1. How to secure API?

Ans: Here's how you can do it:

1. Enable HTTPS:

* **Obtain an SSL/TLS Certificate:** Get a certificate from a trusted Certificate Authority (CA).
* **Configure Spring Boot:**
  + Add the certificate to your project's resources.
  + Update your application.properties or application.yml file to enable HTTPS:

Code

server.port=8443  
server.ssl.key-store=path/to/your/keystore.jks  
server.ssl.key-store-password=your\_keystore\_password  
server.ssl.key-alias=your\_key\_alias

2. Authentication and Authorization:

* **Spring Security:** Use Spring Security to manage authentication and authorization:

Java

@Configuration  
@EnableWebSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http  
 .authorizeRequests()  
 .antMatchers("/api/public/\*\*").permitAll()  
 .antMatchers("/api/admin/\*\*").hasRole("ADMIN")  
 .anyRequest().authenticated()  
 .and()  
 .httpBasic(); *// Or use other authentication methods like OAuth2*  
 }  
}

* **OAuth2 and JWT:** Consider using OAuth2 for authorization and JWT (JSON Web Tokens) for authentication.

3. Input Validation:

* **Validate All Input:**

Validate all input data to prevent SQL injection, cross-site scripting (XSS), and other attacks.

* **Use Spring's Validation Framework:**

Leverage Spring's validation framework to validate request parameters and DTOs.

4. Cross-Site Request Forgery (CSRF) Protection:

* **Enable CSRF Protection:** Spring Security provides CSRF protection out of the box. Ensure it's enabled:

Java

http.csrf().csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());

5. Method-Level Security:

* **Use @PreAuthorize and @PostAuthorize:** Use Spring Security's annotations to secure methods based on roles and permissions.

6. Encrypt Sensitive Data:

* **Encrypt Data at Rest:** Encrypt sensitive data in your database.
* **Encrypt Data in Transit:** Use HTTPS to encrypt data transmitted over the network.

7. Regular Updates:

* **Keep Dependencies Updated:** Regularly update your Spring Boot and Spring Security dependencies to patch known vulnerabilities.

8. Consider API Gateway:

* **Use an API Gateway:** An API gateway can provide additional security features like rate limiting, request validation, and centralized authentication.

9. Security Testing:

* **Perform Regular Security Audits:**

Conduct regular security audits to identify and address potential vulnerabilities.

* **Penetration Testing:**

Consider penetration testing to simulate real-world attacks and identify weaknesses.

2. How Spring security works?

Ans: go through this link : https://www.marcobehler.com/guides/spring-security

3. How will you integrate oauth2 with your application?

Ans: go through this link : https://spring.io/guides/tutorials/spring-boot-oauth2

4. Existing API fetching details with id and now same api should fetch details based on mailId without changing exisiting API.

Ans: You need to use javascript Promise to handle multiple asynchronous calls. Just chain a then after your first fetch, you will be able to access to the Id when it will be available.

5. API Versioning.

Ans: https://www.geeksforgeeks.org/spring-boot-versioning-a-rest-api/

6. How to deploy springboot application on AWS?

Ans: https://www.geeksforgeeks.org/deploy-a-spring-boot-application-with-aws/

7. Write deployment script?

Ans: https://www.geeksforgeeks.org/deployment-of-spring-boot-application-in-jenkins/

8.  Explain the retry mechanism in notification service. How to you handle the queue size considering lot of retries

Ans: A retry mechanism in a notification service is a system designed to automatically attempt to resend a notification if the initial delivery fails due to temporary issues like network glitches or device unavailability, typically using a queue to store failed notifications and retrying them with increasing delay intervals, often following an exponential backoff strategy to avoid overwhelming the system during high failure periods; to manage queue size with many retries, you can implement strategies like dead letter queues (DLQs) to move persistently failing notifications to a separate queue for manual review, set reasonable retry limits, and employ exponential backoff to space out retry attempts effectively.

Key components of a notification retry mechanism:

* **Queue:**

A message queue stores pending notifications, including those that failed delivery attempts, allowing for orderly processing and retries.

* **Retry logic:**

This defines how many times to retry a notification and the delay between attempts, often using an exponential backoff strategy where the delay increases after each failed attempt.

* **Error handling:**

The notification service identifies specific error codes that indicate a temporary issue and should be retried, distinguishing them from permanent errors where further attempts are futile.

* **Dead Letter Queue (DLQ):**

A separate queue where notifications that exceed the retry limit are moved for manual inspection or logging.

Managing queue size with retries:

* **Exponential Backoff:**

This strategy significantly increases the delay between retries after each failure, preventing the system from being overloaded with immediate re-attempts when encountering a burst of failures.

* **Retry limits:**

Set a maximum number of retry attempts for each notification to prevent infinite retry loops and ensure failed notifications eventually reach a DLQ.

* **Prioritization:**

Consider prioritizing critical notifications for faster retries while allowing longer delays for less important ones.

* **Dynamic queue management:**

Monitor queue size and dynamically adjust retry intervals based on current load to prevent backlog buildup.

Implementation considerations:

* **Idempotency:**

Ensure that retrying a notification does not result in duplicate deliveries to the recipient.

* **Logging and monitoring:**

Track retry attempts, error codes, and delivery status to identify potential issues and optimize retry strategy.

* **Alerting:**

Set up alerts to notify operators when retry rates become abnormally high, indicating potential problems with the notification system.

Example scenario:

* A user tries to access their account but receives a "failed login" notification due to a network issue.
* The notification service queues the message and attempts to resend it after a short delay.
* If the delivery fails again, the retry delay is increased exponentially.
* After a set number of retries, the notification is moved to a DLQ for further review if the issue persists.

9. How do you fetch tons of data from Database (Looking for optimized approach)

Ans: To efficiently fetch large amounts of data from a database, focus on optimizing your SQL queries by using appropriate indexes, selecting only necessary columns, avoiding unnecessary joins and subqueries, limiting the data retrieved per request, and considering techniques like data partitioning or batching when dealing with extremely large datasets; additionally, ensure your database schema is well-normalized and monitor performance to identify potential bottlenecks.

Key strategies for optimized data fetching:

* **Proper Indexing:**
  + Create indexes on columns frequently used in WHERE clauses to significantly improve query speed.
  + Analyze your queries to identify which indexes would provide the most benefit.
* **Query Optimization:**
  + **Select specific columns:** Avoid using SELECT \* and only retrieve the columns you actually need.
  + **Limit data retrieved:** Use LIMIT clauses to fetch data in smaller chunks if necessary.
  + **Efficient joins:** Optimize JOIN operations by using the correct join type and avoiding unnecessary joins.
  + **Avoid subqueries:** Rewrite queries to minimize the use of subqueries where possible.
  + **Use EXPLAIN:** Use the EXPLAIN command to analyze your query execution plan and identify potential performance issues.
* **Database Design:**
  + **Normalization:** Ensure your database schema is properly normalized to reduce data redundancy and improve query efficiency.
  + **Partitioning:** For very large datasets, consider partitioning data across multiple tables to improve query performance for specific data subsets.
* **Data Fetching Techniques:**
  + **Batching:** Fetch data in batches instead of retrieving all data at once, especially when processing large datasets.
  + **Streaming:** Utilize streaming APIs to process data as it is retrieved from the database, avoiding the need to load the entire dataset into memory.
* **Other considerations:**
  + **Connection pooling:** Use database connection pooling to reduce the overhead of establishing new connections.
  + **Monitoring performance:** Regularly monitor your database performance to identify bottlenecks and adjust queries accordingly.
  + **Consider NoSQL databases:** Depending on your data structure, a NoSQL database may be more efficient for certain types of data retrieval.

10. Reverse a string String = "Mr ABC" result = "rM CBA". (Looking for optimization in terms of memory).

Ans: https://www.simplilearn.com/tutorials/java-tutorial/reverse-a-string-in-java

11. What is dependency injection. what's the better alternative for @Autowired

Ans: Dependency injection is a programming technique where an object receives the dependencies it needs from external sources, rather than creating them internally, making the object more flexible and testable; in Spring framework, the preferred alternative to @Autowired is constructor injection which explicitly passes dependencies through a class's constructor, generally considered a better practice for clarity and maintainability.

Key points about dependency injection and @Autowired:

* What is @Autowired?:

In Spring, @Autowired is an annotation that allows automatic dependency injection, meaning Spring will automatically find and inject the required dependencies into a class based on type matching.

* Why is constructor injection preferred over @Autowired?:
  + **Explicitness:** It clearly shows which dependencies a class needs by listing them in the constructor.
  + **Immutability:** Encourages creating immutable objects by ensuring all dependencies are set during object creation.
  + **Testability:** Makes unit testing easier as you can easily mock and inject dependencies during test setup.

Example of constructor injection:

Code

public class MyService {

private final MyRepository repository;

@Autowired

public MyService(MyRepository repository) {

this.repository = repository;

}

// ... other methods

}

12. Java 8 Features, Spring boot, Rest Api

13. Default Method and Static Method in Spring boot

14. Reverse of a String up to given characters.

To reverse a string up to a given number of characters, you can convert the string into a character array, then iterate through the array, swapping characters from the beginning and end positions up to the specified character index, effectively reversing only the portion of the string you want.

Here's a breakdown of the steps in Python:

Code

def reverse\_upto(string, num\_chars):

"""

Reverses a string up to the specified number of characters.

Args:

string: The input string.

num\_chars: The number of characters to reverse.

Returns:

The reversed string up to the specified number of characters.

"""

char\_array = list(string) # Convert string to character array

left, right = 0, min(len(char\_array) - 1, num\_chars - 1) # Set pointers

while left < right:

char\_array[left], char\_array[right] = char\_array[right], char\_array[left] # Swap characters

left += 1

right -= 1

return ''.join(char\_array[:num\_chars]) + string[num\_chars:] # Join characters back to string

Explanation:

1. 1. **Convert to character array:**
   * char\_array = list(string): This line turns the input string into a list of characters, allowing for easy manipulation of individual characters.
2. 2. **Set pointers:**
   * left, right = 0, min(len(char\_array) - 1, num\_chars - 1):
     + left starts at the beginning of the array (index 0).
     + right starts at the index just before the desired reversal point, ensuring we don't go out of bounds if num\_chars is larger than the string length.
3. 3. **Swap characters:**
   * while left < right:
     + This loop continues until the pointers cross, meaning we've reached the middle of the reversal section.
     + char\_array[left], char\_array[right] = char\_array[right], char\_array[left]: This line swaps the characters at the current left and right positions.
     + left += 1, right -= 1: Move the pointers one step closer to the middle.
4. 4. **Join back to string:**
   * ''.join(char\_array[:num\_chars]) + string[num\_chars:]:
     + char\_array[:num\_chars] takes only the first num\_chars characters from the reversed array.
     + ''.join(...) joins these characters back into a string.
     + string[num\_chars:] takes the remaining part of the original string after the reversed section and appends it.

Example usage:

Code

print(reverse\_upto("Hello World", 5)) # Output: "olleH World"

Key points:

* This algorithm is efficient as it only iterates through the desired portion of the string.
* Make sure to handle edge cases like when num\_chars is larger than the string length.
* You can adapt this logic to other programming languages with similar string manipulation techniques.

15. Scenario Based Questions on Spring boot and Rest Api Used in Current Project.

16. How to deals with Billions of data

17. How to Deploy application using AWS

18. Queries on database

19. Scenario Based Questions on Implementation, Security Encryption and decryption on Spring boot and Rest API.

20. Current Project Contribution

21. Logic implementation for Multithreading

22. How to deals with Billions of data using Spring boot for a project

23. How to deals with bulk of data using API

24. Cashier Management and Security using API

25. Current Project Details

26. How to secure your rest api

27. Core Java --random number from 1 to 100. One is missing how to find that number with best possible algorithm.

28. What is Containerization

29. How to secure API

30. Versioning of the API

31. Existing API fetching details with id and now same API should fetch details based on mailId without changing existing API.

32. How to make cloud architecture with security.

33. Current Project Details   
34. How to secure your rest api ?  
34. Core Java -  
random number from 1 to 100. One is missing how to find that number with best possible algorithm.  
35. What is Containerization ?  
36. How to secure API ?  
37. Versioning of the API ?  
38. Existing API fetching details with id and now same API should fetch details based on mailId without changing existing API?  
39. How to make cloud architecture with security?