# **Library Management System**

using the WRSPM model. We'll go through each step to illustrate the software development process.

1. W (World)
   1. Real-World Definition: The Library Management System is designed for library staff to manage books and members. Staff can add or remove books, keep track of books borrowed by members, and monitor due dates and stock levels.
   2. -Context: The software will be a web-based application accessible from the library's computer. Librarians will log in to manage books and members, while members can log in to check the books they have borrowed and their return dates.
2. R (Requirements)
   1. User Requirements:
      1. Staff should be able to log in to the system.
      2. Staff should be able to add, update, and remove books.
      3. Staff should be able to add, update, and remove members.
      4. Staff should be able to manage borrowed books and track due dates.
      5. Members should be able to view the books they have borrowed and the corresponding return dates.
   2. Functional Requirements:
      1. Book addition, updating, and deletion functionality.
      2. Member addition, updating, and deletion functionality.
      3. Borrowing and return management functionality.
      4. User login and authorization system.
3. S (Specifications)
   1. Login Screen:
      1. Users can log in using their username and password.
   2. Book Management:
      1. A form to add a new book: book title, author, publisher, ISBN, and stock status.
      2. Options to update and delete books.
   3. Member Management:
      1. A form to add a new member: first name, last name, phone number, and email.
      2. Options to update and delete members.
      3. Borrowing and Returning:
      4. Staff can select a member and book to process a loan.
      5. The system should list all borrowed books with return dates.
4. P (Program)
   1. At this stage, let’s choose the technologies and begin coding the core components of the software.
   2. Technologies:
      1. Backend: C# (.NET Core) or Python (Django)
      2. Database: MySQL, SQLite or MsSQL
      3. Frontend: HTML, CSS, JavaScript (for a simple user interface)

I'm not sure I got this part right.

Hello everyone,  
Today, I’ll be presenting a project I’ve developed—a **Library Management System**. This system is designed to help both librarians and library members manage and track books more easily.

Librarians can log in to add, update, or remove books from the system, manage member information, and keep track of books that are borrowed or returned. On the other side, members can log in to see which books they’ve borrowed and when they need to return them.

### **W - World**

“The first step in the WRSPM model is **World**. In this step, we define the real-world environment where the software will be used.

For the **Library Management System**, the main users are **librarians** and **library members**. Librarians need to manage books and members, while members need to track the books they’ve borrowed. The system will be web-based and accessible from the library’s computer. This sets the stage for the rest of the development process by outlining the overall environment and key users.”

### **Slide 4: R - Requirements**

“Next, we move on to **Requirements**. (Requayrtmıns)This is where we list what the system needs to do. There are two types of requirements: **user requirements** and **functional requirements**.

**User requirements** tell us what the users need:

* Librarians must be able to log in, manage books and members, and track loan information.
* Members need to log in and see what books they have borrowed and when they are due.

**Functional requirements** are more technical:

* The system must allow librarians to perform CRUD operations, which stands for Create, Read, Update, and Delete, for both books and members.
* The system should also manage the borrowing and returning of books, as well as user authentication for security.”

### **Slide 5: S - Specifications**

“Once we’ve defined the requirements, we move to **Specifications**, where we outline how the system will work on a more detailed level.

Here’s how the Library Management System will look:

* We will have a **login page** where both librarians and members can sign in securely.
* A **book management interface** will allow librarians to add, update, or remove books.
* A **member management interface** will handle member details.
* The **borrowing and return interface** will allow librarians to track books loaned out and their return dates.
* Lastly, members will have a **dashboard** to see their borrowed books and upcoming return dates.

These specifications serve as a blueprint for what we will actually code.”

### **Slide 8: EH, EV, SV, SH Breakdown**

I'm not sure I got this part right

**Title**: WRSPM Submodels (EH, EV, SV, SH)

* **Environtment Hidden:** These are things about the people using the system that the system doesn't know or see.
  + In our Library Management System, this could include the librarian's or member's thoughts or personal preferences. For example, the system doesn’t know why a member chose a certain book.
* **EVisible**: These are things in the environment that the system can see.
  + For example, the system can see when a librarian logs in, when a member borrows a book, or when a book is overdue. These actions allow the system to work and make decisions.
* **System V :**These are parts of the system that people using it can see.
  + This includes the login page, the book management screen, and the borrowing/return features. Librarians and members interact with these visible parts of the system.
* **SH :**These are parts of the system that are hidden from people using it.
  + This includes the backend operations like database queries, checking passwords, or tracking which books are due. These happen behind the scenes (siins) and users don’t see them, but they keep the system running.

### **Slide 6: P - Program**

“Now we come to the **Program** phase. This is where we translate the specifications into actual code.

For this system, we’ll use the following technologies:

* **Backend**: Either C# with .NET Core or Python with Django.
* **Database**: We can use MySQL or SQLite to store the data.
* **Frontend**: HTML, CSS, and JavaScript will create the user interface.

### **Slide 7: M – Machine**

I also wanted to tell you briefly about the machine

“After coding the system, the final stage is **Machine**. This step is about deploying the software in a real-world environment.

In our case, the Library Management System will be hosted on a web server, such as Apache or Nginx. Users will access the system through their web browsers, making it easy for librarians and members to use from any computer in the library.

That was my presentation, thank you for listening

Actually, compared to the presentation of others, mine is not very detailed, but at least I tried it.

I think I just followed the order at which I was researching about WRSPM.

I wanted to keep the presentation a little short.

“Hello everyone Today, I'll be presenting the architectural design for a Library Management System.

Our goal here is to design a system that efficiently (efeşınliy) manages library tasks such as user login, borrowing and returning books, and notifying(notifaying) users when books are due. I will cover requirements, chosen architecture, design patterns

2. sayfaya geç

The architecture created for the library management system has a layered structure to support the sustainability(susteynability)and scalability (scalability)of the system.

### **Use Case**

Actors in our System

Member user: Those who can search a book, issue a book, return and renewal of borrowed book.

Librarian: responsible for adding, updating, and removing of books.

System: responsible for sending notifications for overdue books, and calculate fine.

Designing a library management system involves multiple components working together to handle book cataloging, member management, borrowing and returning books, inventory tracking, and more. Here's a high-level design that outlines the primary? components and their interactions?:

### User Interface Layer

This layer is responsible for interacting (interakting)with the end users of the system. This layer includes the following:

Web Application: An interface for library staff and users to manage books, members and transactions. It is built using HTML, CSS, JavaScript and a front-end framework such as React, Angular or Vue.js.

Mobile App: A mobile-friendly interface for accessing library services on the go. Built using native or cross-platform frameworks such as Flutter or React Native.

### **Service Layer**

The Service Layer of the library management system is responsible for encapsulating the core business logic and coordinating the interactions between the various system components.

* User Service
* This service handles member-related operations, such as authenticating users, managing member profiles, and processing checkout and return requests.
* Notification Service:
* Providing automated notifications to members, this service ensures timely communication about overdue books, reservation status updates, and other important library events.
* Transaction Service
* These services manage the entire lifecycle of book borrowing, recording checkouts, returns, and renewals?, while also handling any late fees or fines.
* Reporting Service
* Generating comprehensive? reports on library usage?, inventory, and member activity, this service aids in data-driven decision making and operational? optimization.
* Search Service
* Powering the catalog search functionality, these services enable members to quickly locate and access the books they need, enhancing the overall user experience.

### Application LAyer

This layer contains the core business logic of the system and is responsible for processing requests from the UI layer.

Controller Services: Handle incoming requests, process them, and send responses back to the UI layer. Implemented using a framework like Spring Boot (Java), Express.js (Node.js), or Django (Python).

Business Logic Services: Implement the core functionalities such as book management, catalog management, transaction processing, and reporting.

### **Data Layer**

This layer is responsible for data storage(storıc), retrieval, and management.

Database Management System (DBMS): Stores all the data related to books, members, and transactions.

Data Access Layer (DAL): Manages database interactions. Implements CRUD operations and complex queries using Object-Relational Mapping (ORM) frameworks like Hibernate (Java), Sequelize (Node.js), or SQLAlchemy (Python).

### **Scalabiltiy and Performance Layer**

This layer ensures the system can handle growth and high performance.

Load Balancing: Distributes incoming requests across multiple servers to ensure no single server is overwhelmed. Implemented using tools like NGINX, HAProxy, or cloud-based load balancers.

Caching: Improves performance by caching frequently accessed data using tools like Redis or Memcached.

Auto-Scaling: Automatically adjusts(edjust) the number of servers based on the current load using cloud services like AWS Auto Scaling or Google Cloud's auto-scaling feature.

### 1. Monolithic Architecture

**Advantages:**

Monolithic design is simple and easy to set up. Since everything is in a single application, we can start development faster and manage it with fewer resources. For small teams, this is a strong benefit.

* **Simplicity**: Since it’s developed as a single application, setup and development are simpler and faster, especially at the start.
* **Easier Management**: With all components in the same project, resource management and debugging can be easier.
* **Lower Costs**: Monolithic architecture has lower costs than microservices initially, making it better for small teams and limited resources.

**Disadvantages:**

As the system grows, it becomes harder to scale. Performance might drop as more features and users are added. Updating one part of the system can also affect the entire application.

* **Limited Scalability**: As the system grows or becomes more complex, performance might drop. Any updates or fixes can affect the entire system.
* **Less Flexibility**: Since all components are tightly connected, updating or scaling individual modules separately is difficult.

**When to Use**: If your library management system will stay small to medium-sized with basic functions, a monolithic architecture should be enough.

### 2. Microservices Architecture

Microservices allow us to develop each feature as a separate, independent module. For example, we could scale or update the **User Service** without impacting other parts of the system. This is very helpful if certain services, like **Search**, need to handle high traffic.

**Advantages:**

* **Flexibility**: Microservices are developed and deployed as independent modules, so each service (like User, Notification, or Transaction) can be scaled, updated, or restructured separately.
* **Scalability**: Services with high traffic can be scaled independently, focusing resources on high-demand areas (e.g., the Search Service).
* **Team Independence**: Each service can be managed by a separate team, allowing large teams to work more efficiently.

**Disadvantages:**

Microservices require more setup and management, which can be challenging for small teams. Each service needs its own deployment and communication setup, which adds complexity.

* **Increased Complexity**: Microservices require more management, deployment, and communication, which can be challenging for smaller teams.
* **Higher Costs**: Microservices need more infrastructure, which increases hosting and management costs.
* **Communication Load**: Communication between services increases, so solutions for messaging or API management are needed.

**When to Use**: If your system is expected to grow, serve a large user base, or require frequent updates to independent modules, microservices are a better fit.

### Which One for This Project?

For this project, we might start with a monolithic architecture if our goal is a small to medium-sized system. But if we expect a large user base or need frequent (friquent)updates to certain features, we could consider microservices.

Thank you for listening!

Hello, I would like to talk about a data problem faced by a big company like Discord in the software world and the solution they found for it.

**Slide 1: Title - How Does Discord Store Trillions of Messages?**

Hello everyone, thank you for being here today. In this presentation, we will talk about Discord’s journey(jorniy) in message storage(storıc). We’ll look at the challenges Discord faced as it scaled (skeyıld )up to store trillions of messages and how it tackled (tekılt)? these issues to maintain performance and reliability.(rilaybılity)

**Slide 2: Introduction**

Initially(inişıliy), Discord began with MongoDB but moved to Cassandra in search of a scalable, fault-tolerant, and low-maintenance database. As the user base grew rapidly, Cassandra required more maintenance than expected and faced performance issues. Six years later, Discord has made significant changes to its technical infrastructure(infrasıtrakture) and message storage solutions. This presentation will share the challenges and lessons learned during this process.

**Slide 3: Problem**

In 2017, Discord started using Cassandra with 12 nodes to store billions of messages. By 2022, the number of messages grew to trillions, and they needed 177 nodes to handle this load. However, this increased(inkırist) complexity also led to increased costs and maintenance issues, creating new problems.

**Slide 4: Challenges**

Discord faced three main challenges:

1. **Data Model and Distribution**: Channels with few users sent fewer messages, while high-traffic channels caused bottlenecks(bodılnext), known as "hot partitions(partişıns)."
2. **Read and Write Costs**: While writes were relatively cheap in Cassandra, reads were more costly. A large number of simultaneous(saymılteniyıs)reads to a partition caused performance problems on the nodes.
3. **Maintenance and Compaction**: They tried to manage this with a process called "gossip dance," taking nodes offline and back online to compact data, but this added more complexity.

**Slide 6: Hot Partition Solution**

To address these issues, Discord adopted a two-step approach:

1. **Request Coalescing**: They combined multiple requests for the same data into one query. The first request creates a worker task, while later requests subscribe to that task instead of making new queries. When the task is completed, the result is shared with all subscribers. This approach prevents a spike in database requests and reduces the load on the database layer.
2. **Consistent(konsistınt) Hash-based Routing**: Each request is assigned a routing key, directing it to a specific data service instance. This way, all requests for the same data are centralized(sentrılayzd), avoiding overloading individual nodes.

**Slide 8: Hot Partition Solution**

With the next approach, if someone mentions @everyone on any channel, this does not lead to a sudden increase(inkıris) in the number of requests to the Database. Thus, the provision of data services helps to reduce the load on the database layer.

**Slide 10: Hot Partition Solution**

In the backend, it used the channelID to determine which service the request should be sent to.

As a result, requests for the same channel went to the same data service instance.

**Slide 11: Why ScyllaDB?**

These changes resulted in major performance improvements. But it was not a solution on a large scale. Much less frequently, the hot partition problem appeared.

To solve all the problems, they decided to use ScyllaDB and super disk storage technology.

ScyllaDB is a NoSQL database compatible with Cassandra, but offering higher performance and lower latency.

ScyllaDB's high-performance and low-latency features enable better management of large data streams and high-density requests.

**Slide 12: Data Service Layer and Request Coalescing**

Discord’s data service layer functions as a mediator between the API layer and ScyllaDB clusters, efficiently(efeşinliy) managing database queries. It also uses request coalescing, which means that when multiple users request the same data, only one query reaches the database. This feature greatly reduces the load and boosts performance.

**Slide 13: May 2022 Discord Statement**

By May 2022, Discord noted significant performance improvements: they reduced their infrastructure(infrasırakçır) from 177 Cassandra nodes to just 72 ScyllaDB nodes. Each ScyllaDB node could handle around 9 TB of data, compared to Cassandra’s 4 TB. Message retrieval latency dropped from 40–125 ms to just 15 ms, and message insertion stabilized at 5 ms. This enabled Discord to introduce new product features confidently, knowing they had a stable, reliable data storage solution.

**14**

A graphic showing the impact of goals and key moments on discord during the World Cup, thanks to the stability of the system. (This way they had the chance to watch the match during the meeting...)

I know it was supposed to be a completely failed project, but I came across it while researching and wanted to share it with the solution.