**University of Kotli Azad Jammu & Kashmir**

**Faculty of Computing & Engineering**

**Department Software Engineering**

**Assignment # 01**

**Topic:**

ISPs



**Course:** Web Engineering

**Submitted By:**

**Name: Qura-tul-ain Saleem**

**Roll No: 04**

**Class: BS SE**

**Semester: 6th**

**Submitted to: Ms. Qurat-ul-Ain**

**Submission Date:** 21-10-2024

**Table of Contents**

[What is full stack state? Discuss in detail. 3](#_Toc180322163)

[1. Full Stack State 3](#_Toc180322164)

[1.1. Understanding "State" in Software Development 3](#_Toc180322165)

[1.2. Frontend State 3](#_Toc180322166)

[1.3. Backend State 4](#_Toc180322167)

[1.4. Synchronization Between Frontend and Backend State 5](#_Toc180322168)

[1.5. Challenges in Full Stack State Management 6](#_Toc180322169)

[1.6. Tools and Frameworks for Full Stack State Management 6](#_Toc180322170)

[What is Network and Sessional Attacks? 7](#_Toc180322171)

[1. Network Attacks 7](#_Toc180322172)

[1.1. Denial of Service (DoS) Attack: 7](#_Toc180322173)

[1.2. Distributed Denial of Service (DDoS) Attack: 7](#_Toc180322174)

[1.3. Man-in-the-Middle (MITM) Attack: 7](#_Toc180322175)

[1.4. Packet Sniffing (Passive Eavesdropping): 8](#_Toc180322176)

[1.5. SQL Injection (SQLi): 8](#_Toc180322177)

[1.6. DNS Spoofing (DNS Cache Poisoning): 8](#_Toc180322178)

[1.7. IP Spoofing: 8](#_Toc180322179)

[1.8. Phishing: 8](#_Toc180322180)

[2. Session Attacks 9](#_Toc180322181)

[2.1. Session Hijacking: 9](#_Toc180322182)

[2.2 Cross-Site Scripting (XSS): 9](#_Toc180322183)

[2.3. Replay Attack: 9](#_Toc180322184)

[2.4. Session Fixation: 10](#_Toc180322185)

[2.5. Sidejacking (Session Sniffing): 10](#_Toc180322186)

# ****What is full stack state? Discuss in detail.****

# ****1. Full Stack State****

**Full Stack State** refers to the entire development and operational state of a full-stack application, encompassing both client-side (frontend) and server-side (backend) components, as well as the various states of data, user interfaces, and application behavior across the entire system. To understand the concept more deeply, let’s break it down by looking at the different layers of a full-stack application and how they manage "state" at each level.

## 1**.1. Understanding "State" in Software Development**

In software, "state" refers to the condition of a system or an application at a particular moment in time, determined by the data it holds, the actions it has taken, or the events it has responded to.

For a **full-stack** application, managing the state involves tracking and synchronizing data across:

* The **frontend (client-side)**, where users interact with the application.
* The **backend (server-side)**, which processes logic and interacts with the database.
* The **database (storage layer)**, which stores persistent data.

## 1.2. ****Frontend State****

The **frontend state** deals with the state of the user interface (UI) and how data is managed on the client-side, typically within a browser. Frontend frameworks like React, Vue, Angular, or plain JavaScript are responsible for managing the UI state and user interactions.

#### **Key Concepts of Frontend State:**

1. **UI State**:
   * Manages the appearance and interaction behavior of UI elements.
   * Examples: Open/close status of a modal, active tab in a navigation bar, input fields filled with user data.
2. **Local State**:
   * Data specific to a component or a page.
   * Example: A form that holds user input data temporarily before submission.
3. **Global State**:
   * State that affects multiple parts of the application.
   * Example: Authentication state (logged in/logged out), application-wide settings, or theme preferences.
4. **State Management Libraries**:
   * To manage the complexity of large applications, libraries like **Redux**, **Vuex**, and **Context API** are used to handle the global state.
   * These libraries centralize the state, making it easier to manage the flow of data between components and pages.
5. **Client-Side Caching**:
   * State is sometimes cached locally using **LocalStorage**, **SessionStorage**, or **IndexedDB**, enabling data persistence across page reloads without making server requests.
   * Example: Saving a user’s theme preference so it’s retained on page refresh.

## 1.3. ****Backend State****

The **backend state** refers to how the server handles data and user requests, managing business logic and interacting with the database. The backend is responsible for providing, processing, and storing data used by the frontend.

#### **Key Concepts of Backend State:**

1. **Session State**:
   * Manages user-specific data between different HTTP requests, typically for logged-in users.
   * Stored either in **session cookies**, **in-memory databases** (like Redis), or directly in the database.
   * Examples: Shopping cart details, user login status, preferences.
2. **Server-Side Application State**:
   * Manages the state of the server application, which could include temporary data storage, background tasks, and processing logic.
   * Examples: Handling user uploads, processing payment requests, or performing batch data operations.
3. **Persistent State (Database)**:
   * This is the long-term state that gets stored in the database (e.g., SQL databases like MySQL/PostgreSQL, NoSQL databases like MongoDB).
   * The database stores all the critical application data, including user profiles, posts, comments, transactions, etc.
   * The backend ensures this data is queried, updated, and stored consistently based on user actions from the frontend.
4. **API State**:
   * The backend might interact with other external services via **APIs** (Application Programming Interfaces), adding another layer of state management.
   * Example: An application that fetches external weather data or payment processing data.

## 1.4. ****Synchronization Between Frontend and Backend State****

Managing the state across the **frontend** and **backend** is key to delivering a smooth and consistent user experience. The communication between these layers is often managed through APIs (RESTful or GraphQL) or WebSockets.

#### **Key Mechanisms for Synchronization**:

1. **RESTful APIs**:
   * The frontend sends HTTP requests (GET, POST, PUT, DELETE) to the backend to either retrieve or update data, and the backend responds with the current state.
   * Example: A shopping cart updates after an item is added, which is reflected in the user interface and stored in the server-side session.
2. **GraphQL**:
   * Allows the frontend to query only the specific data it needs, improving efficiency.
   * Example: Instead of multiple endpoints, a single GraphQL request retrieves a user profile, posts, and comments in one go.
3. **WebSockets**:
   * Used for real-time applications (e.g., chat apps, collaborative tools) where the state must be updated continuously.
   * Example: A messaging app updates the chat screen in real-time when a new message arrives from the server.
4. **State Synchronization Patterns**:
   * **Optimistic UI Updates**: The frontend assumes an operation will succeed (e.g., adding an item to a cart) and updates the UI immediately, then syncs with the backend. If the backend operation fails, the UI is corrected.
   * **Polling**: The frontend periodically sends requests to the backend to check for updates.
   * **Real-Time Sync**: Using WebSockets or Server-Sent Events (SSE) for live updates, the backend can push new state data to the frontend as soon as it changes.

## 1.5. ****Challenges in Full Stack State Management****

1. **Consistency**:
   * Keeping the state synchronized across different layers (frontend, backend, and database) is crucial to prevent bugs, such as displaying outdated or incorrect data to users.
2. **Concurrency**:
   * Handling concurrent actions that modify state from multiple users or processes can lead to race conditions or data corruption if not managed properly.
   * Solutions: Implementing transactions in databases, using optimistic or pessimistic locking mechanisms.
3. **Security**:
   * Since the state contains sensitive data, such as user credentials and session tokens, ensuring secure communication (e.g., using HTTPS, encrypting cookies) is critical to prevent session hijacking or data leaks.
4. **Performance**:
   * Efficiently managing state is important for performance, especially in real-time applications or large-scale systems with high traffic. Caching strategies, load balancing, and state partitioning across microservices are often used.
5. **Scalability**:
   * As the application scales, maintaining the state across distributed systems or microservices can become complex. Solutions include using distributed databases, state replication, and partitioning data across services.

## 1.6. ****Tools and Frameworks for Full Stack State Management****

* **Frontend**: React's Context API, Redux, Vuex, Angular’s Services for managing state.
* **Backend**: State management with session management libraries like **Express-session** (Node.js), or using tools like **Redis** for session storage.
* **Database**: Relational databases (MySQL, PostgreSQL) for persistent state or NoSQL solutions (MongoDB, DynamoDB) for flexible, scalable state storage.

# ****What is Network and Sessional Attacks?****

# ****1.**** Network Attacks

Network attacks are attempts to disrupt, misuse, or compromise the integrity, confidentiality, or availability of network resources. They exploit vulnerabilities in a network to gain unauthorized access or cause harm. Network attacks can be classified into two broad categories: **active** and **passive** attacks.

#### **Types of Network Attacks:**

## ****1.1. Denial of Service (DoS) Attack****:

* + **Goal**: Make a network resource or system unavailable to its intended users by overwhelming it with a flood of illegitimate requests.
  + **How it works**: Attackers send a large volume of traffic or data to a target system (server, network device, etc.) to exhaust its resources, making it unable to respond to legitimate requests.
  + **Examples**: SYN flood, ICMP (Ping) flood.

## ****1.2.**** Distributed Denial of Service (DDoS****) Attack****:

* + **Goal**: Same as a DoS attack, but the attack is launched from multiple compromised devices (often forming a botnet) across different locations.
  + **How it works**: A massive amount of requests are sent from many sources simultaneously, making it much harder to stop.
  + **Example**: Mirai botnet DDoS attack.

## ****1.3. Man-in-the-Middle (MITM) Attack****:

* + **Goal**: Intercept communication between two parties without their knowledge to steal data, manipulate messages, or eavesdrop.
  + **How it works**: The attacker positions themselves between two communicating entities (e.g., between a user and a server) to capture or alter the information being exchanged.
  + **Examples**: ARP poisoning, Wi-Fi eavesdropping.

## ****1.4. Packet Sniffing (Passive Eavesdropping)****:

* + **Goal**: Capture and analyze the data packets being transmitted over the network to gain sensitive information like passwords, session tokens, or personal data.
  + **How it works**: Tools like Wireshark are used to capture raw network traffic, which is then analyzed to extract information.
  + **Example**: Capturing unencrypted communication in public Wi-Fi networks.

## ****1.5. SQL Injection (SQLi)****:

* + **Goal**: Inject malicious SQL queries into a network-facing application to manipulate the database.
  + **How it works**: By sending specially crafted SQL queries through input fields (e.g., login forms), attackers can gain access to sensitive data or manipulate database content.
  + **Example**: An attacker uses a vulnerable login page to access or modify the database.

## ****1.6. DNS Spoofing (DNS Cache Poisoning)****:

* + **Goal**: Redirect network traffic by altering DNS records.
  + **How it works**: The attacker poisons a DNS server's cache, making it resolve domain names incorrectly, thereby redirecting users to malicious websites.
  + **Example**: A user tries to visit a legitimate website but is directed to a phishing website instead.

## ****1.7. IP Spoofing****:

* + **Goal**: Impersonate a trusted host by falsifying the source IP address in the packet headers.
  + **How it works**: The attacker sends packets with a forged IP address to make it appear as if the packets are coming from a trusted source, which can be used to bypass security measures or launch other attacks.
  + **Example**: Launching a DDoS attack with spoofed IPs to evade detection.

## ****1.8. Phishing****:

* + **Goal**: Trick users into divulging sensitive information (passwords, credit card numbers) by masquerading as a legitimate entity.
  + **How it works**: Attackers send emails or messages that appear to come from legitimate sources (banks, social networks) to lure users into providing their details on a fake website.
  + **Example**: A user receives an email from a "bank" asking them to reset their password via a phishing link.

# 2. ****Session Attacks****

Session attacks exploit vulnerabilities in the way a session (i.e., an active communication between two systems, such as a user and a server) is managed. They aim to hijack, manipulate, or eavesdrop on the session to gain unauthorized access to information or services.

#### **Types of Session Attacks:**

## ****2.1. Session Hijacking****:

* + **Goal**: Take control of an active user session to gain unauthorized access to the system.
  + **How it works**: The attacker captures or guesses the session token (a unique identifier for a session), allowing them to impersonate the legitimate user.
  + **Examples**:
    - **TCP Session Hijacking**: The attacker intercepts and controls a TCP session by guessing or capturing the session sequence numbers.
    - **HTTP Session Hijacking**: In web applications, session cookies (used to maintain user sessions) are captured and used by the attacker to impersonate the user.

## ****2.2 Cross-Site Scripting (XSS)****:

* + **Goal**: Inject malicious scripts into a web application session to steal session tokens or perform unauthorized actions.
  + **How it works**: The attacker injects a script into a vulnerable web page, and when the victim visits the page, their browser executes the script, which can steal their session cookie or perform other malicious actions.
  + **Example**: A user clicks on a link with a malicious script embedded in it, which allows the attacker to steal the user's session cookie and hijack the session.

## ****2.3. Replay Attack****:

* + **Goal**: Capture and replay a legitimate session to gain unauthorized access.
  + **How it works**: The attacker captures legitimate session data (like a login token or request) and reuses it later to trick the system into granting access.
  + **Example**: An attacker intercepts a user’s login credentials and reuses them later to log into the system as the user.

## ****2.4. Session Fixation****:

* + **Goal**: Force a user to use a specific session ID that the attacker knows, allowing the attacker to hijack the session after the user logs in.
  + **How it works**: The attacker sets a session ID in the user's browser (via URL, cookies, etc.) before they log in, and once the user logs in, the attacker takes over the session by using the same session ID.
  + **Example**: An attacker sends a login URL with a fixed session ID to a victim, and once the victim logs in, the attacker uses the same session ID to access the user’s account.

## ****2.5. Sidejacking (Session Sniffing)****:

* + **Goal**: Steal an unencrypted session token transmitted over the network.
  + **How it works**: The attacker captures network traffic and extracts the session tokens or cookies. They can then use the stolen tokens to impersonate the user.
  + **Example**: An attacker sniffs network traffic in a public Wi-Fi network to capture users’ session cookies and hijack their accounts.