UNIT 5 – Basics of Node.js Server Integration
Assignment 5

Q1. Explain two key advantages of GraphQL over REST APIs in the context of building scalable APIs.

Answer:
GraphQL provides several advantages over REST, especially when building large-scale, data-intensive APIs. Two major advantages are:

1. Precise Data Fetching (No Overfetching or Underfetching)

- In REST, endpoints return fixed structures, often containing more data than required.
- GraphQL allows clients to request **only the fields** they need.
- This reduces payload size and improves performance at scale.

2. Single Endpoint for Complex Data

- REST often requires multiple endpoints to gather related data.
- GraphQL exposes **one unified endpoint**, allowing clients to fetch multiple related resources in a single request.
- This reduces network overhead, improves latency, and simplifies API versioning.

Q2. Describe how pagination and batching can help optimize the performance of GraphQL APIs. Provide an example for each.

Answer:

1. Pagination

Pagination limits the amount of data returned in a single query, reducing load on servers and improving response time.

Example (Cursor-based pagination):

```
query {
  users(first: 5, after: "cursor123") {
    edges {
      node { id name }
      cursor
    }
    pageInfo {
      hasNextPage
    }
  }
}
```

This prevents the server from fetching thousands of records at once.

---

2. Batching (DataLoader)

Batching groups multiple related database requests into a single query to reduce redundant lookups.

Example (DataLoader batching):

```
const userLoader = new DataLoader(async (ids) => {
  return db.users.find({ id: { $in: ids }});
});
```

Instead of 10 queries for 10 posts' authors, batching reduces them to **one DB query**.

Q3. Discuss the key benefits of using serverless computing with Node.js for building event-driven architectures.

Answer:
Serverless platforms (AWS Lambda, Azure Functions) provide significant advantages:

1. Automatic Scaling

Serverless functions scale automatically based on events (e.g., HTTP calls, queue messages).
No server provisioning or autoscaling groups are required.

2. Pay-Per-Use

You pay only when the function runs.
Ideal for variable workloads or event-driven systems like notifications, scheduled jobs, or IoT events.

3. Faster Development & Deployment

No server setup, simplified deployments, and built-in integrations with cloud services (S3, DynamoDB, SQS).

4. High Availability Built-In

Serverless platforms handle fault tolerance and regional redundancy automatically.

Q4. Explain the core concepts of GraphQL, including queries, mutations, and schemas. Illustrate with an example how these elements work together in building a scalable API.

Answer:
GraphQL revolves around three fundamental concepts:

1. Queries

Used for fetching data (read operations).

```
query {
  user(id: 1) {
    name
    email
```

```
  }
}
```

2. Mutations

Used to create, update, or delete data.

```
mutation {
  updateUser(id: 1, name: "Kaushik") {
    id
    name
  }
}
```

3. Schema

Defines the types and structure of your GraphQL API.

```
type User {
  id: ID!
  name: String
  email: String
}

type Query {
  user(id: ID!): User
}

type Mutation {
  updateUser(id: ID!, name: String): User
```

}

How they work together

- **Schema** defines the shape of data.
- **Query** requests data matching the schema.
- **Mutation** modifies data based on schema-defined operations.
- Resolvers connect these to the actual data sources (DB, APIs).

This structure ensures scalability through predictable, strongly-typed interactions.

Q5. Discuss advanced performance optimization techniques like caching and batching in GraphQL. How can these techniques improve API efficiency, and what challenges might arise when implementing them?

Answer:

1. Caching

- Cache results of expensive GraphQL queries.
- Use tools like Redis or in-memory caches.
- Example:
    - Cache full query responses
    - Cache frequently accessed fields
    - Use persisted queries to eliminate repeated parsing

Improvements:

- Reduced DB calls
- Lower latency

- Faster repeated queries

Challenges:

- Invalidating cache when data changes
- Dynamic GraphQL queries make caching more complex than REST

---

2. Batching

- Use DataLoader to group multiple database requests.
- Example: Loading authors of 100 posts → 1 batched DB query.

Improvements:

- Reduces N+1 query problem
- Lowers server load
- Increases throughput

Challenges:

- Misconfigured batching can delay responses
- Must ensure batch size is optimized
- Requires careful resolver design

Q6. Describe best practices for monitoring serverless applications in Node.js. How do tools like AWS CloudWatch or other monitoring services help ensure the reliability of serverless applications?

Answer:

Best Practices:

1. Use Centralized Logging

- Console logs automatically go to CloudWatch.
- Add structured JSON logs for better searchability.

2. Monitor Function Metrics

Track:

- Invocations
- Errors
- Throttles
- Duration
- Cold starts

3. Set up Alerts

- Error rate > threshold
- High latency
- Lambda timeout warnings
- Throttling notifications

4. Use Distributed Tracing

- AWS X-Ray or third-party tools (Datadog, New Relic)
- Helps trace events across microservices

How CloudWatch Helps

- Real-time logs for debugging
- Alarms based on metrics

- Dashboards for performance monitoring
- Alerts via SNS or email
- Traces integration with AWS X-Ray

These enable early detection of issues and maintain reliability.

Q7. Compare and contrast REST and GraphQL in terms of scalability, flexibility, and performance. Discuss specific use cases where GraphQL would be a better choice over REST for API development.

Answer:

Scalability

- **REST:** Scales well using caching, CDNs, and simple endpoints.
- **GraphQL:** Also scalable but requires optimized resolvers and server clustering.

Flexibility

- **REST:** Rigid; fixed responses from endpoints.
- **GraphQL:** Highly flexible; clients fetch exactly what they want.

Performance

- **REST:**
    - Can suffer from overfetching/underfetching.
    - Very fast when using CDNs and caching.
- **GraphQL:**
    - Reduces network calls (single endpoint).
    - But requires more server-side processing.

Use Cases Where GraphQL Is Better:

✔ **Mobile Apps with Limited Bandwidth**
Fetch minimal data to save bandwidth.

✔ **Complex Applications with Nested Data**
Example: Social networks displaying posts + comments + author info in one request.

✔ **Multiple Frontends (Web, Mobile, IoT)**
Each client can request the shape of data it needs.

✔ **Rapid API Evolutions Without Versioning**
GraphQL handles field-level evolution easily.

Q8. Explain the importance of security in GraphQL APIs, especially in the context of data exposure and denial-of-service (DoS) attacks. Describe at least three security practices or techniques that can be used to secure GraphQL APIs in Node.js.

Answer:
GraphQL's flexibility can lead to security issues if not properly controlled:

1. Preventing Data Exposure

Because clients specify fields, they might query sensitive data unless properly restricted.

2. Protecting from DoS Attacks

GraphQL allows deeply nested or expensive queries that can overload the server.

Security Practices:

1. Query Depth Limiting

Restrict how deep a query can go.

graphqlDepthLimit(5)

Prevents expensive nested queries.

---

2. Query Complexity Analysis

Assign weights to fields and reject overly complex queries.

- Helps block "query bombs"
- Common in preventing DoS

---

3. Authentication & Authorization

- Validate JWT tokens in resolvers
- Use field-level authorization
- Ensure sensitive fields require proper permissions

---

4. Rate Limiting & Caching

- Limit queries per client
- Cache repeated queries to reduce load

---

5. Disable Introspection in Production

Prevents attackers from exposing your entire schema.