In my Hospital Management System application, I implemented transaction management to ensure that database operations are executed reliably and consistently. To achieve this, I utilized both declarative and programmatic transactions, each serving different purposes and providing distinct advantages for the application's needs.

For **declarative transactions**, I used the @Transactional annotation. This approach allowed me to automatically manage transaction boundaries, which is particularly useful for operations that involve multiple steps, such as patient registration and ward assignments. By marking methods with @Transactional, I ensured that these multi-step processes are treated as a single transaction, meaning all operations within the method are either fully committed to the database or completely rolled back in the event of an error. This automatic management simplifies code and enhances reliability, as I do not need to manually control transaction logic for these common operations.

On the other hand, **programmatic transactions** were managed using PlatformTransactionManager, providing finer control for more complex workflows. This approach was essential for scenarios such as batch processing, where the application needed to conditionally commit certain operations based on specific criteria or results. By managing transactions programmatically, I could write more granular transaction logic to handle such cases, ensuring that each step of the workflow could be individually assessed and either committed or rolled back, depending on the situation.

In addition to these transaction management strategies, I explored various transaction propagation settings to define how transactions behave when methods are called within other transactional contexts. This was crucial for controlling how nested transactions interact, ensuring that each operation is appropriately isolated or propagated, as needed. I also configured different isolation levels to prevent data inconsistencies, such as dirty reads and phantom reads, which could compromise the integrity of the system’s data.

To further enhance the performance of the Hospital Management System, I implemented caching to store frequently accessed data in memory. By using Spring’s @Cacheable annotation, I was able to significantly reduce the database load for common operations like fetching patient details or ward information. This caching strategy led to faster response times, improving the overall user experience.

To maintain data accuracy in the cache, I also configured cache eviction policies to ensure that outdated data is cleared and the cache is refreshed with updated information as needed. These

policies help maintain a balance between performance gains from caching and the necessity of having up-to-date data available to users.

Overall, these enhancements to the Hospital Management System—combining robust transaction management and efficient caching strategies—have ensured both high performance and reliable data management, effectively supporting the system's operational needs and providing a better experience for its users.