Design Goals, Connectors, and Architectural Style

Design Goals

1. **Usability**: The system ensures intuitive navigation for all users without requiring training, prioritizing user experience.
   - **Trade-off:** Focusing on usability may increase the complexity of the front-end design, requiring more development time. Over-automation of user interactions may reduce flexibility, making the system feel impersonal to some users.
2. **Scalability**: Designed to handle increasing users and data efficiently, crucial for peak times.
   - **Trade-off:** Scaling the system often requires a more complex architecture, which increases initial development time and costs. Automatic scaling features can introduce over-provisioning, leading to unnecessary resource usage during off-peak periods.
3. **Performance**: Key operations like tour applications and updates are optimized to execute swiftly, ensuring responsiveness.
   - **Trade-off:** Optimizing for performance may limit the complexity of certain features, reducing flexibility in system design. Automating tasks like background processing can improve performance but complicates system management and introduces potential delays.
4. **Maintainability**: A modular design allows for easy updates and troubleshooting, minimizing future maintenance efforts.
   - **Trade-off:** Modularity improves maintainability but can create overhead in terms of integration and testing. Automated deployment and testing pipelines simplify updates but require upfront investment in time and resources.
5. **Security**: Robust authentication and data protection measures are implemented to ensure user trust and privacy.
   - **Trade-off:** Strong security measures, such as multi-factor authentication, can impact usability by adding extra steps for users. Automating security monitoring can enhance protection but may lead to false positives or miss nuanced security threats.

These goals balance usability, long-term efficiency, and security.

Connectors

1. **REST APIs**: Used for communication between the React frontend and Express.js backend. This ensures simplicity and scalability but lacks real-time capabilities, which is acceptable for current needs.

2. **Mongoose ORM**: Provides structured access to MongoDB, enhancing schema enforcement and maintainability while slightly trading off raw query performance.
3. **SMTP for Notifications**: Handles email notifications efficiently for user updates and confirmations.

These connectors were chosen for their compatibility and alignment with the system's design goals.

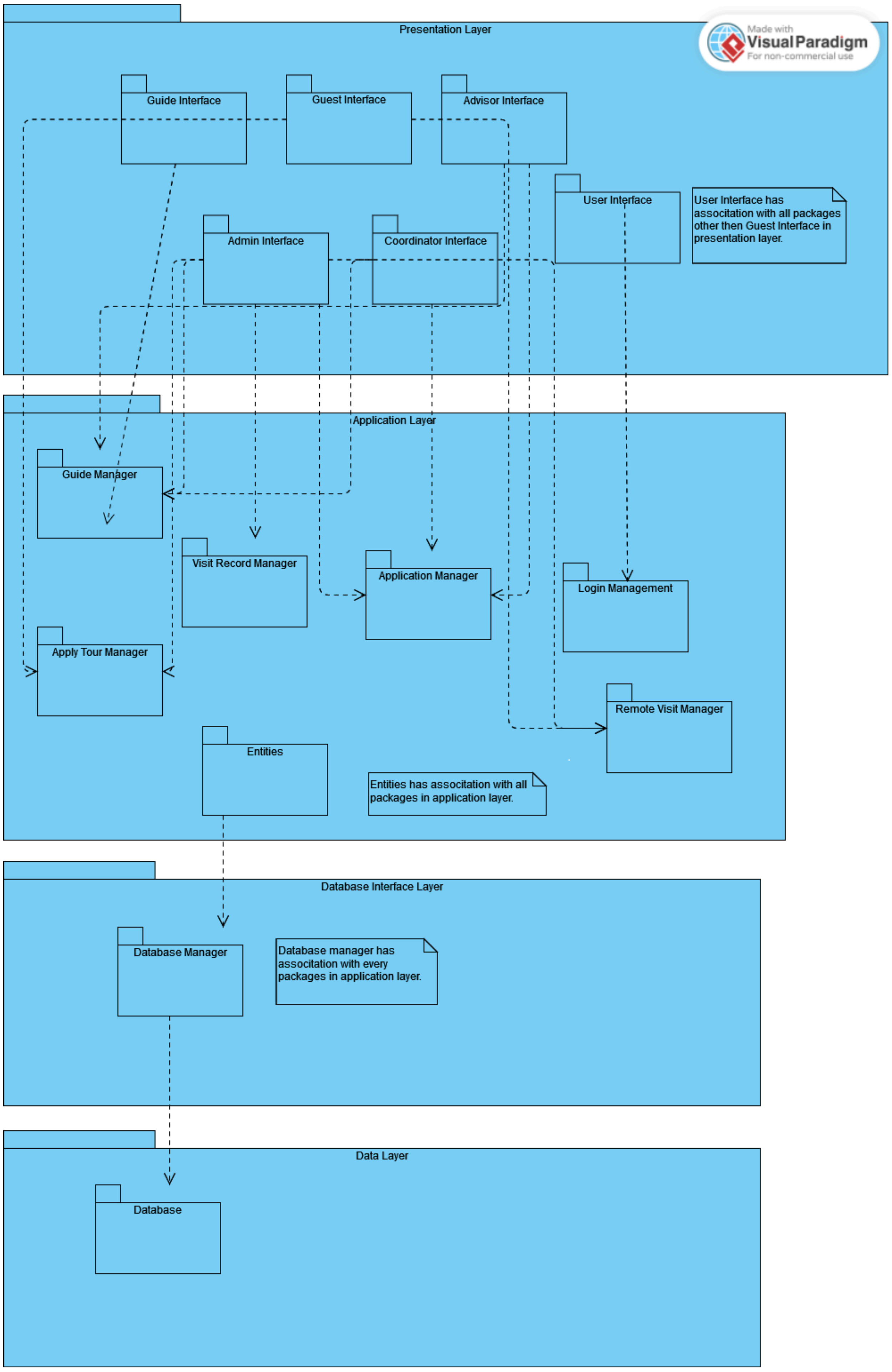
Architectural Style

The system employs a **Layered Architecture**, dividing responsibilities across three layers:

- **Presentation Layer**: React for user interaction.
- **Business Logic Layer**: Express.js for request handling and logic.
- **Data Access Layer**: MongoDB accessed via Mongoose for efficient data storage.

This architecture ensures scalability, maintainability, and flexibility while slightly trading off inter-layer communication speed for long-term benefits.

Subsystem Decomposition Diagram


Presentation Layer
Made with Visual Paradigm
For non-commercial use
Guide Interface
Guest Interface
Advisor Interface
User Interface
User Interface has associtation with all packages other then Guest Interface in presentation layer.
Admin Interface
Coordinator Interface
Application Layer
Guide Manager
Visit Record Manager
Application Manager
Login Management
Apply Tour Manager
Remote Visit Manager
Entities
Entities has associtation with all packages in application layer.
Database Interface Layer
Database Manager
Database manager has associtation with every packages in application layer.
Data Layer
Database