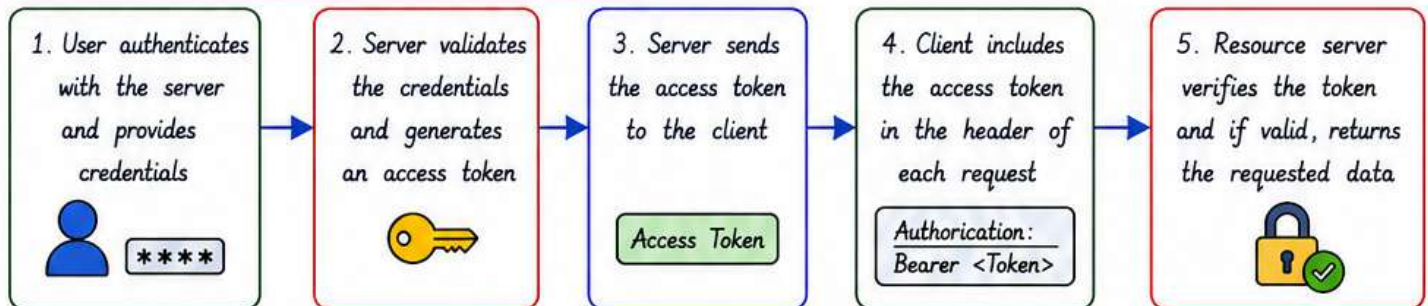
- * An access token is a **credential** used to access **protected resources**.
- * It is usually a **short-lived** token issued by the authentication server.
- * The client includes the access token in each request to access **APIs** or resources.
- * It proves the user's **identity** and **permissions** without sending username and password again.



Example Access Token (JWT)

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM5MTI0IiwiaXNja3BlIjpbInJlYWQiLCJ3cm93c2JldCJleHAiOjE3MTU2MjA0MDB9.SfIKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

18.7.2 How Access Tokens Work?



18.7.3 Components of Access Token (JWT)

Part	Description	Example (JWT)
Header	Contains the type of token and signing algorithm.	<pre>{ "alg": "HS256", "typ": "JWT" }</pre>
Payload	Contains claims about the user and metadata.	<pre>{ "sub": "123", "name": "Bhavitha", "role": "user", "exp": 1715620400, "iat": 1715616800 }</pre>
Signature	Created by combining header, payload and secret key. It ensures integrity.	<pre>HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)</pre>



Key Points :

- ✓ Access tokens are used to access **protected resources**.
- ✓ They are usually **short-lived** for better security.
- ✓ Always include the token in the **Authorization header**.
- ✓ Do not store access tokens in **localStorage**; use **HttpOnly cookies** when possible.
- ✓ Validate the token on every request before granting access.



Key Takeaway:

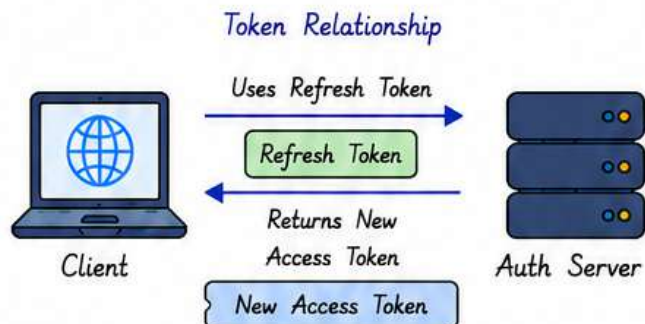


Access tokens are the key to accessing **protected resources** securely. They help in **stateless authentication** and improve the scalability of applications.

18.8 Refresh Tokens

18.8.1 What is a Refresh Token?

- * A refresh token is used to **obtain a new access token** without requiring the user to login again.
- * Access tokens are short-lived, but refresh tokens are **long-lived**.
- * It is stored securely on the client (usually in **HttpOnly cookies**).
- * Refresh tokens are used only to get **new access tokens**, not to access resources directly.
- * It improves user experience by keeping the user **logged in** seamlessly.



Example Refresh Token (Opaque or JWT)

```
def50200c8b1428987f1c6f58f8e1d9a.eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM5ZjZlZSIsImF1dGUiOiJhbm91bnQifQ.EyMsUiLCJleHAiOjE3MTUzMjQwMDB9.HJk3t8ySxQp2mL9V1aBc9ZQwOxYvFfXg2J1r3kQ7w4
```

18.8.2 How Refresh Tokens Work?



18.8.3 Differences: Access Token vs Refresh Token

Feature	Access Token	Refresh Token
Purpose	Access protected resources	Obtain new access token
Lifetime	Short-lived (minutes)	Long-lived (days/weeks)
Usage	Sent with every API request	Used only to get new access token
Storage	Memory (not persistent)	HttpOnly cookie or secure storage
Exposure Risk	Less risky	Higher risk (protect carefully)
Size	Smaller	Usually larger



18.8.4 Best Practices

- * Store refresh tokens in **HttpOnly, Secure** cookies.
- * Use **short expiration** for access tokens.
- * **Rotate refresh tokens** on every use.
- * Invalidate refresh token on **logout**.
- * Bind refresh token to **device or session**.
- * Monitor and **detect token misuse**.
- * **Never expose** refresh tokens in JavaScript.
- * Use **HTTPS** to protect tokens in transit.



Key Points:

- ✓ Refresh tokens help in getting new access tokens **without user login**.
- ✓ Access tokens are **short-lived**, refresh tokens are **long-lived**.
- ✓ Store refresh tokens **securely** and follow best practices.
- ✓ Refresh tokens **improve user experience and security** when used properly.



Key Takeaway:



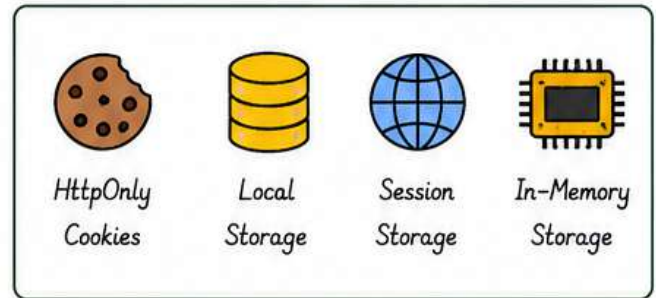
Refresh tokens are the key to **maintaining user sessions securely**. Use them wisely to balance **security** and user **convenience**.

18.9 Token Storage

18.9.1 What is Token Storage?

- * Token storage refers to the way we store **access tokens** and **refresh tokens** on the client side.
- * Proper token storage is important for **security**.
- * Storing tokens insecurely can lead to **token theft** and **unauthorized access**.
- * We should choose the right storage method based on **security** and **convenience**.

Where Can We Store Tokens?



18.9.2 Common Token Storage Methods

Storage Method	Description	Security	Persistence	Best Use
HttpOnly Cookies	Tokens are stored in cookies with HttpOnly flag.	High	Until cookie expires	Best for web applications (recommended)
Local Storage	Tokens are stored in the browser's localStorage .	Medium	Until manually cleared	When cookies cannot be used (e.g., mobile apps)
Session Storage	Tokens are stored in the browser's sessionStorage .	Medium	Until tab/session is closed	For smaller apps or short-lived sessions
In-Memory Storage	Tokens are stored in JavaScript variables.	High	Until page refresh	Best for SPAs for maximum security



18.9.3 Comparison of Token Storage Methods

Storage Method	XSS Protection	CSRF Protection	Persistence Control	Ease of Use	Recommended
HttpOnly Cookies	High	High	High	Easy	✓
Local Storage	Low	Low	High	Easy	!
Session Storage	Low	Low	Medium	Easy	!
In-Memory Storage	High	High	Low	Medium	✓



18.9.4 Best Practices for Token Storage

- * Use **HttpOnly** cookies to store tokens whenever possible.
- * Always use **Secure** flag for cookies in production.
- * Set **SameSite=strict** or **SameSite=lax** to prevent CSRF.
- * Keep access tokens **short-lived**.
- * Use **refresh tokens** to get new access tokens.
- * Never store tokens in **localStorage** if sensitive data is handled.
- * Avoid exposing tokens in URL or client-side scripts.
- * Always use **HTTPS** to transmit tokens.
- * Rotate refresh tokens regularly.



Key Points:

- ✓ Token storage plays a crucial role in application security.
- ✓ HttpOnly cookies are the safest method for storing tokens.
- ✓ Avoid storing tokens in **localStorage** due to XSS risk.
- ✓ Use short-lived access tokens and long-lived refresh tokens.
- ✓ Follow best practices to protect tokens from attacks.



Key Takeaway:



Choose the right storage method based on your application needs and **security** requirements. Always follow **best practices** to store and protect tokens safely.



18.10 Authentication Middleware

18.10.1 What is Authentication Middleware?

- * Authentication middleware is a function that **verifies the identity** of a user before allowing access to protected routes.
- * It checks if the request contains a **valid token** or credentials.
- * If authentication is **successful**, the request proceeds to the next middleware or route handler.
- * If authentication **fails**, access is **denied**.

Where is Authentication Middleware Used?



Protected
Routes



API
Endpoints



User
Sessions



REST
APIs

Purpose

- ✓ Ensure only authenticated users can access resources.
- ✓ Prevent unauthorized access.
- ✓ Extract user information from token (e.g., userId, role).
- ✓ Attach user info to request object for further use.

18.10.2 How Authentication Middleware Works?



18.10.3 Example: JWT Authentication Middleware (Node.js/Express)

```
const jwt = require('jsonwebtoken');
function authenticateToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  if (!authHeader) {
    return res.status(401).json({ message: 'Access denied. No token provided.' });
  }
  const token = authHeader.split(' ')[1]; // Format: Bearer <token>
  if (!token) {
    return res.status(401).json({ message: 'Access denied. Invalid token format.' });
  }
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded; // attach user info to request
    next(); // proceed to next middleware or route handler
  } catch (error) {
    return res.status(403).json({ message: 'Invalid or expired token.' });
  }
}
module.exports = authenticateToken;
```

Explanation:

1. Extract token from Authorization header.
2. Verify token using secret key.
3. If valid, attach decoded user info to req.user.
4. If invalid/expired, return error response.

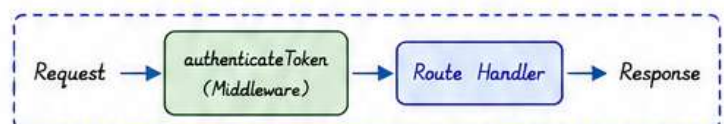


18.10.4 Using Middleware in Routes

```
const express = require('express');
const authenticateToken = require('./middleware/auth');
const app = express();

// Protected route
app.get('/profile', authenticateToken, (req, res) => {
  res.json({ message: 'Welcome', user: req.user });
});
```

Order of Middleware Execution



- * Middleware functions are executed in the order they are defined.

Key Points:

- ✓ Authentication middleware protects routes and APIs.
- ✓ It **verifies the** user's identity using token or session.
- ✓ Attach user info to request object for further processing.
- ✓ Always handle errors and return appropriate status codes.



Key Takeaway:



Authentication middleware acts as a **gatekeeper**. It ensures that only **valid and authenticated users** can access **protected** resources. By centralizing authentication logic, it improves **security** and code reusability.



18.11 Protected APIs

18.11.1 What are Protected APIs?

- * Protected APIs are endpoints that require **authentication** before they can be accessed.
- * They ensure that only **authorized users** can access sensitive data or perform actions.
- * Access to these APIs is usually granted using a valid **access token** (e.g., JWT).
- * It helps in **securing data** and maintaining **user privacy**.

Why Protect APIs?

- ✓ Prevent unauthorized access.
- ✓ Protect sensitive user data.
- ✓ Ensure data integrity and security.
- ✓ Control access based on user roles and permissions.
- ✓ Comply with security best practices.



18.11.2 How Protected APIs Work?



18.11.3 Example: Accessing a Protected API

a) Client Request (Using Token)

```
GET /api/user/profile HTTP/1.1
Host: example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Content-Type: application/json
```

b) Successful Response (200 OK)

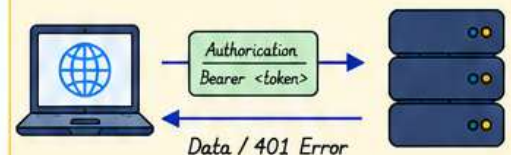
```
{
  "userId": 101,
  "name": "Bhavitha",
  "email": "bhavitha@example.com",
  "role": "user"
}
```

c) Unauthorized Response (401)

```
{
  "message": "Unauthorized",
  "details": "Invalid or expired token"
}
```

Explanation:

1. Client includes access token in the **Authorization** header.
2. Server middleware verifies the token.
3. If the token is valid, the API processes the request.
4. If the token is invalid or expired, server returns **401 Unauthorized**.



18.11.4 Implementing Protected API in Express (Node.js)

```
const express = require('express');
const authenticateToken = require('./middleware/auth');
const app = express();

// Protected Route Example
app.get('/api/user/profile', authenticateToken, (req, res) => {
  res.json({ message: 'Welcome!', user: req.user });
});

// Another Protected Route
app.get('/api/data', authenticateToken, (req, res) => {
  res.json({ data: 'This is protected data.' });
});

app.listen(5000, () => console.log('Server running on port 5000'));
```

How it Works:

1. **authenticateToken** middleware verifies the token.
2. If valid, request proceeds to the route handler.
3. If invalid, middleware sends **401 Unauthorized**.



Key Points:

- ✓ Protected APIs require valid authentication to access.
- ✓ Always send the access token in **Authorization header**.
- ✓ Middleware validates the token before processing the request.
- ✓ Return appropriate status codes (**200** for success, **401** for unauthorized).



Key Takeaway:



Protected APIs are essential for building **secure** applications.
Always **validate tokens** and **handle errors** properly to protect sensitive data.



18.12 Protected React Routes

18.12.1 What are Protected React Routes?

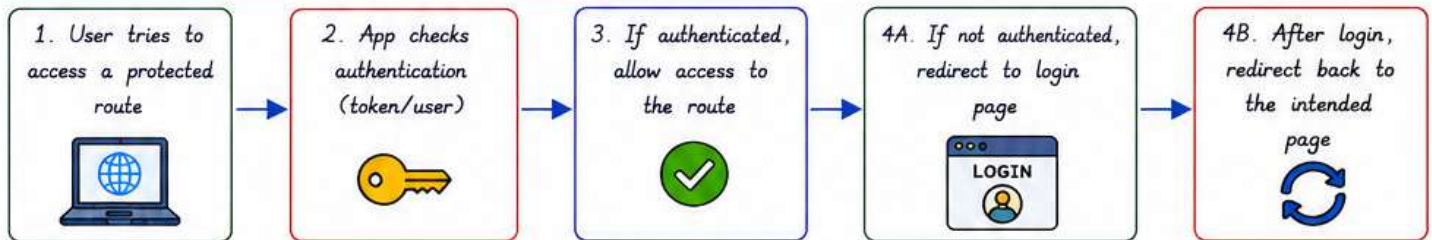
- * Protected React Routes are routes that can be accessed only by **authenticated users**.
- * If a user is **not logged in**, they are redirected to the **login page**.
- * We use **authentication state** (e.g., token/user) to protect routes.
- * It improves **security** and **user experience**.

Why Protect Routes in React?

- ✓ Prevent unauthorized access.
- ✓ Protect private pages (dashboard, profile, settings, etc.).
- ✓ Redirect unauthenticated users to login page.
- ✓ Maintain secure navigation in the app.



18.12.2 How Protected Routes Work?



18.12.3 Example: Protected Route Component

```
// ProtectedRoute.jsx
import { Navigate, useLocation } from 'react-router-dom';
import { useAuth } from '../context/AuthContext';

const ProtectedRoute = ({ children }) => {
  const { user, token } = useAuth(); // get auth state
  const location = useLocation();

  if (!token || !user) {
    // not authenticated -> redirect to login
    return <Navigate to="/login" state={{ from: location }} replace />;
  }
  // authenticated -> render the protected content
  return children;
};

export default ProtectedRoute;
```

Explanation:

1. Check if token (or user) exists.
2. If not, redirect to /login and save the current location.
3. If yes, render the child components (protected pages).

Project Structure

```
src/
├── context/
│   └── AuthContext.jsx
├── components/
│   └── ProtectedRoute.jsx
├── pages/
│   ├── Dashboard.jsx
│   ├── Profile.jsx
│   └── Login.jsx
├── App.jsx
└── main.jsx
```

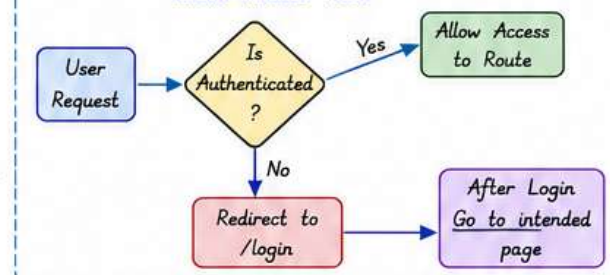
18.12.4 Using Protected Routes in App.jsx

```
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import ProtectedRoute from '../components/ProtectedRoute';
import Dashboard from '../pages/Dashboard';
import Profile from '../pages/Profile';
import Login from '../pages/Login';

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/login" element={<Login />} />
        <Route path="/dashboard" element={
          <ProtectedRoute><Dashboard /></ProtectedRoute>
        } />
        <Route path="/profile" element={
          <ProtectedRoute><Profile /></ProtectedRoute>
        } />
        <Route path="*" element={<Navigate to="/dashboard" />} />
      </Routes>
    </Router>
  );
}

export default App;
```

Route Access Flow



Key Points:

- ✓ Use ProtectedRoute component to protect routes.
- ✓ Check authentication (token/user) before allowing access.
- ✓ Redirect unauthenticated users to login page.
- ✓ Preserve the intended page location and redirect after successful login.



Key Takeaway:



Protected routes in React ensure that only authenticated users can access certain pages. It improves security, user experience, and application flow.

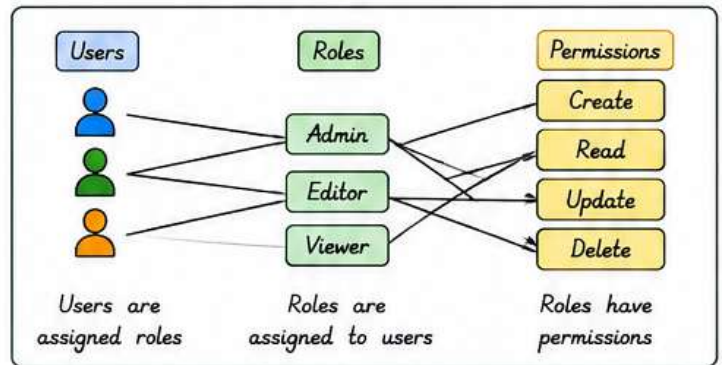


18.13 Role-Based Access Control (RBAC)

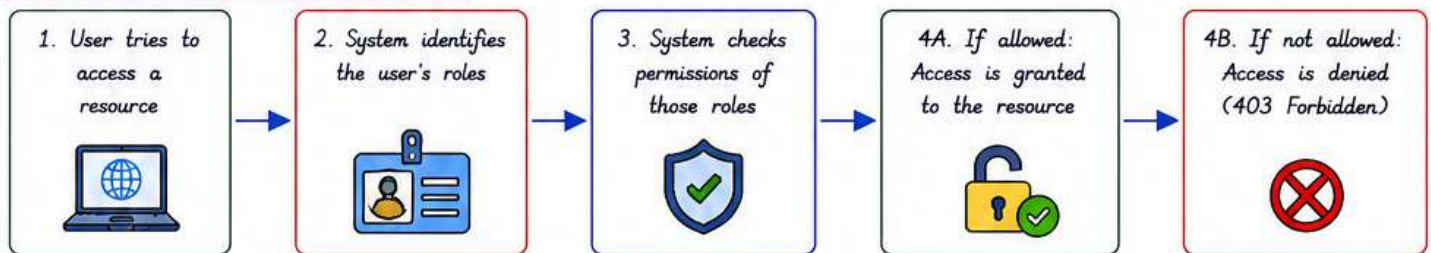
18.13.1 What is RBAC?

- * Role-Based Access Control (RBAC) is an **authorization** mechanism that grants permissions to users based on their **roles**.
- * Instead of assigning **permissions** directly to users, **permissions** are assigned to **roles**.
- * Users are assigned one or more roles, and they get permissions of those roles.
- * RBAC **simplifies** access management and improves **security**.

RBAC Concept



18.13.2 How RBAC Works?



18.13.3 Example: RBAC in an Application

Role	Description	Permissions
Admin	Full access to the system	Create, Read, Update, Delete (All)
Editor	Can manage content	Create, Read, Update
Viewer	Can view content only	Read
Guest	Limited public access	Read (Public only)

18.13.4 Example Scenario

- * Admin can add, update, delete users, and view reports.
- * Editor can create and update articles but cannot delete users.
- * Viewer can only read articles and view dashboard.
- * Guest can only view public pages.



18.13.5 Implementing RBAC (Example in Node.js + Express)

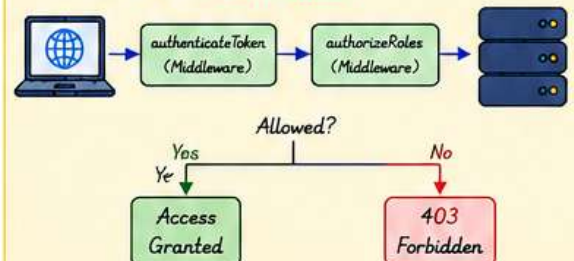
```
// Middleware to check role-based access
function authorizeRoles(...allowedRoles) {
  return (req, res, next) => {
    const user = req.user; // assume user is set by auth middleware
    if (!user) return res.status(401).json({ message: 'Unauthorized' });
    const userRoles = user.roles; // e.g., ['admin', 'editor']

    const hasRole = userRoles.some(role => allowedRoles.includes(role));
    if (!hasRole) {
      return res.status(403).json({ message: 'Forbidden: Access denied' });
    }
    next();
  };
}

// Example: Only admin can access this route
app.get('/admin/dashboard', authenticateToken,
  authorizeRoles('admin'), (req, res) => {
    res.json({ message: 'Welcome Admin!' });
  });
```

How it Works:

1. **authenticateToken** middleware verifies the user and attaches user info to **req.user**.
2. **authorizeRoles** checks if the user has any of the allowed roles.
3. If yes, the request proceeds; otherwise, access is denied with **403 Forbidden**.



18.13.6 Benefits of RBAC



Improves security



Simplifies permission management



Scalable for large applications



Easy to maintain and update permissions



Reduces risk of unauthorized access

Key Points:

- ✓ RBAC assigns permissions to roles, and roles to users.
- ✓ It centralizes access control and makes it easier to manage.
- ✓ Always follow the principle of least privilege.
- ✓ RBAC is widely used in enterprise applications for better security.



Key Takeaway:



RBAC ensures that users can access only what they are authorized to do based on their roles. It improves security, simplifies management, and is a best practice for access control.



18.14 Admin Authentication

18.14.1 What is Admin Authentication?

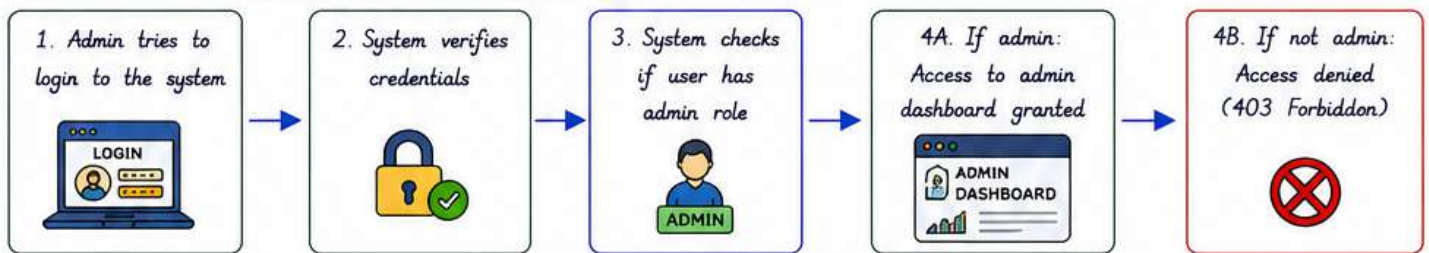
- * Admin authentication is the process of verifying that a user is an **admin** before allowing access to **admin features**.
- * It ensures that only **authorized admins** can manage users, settings, and critical data.
- * It is usually implemented using **role-based access control (RBAC)**.
- * Admin routes and APIs are protected to maintain **security**.

Why Admin Authentication is Important?

- ✓ Prevents unauthorized access to admin panel.
- ✓ Protects sensitive data and system settings.
- ✓ Ensures only admins can perform critical actions.
- ✓ Helps in tracking and logging admin activities.
- ✓ Improves overall application security.



18.14.2 How Admin Authentication Works?



18.14.3 Example: Admin vs Normal User Access

User	Role	Can Access	Cannot Access
Admin User	admin	Admin Dashboard, User Management, Reports, Settings	Nothing (Full Access)
Editor User	editor	Create/Update Articles	Admin Dashboard, User Management
Viewer User	viewer	View Articles	Admin Features, Settings
Guest User	guest	Public Pages	All Restricted Areas

18.14.4 Common Admin Features

- * Dashboard with stats and reports.
- * Manage users (create, update, delete).
- * Manage roles and permissions.
- * Manage content / data.
- * System settings and configurations.
- * View logs and activities.



18.14.5 Implementing Admin Authentication (Node.js + Express)

```
// Middleware to check if user is admin
function isAdmin(req, res, next) {
  if (!req.user) {
    return res.status(401).json({ message: 'Unauthorized' });
  }
  if (req.user.role !== 'admin') {
    return res.status(403).json({ message: 'Access denied. Admins only.' });
  }
  next();
}

// Protected admin route
app.get('/admin/dashboard', authenticateToken, isAdmin, (req, res) => {
  res.json({ message: 'Welcome to Admin Dashboard' });
});

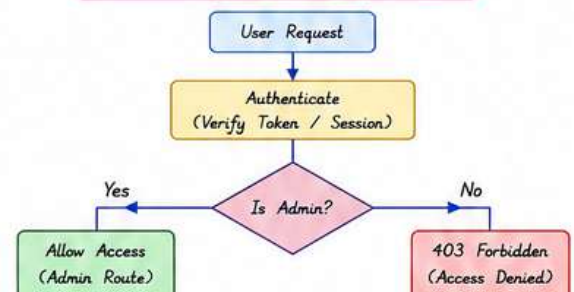
// Example user object (after login)
// req.user = { id: 1, name: 'Bhavitha', role: 'admin', email: 'admin@example.com' }
```

Explanation:

1. User logs in with **email** & password.
2. Credentials are verified.
3. If valid, system checks if `user.role === 'admin'`.
4. If yes, access admin panel.
5. If no, access denied with **403 Forbidden**.



18.14.6 Route Protection Flow



18.14.7 Best Practices



Always verify user role on every request.



Store role in secure token (JWT).



Use RBAC to manage roles and permissions.



Log admin activities for audit.



Never expose admin APIs publicly.

Key Points:

- ✓ Admin authentication ensures only admins can access admin features.
- ✓ It is usually implemented using role-based access control (RBAC).
- ✓ Always protect admin routes and APIs using middleware.
- ✓ Return 403 Forbidden for non-admin users trying to access admin resources.



Key Takeaway:



Admin authentication is essential for secure management of your application. Always verify identity and role before granting access to admin resources.



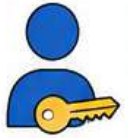
18.15 User Permissions

18.15.1 What are User Permissions?

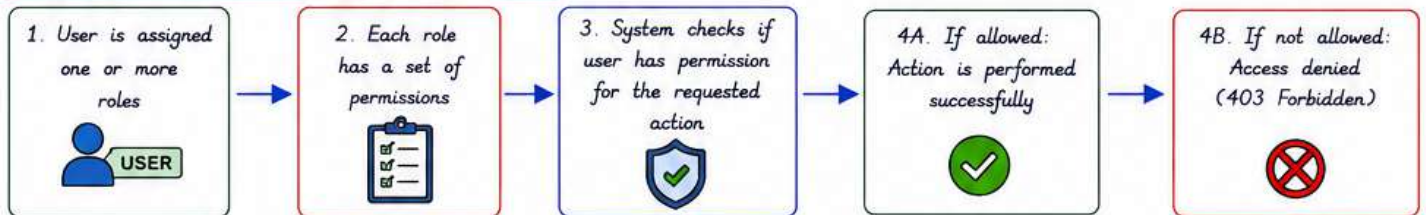
- * User permissions are **specific rights** given to a user that define what **actions** they can perform on **resources**.
- * Permissions are assigned to **roles**, and users inherit permissions through their **roles**.
- * It helps in **controlling access** to features, data, and operations in an application.
- * Fine-grained permissions improve **security** and prevent misuse.

Why Manage User Permissions?

- ✓ Ensure users can access only what they need.
- ✓ Prevent unauthorized actions and data access.
- ✓ Support role-based access control (RBAC).
- ✓ Improve security, compliance, and data privacy.
- ✓ Allow easy management of features and modules.



18.15.2 How User Permissions Work?



18.15.3 Example: Permissions in an Application

Permission Name	Description	Module	HTTP Method
user.create	Create new users	User Management	POST
user.read	View users	User Management	GET
user.update	Update user details	User Management	PUT/PATCH
user.delete	Delete users	User Management	DELETE
post.create	Create new posts	Content	POST
post.read	View posts	Content	GET
post.update	Update posts	Content	PUT/PATCH
post.delete	Delete posts	Content	DELETE
report.view	View reports	Reports	GET
settings.manage	Manage system settings	Settings	ALL

18.15.4 Example: Role-Permission Mapping

Role	Permissions
Admin	All Permissions (Full Access)
Editor	user.read, user.update, post.create, post.read, post.update
Viewer	post.read, report.view
Guest	None (No Access)

Explanation:

1. Each role is given only the permissions it needs.
2. Users inherit permissions from their assigned roles.
3. System verifies permission before any action.
4. If user lacks permission, access is denied with **403 Forbidden**.



18.15.5 Implementing User Permissions (Node.js + Express)

```
// Middleware to check permission
function checkPermission(permission) {
  return (req, res, next) => {
    const user = req.user; // set by authenticateToken middleware
    if (!user) return res.status(401).json({ message: 'Unauthorized' });
    const userPermissions = user.permissions || []; // e.g., ['user.read', 'post.create']
    if (!userPermissions.includes(permission)) {
      return res.status(403).json({ message: 'Forbidden: You do not have permission.' });
    }
    next();
  };
}

// Example: Protecting a route with permission
const express = require('express');
const router = express.Router();
const authenticateToken = require('./middleware/authenticateToken');

// Only users with 'user.create' permission can access this route
router.post('/users', authenticateToken, checkPermission('user.create'), (req, res) => {
  res.json({ message: 'User created successfully' });
});

module.exports = router;
```

18.15.6 Permission Checking Flow



18.15.7 Best Practices

- Use RBAC to assign permissions to roles.
- Follow principle of least privilege.
- Group related permissions together.
- Regularly review roles and permissions.
- Log permission denials and suspicious access.

Key Points:

- ✓ User permissions define what actions a user can perform.
- ✓ Permissions are assigned to roles, and users inherit them.
- ✓ Always verify permissions before allowing access to resources.
- ✓ Return 403 Forbidden when user does not have required permission.



Key Takeaway:

Managing user permissions ensures secure, controlled, and efficient access to application features and data. It is a key part of building a robust and scalable application.



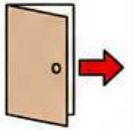
18.16 Logout

18.16.1 What is Logout?

- * Logout is the process of **ending the user session** and **removing authentication information**.
- * It ensures that the user's access **token** or session is **invalidated** and cannot be used again.
- * It **protects** the application from unauthorized access when a user leaves the system.
- * Always provide a **secure logout option** in your application.

Why Logout is Important?

- ✓ Prevents unauthorized access to the user's account.
- ✓ Clears sensitive data from storage.
- ✓ Ends active sessions on the client and server.
- ✓ Protects user privacy and session security.
- ✓ Reduces the risk of session misuse.



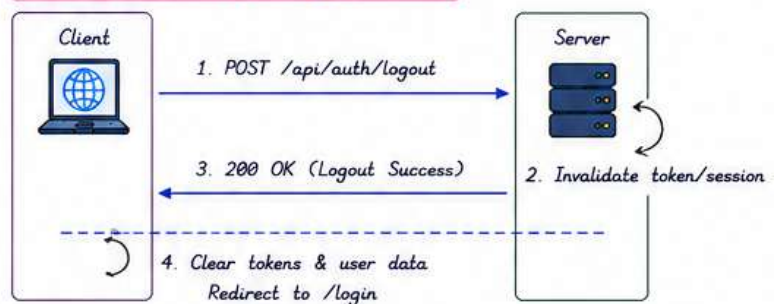
18.16.2 How Logout Works?



18.16.3 What is Cleared on Logout?

Item	What is Cleared?
Access Token	Removed from localStorage / sessionStorage / cookie
Refresh Token	Removed from storage (if used)
User Data	User info removed from state/storage
Session (Server-side)	Session invalidated on the server
Cookies	Authentication cookies are cleared
UI State	User state reset, redirect to login

18.16.4 Example: Logout Flow



18.16.5 Implementing Logout in Node.js + Express

```
// Logout route
app.post('/api/auth/logout', authenticateToken, (req, res) => {
  try {
    // If using blacklist (optional)
    // blacklistToken(req.token);
    // If using sessions, destroy session
    if (req.session) {
      req.session.destroy(err => {
        if (err) return res.status(500).json({ message: 'Logout failed' });
        return res.status(200).json({ message: 'Logout successful' });
      });
    } else {
      // For stateless JWT, client will remove token
      return res.status(200).json({ message: 'Logout successful' });
    }
  } catch (error) {
    res.status(500).json({ message: 'Logout failed' });
  }
});
// Note: With JWT, token is removed on client side.
```

18.16.6 Logout in React (Clearing Token)

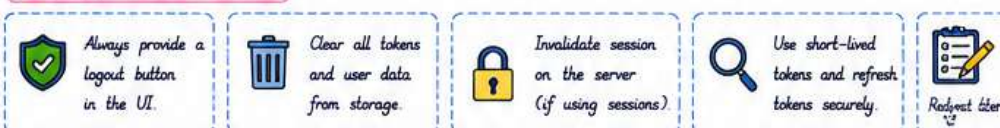
```
// Logout.js
import { useNavigate } from 'react-router-dom';

const logout = () => {
  // Remove tokens from storage
  localStorage.removeItem('token');
  localStorage.removeItem('refreshToken');
  sessionStorage.clear();
  // Clear user data from state (if using context/redux)
  // e.g., setUser(null);

  // Redirect to login page
  const navigate = useNavigate();
  navigate('/login');
};

export default logout;
```

18.16.7 Best Practices



Key Points:

- ✓ Logout terminates the user session and removes authentication data.
- ✓ Invalidate tokens/sessions on the server (if applicable).
- ✓ Always clear tokens, user info, and state on the client.
- ✓ Redirect the user to the login page after logout.



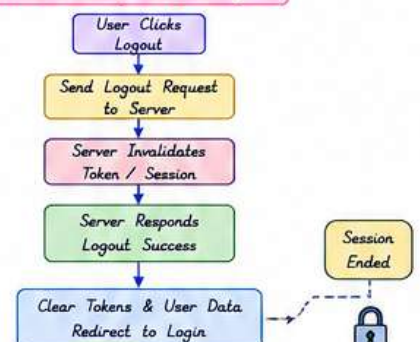
Key Takeaway:



A secure logout protects user accounts and ensures that no one else can access the application using the user's session or tokens.



18.16.8 Logout Flow Diagram



18.17 Password Reset

18.17.1 What is Password Reset?

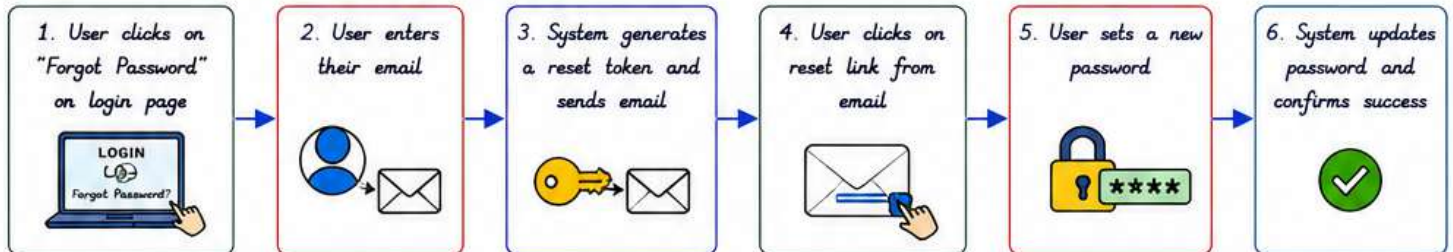
- * Password reset allows users to securely reset their password if they forget it.
- * It helps users regain access to their account without compromising security.
- * It is done using a reset link or a one-time token sent to the user's email.
- * Always use **secure** and **time-limited tokens** for password reset.

Why Password Reset is Important?

- ✓ Helps users recover access to their account.
- ✓ Secure process prevents unauthorized access.
- ✓ Improves user experience.
- ✓ Reduces support requests.
- ✓ Protects user data and privacy.



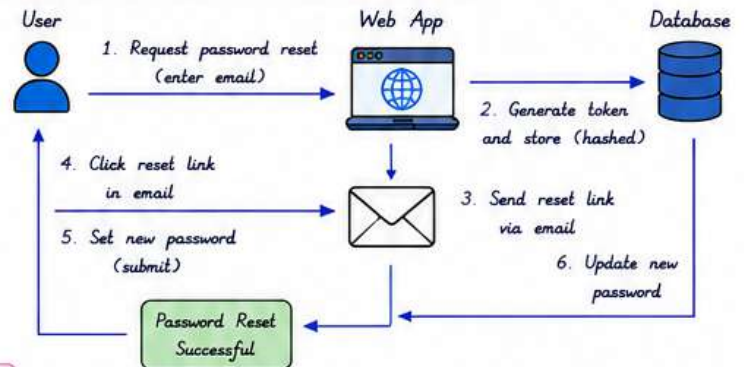
18.17.2 How Password Reset Works?



18.17.3 Token / Link Details

Property	Description
Token Type	Random, unique, hard to guess
Stored In	Database (hashed)
Expiry Time	Short-lived (e.g., 15 to 60 minutes)
Usage	One-time use only
After Use	Token is invalidated / deleted
Delivery	Sent via email to registered user

18.17.4 Example: Password Reset Flow



18.17.5 Implementing Password Reset (Node.js + Express)

```
// Request password reset (send email with token)
app.post('/api/auth/forgot-password', async (req, res) => {
  const { email } = req.body;
  const user = await User.findOne({ email });
  if (!user) return res.status(404).json({ message: 'Email not found' });

  const token = crypto.randomBytes(32).toString('hex');
  const resetToken = crypto.createHash('sha256').update(token).digest('hex');
  user.resetToken = resetToken;
  user.resetTokenExpiry = Date.now() + 30 * 60 * 1000; // 30 minutes
  await user.save();

  const resetLink = `${FRONTEND_URL}/reset-password/${token}`;
  // sendEmail(user.email, 'Password Reset', resetLink);
  res.json({ message: 'Reset Link sent to your email' });
});

// Reset password (use token)
app.post('/api/auth/reset-password/:token', async (req, res) => {
  const token = crypto.createHash('sha256')
    .update(req.params.token).digest('hex');
  const user = await User.findOne({ resetToken: token, resetTokenExpiry: { $gt: Date.now() } });
  if (!user) return res.status(400).json({ message: 'Invalid or expired token' });

  user.password = await bcrypt.hash(req.body.password, 10);
  user.resetToken = undefined;
  user.resetTokenExpiry = undefined;
  await user.save();
  res.json({ message: 'Password reset successful' });
});
```

18.17.6 Explanation

1. User requests password reset by entering email.
2. System generates a secure token and stores its hashed value with expiry.
3. Reset link with token is sent to user's email.
4. User clicks the link and sets a new password.
5. System validates token, updates password and invalidates the token.



18.17.7 Best Practices

- * Use short-lived tokens (15 to 60 minutes).
- * Store only hashed tokens in the database.
- * Allow token to be used only once.
- * Limit password reset requests to prevent abuse.
- * Always send reset link to registered email only.
- * Use HTTPS for all reset requests.



Key Points:

- ✓ Password reset helps users recover account access securely.
- ✓ Always use secure, time-limited, one-time tokens.
- ✓ Send reset link only to the registered email.
- ✓ Invalidate token after successful password reset.



Key Takeaway:



A well-implemented password reset process balances usability and security, ensuring users can recover access without risking their account.



18.18 Email Verification

18.18.1 What is Email Verification?

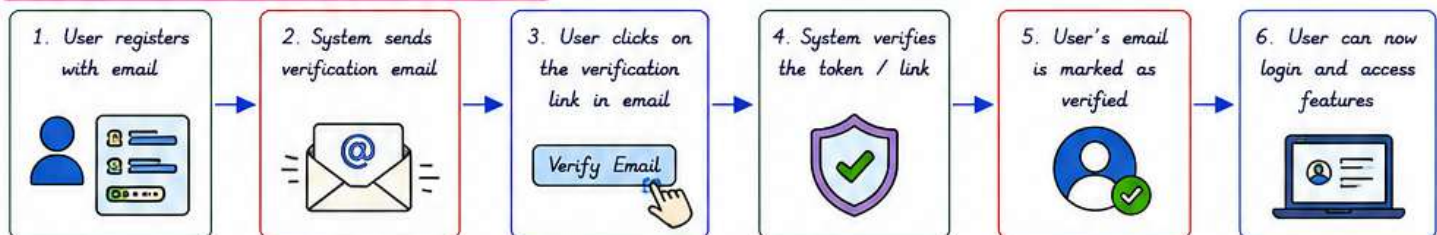
- * Email verification is the process of confirming that a user's email address is **valid** and **belongs to them**.
- * It is usually done by sending a **verification link** or **code** to the user's email.
- * It helps **prevent fake registrations** and improves account security.
- * Only **verified** users are allowed to access certain features or APIs.

Why Email Verification is Important?

- ✓ Ensures valid and reachable email addresses.
- ✓ Prevents fake and spam registrations.
- ✓ Improves account security.
- ✓ Helps in password reset and notifications.
- ✓ Builds trust and credibility in the system.



18.18.2 How Email Verification Works?



18.18.3 Verification Token / Link Details

Field	Description
Token	Random, unique string used for verification
Stored In	Database (user table)
Expiry Time	Short-lived (e.g., 15 to 60 minutes)
Usage	Used only once
After Use	Token is invalidated / deleted
Delivery	Sent via email to the user

18.18.4 Example: Email Verification Flow



18.18.5 Implementing Email Verification (Node.js + Express)

```
// 1. Generate token and send email
const crypto = require('crypto');
function generateToken() {
  return crypto.randomBytes(32).toString('hex');
}
// After user registration
const token = generateToken();
user.verifyToken = token;
user.tokenExpiry = Date.now() + 30 * 60 * 1000; // 30 minutes
user.isVerified = false;
await user.save();

// Send email with verification link
const link = `${FRONTEND_URL}/verify-email/${token}`;
sendEmail(user.email, 'Verify Your Email',
  `Please verify your email: ${link}`);

// 2. Verify email (when user clicks link)
app.get('/verify-email/:token', async (req, res) => {
  const token = req.params.token;
  const user = await User.findOne({ verifyToken: token });
  if (!user) return res.status(400).send('Invalid or expired link');
  if (user.tokenExpiry < Date.now()) return res.status(400).send('Link expired');
  user.isVerified = true;
  user.verifyToken = undefined;
  user.tokenExpiry = undefined;
  await user.save();
  res.send('Email verified successfully! You can now Login.');
```

18.18.6 Explanation

1. After registration, a unique token is generated and stored.
2. The token is sent to the user's email as a verification link.
3. When the user clicks the link, the token is verified.
4. If valid and not expired, the user's email is marked as verified.
5. The token is removed to prevent reuse.

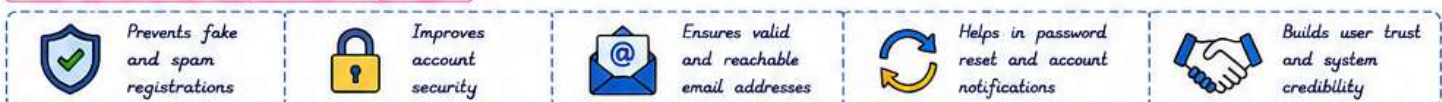


18.18.7 Best Practices

- * Use secure random tokens.
- * Set token expiry (15-60 minutes).
- * Allow token to be used only once.
- * Invalidate token after successful verification.
- * Always send verification link to registered email only.
- * Use HTTPS for all verification requests.



18.18.8 Benefits of Email Verification



Key Points:

- ✓ Email verification ensures that the user owns a valid email address.
- ✓ It prevents fake accounts and improves security.
- ✓ Use time-limited tokens and allow one-time usage only.
- ✓ Always redirect to login page after successful verification.



Key Takeaway:



Email verification is a simple but crucial step to secure user registration and build a trustworthy application.

