Project Design Phase-II Technology Stack (Architecture & Stack)

|  |  |
| --- | --- |
| Date | 27 October, 2023 |
| Team ID | PNT2022TMID591061 (team id-591061) |
| Project Name | Project - Wanderlust: A Personalized Travel Planning And Tracking App |
| Maximum Marks | 4 Marks |

Technical Architecture:

The Deliverable shall include the architectural diagram as below and the information as per the table1 & table 2

Example: Order processing during pandemics for offline mode

Reference: <https://developer.ibm.com/patterns/ai-powered-backend-system-for-order-processing-during-pandemics/>

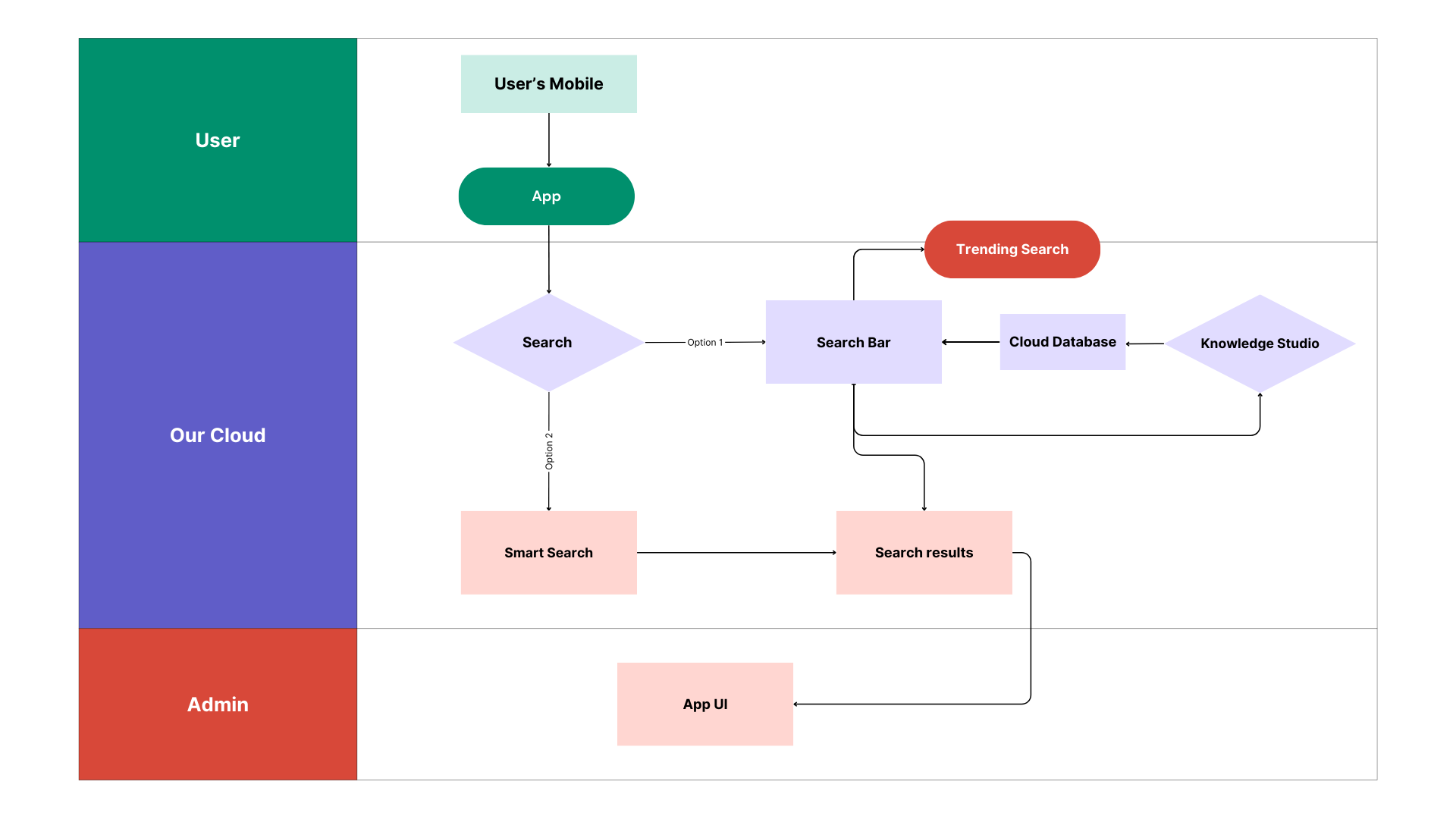


Table-1 : Components & Technologies:

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Component** | **Description** | **Technology** |
| 1. | User Interface | User can login the App and use it for organizing and planning their travel or trips , also allows user to get personalized feed of recommended accommodations based on the locations | HTML, CSS, JavaScript / Angular Js / React Js etc. |
| 2. | Application Logic-1 | Login Process | HTML, CSS, MySQL Php. |
| 3. | Application Logic-2 | Search for different locations and accommodations in the App itself. | Python, HTML, CSS, Kotlin. |
| 4. | Application Logic-3 | Recommendation of accommodations based on the locations. | Python, HTML, CSS, Kotlin.  Google Maps API, Google locations API. |
| 5. | Database | Locations ,Accommodations, Costs, Text , numeric, alphanumeric and Date and Time type, data type. | MySQL, NoSQL, Firrebase Realtime Database. |
| 6. | Cloud Database | Database Service on Cloud | IBM DB2, IBM Cloudant etc. |
| 7. | File Storage | File storage requirements | IBM Block Storage or Other Storage Service or Local Filesystem |
| 8. | External API-1 | For displaying maps and handling location-based services. | Google Maps API |
| 9. | External API-2 | For retrieving information about places, including details like name, address, user ratings, and reviews. | Google Places API |
| 10. | Machine Learning Model | Manage UI-related data in a lifecycle-conscious way. | ViewModel. |
| 11. | Infrastructure (Server / Cloud) | Cloud Server Configuration : | Local, Cloud Foundry, Kubernetes, etc. |

Table-2: Application Characteristics:

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Characteristics** | **Description** | **Technology** |
| 1. | Open-Source Frameworks | * Kotlin Coroutines * Koin * SQLDelight * Accompanist | Technology of Opensource framework |
| 2. | Security Implementations | * Authentication and Authorization * Data Storage * Code Obfuscation and Proguard * Authentication Tokens | * OAuth or JWT * Android's Secure Storage APIs or EncryptedSharedPreferences * Proguard or R8 * JWT |
| 3. | Scalable Architecture | Presentation Layer (Android Compose UI):  Scalability Through Compose: Android Compose allows for the creation of highly modular and reusable UI components. As your app's UI requirements grow, Compose facilitates the development of scalable and maintainable UI elements. New features and screens can be added without disrupting existing functionality, ensuring smooth scalability.  Logic Layer (Backend Services):  Scalability of Backend Services: By separating the presentation layer from the backend logic, you can scale each layer independently. Backend services can be designed as stateless, allowing them to handle an increasing number of user requests efficiently. Implementing load balancing techniques and deploying the backend on scalable cloud platforms ensures that the logic layer can handle a growing user base and data processing demands.  Data Layer (Database and Data Processing):  Scalable Database Solutions: Utilize scalable database solutions, such as cloud-based databases or NoSQL databases, which can handle large volumes of data and read/write operations. Proper indexing and sharing techniques can be applied to distribute the database load efficiently as data grows.  **Microservices Architecture:**  Microservices architecture breaks down the application into smaller, independent services, each responsible for a specific functionality. For your travel app:   * **Modular Development:** Microservices allow developers to work on isolated modules or services independently. New features can be added or existing ones can be updated without affecting the entire application, enabling faster development and deployment of new functionalities. * **Scalability Through Service Independence:** Each microservice can be deployed and scaled independently based on its specific requirements. For example, the recommendation engine responsible for personalized accommodations can be scaled independently from other services. This ensures that resources are allocated precisely where they are needed, optimizing efficiency and responsiveness. * **Fault Tolerance and Redundancy:** Microservices can be designed for fault tolerance. If one service fails, it does not bring down the entire application. Redundancy and load balancing can be implemented to ensure continuous service availability, enhancing the app's reliability. * **Third-Party Integrations:** Microservices can be specialized to handle third-party integrations (e.g., payment gateways, mapping services). By isolating these integrations into microservices, they can be individually scaled based on usage patterns without impacting other parts of the application. | Technology used |

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Characteristics** | **Description** | **Technology** |
| 4. | Availability | **\*\*1. \*\* Load Balancers:**   * **Efficient Traffic Distribution:** Load balancers distribute incoming network traffic across multiple servers. In the context of your travel app, load balancers ensure that user requests are evenly distributed among several servers. This prevents any single server from becoming overloaded, ensuring consistent performance even during high traffic periods. * **Fault Tolerance:** Load balancers can detect unhealthy servers and redirect traffic to healthy ones. If a server fails, the load balancer automatically routes traffic to functioning servers, minimizing downtime and providing continuous service availability. * **Scalability**: Load balancers enable seamless scalability. As the user base grows, additional servers can be added, and the load balancer will distribute the traffic efficiently. This dynamic scaling ensures that your app can handle a higher volume of users without compromising performance.   **\*\*2. \*\* Distributed Servers:**   * **Redundancy:** Distributing your application across multiple servers, possibly in different geographical locations, provides redundancy. If one server or data center experiences issues, other servers can handle the traffic, ensuring continuous availability. * **Disaster Recovery:** Having servers in different regions or data centers provides a level of disaster recovery. If one region experiences a network outage or natural disaster, users can be redirected to servers in another region, minimizing service disruptions. * **Data Replication:** For personalized feeds and recommendations, data replication can be used. By replicating data across multiple servers, each server can serve personalized content without relying on a single centralized database. This approach enhances responsiveness and availability.   **\*\*3. \*\* Caching Mechanisms:**   * **Content Delivery Networks (CDN):** CDNs cache static content (images, stylesheets, etc.) in servers distributed globally. When a user requests content, it is served from the nearest CDN server, reducing latency and ensuring fast content delivery. This is particularly useful for image-heavy travel apps where fast loading of images is crucial for user experience.   **Edge Caching:** Edge caching brings content closer to users by caching it in servers located near user populations. This reduces the distance data needs to travel, improving response times and availability.  **\*5. \*\* Content Delivery Optimization:**   * **Data Compression:** Compress data sent to clients to reduce bandwidth usage. This is especially relevant for mobile apps where data usage affects user experience and costs. * **Optimized Data Fetching:** Use efficient algorithms and data structures to fetch personalized recommendations. Minimize unnecessary data transfers and ensure that only relevant data is transmitted, reducing server load and response times. | Technology used |
| 5. | Performance | **1. Number of Requests Per Second (RPS)**  - Optimized Backend Endpoints: Design backend APIs to be efficient and optimized. Reduce unnecessary data in API responses and fetch only the required information.  - Batch Requests: Instead of making multiple small requests, batch API requests where possible to reduce the overhead of multiple HTTP connections.  **2. Cache Mechanisms:**  - Client-Side Caching: Implement caching mechanisms on the client side to store static content locally. Compose provides mechanisms to efficiently manage local state, which can be used to cache UI components and data.  - Server-Side Caching: Utilize server-side caching strategies to cache frequently accessed data and responses. Consider using in-memory caches or distributed caching solutions like Redis to store dynamic data temporarily.  **3. Content Delivery Networks (CDNs):**  - Static Content CDN: Use CDNs to cache and deliver static assets such as images, stylesheets, and JavaScript files. This reduces the load on your servers and accelerates content delivery to users.  - Edge Caching: Leverage edge caching provided by CDNs. Cached content stored at edge locations near users reduces latency and improves response times, enhancing the app's overall performance.  **4. Optimized Data Fetching:**  - Lazy Loading: Utilize lazy loading mechanisms in Compose to load data and UI components only when they are needed. This ensures that resources are used efficiently, especially in lists or grids where not all items are visible at once.  - Pagination: Implement pagination for long lists to load data incrementally, reducing the initial load time and ensuring a more responsive user interface.  **5. Image Loading and Compression:\*\***  - Image Loading Libraries: Use efficient image loading libraries like Coil or Glide. These libraries handle image caching, loading, and display, optimizing the loading of images in your app.  - Image Compression: Compress images to reduce file sizes before they are loaded into the app. This reduces bandwidth usage and speeds up image loading times.  **6. Minimize Network Requests:**  - Data Bundling: Bundle multiple API requests into a single request to reduce the number of network calls. Aggregate data on the server side to minimize the data sent over the network.  - WebSocket: Consider using WebSocket for real-time updates instead of polling the server at regular intervals. WebSocket allows bidirectional communication between the client and server, enabling real-time notifications and updates.  **7. Optimized Database Queries:**  - Database Indexing: If your app uses a local database, ensure that database queries are optimized and utilize proper indexing. Indexes speed up data retrieval, especially when dealing with large datasets.  - Offline Support: Implement local data storage and synchronization mechanisms to provide basic functionality even when the user is offline. This enhances the user experience and ensures the app remains usable in adverse network conditions.  **8. Code and UI Optimization:**  - UI Rendering Optimization: Compose provides tools to optimize UI rendering. Utilize the `remember` function and `state` to manage UI state efficiently. Avoid unnecessary recompositions by optimizing your UI components.  - Code Splitting: Split your code into smaller modules to reduce the initial app download size. This is particularly important for large applications with extensive features.  **9. Performance Monitoring and Profiling:**  - Profiling Tools: Use Android Profiler and other profiling tools to identify performance bottlenecks in your app. Analyze CPU usage, memory allocation, and network activity to pinpoint areas that need optimization.  - Crash Reporting: Implement crash reporting tools to identify and fix issues quickly. Unhandled crashes can severely impact user experience and need to be addressed promptly.  By carefully considering these design strategies, your travel app built with Android Compose can deliver high performance, ensuring a smooth and responsive user experience, even in scenarios involving a high number of requests, personalized content, and real-time recommendations. Regular testing, profiling, and optimization are key to maintaining optimal performance as your app evolves. | Technology used |

References:

<https://c4model.com/>

<https://developer.ibm.com/patterns/online-order-processing-system-during-pandemic/> <https://www.ibm.com/cloud/architecture>

<https://aws.amazon.com/architecture>

<https://medium.com/the-internal-startup/how-to-draw-useful-technical-architecture-diagrams-2d20c9fda90d>