**High Availability**

High Availability in java: it refers to designing and implementing system Applications

It typically involves redundancy and fault tolerance and ability

Recover quickly from failures.

Some of strategies for implementing high availability in java:

* Load Balancing
* Data base Replication
* Clustering
* Catching
* Failover Mechanisms
* Transaction management
* Cloud services
* Health Monitoring
* Distributed Systems
* Recovery and Backup
* Logging and Monitoring

It involves a combination of Software and Hardware and operational practices.

**Auto Scaling**

* Auto Scaling in java it allows you to Automatically and adjust the number of computer resources.
* Implementing auto scaling in Java typically involves leveraging cloud services that provide auto scaling capabilities.

How to achieve auto scaling in java applications:

**Choose a Cloud Provider:**

Select a cloud provider that supports auto scaling. Common options include Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), and others.

Auto scaling is a powerful tool, but it requires careful configuration and testing to ensure optimal performance and reliability for your Java applications.

**Resiliency**Top of Form

* Resilience in software refers to a system's ability to recover gracefully from failures or disruptions, ensuring continued functionality and minimal impact on users.
* Achieving resilience in Java applications involves implementing practices and patterns that enhance fault tolerance and recovery.

Several strategies for building resiliency in java applications:

Circuit breaker pattern

Retry Mechanism

Timeouts

Fallback Mechanisms

Asynchronous programming

Health checks

Monitoring and Logging

Fault Injection Testing

**Observability**

* In java Applications observability is crucial for monitoring, debugging, and optimizing performance.
* It refers to the ability to understand and gain insights into the internal workings of a system by collecting and analysing relevant data.

Tools For enhancing the Observability in java applications:

**Logging:**

1. Use logging frameworks like SLF4J and Log back or loggers provided by the Java standard library.
2. Log statements should include relevant context, timestamps, and severity levels.

**Metrics:**

Integrate metrics to quantify and measure various aspects of your application's performance. Common metrics include CPU usage, memory consumption, response times, and throughput. Libraries like Micrometer, Dropwizard Metrics, or Prometheus client libraries can help with metric collection.

**Tracing:**

Use distributed tracing to monitor the flow of requests across different components and services. Tools like OpenTelemetry, Zipkin, or Jaeger can be used to trace and visualize the path of requests as they traverse through your application.

**Health Checks:**

Implement health checks to assess the overall health of your application and its dependencies. These checks can be monitored to quickly identify issues and trigger automated recovery mechanisms. Spring Boot Actuator, for example, provides built-in health check endpoints.

**Auditing:**

Implement auditing to track and log important events or changes within your application. This is especially useful for security and compliance purposes. Audit logs should include details such as who performed the action, when, and what changes were made.

**Alerting:**

Set up alerting mechanisms to receive notifications when predefined thresholds are exceeded or when specific events occur. This allows you to proactively address issues before they impact users. Tools like Prometheus Alert manager or Grafana can assist with alerting.

**Centralized Logging and Monitoring:**

Aggregate logs and metrics in a centralized location for easier analysis. Tools like ELK stack (Elasticsearch, Logstash, Kibana), Grafana, or Prometheus can help you centralize and visualize data for monitoring and troubleshooting.

**Profiling:**

Use profiling tools to analyse the performance of your Java application. Profilers like Visual VM, Your Kit, or Java Flight Recorder can help identify bottlenecks, memory leaks, and other performance issues.

**Error Tracking:**

Implement error tracking to capture and analyse exceptions and errors that occur in your application. Tools like Sentry, Rollbar, or New Relic can provide insights into error occurrences, frequency, and associated stack traces.

**Correlation IDs:**

Include unique correlation IDs in logs and traces to correlate events across different components. This helps trace the flow of requests and identify the root cause of issues more easily, especially in distributed systems.

**Real-time Monitoring:**

Consider real-time monitoring solutions that provide immediate insights into the performance and behaviour of your application. Dashboards and real-time visualizations can be valuable for quickly identifying and addressing issues.

**Continuous Improvement:**

Regularly review and update your observability practices based on evolving requirements and the changing nature of your application. Continuously seek ways to enhance the visibility into your system.

**Logging**

 Logging is an API (Application programming interface ) that provides the ability to trace out the errors of the applications. When an application generates the logging call, the Logger records the event in the Log Record. After that, it sends to the corresponding handlers or appenders

There are the five logging handlers in Java:

Stream Handler: It writes the formatted log message to an Output Stream.

Console Handler: It writes all the formatted log messages to the console.

File Handler: It writes the log message either to a single file or a group of rotating log files in the XML format.

**RTO in java**

Recovery Time Objective, is a metric that defines the maximum acceptable downtime for a system or application after a failure or disruption. It represents the targeted duration within which a system should be restored to normal operation following a failure.

Achieving a specific RTO involves planning and implementing strategies for disaster recovery and system resilience.

**RPO in java**

Recovery Point Objective, is a critical metric in disaster recovery and business continuity planning.

It represents the maximum allowable data loss that an organization is willing to tolerate in the event of a system failure or disaster.

In the context of Java applications, achieving a specific RPO involves implementing strategies to ensure that data is regularly backed up and that recovery points are maintained within acceptable limits.

**OSI Models**

* OSI stands for Open System Interconnection is a reference model that describes how information from a [software](https://www.javatpoint.com/software) application in one [computer](https://www.javatpoint.com/what-is-computer) moves through a physical medium to the software application in another computer.
* OSI consists of seven layers, and each layer performs a particular network function.
* OSI model was developed by the International Organization for Standardization (ISO) in 1984, and it is now considered as an architectural model for the inter-computer communications.
* OSI model divides the whole task into seven smaller and manageable tasks. Each layer is assigned a particular task.
* Each layer is self-contained, so that task assigned to each layer can be performed independently.

Layers of OSI model

1. Physical Layer
2. Direct Link Layer
3. Network Layer
4. Transport Layer
5. Session Layer
6. Presentation Layer
7. Application Layer

.