**Question 13**

**How would you set up monitoring for the React Native mobile app’s API endpoints?**

**Answer:**

### **1. Define Monitoring Objectives**

Focus on:

* **Uptime**: Ensuring the API is accessible.
* **Performance**: Measure latency, response times, and throughput.
* **Errors**: Track error rates and response codes (e.g., 4xx, 5xx).
* **Usage Patterns**: Monitor the number of requests, popular endpoints, and user behaviors.

### **2. Set Up Server-Side Monitoring**

Using tools to monitor the APIs directly from the server:

* **Application Performance Monitoring (APM)**: Implementing tools like **New Relic**, **Datadog**, or **Dynatrace** to provide insights into API performance, database queries, and server health.
* **Logs**:
  + Using a centralized logging service like **ELK Stack** (Elasticsearch, Logstash, Kibana) or **AWS CloudWatch**.
  + Loging requests and responses, including headers, payload, and response times.
  + Including metadata such as user ID, app version, and device type for React Native-specific insights.

**3. Implement API Health Checks**

* Using tools like **Postman Monitor** or **Pingdom** to schedule health checks.
* Automating checks for:
  + Endpoint availability.
  + Expected responses.
  + Authentication and authorization flows.

### **4. Instrument the React Native App**

To gather client-side API metrics, instrument the React Native app for:

#### **a. Network Interception**

Using libraries to intercept API calls and capture data:

* **Axios Interceptors** (for Axios-based apps)
* **React Native Network Inspector** or custom middleware for fetch.

#### **b. Error Reporting**

Using error reporting tools like **Sentry**, **Bugsnag**, or **Crashlytics**:

* Logging API request errors and categorize them by endpoint, error type, and affected users.

#### **c. Performance Monitoring**

* Using tools like **Firebase Performance Monitoring** to track API call latency and errors.
* Adding custom traces for critical API endpoints.

### **5. Visualize Metrics**

* **Dashboards**:
  + Using platforms like **Grafana**, **Datadog**, or **AWS CloudWatch** to visualize metrics like response times, errors, and throughput.
  + Creating specific dashboards for endpoints critical to the app (e.g., user login, payment processing).
* **Alerts**:
  + Configuring alerts for:
    - High response times (e.g., >500ms).
    - Error rates above a threshold (e.g., >5% for 5xx responses).
    - Downtime.
  + Using tools like **PagerDuty**, **OpsGenie**, or **Slack notifications** for alert delivery.

### **6. Simulate User Behavior**

* **Synthetic Monitoring**:
  + Using tools like **New Relic Synthetic Monitoring** or **Checkly** to simulate user interactions and monitor API behavior under realistic conditions.
* **Load Testing**:
  + Using tools like **Apache JMeter**, **k6**, or **Gatling** to identify performance bottlenecks under stress.

### **7. Enable Distributed Tracing**

Using distributed tracing tools like **Jaeger** or **Zipkin** to trace API requests across services. This helps:

* Identify bottlenecks across microservices.
* Measure end-to-end response times.

### **8. Monitor Backend Resources**

Monitoring the backend infrastructure to ensure it supports API requests:

* **Server Metrics**: CPU, memory, disk usage, and network throughput.
* **Database Performance**: Query response times, connection pool usage, and indexes.
* **Caching Layers**: Cache hit/miss rates if using tools like Redis or Memcached.

### **9. Enable Real-User Monitoring (RUM)**

RUM tools like **Datadog RUM** or **New Relic RUM** provide insights into:

* Actual API response times as experienced by users.
* Errors and performance issues on specific devices or regions.

### **10. Iterate and Improve**

* Continuously analyze logs and metrics to identify trends.
* Perform post-mortem analyses after incidents to prevent recurrence

**Question 12**

**Explain how you would debug high latency in the Node.js microservices.**

**Answer;**

* **Log Analysis:** Check logs for error messages and performance metrics.
* **Performance Profiling:** Use tools like Node.js profiler or Clinic.js to identify bottlenecks.
* **Resource Monitoring:** Monitor resource usage (CPU, memory) to identify potential issues.
* **Database Queries:** Ensure database queries are optimized and not causing delays.
* **Network Latency:** Check for network-related issues that may be causing high latency.