Company Overview – Ginger Media Group

Ginger Media Group is a digital and offline advertising agency founded in 2015 and headquartered in Bangalore, Karnataka. The company specializes in delivering creative campaign solutions that help businesses of all sizes achieve their marketing goals. With a track record of serving over 1,000 clients across India, Ginger Media Group has successfully executed thousands of projects focused on lead generation, branding, and targeted marketing strategies.

The company offers end-to-end advertising services that combine innovative ideas with practical execution, enabling brands to reach their audience effectively across both digital and offline channels. Their client base spans diverse industries, and their campaigns are tailored to meet specific business objectives, from increasing visibility to driving measurable sales results.

Culture and Work Environment

Ginger Media Group fosters a friendly, team-oriented, and collaborative workplace. The company emphasizes open communication, mutual respect, and celebrating both big achievements and small wins. Employees often engage in team dinners, birthday celebrations, and informal gatherings, building strong connections beyond day-to-day work.

The environment is relaxed yet performance-driven, with a focus on creativity, shared success, and maintaining positive energy in the workplace. Collaboration across teams is encouraged, ensuring that employees feel supported in both their professional growth and personal well-being.

Expectations from Freshers

Ginger Media Group seeks fresh graduates who are eager to learn, adaptable, and ready to contribute to real-world advertising projects. Key expectations include:

- **Technical Skills:** Basic knowledge of backend development tools and technologies, particularly for relevant roles involving Node.js, REST APIs, AWS, and MySQL. Familiarity with general programming concepts and web application workflows will be beneficial.
- **Learning and Adaptability:** Willingness to quickly learn new tools, advertising technologies, and industry practices. Freshers should be open to working on diverse campaigns and adapting to evolving client needs.
- **Communication and Teamwork:** Strong interpersonal skills and the ability to clearly communicate ideas are essential. A collaborative mindset and readiness to work closely with team members from different departments are valued.
- **Workplace Commitment:** Openness to working from the Bangalore office and engaging in hands-on campaign activities. Freshers should actively participate in discussions, brainstorming sessions, and team initiatives.
- **Positive Attitude:** An enthusiastic, problem-solving approach and readiness to take on new challenges as the company continues to grow.

Ginger Media Group offers an inclusive, learning-focused environment where newcomers can develop both technical and interpersonal skills, contribute to innovative campaigns, and grow alongside a dynamic team in the fast-paced advertising industry.

Ginger Media Group – Backend Developer Interview Process

1. **Screening Round (Virtual)**

   This is the first interaction, where they go through your resume and ask some basic questions. The aim is to check if your skills, background, and communication style match the role. Expect light technical discussions and questions about your past work or projects.

2. **Technical Round 1 (Virtual)**

   Here, they focus on the basics of backend development. You can expect questions on **Node.js**, **REST APIs**, and **databases** like MySQL. They might also touch on the basics of **AWS services**. This round can include small coding tasks, problem-solving exercises, and some simple system design questions to test your practical understanding.

3. **Technical Round 2 (Virtual)**

   This is a deeper technical round. The questions can get more complex — such as advanced **Node.js** concepts, API performance and scalability challenges, or real-world scenario problems. You might also discuss integrating AWS services like **EC2** or setting up **CI/CD pipelines**. Expect some problem-solving tasks where they look for clean, optimized solutions.

4. **HR & Final Discussion (Virtual)**

   The last round is more about you as a person — how you fit into the company's culture, your career goals, and how you handle workplace situations. You'll also discuss salary expectations, notice period, and any other final clarifications before the offer stage.

Topics to prepare:

Technical Round 1 – Core Backend Fundamentals

In this round, the interviewer usually wants to check if you have a solid foundation in backend development. They'll focus on how well you understand **Node.js**, **REST APIs**, **databases**, and some basic **AWS concepts**.

1. Node.js Basics

You should be comfortable explaining **asynchronous programming** — how Node.js handles tasks without blocking, using the **event loop**, **callbacks**, **promises**, and **async/await**. Expect to walk through how Node.js processes an HTTP request and sends a response.

2. REST API Development

Be prepared to design RESTful endpoints — for example, a set of routes for CRUD operations (Create, Read, Update, Delete). You should clearly explain the **request/response cycle** and how you'd structure an API in a real project.

3. MySQL – Database Management

They may give you simple SQL tasks like fetching data with SELECT, inserting with INSERT, or updating/deleting records. You should understand how to use **joins** to combine related tables and what **normalization** means. Also, know how to connect a Node.js app to a MySQL database.

4. AWS Fundamentals

You'll need a basic understanding of **cloud hosting** and why it's used. Know what services like **S3**, **EC2**, and **RDS** are for, and be able to explain how you might deploy a small backend application to AWS.

5. Basic Error Handling and Debugging

Be ready to discuss common backend issues (like unhandled promise rejections or database connection errors) and how you'd debug them.

6. General Coding & Problem Solving

They might throw in small logic problems — things involving arrays, objects, or string manipulation — to see how you think through solutions.

Technical Round 2 – Advanced Backend & Application Architecture

This round goes deeper into **advanced backend topics** and **real-world application design**. Here, they're checking if you can build and maintain scalable, secure systems.

1. Advanced Node.js Concepts
Expect questions about **middleware functions** in Express.js, how authentication works (JWT vs. sessions), and how to structure a project so it's easy to maintain.

2. API Security & Performance
You might need to explain how to secure an API with **rate limiting**, **input validation**, and **proper error handling**. They could also ask how you'd improve API performance — for example, by caching results or optimizing database queries.

3. AWS Services – In Depth
Here, they'll expect more than just knowing service names. You should be able to talk about setting up and managing **EC2 instances**, deploying using **CI/CD pipelines**, and integrating extra AWS services like **Lambda** or **SQS** when needed.

4. Database Optimization
Know how to speed up queries with **indexing**, manage **transactions**, and write more complex SQL (like multi-table joins, aggregate functions, and stored procedures).

5. System Design & Scaling
You may be asked to design a scalable backend for a given problem. You should understand **stateless architectures**, how to handle **concurrent requests**, and how to identify bottlenecks.

6. Scenario-Based Problem Solving
They might give you a real-world scenario — for example, adding a new feature to an existing backend or integrating a third-party service — and ask how you'd approach it.

Commonly Asked Questions:

1. What is the Node.js event loop and how does it work?

The **event loop** is at the heart of Node.js — it's what makes Node capable of handling many requests **without creating multiple threads**.
Node.js is **single-threaded** but still highly scalable because it uses the event loop to manage asynchronous operations.

How it works (simplified steps):

1. **Node starts** and initializes the event loop.
2. **Requests come in** — Node quickly hands off long-running tasks (like file reads, database queries, API calls) to the system's background workers.
3. When a task is complete, the **callback** is added to a queue.
4. The event loop **picks tasks from the queue** and executes their callbacks.
5. This continues indefinitely as long as there are tasks.

Example:

```
console.log('Start');

setTimeout(() => {
  console.log('Timer finished');
}, 2000);

console.log('End');

```

Output:

```
Start
End
Timer finished
```

Here, the setTimeout is handled asynchronously, allowing Node to continue executing without waiting.

Interview Tip:
Explain that this is why Node can handle **thousands of concurrent requests** efficiently — because the event loop avoids blocking the main thread.

Youtube Link: https://youtu.be/1_EVy3tls0k

---

2. What are the differences between callbacks, promises, and async/await?

All three are ways to handle **asynchronous operations** in JavaScript/Node.js, but they differ in syntax and readability.

Callbacks

- A **function passed as an argument** to be called when a task completes.
- Can cause **callback hell** if nested too much.

```
fs.readFile('file.txt', (err, data) => {
  if (err) throw err;
  console.log(data.toString());
});

```

Promises

- Represent a value that will be available **now, later, or never**.
- Have .then() and .catch() for handling results and errors.

```
readFilePromise('file.txt')
  .then(data => console.log(data))
  .catch(err => console.error(err));

```

Async/Await

- Built on top of promises.
- Makes asynchronous code look like synchronous code.

```
async function readFileData() {
  try {
    const data = await readFilePromise('file.txt');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}

```

Key Difference Table:

Feature	Callbacks	Promises	Async/Await
Readability	Low (nested)	Medium	High
Error Handling	Via callback parameters	.catch()	try...catch
Chaining	Hard	Easy	Easy

Youtube Link: https://youtu.be/JRNToFh3hxU

3. What is callback hell and how do you avoid it?

Callback hell happens when you have **many nested callbacks** — making code hard to read, maintain, and debug.

Example of callback hell:

```
doTask1((err, res1) => {
  doTask2(res1, (err, res2) => {
    doTask3(res2, (err, res3) => {
      console.log(res3);
    });
  });
});

```

Problems:

- Hard to read and debug.
- Error handling becomes messy.

Ways to avoid it:

1. **Use Named Functions**

```
function afterTask1(err, res1) { ... }
doTask1(afterTask1);

```

2. **Use Promises**
3. **Use Async/Await** for cleaner code.
4. **Modularize** logic into smaller functions.

Youtube Link: https://youtu.be/NOlOw03qBfw

4. What is middleware in Express.js and how is it used?

In Express.js, **middleware** is a function that runs **between** receiving a request and sending a response.

Purpose:

- Process request data.
- Run authentication checks.
- Log request details.
- Handle errors.

Basic structure:

```
app.use((req, res, next) => {
  console.log(`Request for ${req.url}`);
  next(); // Pass control to next middleware
});

```

Types of middleware:

1. **Application-level** (runs for all routes)

2. **Router-level** (runs for specific routes)
3. **Built-in** (e.g., express.json())
4. **Error-handling** middleware

Youtube Link: https://youtu.be/iIfkuL3v-b8

5. Explain the types of streams available in Node.js.

Streams are used for **reading/writing data in chunks** rather than loading all at once.

Types:

1. **Readable** — for reading data (fs.createReadStream()).
2. **Writable** — for writing data (fs.createWriteStream()).
3. **Duplex** — both read and write (e.g., TCP sockets).
4. **Transform** — duplex stream that also modifies data (e.g., compression).

Example:

```
const fs = require('fs');
const readStream = fs.createReadStream('file.txt');
readStream.on('data', chunk => {
  console.log(`Received ${chunk.length} bytes`);
});

```

Youtube Link: https://youtu.be/EcznOgzOdxI

6. What is the Reactor Pattern in Node.js?

The Reactor Pattern is the **design pattern** Node.js uses to handle **I/O operations asynchronously**.

Key points:

- Uses an **event demultiplexer** (the event loop).
- Registers events with callbacks.
- Executes callbacks when events are ready.

It's the reason Node can handle **non-blocking I/O** efficiently.

Youtube Link: https://youtu.be/mgaCXlCcXsE

---

7. How do you handle errors in a Node.js application?

Ways to handle errors:

1. **Error-first callbacks** — Pass error as the first argument.
2. **Try-Catch** — For synchronous and async/await code.

3. **Global error handling** — process.on('uncaughtException').
4. **Centralized error middleware** in Express.

Example in async/await:

```
try {
  const data = await readFilePromise('file.txt');
} catch (err) {
  console.error('Error reading file:', err);
}

```

Youtube Link: https://youtu.be/dfFSgB3Q84I

8. What is clustering in Node.js and why is it used?

Clustering allows you to **use all CPU cores** by creating multiple Node.js processes.

Without clustering:

- Node runs in a single process.
- Only one CPU core is used.

With clustering:

- Multiple worker processes share the same server port.
- Improves performance for heavy workloads.

Example:

```
const cluster = require('cluster');
const os = require('os');

if (cluster.isMaster) {
  os.cpus().forEach(() => cluster.fork());
} else {
  require('./server');
}

```

Youtube Link: https://youtu.be/6lHvks6R6cI

9. What are event emitters and how do you use them?

Event emitters allow **custom events** to be created and handled.

Example:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

emitter.on('greet', name => {
  console.log(`Hello, ${name}`);
});

```

```
emitter.emit('greet', 'John');

```

They're widely used in Node core modules (like http, fs).

Youtube Link: https://youtu.be/wINRm5arVlM

---

10. What is the difference between process.nextTick() and setImmediate()?

Both schedule code to run **asynchronously**, but timing differs.

- **process.nextTick()** — Runs **before** the next event loop iteration, immediately after the current function.
- **setImmediate()** — Runs **on the next event loop iteration**, after I/O events.

Example:

```
process.nextTick(() => console.log('nextTick'));
setImmediate(() => console.log('setImmediate'));
console.log('Start');

```

Possible output:

```
Start
nextTick
setImmediate
```

Youtube link: https://youtu.be/Bu2C31shcOQ

REST APIs Questions:

1. What are the key principles of REST?

REST (Representational State Transfer) is an architectural style for building APIs. Its principles are:

1. **Statelessness**
   - Every request from the client to the server must contain all the information needed to process it.
   - The server does not store client session state between requests.
   - Example: If you log into a website, the server doesn't "remember" you — instead, your token is sent with every request.
2. **Client–Server Separation**
   - The client (frontend) and the server (backend) are independent.
   - The client is only responsible for the user interface, and the server handles the business logic and data storage.

   - Example: A mobile app (client) can work with the same backend server as a website.
3. **Uniform Interface**
   - REST APIs use consistent resource naming (URLs) and HTTP methods.
   - Example: /users/123 always refers to user with ID 123, and a GET request always retrieves data.
4. **Resource-Based**
   - Everything is considered a resource, identified by a URL.
   - Example: /products refers to all products; /products/5 refers to product with ID 5.
5. **Cacheability**
   - Responses should be cacheable to improve performance.
   - Example: A GET /products call can be cached for 5 minutes to avoid hitting the database repeatedly.
6. **Layered System**
   - The API system can have multiple layers (e.g., load balancer, cache, proxy) without the client knowing.

Youtube links: https://youtu.be/pJ83mmqcvoQ

---

2. Explain the difference between GET, POST, PUT, and DELETE HTTP methods.

These are the four main CRUD operations in REST APIs:

1. **GET** – Retrieve data
   - Used to fetch resources from the server.
   - Example: GET /users returns a list of all users.
   - **Idempotent**: Calling it multiple times gives the same result (no changes happen).
2. **POST** – Create data
   - Used to create new resources on the server.
   - Example: POST /users with {name: "John"} creates a new user.
   - **Not idempotent**: Multiple POSTs create multiple records.
3. **PUT** – Update or replace data
   - Used to fully update a resource.
   - Example: PUT /users/1 with {name: "Jane", age: 25} replaces the entire record.
4. **DELETE** – Remove data
   - Used to delete a resource.
   - Example: DELETE /users/1 removes that user from the system.

Youtube links: https://youtu.be/sX9GdBQE_uo

---

3. How do you handle error responses in REST APIs?

Error handling in REST APIs should be consistent and clear.
Best practices:

1. **Use proper HTTP status codes**
    - 400 Bad Request – Client sent invalid data.
    - 401 Unauthorized – Missing or invalid authentication.
    - 404 Not Found – Resource doesn't exist.
    - 500 Internal Server Error – Something went wrong on the server.
2. **Return structured JSON error messages**

```
{
  "error": "Invalid Input",
  "details": "Email field is required",
  "code": 400
}

```

3. **Log errors internally** for debugging but show user-friendly messages to clients.

Youtube links:

---

4. How do you secure a REST API?

Securing REST APIs involves multiple layers:

1. **Authentication** – Verify the identity of the user.
    - Use JWT (JSON Web Token), OAuth2, or API keys.
2. **Authorization** – Ensure the user has permission for the action.
    - Example: A normal user should not be able to access admin routes.
3. **HTTPS Only** – Always encrypt data in transit to avoid man-in-the-middle attacks.
4. **Rate Limiting** – Limit requests per user/IP to prevent abuse.
    - Example: Max 100 requests per minute.
5. **Input Validation** – Sanitize and validate incoming data to avoid SQL injection or XSS.

Youtube links: https://youtu.be/dZ2CkvxuWIo

---

5. What is the difference between PUT and PATCH?

- **PUT** – Updates the entire resource. If a field is missing in the request, it will be reset or removed.
  Example:
  Current Data:
- { "name": "John", "age": 25, "city": "Delhi" }

  PUT Request:

{ "name": "John" }

Result:

{ "name": "John", "age": null, "city": null }

- **PATCH** – Updates only the specified fields, leaving others unchanged.
  Example:
  PATCH Request:
- { "name": "John" }

  Result:

  { "name": "John", "age": 25, "city": "Delhi" }

Youtube links: https://youtu.be/LJajkjI5RHE

---

6. How do you implement versioning in REST APIs?

API versioning ensures that changes don't break existing clients.
Ways to do it:

1. **URI Versioning** – Most common
   - Example: /v1/users and /v2/users
2. **Header Versioning** – Version specified in the request header.
   - Example: Accept: application/vnd.myapi.v2+json
3. **Query Parameter** – Version passed in the query string.
   - Example: /users?version=2

Youtube links: https://youtu.be/vsb4ZkUytrU

---

7. What HTTP status codes do you commonly use and what do they mean?

- **200 OK** – Request successful.
- **201 Created** – Resource created successfully.
- **204 No Content** – Request successful but no data returned.
- **400 Bad Request** – Invalid client request.
- **401 Unauthorized** – Authentication required.
- **403 Forbidden** – Client authenticated but doesn't have access.
- **404 Not Found** – Resource not found.
- **500 Internal Server Error** – Server failure.

Youtube links: https://youtu.be/qmpUfWN7hh4

---

8. What are idempotent HTTP methods?

An **idempotent method** is one where making the same request multiple times results in the same outcome.

- **GET, PUT, DELETE** are idempotent.
    - Example: Deleting the same resource twice — first time deletes it, second time it still results in "not found" but no extra effect.
- **POST** is **not** idempotent because it creates new records each time.

Youtube links: https://youtu.be/YlS-I-hCGzA

---

9. Explain CORS and how to handle it.

CORS (Cross-Origin Resource Sharing) is a security mechanism in browsers that restricts JavaScript code from making requests to a different domain than the one that served the page.

- Example: If your frontend is hosted at frontend.com and your API is at api.com, the browser will block requests unless CORS is enabled.

How to handle it:

- On the server, send the correct HTTP headers:
- Access-Control-Allow-Origin: https://frontend.com
- Access-Control-Allow-Methods: GET, POST, PUT, DELETE
- Access-Control-Allow-Headers: Content-Type, Authorization
- In development, you can allow * for all origins, but in production, restrict to trusted domains.

Youtube links: https://youtu.be/E6jgEtj-UjI

AWS Services (including EC2)

1. What is an EC2 instance and how do you secure it?

Explanation:
Amazon EC2 (Elastic Compute Cloud) is a service that lets you rent virtual servers in the cloud to run your applications. Think of it like renting a computer from AWS that you can access from anywhere over the internet. You can choose the operating system, processing power, storage, and network settings.

Security Best Practices in an Interview Context:
When securing an EC2 instance, you should demonstrate awareness of both AWS tools and general security practices:

- **Use Security Groups & NACLs** – Configure them to allow only necessary inbound/outbound traffic (e.g., allow SSH only from your IP).
- **Use Key Pairs** – Always use AWS key pairs (private key .pem file) for SSH authentication instead of passwords.
- **Keep Software Updated** – Regularly patch and update OS and application software to close vulnerabilities.
- **Disable Root Login** – Create separate users with least privilege access.
- **Use IAM Roles** – Assign AWS permissions through roles instead of embedding credentials in code.
- **Enable CloudWatch Alarms** – Monitor for unusual activity.
- **Enable AWS Inspector / GuardDuty** – For automated vulnerability scanning.

Example Interview Sentence:
"An EC2 instance is a virtual server hosted in AWS. To secure it, I ensure least privilege access, restrict ports with security groups, use IAM roles for credentials, patch systems regularly, and monitor activity using CloudWatch and GuardDuty."

Youtube Link: https://youtu.be/oD-SJhUmMTY

---

2. What is auto-scaling in AWS and how does it work?

Explanation:
Auto Scaling in AWS automatically adjusts the number of EC2 instances based on demand. This ensures that you have the right amount of resources running at all times — not too many when traffic is low, and not too few when traffic spikes.

It works by:

1. **Defining Policies** – Rules that decide when to add or remove instances based on CPU usage, network load, or custom CloudWatch metrics.
2. **Scaling Out** – Adding more instances when load increases.
3. **Scaling In** – Removing instances when load decreases to save costs.
4. **Integration with Load Balancers** – Works with Elastic Load Balancer so traffic is automatically distributed.

Example Interview Sentence:
"Auto Scaling ensures optimal performance and cost-efficiency by automatically adding instances when usage increases and removing them when demand drops."

Youtube Link: https://youtu.be/st4qpzz2FGc

---

3. Explain the purpose of AWS IAM and how it is used.

Explanation:
AWS IAM (Identity and Access Management) is a service for controlling access to AWS resources. It ensures that only the right people and systems can access specific resources.

Key Features:

- **Users** – Individual identities with permissions.
- **Groups** – Collections of users with common permissions.
- **Roles** – Temporary permissions for AWS services or applications.
- **Policies** – JSON-based documents that define permissions.

How it's Used:

- Create users for each person instead of sharing one account.
- Assign least privilege permissions using policies.
- Use Multi-Factor Authentication (MFA) for extra security.
- Assign roles to EC2 instances so they can access AWS services without storing keys.

Example Interview Sentence:
"IAM is AWS's security gatekeeper. I use it to create separate users, enforce least privilege, enable MFA, and assign roles so applications securely interact with AWS resources."

Youtube Link: https://youtu.be/hAk-7ImN6iM

---

4. What is AWS VPC and its main components?

Explanation:
A VPC (Virtual Private Cloud) is your own isolated network within AWS where you can launch AWS resources like EC2 instances.

Main Components:

- **Subnets** – Segments of the network (Public subnets for internet-facing resources, Private subnets for internal systems).
- **Route Tables** – Rules that control traffic flow.
- **Internet Gateway** – Lets resources in public subnets connect to the internet.
- **NAT Gateway** – Lets resources in private subnets access the internet securely.
- **Security Groups** – Instance-level firewalls.
- **NACLs (Network ACLs)** – Subnet-level firewalls.

Example Interview Sentence:
"A VPC is like your own private data center inside AWS. It gives you full control over networking, IP addressing, and routing for your resources."

Youtube Link: https://youtu.be/7_NNlnH7sAg

---

5. How does AWS CloudWatch help in monitoring?

Explanation:
AWS CloudWatch is a monitoring service for AWS resources and applications.

Capabilities:

- **Metrics Monitoring** – CPU usage, memory, disk, network.
- **Logs** – Collect and analyze logs from applications and AWS services.
- **Alarms** – Send alerts when metrics cross thresholds.
- **Dashboards** – Custom visual monitoring.

Example:
If CPU usage of an EC2 instance exceeds 80%, CloudWatch can trigger an auto-scaling action or send an alert.

Example Interview Sentence:
"CloudWatch gives real-time visibility into system health. I use it to monitor key metrics, set alarms, and automate responses to performance issues."

Youtube Link: https://youtu.be/Yxl7e88cTAQ

---

6. What are Elastic Load Balancers and why are they used?

Explanation:
An Elastic Load Balancer (ELB) automatically distributes incoming network traffic across multiple targets like EC2 instances to improve performance and reliability.

Benefits:

- **High Availability** – If one instance fails, traffic is routed to healthy ones.
- **Scalability** – Works with Auto Scaling.
- **Security** – Can terminate SSL/TLS connections.

Types:

- **Application Load Balancer (ALB)** – For HTTP/HTTPS, supports routing rules.
- **Network Load Balancer (NLB)** – For TCP/UDP, very fast and handles millions of requests.
- **Gateway Load Balancer (GWLB)** – For security appliances.

Youtube Link: https://youtu.be/Wjc_ka1W54g

---

7. What is the use case for AWS Snowball?

Explanation:
AWS Snowball is a physical device AWS sends to you to move large amounts of data securely into or out of AWS.

Use Cases:

- Data migration to AWS when internet bandwidth is too slow.
- Disaster recovery backups.
- Transferring terabytes or petabytes of data quickly.

Youtube Link: https://youtu.be/HxEhe0yVyHk

---

8. Describe the difference between AWS S3 and EBS.

Explanation:

- **S3 (Simple Storage Service)** – Object storage for files, images, backups. Accessible over the internet, highly durable, unlimited storage.
- **EBS (Elastic Block Store)** – Block storage for EC2 instances, works like a hard disk attached to a server, good for databases or OS storage.

Youtube Link: https://youtu.be/MsPZVa6hfJc

9. What is AWS Direct Connect and when would you use it?

Explanation:
AWS Direct Connect is a dedicated, private network connection between your on-premises data center and AWS.

Use When:

- You need high-speed, low-latency connection.
- You want secure data transfer without going over the public internet.
- Large data transfer volumes are required regularly.

Youtube Link: https://youtu.be/nCLQCrv-MZA

SQL Questions:

1. What is database normalization? Explain different normal forms.

Answer:
Database normalization is a process of organizing data in a database so that it reduces **redundancy** (duplicate data) and ensures **data integrity** (accuracy and consistency). It involves structuring tables and relationships in such a way that every piece of data is stored only once, and dependencies between data are logical.

Why normalization?

- Avoids duplication of data
- Makes the database more efficient
- Prevents update, insert, and delete anomalies

Normal Forms:

1. **First Normal Form (1NF)**
   - Each column should have atomic (indivisible) values.
   - No repeating groups or arrays.
   - Example: Instead of storing Phone: 12345, 67890 in one cell, create separate rows.
2. **Second Normal Form (2NF)**
   - Must be in **1NF**
   - All non-key columns must depend entirely on the **primary key** (not just part of it).
   - Removes **partial dependency** in composite keys.
3. **Third Normal Form (3NF)**
   - Must be in **2NF**
   - No **transitive dependency** (non-key columns shouldn't depend on other non-key columns).
   - Example: If City → State, and State → Country, store State and Country in a separate table.
4. **Boyce-Codd Normal Form (BCNF)**
   - A stricter version of **3NF**
   - Every determinant must be a candidate key.
   - Fixes anomalies not covered by 3NF.

Youtube Link: https://youtu.be/Fb9jZsCETVA

---

2. What are indexes and how do they improve performance?

Answer:
An **index** in MySQL is like a **table of contents in a book**. Instead of searching every page for a keyword, the index helps MySQL jump directly to the location where the data is stored.

How indexes improve performance:

- Speeds up SELECT queries because MySQL doesn't scan the whole table (full table scan).
- Allows faster sorting and filtering.

Types of Indexes in MySQL:

1. **Primary Index** – Created automatically on a primary key column.
2. **Unique Index** – Ensures all values in a column are unique.
3. **Full-Text Index** – Used for searching large text fields.
4. **Composite Index** – Indexes multiple columns together.

Drawback: Indexes take additional space and slow down INSERT/UPDATE operations because the index must be updated as well.

Youtube Link : https://youtu.be/kv3jC0P4gOc?si=V_p0BoeBuHDZ9O8p

3. Explain different types of JOINs in MySQL.

Answer:
A **JOIN** is used to combine rows from two or more tables based on a related column.

Types:

1. **INNER JOIN** – Returns rows where there's a match in both tables.

   ```
   sql
   CopyEdit
   SELECT employees.name, departments.dept_name
   FROM employees
   INNER JOIN departments
   ON employees.dept_id = departments.id;
   ```

2. **LEFT JOIN** – Returns all rows from the left table, and matching rows from the right table. Non-matching rows from the right table will be NULL.
3. **RIGHT JOIN** – Returns all rows from the right table, and matching rows from the left table.
4. **FULL OUTER JOIN** – Returns all rows when there is a match in either table (not directly supported in MySQL, can be done using UNION).
5. **CROSS JOIN** – Returns a **cartesian product** of the two tables.

Youtube Link: https://youtu.be/SYa2GQDT_g4

4. What are transactions and how does MySQL handle them?

Answer:
A **transaction** is a group of SQL operations that are executed together as a single unit. Either all changes are committed, or none are applied.

MySQL handles transactions using the InnoDB storage engine and follows the **ACID** properties.

Example:

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

If any step fails, you can ROLLBACK to undo changes.

Got it — I'll **fully elaborate** these answers so they sound strong and confident when you explain them in an interview, while also showing your understanding with examples and reasoning.

5. How do you optimize slow queries?

Slow queries can affect the performance of your application, especially when the dataset grows. Optimizing them is a mix of **good indexing**, **query rewriting**, and **data structure improvements**.

Detailed Strategies:

1. **Use Indexes Effectively**
   - Indexes work like a book's table of contents, allowing MySQL to jump to the relevant row instead of scanning the entire table.
   - Create indexes on columns that appear frequently in WHERE, JOIN, and ORDER BY clauses.
   - Example:
   - CREATE INDEX idx_customer_name ON customers(name);

     This helps if you often search customers by name.

2. **Avoid SELECT ***
   - Fetching all columns means MySQL has to read unnecessary data from disk into memory.
   - Instead, only select the required columns:
   - SELECT name, email FROM customers WHERE status = 'active';
3. **Use EXPLAIN to Analyze Queries**
   - The EXPLAIN keyword shows how MySQL will execute your query. It tells you if indexes are being used or if a full table scan is happening.
   - EXPLAIN SELECT name FROM customers WHERE email = 'abc@example.com';
4. **Limit Results for Large Data Sets**
   - If you don't need all records at once, use LIMIT to reduce data load:
   - SELECT * FROM orders LIMIT 50;
5. **Normalize Tables Where Possible**
   - Removing redundant data makes the database smaller and queries faster. For example, instead of storing the same city_name in multiple rows, store it in a cities table and reference it by ID.
6. **Avoid Subqueries if JOINs Can Work Better**
   - Subqueries often require creating a temporary table in memory, slowing down performance.
   - Replace:
   - SELECT * FROM customers WHERE id IN (SELECT customer_id FROM orders);

     With:

     SELECT DISTINCT c.* FROM customers c JOIN orders o ON c.id = o.customer_id;

Youtube link: https://youtu.be/3pu7hoR1HbU

6. What is ACID compliance?

ACID stands for the **four properties of a reliable database transaction**. In MySQL, the InnoDB engine is fully ACID-compliant, which ensures your data stays correct and safe.

Detailed Explanation:

1. **Atomicity** – *All or nothing.*
   - A transaction is treated as a single unit. If any step fails, the whole transaction is rolled back.
   - Example: If you transfer money between accounts, both debit and credit steps must succeed; otherwise, neither should.
2. **Consistency** – *Valid state before and after.*
   - Data must follow rules, constraints, and relationships before and after a transaction.
   - Example: If an order references a non-existing customer ID, the transaction fails.
3. **Isolation** – *Transactions don't affect each other.*
   - If two people edit the same record at the same time, changes won't interfere.
   - Example: One user's balance update won't be visible to another until it's committed.
4. **Durability** – *Data is permanent.*
   - Once a transaction is committed, data remains saved even after a crash or power loss.
   - MySQL achieves this with write-ahead logs and disk persistence.

Youtube link: https://youtu.be/vNbslx32izM

7. How do you backup and restore MySQL databases?

Backup (using mysqldump):

```
mysqldump -u root -p database_name > backup.sql
```

- mysqldump creates a text file containing all SQL commands needed to recreate the database.
- -u is the username, -p prompts for a password.

Restore:

```
mysql -u root -p database_name < backup.sql
```

- This runs all SQL commands from backup.sql to recreate tables and data.

Best Practices:

- Always test backups by restoring to a staging environment.
- Automate backups with cron jobs for production databases.
- Use incremental backups for large datasets.

Youtube link: https://youtu.be/rLvB4Xvb5PY

---

8. What are stored procedures and when would you use them?

A **stored procedure** is a set of SQL statements stored in the database that can be executed on demand.

Advantages:

- **Reusability** – Write once, call multiple times.
- **Performance** – Precompiled, so execution is faster.
- **Security** – Prevents direct table access, letting you control what's exposed.

Example:

```
DELIMITER //
CREATE PROCEDURE GetAllEmployees()
BEGIN
   SELECT * FROM employees;
END //
DELIMITER ;
```

Call the procedure:

```
CALL GetAllEmployees();
```

When to Use:

- Complex business logic that's repeated often.
- Preprocessing data before sending to the application.
- Enforcing data consistency without relying solely on application code.

Youtube Links: https://youtu.be/lYitPp6UWOQ

EC2 (Elastic Compute Cloud) Questions:

1. What is EC2 and how does it differ from other AWS compute services?
Amazon EC2 (Elastic Compute Cloud) is a cloud-based virtual server service that allows you to run applications without buying and maintaining physical servers. Instead of owning hardware, you rent computing capacity from AWS and pay for what you use.

The key word here is "elastic" — EC2 lets you increase or decrease computing resources as your needs change, often within minutes.

It differs from other AWS compute services mainly in **control and flexibility**:

- With EC2, you have full control over the server environment — you choose the operating system, configure the storage, install software, and control network settings.
- AWS Lambda, on the other hand, is serverless — you only run code without managing servers at all. This is great for short-running tasks but not for hosting long-running applications.
- AWS Elastic Beanstalk automates the setup and scaling for you — it still uses EC2 under the hood but removes the need for manual server management.
- AWS Fargate focuses on running containers without having to manage EC2 instances.

Think of EC2 as a blank canvas where you have maximum control, while other services are more managed or specialized for certain use cases.

Youtube Link: https://youtu.be/_0vveSJpWs4

---

2. How do you launch and connect to an EC2 instance?
Launching an EC2 instance is like setting up a new virtual machine in the cloud:

1. **Sign in to the AWS Management Console** and open the EC2 dashboard.
2. **Click "Launch Instance"** and choose an Amazon Machine Image (AMI) — this is the OS template, such as Ubuntu, Amazon Linux, or Windows.
3. **Select an Instance Type** based on your CPU, memory, and performance needs.
4. **Configure instance details** — number of instances, network settings (VPC, subnet), IAM role, and shutdown behavior.
5. **Add storage** — decide the size and type of your EBS volume (SSD or HDD).
6. **Configure security group** — define inbound and outbound traffic rules (e.g., allow SSH for Linux or RDP for Windows).
7. **Generate or use an existing key pair** — you'll use this to securely connect to the instance.
8. **Launch the instance** and wait until its status is "running."

To connect:

- For Linux, use an SSH client with the `.pem` key you downloaded. Example:
  ```
  ssh -i your-key.pem ec2-user@your-public-ip
  ```
- For Windows, use RDP by downloading the remote desktop file from the console and entering your password (which you decrypt using the `.pem` file).

Youtube Link: https://youtu.be/Qs9Ym-83gkg

---

3. What security best practices should you follow for EC2?
Security is critical because EC2 is accessible over the internet:

- **Use key pairs instead of passwords** for SSH/RDP access. This is more secure and harder to brute-force.
- **Limit inbound traffic** — only open necessary ports in the security group (e.g., port 22 for SSH, 80/443 for web traffic).

- **Regularly patch the OS** — update packages to fix vulnerabilities.
- **Use IAM roles** instead of embedding credentials inside your instance — this avoids accidental leaks.
- **Enable detailed logging** with CloudWatch and store logs securely.
- **Encrypt data at rest and in transit** — use EBS encryption and HTTPS connections.
- **Use a bastion host or VPN** for connecting to instances instead of exposing them directly to the internet.
- **Terminate unused instances** — don't leave idle servers running unnecessarily, as they could become security liabilities.

Youtube Link: https://youtu.be/En6kC_QYqBg

---

4. How does EC2 auto-scaling work?
EC2 Auto Scaling automatically adjusts the number of instances in a group to meet demand:

- You define a **Launch Template** or **Launch Configuration** that specifies instance type, AMI, and settings.
- You create an **Auto Scaling Group (ASG)** with a minimum, maximum, and desired number of instances.
- You define **scaling policies** — for example, "Add two instances when CPU usage exceeds 80% for 5 minutes" or "Remove one instance when CPU usage falls below 40% for 10 minutes."
- AWS automatically launches or terminates instances to keep your application responsive and cost-efficient.

Auto-scaling ensures you never overpay for idle resources and that your application can handle sudden traffic spikes without downtime

Youtube Link: https://youtu.be/LrVWHCWnecI

---

5. What are EC2 instance types and how do you choose one?
EC2 offers different instance families, each optimized for specific workloads:

- **General Purpose (e.g., t3, m5)** — Balanced CPU, memory, and networking for most applications.
- **Compute Optimized (e.g., c5)** — High CPU performance for compute-intensive tasks like gaming servers or scientific modeling.
- **Memory Optimized (e.g., r5, x1)** — Large memory for in-memory databases, big data analytics.
- **Storage Optimized (e.g., i3, d2)** — High disk throughput for large datasets, data warehousing.
- **Accelerated Computing (e.g., p3, g4)** — GPU-powered for AI, machine learning, and video rendering.

To choose:

- Understand your workload needs (CPU-heavy, memory-heavy, GPU processing, or storage-intensive).
- Consider your budget — smaller instances cost less but may not handle peak loads.
- Test with different instance types to find the best balance between performance and cost

Youtube Link: https://youtu.be/oOOxYl7RAdk

---

6. How can you troubleshoot connectivity issues on EC2?
If you can't connect to your EC2 instance, check these:

- **Security group rules** — Ensure the inbound rules allow your IP to connect on the right port (22 for SSH, 3389 for RDP).
- **Network ACLs** — Make sure they allow inbound/outbound traffic.
- **Elastic IP** — If your instance doesn't have a public IP, assign one.
- **Key pair issues** — Check that you're using the right `.pem` file with correct permissions.
- **OS firewall** — The instance's internal firewall (e.g., `iptables` or `ufw`) might be blocking traffic.
- **Route table** — Verify that the route table in your VPC points internet-bound traffic to an internet gateway.
- **Instance state** — The instance must be in "running" state and have passed health checks.

Youtube Link: https://youtu.be/LuU2sbYHVg0

---

7. How do you monitor EC2 instance health?
You can monitor EC2 using AWS CloudWatch:

- **Basic metrics** — CPU utilization, network traffic, disk reads/writes.
- **Custom metrics** — Application-specific metrics, such as request counts or error rates.
- **Status checks** — AWS automatically performs two checks:
  1. **System status check** — Ensures AWS infrastructure hosting the instance is working.
  2. **Instance status check** — Verifies the OS is running and reachable.
- **Alarms** — Set thresholds so that you get notified (via email/SMS) when performance drops or an issue occurs.

You can also install the **CloudWatch Agent** for detailed OS-level metrics like memory usage and disk space.

Youtube Link: https://youtu.be/IeW5ISqYpmYa

---

8. Explain how you would deploy a web app on EC2.
Here's a simple flow for deploying a web application:

1. **Launch an EC2 instance** with the OS you prefer (Ubuntu, Amazon Linux).
2. **Connect via SSH** and update packages.
3. **Install required software** — e.g., Apache/Nginx for web hosting, MySQL/PostgreSQL for database, or Node.js/Java for app runtime.
4. **Upload your application files** — use `scp`, `git clone`, or CI/CD pipelines.
5. **Configure the application** — set environment variables, database connections, etc.
6. **Open required ports** in the security group (e.g., 80 for HTTP, 443 for HTTPS).
7. **Test the application** using the instance's public IP.
8. (Optional) **Attach a domain name** using Route 53 and configure HTTPS with AWS Certificate Manager.
9. **Enable auto-scaling and load balancing** if expecting high traffic.

This process can be manual or automated using scripts, Ansible, or AWS CodeDeploy for continuous delivery.

Youtube Link: https://youtu.be/bZqLeiid5ZM

CI/CD Questions:

1. What is the difference between Continuous Integration, Continuous Delivery, and Continuous Deployment?

Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment sound similar, but they represent different stages in the modern software release process:

- **Continuous Integration (CI)**:
  CI is about **integrating code changes frequently** into a shared repository (often multiple times a day).
  Every time a developer pushes code, an **automated build** is triggered to compile the code, run tests, and check that nothing breaks.
  Example: In a Node.js project, pushing changes to GitHub triggers a GitHub Actions workflow that runs ESLint, Jest unit tests, and builds the application.

  Purpose: Catch bugs early, improve code quality, and make integration painless.

- **Continuous Delivery (CD)**:
  Continuous Delivery extends CI by ensuring that the code is **always in a deployable state**.
  After CI passes, CD prepares the application for deployment and may push it to a staging or pre-production environment.
  Example: After tests pass, the pipeline automatically deploys the Node.js app to a staging server where QA can test it before production release.

Purpose: Automate the release process up to the final decision point; deployment to production is still a **manual approval** step.

- **Continuous Deployment**:
  Continuous Deployment goes one step further — it **automatically deploys every code change to production** without human approval, as long as it passes all automated tests and checks.
  Example: You commit to `main` → CI builds & tests → pipeline deploys it directly to production.

  Purpose: Deliver features to users faster, reduce release delays, and make deployments routine.

Summary:

- CI = Integrate & test frequently
- Continuous Delivery = Ready to deploy anytime (manual trigger for prod)
- Continuous Deployment = Deploy to production automatically after tests

Youtube Link: https://youtu.be/9TN4x2pMPpE

---

2. What are common CI/CD tools you have used or know about?

There are several CI/CD tools, each with its own strengths:

- **Jenkins** – Open-source, highly configurable, supports plugins, works with almost any tech stack.
- **GitHub Actions** – Integrated with GitHub, great for repo-based workflows, YAML-based pipelines.
- **GitLab CI/CD** – Built into GitLab, supports versioned pipelines in `.gitlab-ci.yml`.
- **CircleCI** – Cloud-native, optimized for speed and parallel execution.
- **Azure DevOps Pipelines** – Part of Azure DevOps suite, integrates well with Microsoft ecosystem.
- **Bitbucket Pipelines** – Integrated with Bitbucket repos.
- **AWS CodePipeline** – AWS-native CI/CD service.

Example: In a Node.js project, I might use GitHub Actions because:

1. The repository is already on GitHub.
2. YAML workflows are simple to configure.
3. Easy integration with npm, Jest, ESLint, and deployment scripts.

Youtube Link: https://youtu.be/VcXSnFe2I8k

---

3. How would you set up a CI/CD pipeline for a Node.js application?

Step-by-step approach:

1. **Version Control Setup**
   - Store the Node.js code in GitHub/GitLab/Bitbucket.
   - Use branching strategy (`main` for production, `develop` for testing).
2. **Create the CI Pipeline**
   - On push or pull request, run:
     - `npm install` (to install dependencies)
     - `npm run lint` (to ensure code style)
     - `npm test` (to run unit tests with Jest/Mocha)
     - `npm run build` (to compile/minify if needed)
3. **Set up CD (Delivery/Deployment)**
   - After tests pass:
     - Deploy to staging environment automatically.
     - If approved, deploy to production (Continuous Delivery) or deploy automatically (Continuous Deployment).
4. **Example GitHub Actions workflow:**

```
name: Node.js CI/CD
on:
  push:
    branches: [ main, develop ]
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Use Node.js 18
        uses: actions/setup-node@v3
        with:
          node-version: 18
      - run: npm install
      - run: npm run lint
      - run: npm test
      - run: npm run build
      - name: Deploy to Staging
        run: bash deploy-staging.sh
```

Youtube link: https://youtu.be/WG_xH9AEP-g

---

4. How do you handle rollbacks in a CI/CD pipeline?

Rollback strategies are critical for reducing downtime when a release fails:

- **Blue-Green Deployment**:
  - Keep two environments — Blue (current production) and Green (new version).
  - If Green fails, switch traffic back to Blue instantly.
- **Canary Deployment**:
  - Release the new version to a small percentage of users.
  - If errors spike, roll back before full rollout.
- **Version Tagging in CI/CD**:
  - Tag releases in Git (`v1.0.2`, `v1.0.3`).

    - If needed, redeploy the last stable tag.
- **Database Rollback Plan**:
    - Keep SQL rollback scripts ready for schema changes.
- **Example in practice:**
  In a Node.js + Docker + Kubernetes setup, I store older container images in a registry. If a new release fails, the pipeline triggers `kubectl rollout undo deployment my-app` to revert to the previous stable image.

Youtube Links: https://youtu.be/G8fYYrxqF5E

---

5. What are the common challenges you face when setting up CI/CD pipelines?

- **Long Build Times** – Can be solved by caching dependencies (e.g., `node_modules`).
- **Flaky Tests** – Cause false failures; fix by stabilizing test cases.
- **Secrets Management** – Need to store API keys securely (e.g., GitHub Secrets, Vault).
- **Environment Differences** – Works locally but fails in pipeline; use Docker for consistency.
- **Rollback Readiness** – Not having a rollback plan in place.
- **Complex Merge Conflicts** – Require good branching strategy.

Youtube link: https://youtu.be/m1oMj29P--Y

---

6. How do you automate testing in CI pipelines?

Automating testing ensures every change is validated without manual effort:

- **Unit Tests**: Run using Jest or Mocha after build.
- **Integration Tests**: Test APIs and database interactions.
- **Static Analysis**: Use ESLint to check for code style and potential issues.
- **Example:**
  In GitHub Actions, after code checkout:
    - `npm install`
    - `npm run lint`
    - `npm test -- --coverage` (to ensure minimum coverage)

This ensures no code is merged without passing tests.

Youtube Link: https://youtu.be/EcaNjCZI7eY

---

7. What is Infrastructure as Code (IaC) and how does it relate to CI/CD?

IaC means managing infrastructure (servers, networks, databases) using code, instead of manual setup.

- Tools: Terraform, AWS CloudFormation, Ansible.
- Benefits: Consistency, version control, easy rollback.

Relation to CI/CD:

- IaC scripts can be part of the pipeline, so when deploying a Node.js app, the CI/CD system first provisions/updates the infrastructure automatically before deploying code.
- Example:
    - Step 1: Terraform creates AWS EC2 and RDS instances.
    - Step 2: CI/CD deploys Node.js app to EC2.

Youtube link: https://youtu.be/B80NfQaCwxY

---

8. How does containerization (e.g., Docker) fit into CI/CD workflows?

- Docker packages the application and dependencies into a single **image** so it runs consistently across environments.
- In CI/CD:
    1. **Build Stage**: CI builds a Docker image after tests pass.
    2. **Push Stage**: Image is pushed to a registry (Docker Hub, ECR).
    3. **Deploy Stage**: CD pulls the image and deploys it to production (e.g., Kubernetes).

Example:

- Step 1: CI builds image `myapp:1.0.5`.
- Step 2: Push to Docker Hub.
- Step 3: Production server pulls and runs it — guaranteeing the same behavior as in development.

Link: https://youtu.be/JyEwKm-OfxA

HR Questions:

1. Tell me about yourself
Answer:
My name is [Your Name], and I recently graduated in Computer Science from [Your College]. I enjoy solving problems and building useful tools with technology. I have a good understanding of Node.js and databases, and I'm excited to apply what I've learned in a real job. I'm hardworking, willing to learn, and ready to contribute as a backend developer at Ginger Media Group.

---

2. Why do you want to work at Ginger Media Group?
Answer:
I admire Ginger Media Group because it works on creative projects that help brands grow. I like the company's positive work culture and focus on learning. I believe working here will help me grow my skills while contributing to meaningful projects.

---

3. What are your strengths?
Answer:
I learn quickly and adapt well to new situations. I can stay focused on tasks, meet deadlines, and work well with others. I also pay attention to detail, which helps me deliver good-quality work.

---

4. What is your biggest weakness?
Answer:
Sometimes I spend extra time double-checking my work to make sure it's correct. This helps me avoid mistakes, but I'm learning to balance accuracy with speed.

---

5. Can you work under pressure?
Answer:
Yes. During college, I often had to finish multiple assignments in a short time. I stayed calm, planned my work, and focused on one task at a time. This approach helps me work well under pressure.

---

6. Where do you see yourself in five years?
Answer:
In five years, I want to be a skilled backend developer, taking on bigger responsibilities and helping my team with important projects. I also hope to mentor new team members.

---

7. Why should we hire you?
Answer:
I have the skills you're looking for, and I'm eager to learn and work hard. I'm motivated, dependable, and ready to start contributing to the team from day one.

---

8. How do you handle feedback and criticism?
Answer:
I see feedback as a way to get better. I listen carefully, try to understand the points made, and then work on improving my performance.

9. Are you willing to relocate and work from the office?
Answer:
Yes, I'm happy to relocate and work from the office in Bangalore. I think working in person will help me learn faster and connect better with the team.

10. Do you have any questions for us?
Answer:
Yes. Could you tell me about the training process for new team members and the kind of projects I might work on first?