Donnerstag, 23. Februar 2023 19:32

What is Distributed System? And example

A distributed system is a collection of independent computers that work together as a single system to provide a common service. The computers in a distributed system are connected to each other through a network, and they communicate and coordinate their actions by passing messages to one another.

Distributed systems are designed to provide certain benefits, such as increased reliability, scalability, and fault tolerance. By distributing the workload across multiple computers, a distributed system can handle more users and more data than a single computer could. Additionally, if one computer in the system fails, the other computers can continue to provide service, reducing downtime and improving reliability.

Examples of distributed systems include web applications that run on multiple servers, distributed databases that span multiple machines, and cloud computing platforms that provide on-demand access to computing resources from anywhere in the world.

Example :- CDN (By distributing the content across multiple servers, a CDN can deliver it to users more quickly than a single server could.)
P2p network
Telecomunication

Use of distibuted system ?

1. Web Applications: Many web applications, such as social media platforms, e-commerce websites, and search engines, are built on top of distributed systems. By distributing the workload across multiple servers, these applications can handle large volumes of traffic and provide fast response times to users.
2. Cloud Computing: Cloud computing platforms, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform, use distributed systems to provide on-demand access to computing resources, such as servers, storage, and databases.
3. Big Data: Distributed systems are also used in big data applications, such as Hadoop and Spark, to process and analyze large volumes of data. By distributing the data and processing across multiple servers, these systems can handle large data sets that would be too big for a single server to handle.
4. Internet of Things (IoT): Distributed systems are also used in IoT applications, where multiple devices, such as sensors and actuators, are connected to a network. By distributing the processing and storage of data across multiple devices, IoT applications can handle real-time data processing and analysis.
5. Scientific Computing: Distributed systems are also used in scientific computing applications, such as climate modeling and simulations. By distributing the workload across multiple servers, these applications can handle complex simulations and calculations that would be too computationally intensive for a single server to handle.

What is middle where ?

It can be described as “software glue” that enables communication and data management for distributed applications12.

Some examples of middleware are:

- Database middleware that allows applications to access data from different sources
- Message-oriented middleware that enables asynchronous communication between applications
- Web middleware that supports web development and hosting
- Platform middleware that provides a runtime environment for application logic3

The default middleware server for Spring Boot applications is Apache Tomcat, but it can also be Jetty, Undertow, or Liberty1.

Spring and spring boot ?

Spring and Spring Boot are both Java frameworks that provide various features for developing applications. The main difference between them is that Spring Boot is built on top of Spring and provides additional features to simplify and speed up development[13].

Spring Boot is basically an extension of the Spring framework, which eliminates the boilerplate configurations required for setting up a Spring application.

It takes an opinionated view of the Spring platform, which paves the way for a faster and more efficient development ecosystem.

Some of the key differences are:

- Spring Boot has an embedded middleware server, while Spring requires a separate server[1]

- Spring Boot has auto-configuration and starter dependencies, while Spring requires manual configuration and dependency management[134]

- Spring Boot supports microservice-based architecture, while Spring supports any kind of application[24]

DI and IOC

Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. We most often use it in the context of object-oriented programming.

oC enables a framework to take control of the flow of a program and make calls to our custom code. To enable this, frameworks use abstractions with additional behavior built in. **If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.**

The advantages of this architecture are:

- decoupling the execution of a task from its implementation
- making it easier to switch between different implementations
- greater modularity of a program
- greater ease in testing a program by isolating a component or mocking its dependencies, and allowing components to communicate through contracts

We can achieve Inversion of Control through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

DI

Dependency injection is a pattern we can use to implement IoC, where the control being inverted is setting an object's dependencies.
Constructor-Based Dependency Injection
Setter-Based Dependency Injection
Field-Based Dependency Injection(Autowired)

The Spring IoC Container

An IoC container is a common characteristic of frameworks that implement IoC.

In the Spring framework, the interface *ApplicationContext* represents the IoC container. The Spring container is responsible for instantiating, configuring and assembling objects known as *beans*, as well as managing their life cycles.

Autowired

@Autowired is an annotation that enables dependency injection for Java classes in Spring Framework1. It allows Spring to automatically inject dependencies into fields, methods, or constructors by matching the data type2. This annotation can simplify your code by eliminating manual configuration.

App engin
Google App Engine (often referred to as GAE or simply App Engine) is a cloud computing platform as a service for developing and hosting web applications in Google-managed data centers. Applications are sandboxed and run across multiple servers.[2] App Engine offers automatic scaling for web applications—as the number of requests increases for an application, App Engine automatically allocates more resources for the web application to handle the additional demand.[3]

What is basic scaling in App Engine?

Basic scaling **creates instances when your application receives requests**. Each instance will be shut down when the application becomes idle. Basic scaling is ideal for work that is intermittent or driven by user activity.

Components in spring boot

- Spring Boot Starters
- Spring Boot AutoConfigurator
- Spring Boot CLI
- Spring Boot Actuator
- Spring Initilizr

How authentication managed in spring boot ?

Session based or jwt auth . Other third party OAuth

What is JPA and explain

JPA stands for Java Persistence API (Application Programming Interface). It is a Java specification that gives some functionality and standard to ORM tools. It is used to examine, control, and persist data between Java objects and relational databases.

Hibernate is a java framework and ORM (Object Relation Mapping) tool that is used to provide the implementation of the JPA methods. How does JPA Work? JPA is an abstraction that is used to map the java object with the database.

Jpa relations

Java Persistence API (JPA) is a specification for object-relational mapping (ORM) in Java. JPA defines a set of annotations that can be used to map Java objects to database tables, and provides a standardized way to work with relational databases in Java.

JPA provides support for several types of relationships between entities in a relational database. The most common types of relationships are:

One-to-One (1:1) Relationship: In a one-to-one relationship, each entity in one table is associated with only one entity in another table, and vice versa. For example, a student and a student's address could have a one-to-one relationship, where each student has only one address and each address belongs to only one student.

One-to-Many (1:N) Relationship: In a one-to-many relationship, each entity in one table is associated with many entities in another table, but each entity in the other table is associated with only one entity in the first table. For example, a department and its employees could have

a one-to-many relationship, where each department has many employees, but each employee belongs to only one department.

Many-to-One (N:1) Relationship: In a many-to-one relationship, many entities in one table are associated with one entity in another table. For example, many employees could belong to one department.

Many-to-Many (N:N) Relationship: In a many-to-many relationship, each entity in one table can be associated with many entities in another table, and vice versa. For example, a student could be enrolled in many courses, and each course could have many students.

JPA provides annotations, such as @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany, to define these relationships between entities in Java classes. These annotations specify the mapping between the Java object model and the relational database tables, and allow JPA to generate the necessary SQL queries to manage the relationships between entities in the database.

Dataflow in spring boot

In Spring Boot, data typically flows through a series of layers, each responsible for a different aspect of the application's functionality. The most common layers in a Spring Boot application are the controller layer, service layer, and repository layer.

Controller Layer: The controller layer is responsible for handling incoming HTTP requests and sending back HTTP responses. Controllers receive input from the user, validate it, and pass it to the service layer for further processing. Controllers are typically implemented as Spring MVC controllers, which are annotated with @Controller or @RestController.

Service Layer: The service layer is responsible for business logic and processing of data. Services receive data from the controller layer, perform any necessary processing or manipulation, and pass the data to the repository layer for persistence. Services are typically implemented as Spring services, which are annotated with @Service.

Repository Layer: The repository layer is responsible for persistence of data. Repositories receive data from the service layer, interact with the database or other data storage systems, and return the data back to the service layer. Repositories are typically implemented as Spring Data JPA repositories, which are interfaces that define a set of methods for data access.

The data typically flows from the controller layer to the service layer, and then to the repository layer for persistence. When a user sends an HTTP request to the application, the request is first handled by the controller layer, which receives the input from the user and validates it. The controller then passes the data to the service layer for further processing.

The service layer performs any necessary business logic or data manipulation, and then passes

the data to the repository layer for persistence. The repository layer interacts with the database or other data storage systems to store the data, and then returns the data back to the service layer. The service layer then prepares the data to be sent back to the user, and the controller layer sends an HTTP response back to the user.

App engin language support

Google App Engine primarily supports Go, PHP, Java, Python, Node. js, . NET, and Ruby applications,although it can also support other languages via "custom runtimes"

Spring MVC *@Controller*

We can annotate classic controllers with the *@Controller* annotation. This is simply a specialization of the *@Component* class, which allows us to auto-detect implementation classes through the classpath scanning.

We typically use *@Controller* in combination with a *@RequestMapping* annotation for request handling methods.

Spring MVC *@RestController*

@RestController is a specialized version of the controller. It includes the *@Controller* and *@ResponseBody* annotations, and as a result, simplifies the controller implementation:

what is post mapping ?

@PostMapping is a Spring MVC annotation that maps HTTP POST requests to a specific controller method in a Spring Boot application.

When a client sends an HTTP POST request to the server, it typically includes data in the request body that the server needs to process. The @PostMapping annotation is used to map the URL of the request to a specific controller method that will handle the processing of the request body.

Life cycle

In the context of specific technologies, such as Spring Boot, lifecycle management may refer to the management of specific aspects of an application's lifecycle, such as the lifecycle of beans or components within the application. Spring Boot provides a number of tools and features that can help manage the lifecycle of components within an application, such as dependency injection and bean lifecycle callbacks.

For example, in Spring Boot, the lifecycle of a bean can be managed using annotations such as

@PostConstruct and @PreDestroy. The @PostConstruct annotation is used to annotate a method that should be called after a bean has been constructed, but before it is returned to the application context. The @PreDestroy annotation is used to annotate a method that should be called before a bean is destroyed or removed from the application context.

Overall, lifecycle management is an important concept in software development, and it plays a critical role in ensuring that software applications are developed and maintained in a consistent and effective manner.

bean lifecycle in Spring Boot:

Bean instantiation: When a bean is first created, it is instantiated by the Spring container. The container creates a new instance of the bean class using the default constructor or a factory method.
Dependency injection: After a bean has been instantiated, the container injects any dependencies that the bean requires. This is typically done using setter injection or constructor injection.
Bean initialization: Once all dependencies have been injected, the container initializes the bean. This includes calling any initialization methods specified by the bean, such as methods annotated with @PostConstruct.
Use: After initialization, the bean is available for use by other components within the Spring application context.
Destruction: When the Spring application context is shut down, any beans that were created by the context are destroyed. This includes calling any destruction methods specified by the bean, such as methods annotated with @PreDestroy.

Spring Boot provides several mechanisms for managing the lifecycle of beans, including annotations such as @PostConstruct and @PreDestroy, as well as the InitializingBean and DisposableBean interfaces. Developers can also create custom lifecycle callbacks using BeanPostProcessor and BeanFactoryPostProcessor.

lazy and eager loading

In the context of database queries and object-relational mapping (ORM) frameworks like Hibernate used in Spring Boot, lazy loading and eager loading are two strategies for fetching data from the database.

Eager loading refers to the strategy of fetching all the required data for a particular object or entity and its related objects or entities at the time the query is executed. In other words, all the associated data is loaded into memory along with the main object. This can lead to performance issues if the object and its associations contain a large amount of data.

On the other hand, lazy loading is a strategy where only the required data is loaded into memory at the time of query execution, and related data is loaded on-demand as it is accessed by the application. This can help to improve performance by reducing the amount of

unnecessary data that is loaded into memory.

In Spring Boot, lazy loading can be implemented using annotations like @OneToMany and @ManyToMany, which tell Hibernate to fetch related entities lazily. For example, if a parent entity has a @OneToMany relationship with a child entity, we can annotate the relationship with @OneToMany(fetch = FetchType.LAZY) to instruct Hibernate to fetch the child entities lazily.

Similarly, eager loading can be implemented using annotations like @ManyToOne, @OneToOne, and @ManyToMany, which tell Hibernate to fetch related entities eagerly. For example, if a parent entity has a @ManyToOne relationship with a child entity, we can annotate the relationship with @ManyToOne(fetch = FetchType.EAGER) to instruct Hibernate to fetch the child entities eagerly.

Overall, the choice between lazy loading and eager loading depends on the specific use case and performance requirements of the application. Lazy loading is generally preferred for large and complex data models, while eager loading may be more appropriate for smaller data models or cases where the associated data is frequently accessed.