# DSAM

## Distributed System and Middleware

A distributed system is a collection of computer programs that utilize computational resources across multiple, separate computation nodes to achieve a common, shared goal. A distributed system is a collection of independent computers that appears to its users as a single coherent system

**Challenge when making software for Distributed systems**
Failures, Security, Consistency, Network Latency, Resource Allocation and Balancing, Concurrency and Synchronization, Compatibility.

**Middleware**
**Middleware** resides between the application and the network, serving as a bridge.

Middleware is the software between
application and operating system (local view)
service user and service provider (global view)

Vertical and Horizontal Clustering (Across nodes)

Asynchronous Transactions (**Loosely Coupled)**
Synchronous Transactions (**Highly Coupled)**

**Explicit Use**: Directly telling clients which middleware to use.
**Transparent Use**: Middleware operates silently, without clients needing to know about it.

**Software architecture design** maps the problem domain to the solution domain considering forces and risks

**Kruchten's 4+1 Views**
Logical View, Process View, Physical View, Development View, Use Case View
Diagrams using UML consisting of Components and Connectors

(BFT) is a concept in distributed systems.
Problem Statement: Imagine a network of computers (nodes) that need to agree on a common decision, such as validating a transaction

or reaching consensus.

———————

## Spring Boot

Spring Boot streamlines Spring Framework development.
@SpringBootApplication

**Annotations** in Spring Boot are a form of metadata that provides additional information about a program. Annotations play a crucial role in configuring Spring Boot applications, defining beans, and managing components.
**@Autowired** indicates **automatic dependency injection**.
**@Configuration** simplifies bean configuration in Spring Boot by allowing you to define beans directly in Java code.
**@Controller**: mark class for component scanning, generating a bean.
**@Entity** signifies that it represents a **table** in the database (JPA)

**Bean** in an object, managed by the IoC container, simplify dependency management and enhance the flexibility of your application. Without IoC, we manually create objects using constructors. With Spring's IoC container, we define beans and their relationships in configuration metadata (XML, annotations, or Java code). The container handles object creation and wiring.

**IoC container** manages the lifecycle of objects (beans) within your application. Instead of objects creating their own dependencies, they delegate the task of constructing dependencies to the IoC container. The container handles object instantiation, configuration, and wiring.

- BookService depends on a BookRepository (data access layer). We inject the repository via constructor (@Autowired).

```
 @Autowired
    public BookService(BookRepository 
bookRepository) {
        this.bookRepository = bookRepository;
    }
```

- BookController depends on BookService. Again, we inject it via constructor.
- Spring Boot's IoC container manages the lifecycle of these

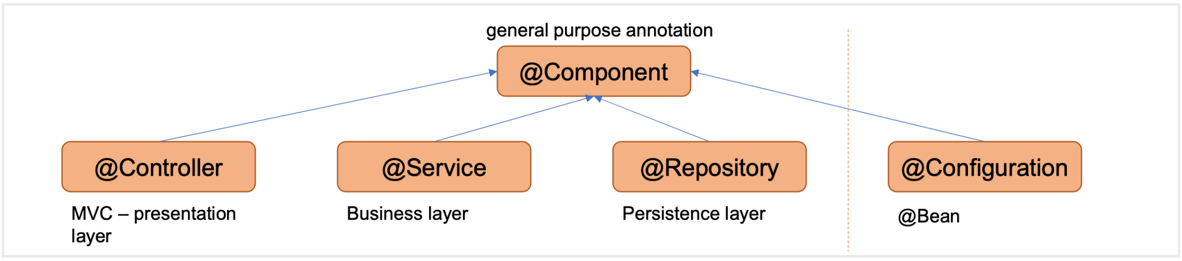
beans.
- When an HTTP request hits /books, the BookController calls the BookService to fetch book data.

@Autowired indicates **automatic dependency injection**.

**@Configuration** simplifies bean configuration in Spring Boot by allowing you to define beans directly in Java code.

**Spring Boot profiles and properties**
**Profiles** allow you to configure your application differently based on the environment. By activating a profile, you choose which beans and properties are active in your application.
Properties are key-value pairs used for configuration.
```
@Component
@Profile("dev") // Only active during development
public class DevDataSourceConfig {
    // Dev database configuration
}
```
**application.properties** is the primary place to configure properties

**Classpath** scanning automatically detects classes (components) in your application without you explicitly specifying them. (@ComponentScan)



**Lombok** plugin. Annotate your model classes with @Data. Getters, setters at compile time.

**Spring MVC** facilitates handling HTTP requests and responses in web applications.
- **Model**: Represents the data and business logic.
- **View**: Renders the data (usually as HTML pages).
- **Controller**: Manages the flow between the model and view.

- You define **controllers** (annotated with @Controller) that

handle specific URLs.
- When a request arrives, Spring MVC routes it to the appropriate controller method.
- The controller processes the request, interacts with the model, and prepares data for the view (templating engine).
- Finally, the view renders the response (usually an HTML page).

**@Controller**: mark class for component scanning, generating a bean.

**Validation**
- Use Java Bean Validation API
- OR Declare validation at your model classes (Movie)
- Specify validation in your controller and change method signature of processMovie
- Adapt your view to display errors (movies.html)

@NotNull(message = "Title must be set")

**JPA (Java Persistence API)**
JPA is a component in Spring Boot for data persistence and managing relational data.

Working:
JPA helps you connect Java objects to database tables (Mapping). Each entity corresponds to a table, and each object becomes a row in that table. It converts Java fields (like name, age, etc.) into columns in the table. When you save an object, JPA inserts a row in the table. EntityManager helps to Save, Update, and Retrieve. Query with JPQL (Java Persistent Query Language).

- Hibernate is used as the default ORM (Object <--> Relational mapping) implementation
- @Entity (JPA Annotation marking the class as a JPA Entity)
- **Object-Relational Mapping (ORM)**: JPA handles the mapping between your Java objects and the database tables.

```
@Data
@NoArgsConstructor
@Entity
public class Movie {
}
```

Using **@NoArgsConstructor** for JPA to work correctly. It instructs Lombok to generate a **parameterless constructor** (a constructor with no arguments) for the annotated class.

```
@OneToMany(mappedBy = "bottle")
private List<Crate> crates;
```

**@JoinTable** defines the join table, and @JoinColumn specifies the linking column. Together, they orchestrate the dance of many-to-many relationships in Spring Boot.

Create repository interfaces for each entity by extending **CrudRepository**. **CrudRepository** is part of the **Spring Data JPA framework**. It provides convenient methods for performing **CRUD (Create, Read, Update, Delete)** operations on entities (i.e., data objects) in a **relational database**.

**EAGER** (default for @ManyToOne and @OneToOne) and **LAZY** (default for @OneToMany, @ManyToMany) are static properties and you cannot switch between them at runtime.

**EntityGraph** allows you to **customise the fetching strategy** for related entities. EntityGraphs allow you to fine-tune how related entities are fetched, optimising performance and memory usage. Entity Graphs allow you to specify which related entities should be eagerly loaded when querying the database. They help optimize performance by avoiding unnecessary lazy loading and N+1 query problems.

**Named EntityGraph**: Defined globally and can be reused across multiple queries.
**Dynamic EntityGraph**: Created programmatically at runtime based on specific criteria.
In your repository interface, specify the EntityGraph to be used.

```
public interface AuthorRepository extends
JpaRepository<Author, Long> {
    @EntityGraph(attributePaths = "books")
    Author findById(Long id);
}
```

**Transactions** allow related operations to be executed as a single unit. They ensure that either all database operations within a transaction are **committed** (persisted) or **rolled back** (undone) together.
When applied to a method or class, it wraps the method execution in a transaction. This means that either all database operations within that method are **committed** together, or they are **rolled back** together if an exception occurs.

**JDBC** (Java Database Connectivity) is an API that allows Java applications to interact with relational databases (Low-level, using SQL queries)

**ACID** is a set of properties that ensure reliable and consistent database transactions
ACID (Atomicity, Consistency, Isolation, Durability)

**Spring Security**
Adding spring security starter to the project implementation 'org.springframework.boot:spring-boot-starter-security'
Use Authentication and Authorization
Use HTTPS (Obtain an SSL certificate)
Conduct security testing, including penetration testing and vulnerability scanning.

REST **(Representational State Transfer)** isn't just a standard; it's an approach that helps you build web-scale systems by leveraging the power of HTTP and adhering to specific constraints.
RESTful services use standard HTTP methods:

- **GET**: Retrieve data (read).
- **POST**: Create new resources.
- **PUT**: Update existing resources.
- **DELETE**: Remove resources.

Statelessness:

- Each request from the client to the server must contain all necessary information.
- Servers don't store client state between requests.

**HATEOAS** is about enriching your API responses with hypermedia links. Instead of just returning data, your API provides links to related resources. These links guide clients on what actions they can take next.

**Testing**
**Unit Testing**
Test a single class in isolation or a method of this class
**Integration Testing**
Test the interaction of various components

**Stub, Mock, and Spy**
Interaction-based testing, where we focus on how objects interact with each other. By using Spock or Mockito.

- **Stubs** provide **dummy behaviour** for specific methods.
- **Mocks** are placeholders for real objects or dependencies.
- **Spies** allow you to **spy on real objects**.

Spring Boot offers various ways to **handle exceptions**, from controller-specific methods to global handlers.
@ExceptionHandler and @HandlerExceptionResolver

——————

## Cloud

Goal: Secure, scalable, flexible, Cost-Efficient way to store and manage your data.

**Cloud Provider Tasks**
Service Management
Software Lifecycle Management
Infrastructure Management

**Cloud Tenancy Models**
Public, Private, Hybrid

Cloud computing relies on a technology called virtualisation.

**IaaS (Infrastructure as a Service):**
IaaS delivers on-demand infrastructure resources via the cloud, such as compute power, storage, networking, and virtualization.
**Responsibilities**:
You don’t manage your own data center infrastructure.

You're responsible for the operating system, middleware, virtual machines, and any apps or data.
**Example**: Imagine renting a car—you get the vehicle (compute, storage, etc.), but you're responsible for driving and maintaining it.
e.g. Virtual Machines (AWS EC2)

**PaaS (Platform as a Service):**
PaaS provides a complete, ready-to-use platform for developing, running, and managing applications.
**Responsibilities**:
You focus on building applications without worrying about underlying infrastructure.
The cloud provider handles hardware, OS, and middleware.
**Example**: Think of taking a cab—you get to your destination (application development) without dealing with driving or car maintenance.
e.g. AWS Elastic Beanstalk, Google App Engine

**SaaS (Software as a Service):**
SaaS gives you ready-to-use, cloud-hosted application software.
**Responsibilities**:
You don't manage servers, OS, or application infrastructure.
Simply use the software via an internet connection.
**Example**: Boarding a bus or subway—you hop on and reach your destination (using software) without worrying about driving or routes.
e.g. Microsoft 365

**Function as a Service (FaaS)**
Developers focus on writing code (specific functions) without managing servers or infrastructure.
FaaS is an event-driven computing model.
E.g. AWS Lambda (Image Processing), Google Cloud Functions. Microsoft Azure Functions

- PaaS requires some capacity planning and configuration.
- FaaS doesn't require capacity planning; it scales readily.

**Challenges:**
Provider Limits, Cold Start, Network Latency

Steps: Make the Function, Deploying API, Setting up DynamoDb, Infrastructure automation with CloudFormation
```
AWS Lambda Function Template
public class AllItemsHandler
```

```
                implements RequestHandler<Object,
String>
```

**Traditional 3-tier Architecture**: Presentation, Application, and Data Layer

**Container as a Service (CaaS)**
Containers are lightweight, standalone software units that package applications along with their dependencies.

**Hypervisors** create an abstraction layer that allows multiple virtual machines (VMs) to run on a single physical host. Each VM runs its own complete operating system, including the kernel. Hypervisors allocate dedicated resources (CPU, memory, storage) to each VM.

**Docker** image is a template; a container is a running instance of that template.
E.g. AWS ECS (Elastic Container Service)
Closer to IaaS

**Kubernetes** (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications.
Multiple physical or virtual machines Nodes are joined to a Kubernetes Cluster which is controlled by a Master.

- A pod contains containers
- A kubelet controls pods on a node (VM)
- Multiple physical or virtual machines Nodes are joined to a Kubernetes Cluster
- A master controls a cluster

e.g. AWS EKS (Elastic Kubernetes Service)
Closer to IaaS but is a CaaS

**Backend as a Service (BaaS)**
Focus on building the frontend of their applications while outsourcing the complexities of the backend.
Features usually include
• Authentication
• Analytics
• Push notifications
• Storage
E.g. AWS Amplify, Google Firebase

**Google App Engine**
GAE is made for running web applications! Automatic scaling, Resource Management, Global Reach, Serverless Benefits and loadbalancing.
With GAE’s Java runtime, you can deploy **executable JAR files**.

- **App Engine Mail** allows you to send emails from your GAE applications.
- **App Engine Memcache** provides an in-memory caching service. It improves application performance by reducing database and API calls.

**Firestore** is a **NoSQL document database** designed for scalability, high performance, and ease of application development.
**Documents**

- In Firestore, a **document** is the fundamental unit of data storage.
- It represents a single piece of information or an entity within a **collection**.
- Unique ID, Fields, and No Schema
- Define rules to control who can read, write, or modify documents.

**Document Limits**
1 MB in size
40 thousand indexed fields
1 write per second sustained write rate

```
Collection: Users
  Document: user123
    Fields:
      - name: "Alice"
      - email: "alice@example.com"
      - age: 30
```

## Firestore Native – CRUD I

❑ Getting hold of a firestore reference

```
FirestoreOptions firestoreOptions =
    FirestoreOptions.getDefaultInstance().toBuilder()
        .setProjectId(projectId)
        .setCredentials(GoogleCredentials.getApplicationDefault())
        .build();
Firestore db = firestoreOptions.getService();
```

❑ Create

```
DocumentReference docRef = db.collection("blogentries").document("firestore");
DocumentReference docRef =
db.collection("blogentries").document("firestore").collection("comments")…
```

→ implicitly creates collection / document reference if not existent

```
Map<String, Object> data = new HashMap<>();
data.put("title", "Firestore");
data.put("meta", "concept, data modeling");
ApiFuture<WriteResult> result = docRef.set(data);
```

→ replaces existing data

→ yields an ApiFuture; there is some work to be done behind the scenes

→ adding a POJO is possible as well

## Firestore Native – CRUD II

FS

❑ Update

```
docRef.update("title", "Firestore Native");
```

OR

```
Map<String, Object> data = new HashMap<>();
data.put("title", "Firestore");
docRef.update(data)
```

❑ Read

```
ApiFuture<QuerySnapshot> query = db.collection("blogentries").get();
QuerySnapshot querySnapshot = query.get(); // essentially two gets
List<QueryDocumentSnapshot> docs = querySnapshot.getDocuments(); // then iterate
System.out.println(document.getData().get("title"));
//Similarly
db.collection("blogentries").document("firestore").listCollections();
```

❑ Delete

```
ApiFuture<WriteResult> res = db.collection("blogentries").document("firestore").delete();
```

→ note: sub-collections need to be deleted manually!

OR

```
Map<String, Object> upd = new HashMap<>();
upd.put("meta", FieldValue.delete());
ApiFuture<WriteResult> res = docRef.update(upd);
```

## Firestore Native – Query Example

- Simple Query

```
CollectionReference entries = db.collection("blogentries");
Query query = entries.whereEqualTo("title", "firestore");
// get query results
ApiFuture<QuerySnapshot> querySnapshot = query.get();
// then iterate
for (DocumentSnapshot document : querySnapshot.get().getDocuments()) { .. }
```

- Compound Query

```
Query compQ = entries.whereEqualTo("author", "andreas").whereGreaterThan("rating", 5);
```

→ Note: One range filter max. per compound query

- Collection Group Query

```
Query jComs = db.collectionGroup("comments").whereEqualTo("author", "johannes");
ApiFuture<QuerySnapshot> querySnapshot = jComs.get();
for (DocumentSnapshot document : querySnapshot.get().getDocuments()) {…}
```

## Firestore Native - Transactions

- Firestore offers two „sorts“ of transactions
  - read and write operations on one or more documents
    → called *transactions* in Google documentation

```
ApiFuture<Void> ftTx = db.runTransaction(transaction -> {
  Map map = transaction.get(docRef).getData();
  rating = map.getDouble("rating");
  transaction.update(docRef, "rating", rating * 1.1);
  return null;
});
```

  - set of write operations on one or more documents
    → called *batched write* in Google documentation

```
WriteBatch batch = db.batch();
batch.set(..); batch.update(..); batch.delete(..)
ApiFuture<List<WriteResult>> future = batch.commit();
```

- Liabilities
  - Read operations must come before write operations
  - Maximum of 500 affected documents