1241EA
Lita Naomi
Chiorean Rebeca
Adascalitei Ana-Maria

Project Name: Locafy

Evaluate three different software architectures for your system:

1. Monolithic architecture:

Source of inspiration :
https://medium.com/@vino7tech/monolithic-and-microservice-architectures-in-spring-boot-6a294e507dea

- Describe how the system would be structured in that style (main components, interactions, and data flow).

  The entire system is packaged as **one Spring Boot JAR/WAR**. Even though the code is organized into packages such as `auth`, `search`, or `reviews`, they are not separate services. They compile, run, and scale together.

  Main Components -> Modules (packages):

  - **Auth** — Spring Security with JWT or session-based authentication
  - **Users** — user accounts and profile management
  - **BusinessProfile** — business listings, metadata, and image handling
  - **Reviews** — review CRUD and rating aggregation
  - **Favorites** — user bookmarks
  - **Search** — pluggable search strategies (Strategy pattern)
  - **Notifications** — in-app/email/ push notifications
  - **Admin/Moderation** — administrative actions and content moderation
  - **MapAdapter** — Google Maps geocoding integration
  - **CacheManager** — caching layer (local memory or Redis)
  - **EventBus** — in-process event dispatcher for domain events
  - **StorageService** — local filesystem storage

Data Flow

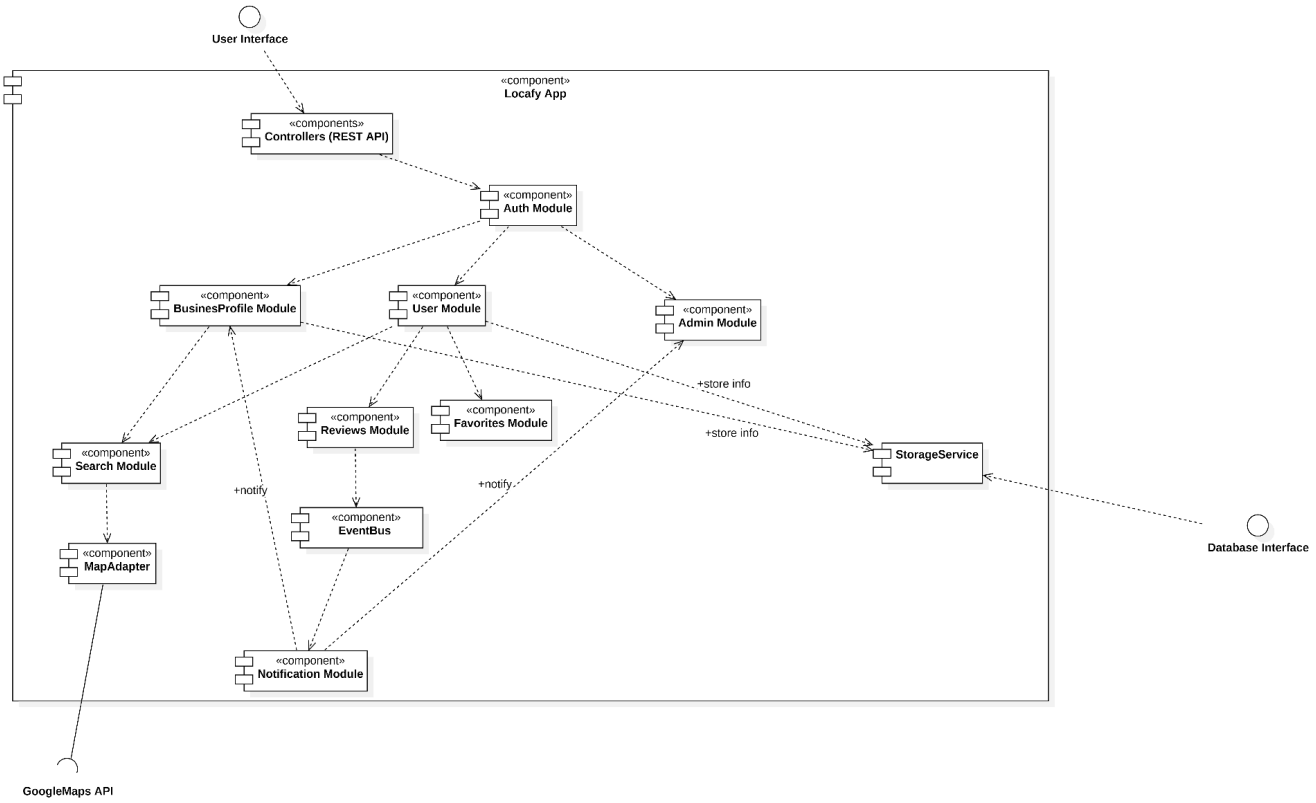
- **Database:** A single relational database shared by the entire system, running on the same machine (no Dockerized or independent DB components).
- **File Storage:** Images stored on local disk on the same computer running the application without Docker containerization.

Interactions:

- Client → single HTTP REST API (Spring MVC / controllers).
- SearchStrategy is selected at runtime (Strategy pattern) inside the SearchService.
- When a review is posted: ReviewController → ReviewService saves in DB → in-process EventBus publishes ReviewCreatedEvent → NotificationService, AnalyticsService, ReputationService react synchronously/asynchronously in same process (threads).
- CacheManager is a singleton Spring bean (e.g., backed by Redis or local cache).
- GoogleMapsAdapter called for geocoding on profile creation or search.

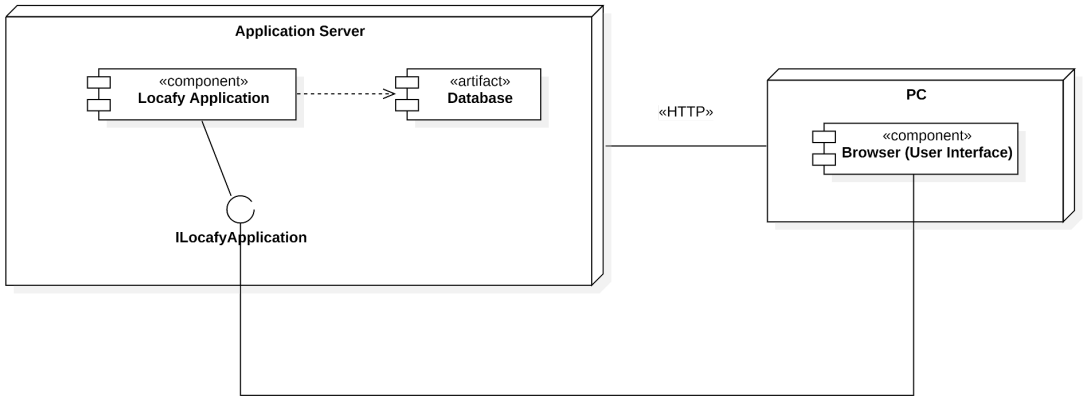
- Provide a deployment and component digrams to illustrate the architecture. Component Diagram:**->** Shows **what** the main parts of your system are and **how they interact**.

Component Diagram Monolithic Architecture



Deployment Diagram: **->** Shows **where** each component runs — servers, containers, databases, etc

Deployment Diagram Monolithic Architecture



- Discuss the pros and cons of using this style for your project.

  Source: https://www.geeksforgeeks.org/system-design/monolithic-architecture-system-design/

  Pros:

  - **Simplicity:** Monolithic architectures offer straightforward development and deployment processes.
  - **Cost-Effectiveness:** Compared to distributed systems, they require simpler deployment settings and less infrastructure overhead.
  - **Performance:** Because everything is operating within the same process, monolithic systems can occasionally offer higher performance because there is less communication overhead between components.
  - **Security:** With fewer inter-service communication points, monolithic systems may have a reduced attack surface, making them potentially more secure, especially if proper security measures are implemented.

Cons:

- **Scalability:** you must scale whole app even if only Search or Image upload is heavy.
- **Deployment friction:** entire app redeploys for small changes.
- **Team/ownership**: harder to split work among multiple teams; risk of merge conflicts.
- **Technology lock-in:** everything must use same stack/version; harder to adopt new tech per component.
- **Fault isolation:** a bug in one module can bring down whole app.

2. Microservices architecture:

Source of inspiration : https://medium.com/@vino7tech/monolithic-and-microservice-architectures-in-spring-boot-6a294e507dea

- Describe how the system would be structured in that style (main components, interactions, and data flow).

Each service focuses on a narrow domain and encapsulates its own data and logic:

Main Components -> Services:

- **Auth Service**: handles registration, login, roles.
- **User Service**: Manages basic user profiles.
- BusinessUser Service: Manages business user profiles.
- **Business Service**: manages business profiles, images and metadata.
- **Image Service**: receives image upload, stores the image in the database
- **Search Service**: search indexing and queries
- **Review Service**: persists reviews, exposes review APIs.
- **Favorite Service:** Handles user favorites and saved business lists.
- **Notification Service:** Sends push, email, or in-app notifications in response to events.
- **Reputation / Analytics Service:** Computes scores, aggregates metrics, and performs statistical analysis.
- **Gateway/API Gateway**: single public endpoint (routes to services, central auth check).

- **Event Broker**:RabbitMQ for events (`review.created, business.updated`).
- **Databases**: each service owns its DB

Interactions:

- **API Gateway**
  A single public entry point (e.g., Spring Cloud Gateway or NGINX).
  Handles routing, authentication checks, rate limiting, and cross-cutting concerns.
- **Service Discovery**

  Eureka allows services to find each other dynamically.

  Source:
  https://www.geeksforgeeks.org/advance-java/spring-boot-eureka-server/

- **Config Server**
  Centralized configuration service using Spring Cloud Config (Git-backed).
- **Event Broker**
  RabbitMQ for asynchronous, event-driven communication:
  - `review.created`
  - `business.updated`
  - `image.uploaded`

Example of use — Posting a Review

1. Client → API Gateway → Review Service
   The Review Service validates the request and writes the review to its own Review DB.
2. Review Service → RabbitMQ
   Publishes an event: `review.created`
3. Event Consumers:

   Notification Service sends alerts to the business owner -> Reputation Service recalculates the business rating score - > Analytics Service updates dashboards -> Search Service consumes the event to update its Elasticsearch index

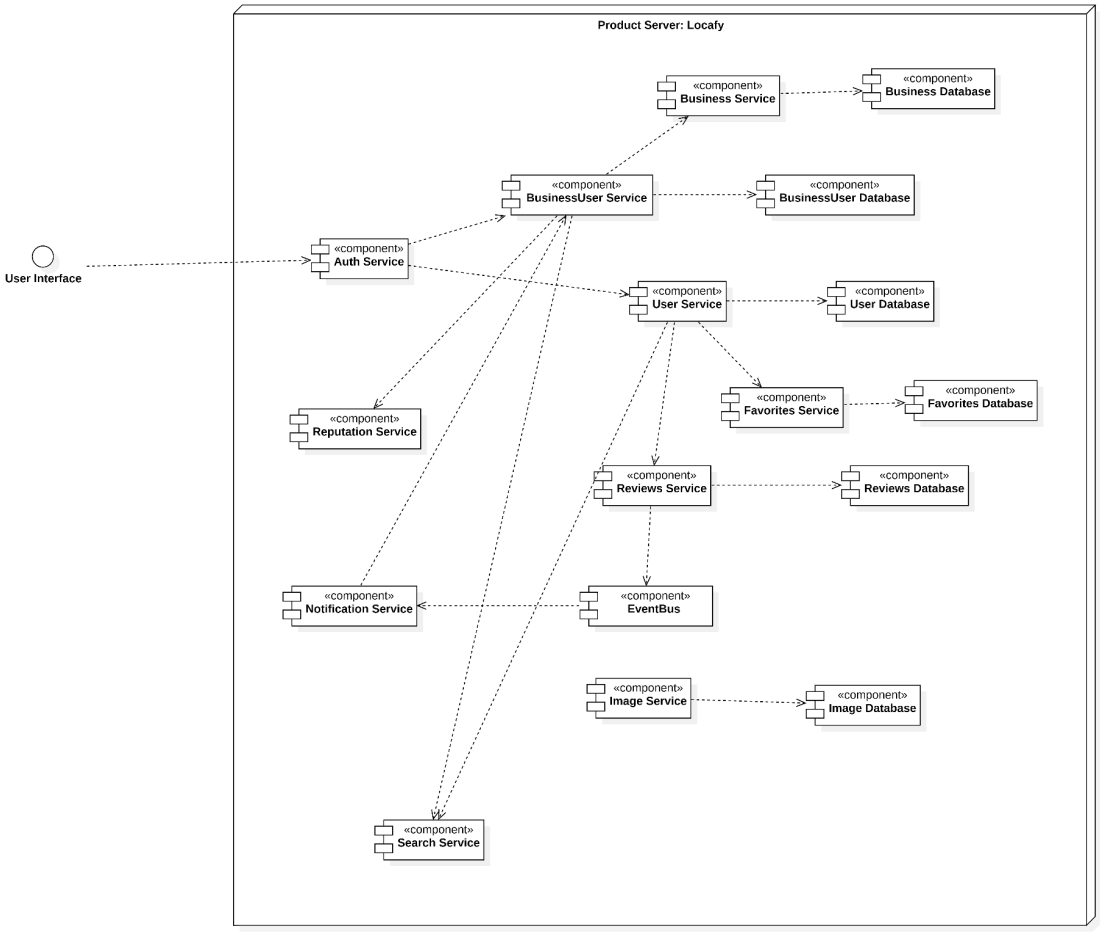
Data flow examples:

Each service manages its own data store (decentralized data management):

- Auth DB (e.g., Postgres)
- User DB
- Business DB
- Review DB
- Favorites DB
- Notification DB (optional)
- Analytics DB (warehouse / time-series DB)

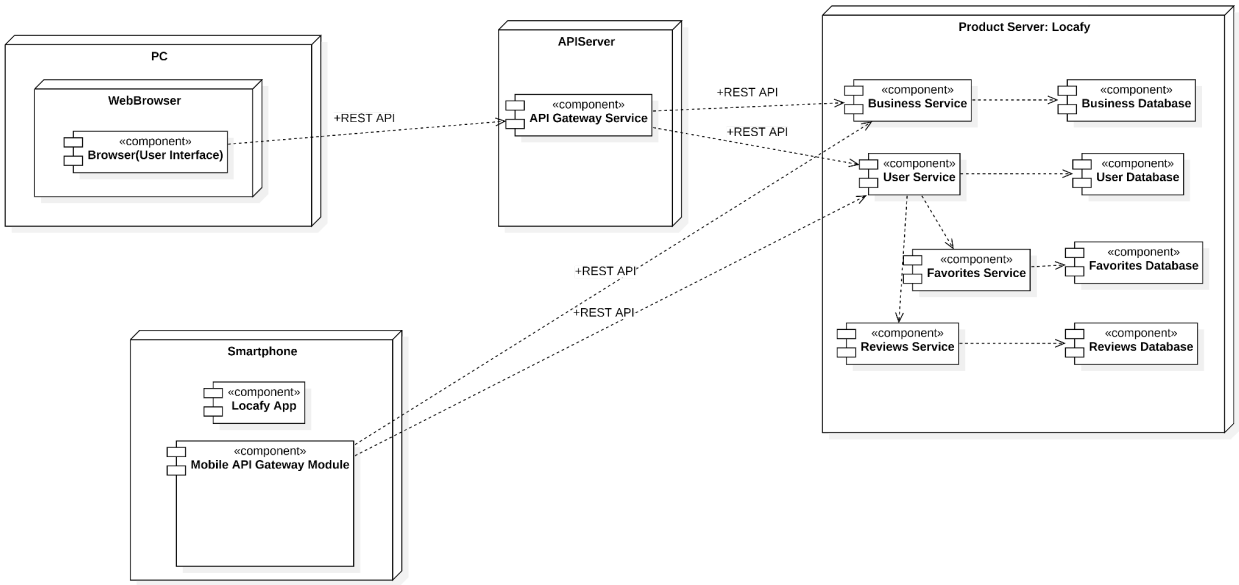
- Provide a deployment and component digrams to illustrate the architecture. . Component Diagram:**->** Shows **what** the main parts of your system are and **how they interact**.

Component Diagram Microservices Architecture



- Deployment Diagram: **->** Shows **where** each component runs — servers, containers, databases, etc

Deployment Diagram Microservices Architecture



- Discuss the pros and cons of using this style for your project.
  Pros

  - **Scalability**: Services can be scaled independently based on demand.
  - **Flexibility**: Each microservice can be developed, deployed, and maintained by different teams using different technologies.
  - **Resilience:** If one service fails, the rest of the application can continue to function.
  - **Faster Deployment:** Independent services allow for faster iteration and deployment of specific parts of the system.

  Cons

  - **Increased Complexity:** Managing multiple services can introduce operational complexity in areas like deployment, monitoring, and communication.
  - **Data Consistency:** Managing consistency between microservices, especially when they have separate databases, can be challenging.
  - **Latency:** Inter-service communication over the network introduces latency compared to in-process communication in a monolith.

- **Testing Challenges**: Testing distributed systems requires more effort than a single monolithic system.

3. Another distributed architecture style of your choice: 3 Layer Architecture

Source: https://www.geeksforgeeks.org/springboot/spring-boot-architecture/

- Describe how the system would be structured in that style (main components, interactions, and data flow).

Main Components -> Modules (packages):

1. Presentation Layer

   Handles HTTP requests (GET, POST, PUT, DELETE), request validation, authentication flow entry points, and JSON serialization/deserialization.

   - **AuthController** — authentication endpoints (login, register, refresh tokens)
   - **UserController** — user profile CRUD and account management
   - **BusinessProfileController** — business listing CRUD, metadata handling
   - **ReviewController** — review CRUD and rating-related endpoints
   - **FavoritesController** — user favorites management
   - **SearchController** — search endpoints, forwarding to SearchService
   - **AdminController** — administrative operations and moderation tools
   - **NotificationController** — notification preferences and delivery settings

2. Business Logic/ Application Layer

   Implements the application's core logic, integrates Spring Security, enforces rules, applies validations, coordinates workflows, and manages transactions (`@Transactional`).
   Communicates with the Persistence Layer.

- **AuthService** — authentication logic using JWT or session-based mechanisms
- **UserService** — manages user accounts, profile updates, and validation
- **BusinessProfileService** — handles business listing logic and metadata processing
- **ReviewService** — manages review creation, updates, deletion, and rating aggregation
- **FavoritesService** — user favorites storage, retrieval, and business rules
- **SearchService** — executes pluggable search strategies (Strategy Pattern)
- **NotificationService** — handles in-app, email, or push notifications
- **AdminModerationService** — moderation workflows, admin rule enforcement
- **EventBus** — in-process event dispatcher for domain events
- **CacheManager** — caching operations using local memory or Redis for performance
- **MapAdapter** — integrates with external geocoding services (e.g., Google Maps API)

3. 3. Persistence Layer (Repositories & Data Access)
   Manages all interaction with the database using Spring Data JPA or Hibernate. Maps Java objects to DB tables and performs CRUD operations.
   - **UserRepository** — persistent storage for users
   - **BusinessProfileRepository** — stores business listings and metadata
   - **ReviewRepository** — review storage, rating data access
   - **FavoritesRepository** — manages user favorites
   - **NotificationRepository** — notification preferences
   - **AdminRepository** — admin/moderation data
   - **SearchPersistenceAdapters** — custom DB access for search indexing or lookups

4. Data Layer
   Stores all permanent application data. Can be relational or NoSQL.
   - **Relational Database** (e.g., PostgreSQL, MySQL) — primary storage for structured data
   - **Redis (optional)** — used by CacheManager for caching

Interactions:

Client → Presentation Layer (Controllers)
All requests from the frontend client pass through REST controllers.
Controllers handle validation, authentication checks, and JSON parsing before delegating logic to the Business Layer.

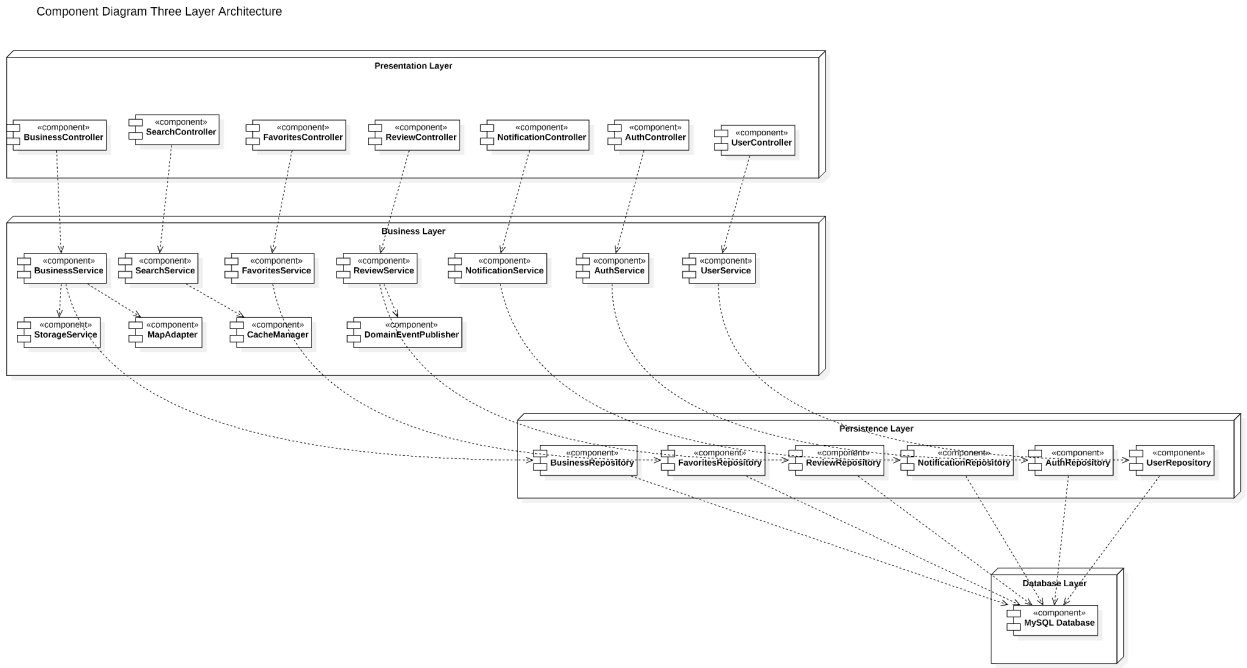
Search Strategy selected at runtime
Inside the `SearchService,` the appropriate `SearchStrategy` implementation (e.g., text-based search, location-based search) is chosen dynamically using the **Strategy Pattern**, based on the request parameters.

Review posting workflow
When a new review is created:
ReviewController → ReviewService (business logic) → ReviewRepository (DB write)
After saving the review, the Business Layer triggers an in-process domain event:
EventBus publishes `ReviewCreatedEvent`.

Data flow:

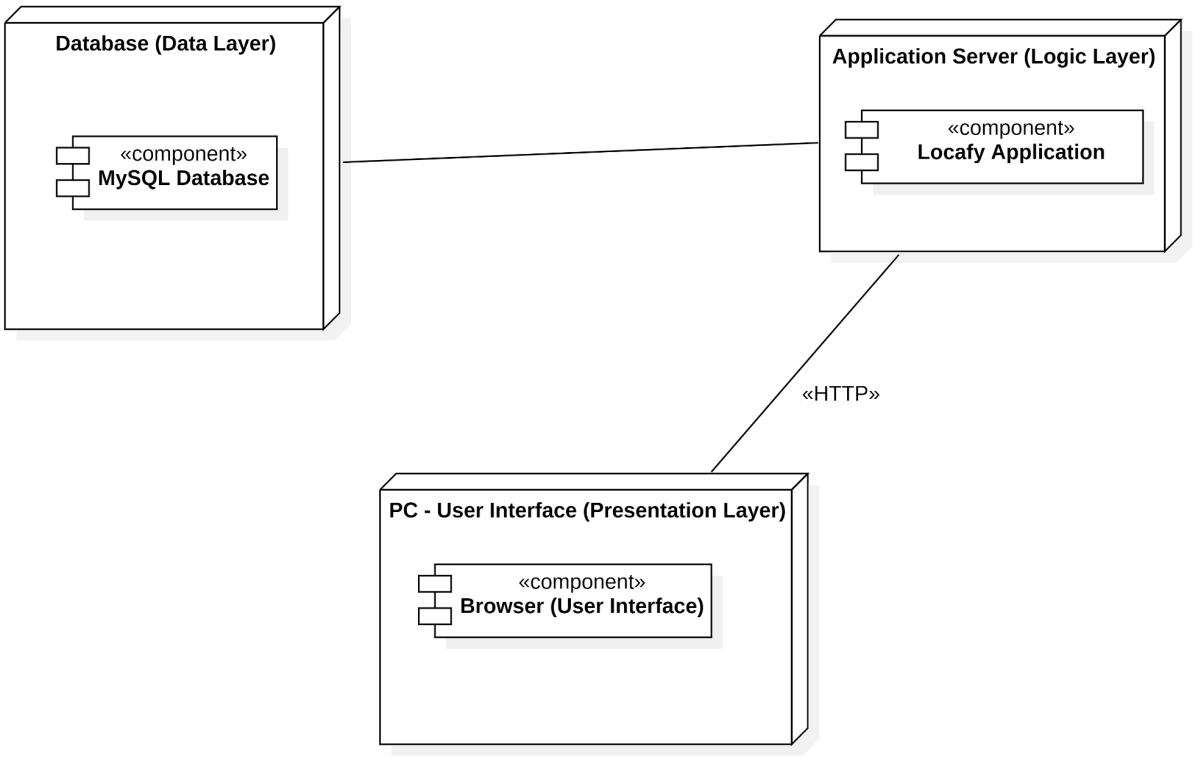
Database Access
All persistent data (users, businesses, reviews, favorites, notifications, search metadata) is managed through the **Persistence Layer**, which uses Spring Data JPA repositories.
These repositories communicate with **a single relational database** (e.g., PostgreSQL, MySQL).
The database runs on the same machine or server instance as the Spring Boot application—no separate containers or distributed databases.

- Provide a deployment and component digrams to illustrate the architecture. . Component Diagram:**->** Shows **what** the main parts of your system are and **how they interact**.



- Deployment Diagram: **->** Shows **where** each component runs — servers, containers, databases, etc

Deployment Diagram Three Layer Architecture



- Discuss the pros and cons of using this style for your project.

Pros:

- The key three-tier benefit is improved scalability, since the application servers can be deployed on many machines. Also, the database does not make longer connections with every client – it only requires connections from a smaller number of application servers.
- It improves data integrity. Here, all the updated information goes through the second tier. The second tier can ensure that only important information is allowed to be updated in the database and

the risk of unreliable client applications corrupting information is removed.

- Security is improved since the client does not have direct access to the database; it is more difficult for a client to obtain unauthorized data. Business logic is more secure because it is stored on a secure central server
- The added modularity makes it easier to modify or replace one tier without affecting the other tier.

Cons:

- It is more complex than the 2-tier client-server computing model, because it is more difficult to build a 3-tier application compared to a 2-tier application. The points of communication are doubled.
- The client does not maintain a persistent database connection.
- A separate proxy server may be required.
- Improve complexity or effort.
- The physical separation of application servers containing business logic functions and database servers containing databases may be something that affects performance.

Compare the three alternatives and explain which one you consider the most suitable for your project, and why.

The most suitable one is the three layer architecture. This solution is the best for a medium sized application.