**Clync App**

**General Specifications for Mobile Application:**

**Mobile application Apps:**

* IOS mobile application
* Android mobile application

**- App Users:**

* Individuals

**- Mobile phone number mask to be used in mobile application:**

* +20

**- Interface language of mobile applications:**

* English
* Arabic

**- Currency to be used in mobile application:**

* EGP

**Features/Modules (Mobile Apps)**

|  |  |
| --- | --- |
| Feature | Description |
| User Registration | To register in the Clync application |
| User login/authorization | To get authorized in the Clync application – Login |
| Quick Access Bar | List of quick functional icons for quick access |
| Profile setting page (Profile) | View and edit the profile |
| Profile setting page(setting) | Account, support and about section |
| QR code (User, Circle) |  |
| Home | Designing all functional Buttons |
| Favorite Clync Friends |  |
| View Latest Transactions |  |
| View Payment Requests |  |
| Recent Clync Circles |  |
| Financial Hub | Spending Analysis (Adding 5 days in development due to the dependency on Backend and work on backend increased) |
| Create/edit pocket | create pocket with Planning and Saving, edit pocket, manual payment |
| Get Clync Card | Document scan for android and ios and order card |
| Manage Card | Edit Pin, freeze, top up card, add or remove |
| Top up/Cash (select from the card) |  |
| Top up/Cash (select from the Bank Account) |  |
| Request Money R2P |  |
| Send Money P2P |  |
| Transactions History | View and allocate transaction |
| Social Hub | Unsettled Receipts, Friends List, access Circle, add Circle |
| Friends List | Friend list and search friend, add and remove friend |
| Access Circles | Integrate chat module |
| Add/Create Circles |  |
| Settle Button |  |
| Collect Button |  |
| Split Receipt Evenly |  |
| Split Receipt by Item/Unevenly |  |
| Create/Access Direct Messaging | Start and delete the individual chat |
| Services Hub | My Vouchers, Notifications |
| Gift Transactions |  |
| Gift Cards | Also integrate payment gateway |
| E-Vouchers |  |
| Pay Services | Also integrate payment gateway |

**Software Stack:**

Here's an overview of the software technologies and frameworks used in the development of Clync mobile application:

1. **Flutter:** The primary framework used for building the mobile app. Flutter allows for cross-platform development using a single codebase written in the Dart programming language.
2. **GetX:** A state management library for Flutter that simplifies dependency injection, routing, and state management. GetX provides a lightweight and efficient solution for managing app state and navigation.
3. **Dart:** The programming language used with Flutter. Dart is a modern, object-oriented language with features such as strong typing, asynchronous programming, and a rich set of libraries.
4. **Firebase:** A mobile and web application development platform provided by Google. Firebase offers a suite of backend services, including authentication, real-time database, cloud storage, and cloud messaging, which are commonly used in mobile app development.
5. **Flutter Packages:** Various third-party packages and libraries available on pub.dev, the official package repository for Dart and Flutter. These packages provide additional functionalities and integrations, such as HTTP requests, SQLite database management, push notifications, and more.
6. **Material Design (or Cupertino Design):** Design guidelines provided by Google (for Android) and Apple (for iOS) that help maintain consistency and provide best practices for creating visually appealing and intuitive user interfaces. Flutter offers widgets that implement these design languages, allowing developers to create apps that look and feel native to each platform.
7. **Android Studio (or Visual Studio Code):** Integrated development environments (IDEs) commonly used for Flutter development. These IDEs provide features such as code editing, debugging, and project management, making the development process more efficient.

By leveraging these technologies and frameworks, mobile application benefits from a powerful combination of cross-platform development, efficient state management, backend services, and a rich ecosystem of third-party packages, resulting in a high-quality and feature-rich user experience.

Here's an Overview of the **backend API development** technology stack:

**Node.js:** A server-side JavaScript runtime environment used for building scalable and efficient web applications. Node.js allows for non-blocking, event driven I/O operations, making it ideal for handling asynchronous tasks in real-time applications.

**Express.js:** A minimalist web application framework for Node.js, Express.js simplifies the process of building robust APIs and web servers. It provides a thin layer of fundamental web application features, allowing developers to create RESTful APIs quickly and efficiently.

**PostgreSQL:** An open-source relational database management system known for its reliability, performance, and advanced features. PostgreSQL is highly extensible and supports a wide range of data types, making it suitable for handling complex data structures in backend applications.

**Prisma ORM:** A modern database toolkit and Object-Relational Mapping (ORM) tool for Node.js and TypeScript. Prisma simplifies database access and manipulation by providing a type-safe and auto-generated query builder. It supports various databases, including PostgreSQL, MySQL, and SQLite, and offers features like schema migrations and data seeding.

**Azure Cloud Storage:** Microsoft's cloud storage solution that provides scalable, secure, and highly available storage for web applications. Azure Cloud Storage offers various storage options, including Blob Storage for unstructured data, Table Storage for NoSQL data, Queue Storage for messaging between application components, and File Storage for file sharing in the cloud.

This technology stack combines the power of Node.js for server-side development, Express.js for building RESTful APIs, PostgreSQL for data storage, Prisma ORM for seamless database interaction, and Azure Cloud Storage for scalable and reliable cloud storage solutions. Together, they enable developers to create scalable, performant, and maintainable backend APIs for a wide range of applications.

**Architecture Style:**

Here's an overview of the foundational architectural design principles and design patterns employed in our mobile application development:

**Architectural Design Principles:**

* **Separation of Concerns (SoC):** Divide the app into UI, logic, and data layers.
* **Single Responsibility Principle (SRP):** Each component has one job.
* **Dependency Injection (DI):** Manage dependencies for loose coupling.
* **Immutable State:** Ensure predictability and prevent side effects.
* **Reactive Programming:** Widgets react to state changes automatically.

**Design Patterns:**

* **MVVM (Model-View-ViewModel):** ViewModel bridges View and Model, ensuring separation of concerns.
* **Observer Pattern:** Widgets subscribe to state changes, maintaining a reactive UI.
* **Factory Pattern:** Flexible object creation without specifying concrete classes.
* **Repository Pattern:** Abstracts data source details for clean data access.
* **Service Locator Pattern:** Centrally manages dependencies for modularization and scalability.

The purpose of using these principles and patterns is to promote modularity, testability, and scalability in Flutter apps using GetX.

Here's an overview of the architectural design principles and design patterns employed in the **backend API development technology stack** mentioned:

**Architectural Design Principles:**

* **Separation of Concerns (SoC):** Divide the application into distinct layers such as routing, controllers, services, and data access to ensure each component has a single responsibility and is easily maintainable.
* **Modularity:** Encapsulate related functionalities into modules or packages, promoting code reusability and scalability.
* **Dependency Injection (DI):** Utilize dependency injection to manage component dependencies and promote loose coupling between modules, enhancing testability and flexibility.
* **Scalability:** Design the application to handle increased load and traffic by employing scalable architectural patterns such as microservices, load balancing, and distributed caching.
* **Security:** Implement security measures such as authentication, authorization, data validation, and encryption to protect against common security threats like SQL injection, cross-site scripting (XSS), and unauthorized access.

**Design Patterns:**

* **MVC (Model-View-Controller):** Divide the application into three interconnected components: Models (data), Views (presentation layer), and Controllers (business logic). This promotes separation of concerns and enhances maintainability.
* **Repository Pattern:** Abstract the data access layer by providing a consistent interface for accessing and manipulating data. This pattern facilitates code reuse, testability, and enables switching between different data storage technologies seamlessly.
* **Factory Pattern:** Use factories to create instances of objects dynamically without specifying their concrete classes. This promotes flexibility and decouples object creation from the client code.
* **Middleware Pattern:** Utilize middleware functions to preprocess requests, execute additional logic, and modify responses before they reach the final handler. This pattern enhances modularity, reusability, and allows for cross-cutting concerns like logging, authentication, and error handling.
* **Decorator Pattern:** Attach additional responsibilities to objects dynamically by wrapping them with decorator objects. This pattern allows for the incremental addition of features to objects without modifying their structure, promoting code reuse and extensibility.

By adhering to these architectural design principles and design patterns, developers can create robust, scalable, and maintainable backend APIs using the technology stack mentioned. These principles and patterns promote modular, loosely coupled, and highly cohesive code, leading to enhanced flexibility, testability, and performance of the application

**Components:**

Detailed breakdown of the various components/modules is defined in table under General Specifications for Mobile Application.

**Databases Dictionary:**

Clync Database ERD (entity relationship diagram) is attached as PDF. It contains the complete ERD defining database schemas, tables, and relationships.

**Environment Sizing:**

Following detailed guidelines for environment ensure the Clync App's robust, efficient, and scalable operations across all environments, providing a reliable and high-quality user experience:

**Development Environment (Azure Dev Environment):**

* **Purpose:** Supports coding, debugging, and testing by developers.
* **Requirements:**
  + Azure VM Tier: Standard B2s
  + CPU: 2 vCPUs
  + Memory: 8 GB RAM
  + Storage: 100 GB SSD (Azure Managed Disk)
  + Instances: 1
  + Network: Azure Virtual Network with moderate speed, sufficient for code syncing and version control access

**Testing Environment (Azure Test Environment):**

* **Purpose:** Conducts functional, integration, and regression testing.
* **Requirements:**
  + Azure VM Tier: Standard D4s v3
  + CPU: 4 vCPUs
  + Memory: 16 GB RAM
  + Storage: 200 GB SSD (Azure Managed Disk)
  + Instances: 1 (to simulate diverse user environments and parallel test executions)
  + Network: Azure Virtual Network with high speed for efficient deployment and testing

**Production Environment (Azure Production Environment):**

* **Purpose:** Live deployment for end users ensuring high performance and reliability.
* **Requirements:**
  + Azure VM Tier: Standard D8s v3
  + CPU: 8 vCPUs
  + Memory: 32 GB RAM
  + Storage: 500 GB SSD (Azure Managed Disk)
  + Instances: 3 (to ensure high availability, fault tolerance, and load distribution)
  + Load Balancer: Nginx Load Balancer to distribute incoming traffic evenly
  + Database: Azure Database for PostgreSQL with primary and replica instances (appropriate tier based on workload)
  + Network: Azure Virtual Network with high speed and redundancy for continuous availability
  + Security: Azure Security Center for robust measures including encryption, firewalls, and intrusion detection

**Disaster Recovery Environment (Azure DR Environment):**

* Purpose: Backup environment to ensure quick recovery during failures.
* Requirements:
  + Azure VM Tier: Standard D4s v3
  + CPU: 4 vCPUs
  + Memory: 16 GB RAM
  + Storage: 200 GB SSD (Azure Managed Disk)
  + Instances: 4
  + Data Backup: Azure Blob Storage for regular snapshots, asynchronous replication from production to minimize data loss
  + Network: Azure Virtual Network with high speed and redundancy to ensure swift recovery

**Benchmark for Production:**

**Performance Benchmarks and Key Matrics**

* **Response Time:**
  + **API Response Time:** < 200ms for 95% of requests
  + **Page Load Time:** < 1 second for 90% of user interactions
* **Throughput:**
  + **Transactions per Second (TPS):** 500 TPS sustained, 1000 TPS during peak times
  + **User Concurrency:** Support for at least 10,000 concurrent users
* **Scalability:**
  + **Horizontal Scalability:** Seamless addition of instances to handle increased load
  + **Database Scalability:** PostgreSQL clustering and read replicas for high availability and load distribution
* **Data Metrics:**
  + **Database Query Performance:** Average query execution time < 100ms, with optimizations for complex queries
  + **Data Consistency:** Ensure strong consistency, particularly for transactional operations to maintain data integrity