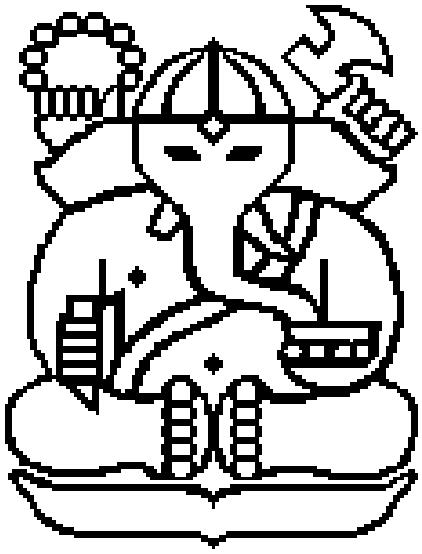
Pengembangan Perangkat Lunak Terdistribusi pada Domain Ticketing dan Reservation IF4031 Pengembangan Aplikasi Terdistribusi

oleh:

Chiquita Ahsanunnisa / 13521129

Rava Maulana / 13521149

Made Debby Almadea Putri / 13521153



PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2023/2024

Analisis Masalah

Sistem terdiri atas tiga *service* yang berhubungan. *Service* tersebut terdiri atas *client app*, *ticket app*, serta *payment app*. Berikut merupakan *requirement* untuk setiap *service*.

1. **Client App**
    - Memiliki CRUD *functionality* untuk manajemen data pengguna.
    - Menyimpan data pengguna secara persisten.
    - Memungkinkan pengguna tertentu untuk melakukan booking terhadap kursi dari event tertentu yang terdaftar pada ticket app.
    - Menyimpan data historis booking tiket masing-masing pengguna termasuk status bookingnya
    - Menyimpan data *booking* tiket yang berhasil maupun gagal

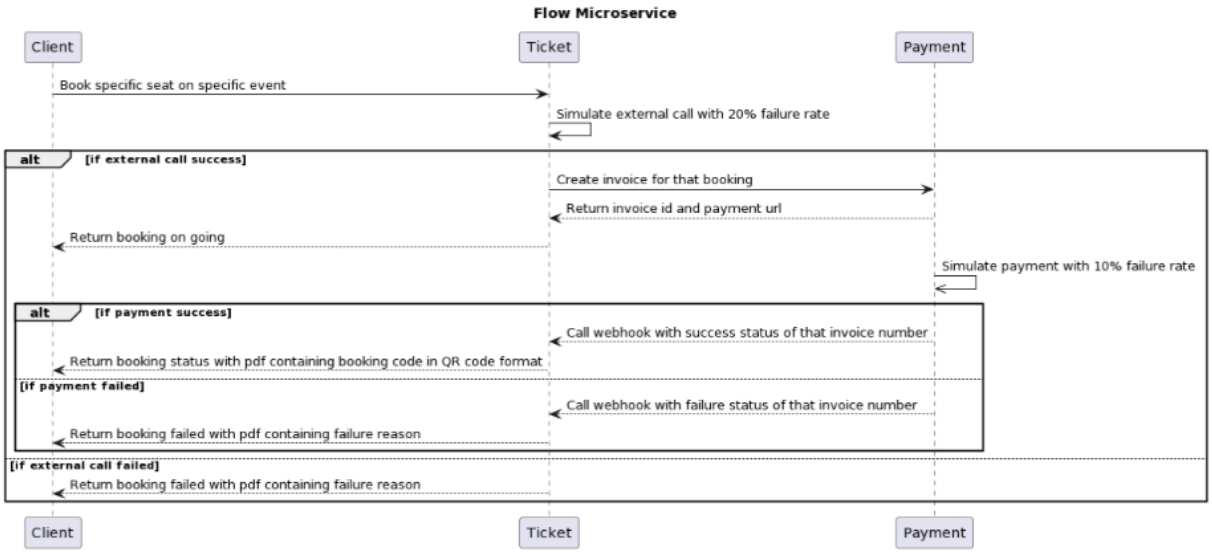
2. **Ticket App**
    - Ticket app harus memiliki CRUD functionality untuk manajemen data event dan kursi yang tersedia
    - Ticket app harus menyimpan data kursi yang tersedia yang setidaknya memuat ID Kursi, status (OPEN | ON GOING | BOOKED)
    - Karena Ticket App tidak menyelenggarakan satupun dari event yang ada, melainkan hanya sekedar menjadi penengah antara customer dan penyelenggara event tertentu, maka dalam proses pemesanan Ticket app akan melakukan pemanggilan eksternal ke sistem yang berkaitan (sistem event X, Y, dst). Pemanggilan eksternal ini terjadi secara synchronous dan hanya berperan untuk hold kursi dari event tertentu. Pemanggilan eksternal ini memiliki tingkat kegagalan 20% (simulasikan dengan random saja)
    - Setelah Ticket App berhasil melakukan hold dari kursi dari event tertentu, ticket app akan membuat invoice ke Payment App. Payment App kemudian akan mengembalikan invoice number dan url untuk membayarnya ke Ticket App yang kemudian akan diteruskan pada Client App.
    - Proses “hold seat” yang dilakukan dari client ke ticket app terjadi secara synchronous. Ticket app harus tetap dapat menerima booking meskipun sedang ada booking yang diproses (belum selesai diproses)
    - Ticket app akan memberitahu client app apabila ada booking yang berhasil/gagal.
    - Apabila booking berhasil, ticket app akan meng-generate PDF hasil booking dan mengirimkannya juga ke client (proses/prosedur storing & delivery dibebaskan)
    - Ticket app akan memiliki endpoint untuk client app dapat secara manual mengecek status dari booking tertentu.

3. **Payment App**
    - Payment App harus menyimpan data invoice baik yang berhasil maupun tidak secara persisten

- Payment App akan memproses pembayaran dengan rate kegagalan 10% (simulasikan dengan random)
- Pemrosesan pembayaran ini terjadi secara *asynchronous*, dan apabila prosesnya sudah selesai (berhasil maupun gagal), Payment App akan melakukan pemanggilan webhook (push message) ke WEBHOOK_URL yang berisikan status keberhasilan dari pembayaran tersebut.
- Payment App menyimpan konfigurasi berupa WEBHOOK_URL yang merupakan URL/Endpoint milik Ticket App.
- WEBHOOK_URL akan dipanggil setiap proses payment selesai dilaksanakan (baik sukses maupun gagal)

4. **Spesifikasi bonus**
   Reservasi tiket dimana pengguna dapat “mengantri” kursi tertentu yang sudah terisi dengan harapan apabila ada yang membatalkan ticket atas kursi tersebut, maka pengguna yang mengantri terlebih dahulu dapat memperoleh kursi tersebut.

Berikut merupakan *sequence diagram* yang menggambarkan alur pada ketiga *service* tersebut.



Berdasarkan lingkupnya, jenis kegagalan yang mungkin timbul dalam sistem dapat diklasifikasikan menjadi dua jenis, yaitu intra-service dan inter-service. Kegagalan intra *service* terjadi dalam lingkup satu *service* spesifik, sedangkan kegagalan *inter-service* terjadi antara dua atau lebih *service*. Berikut merupakan masalah yang kemungkinan akan timbul baik *intra-service* ataupun *inter-service*.

1. **Intra Service**
   - **Client App**
     a. Pengiriman email hasil booking dapat saja gagal sehingga pengguna tidak pernah tahu apakah booking yang dilakukan berhasil atau gagal.

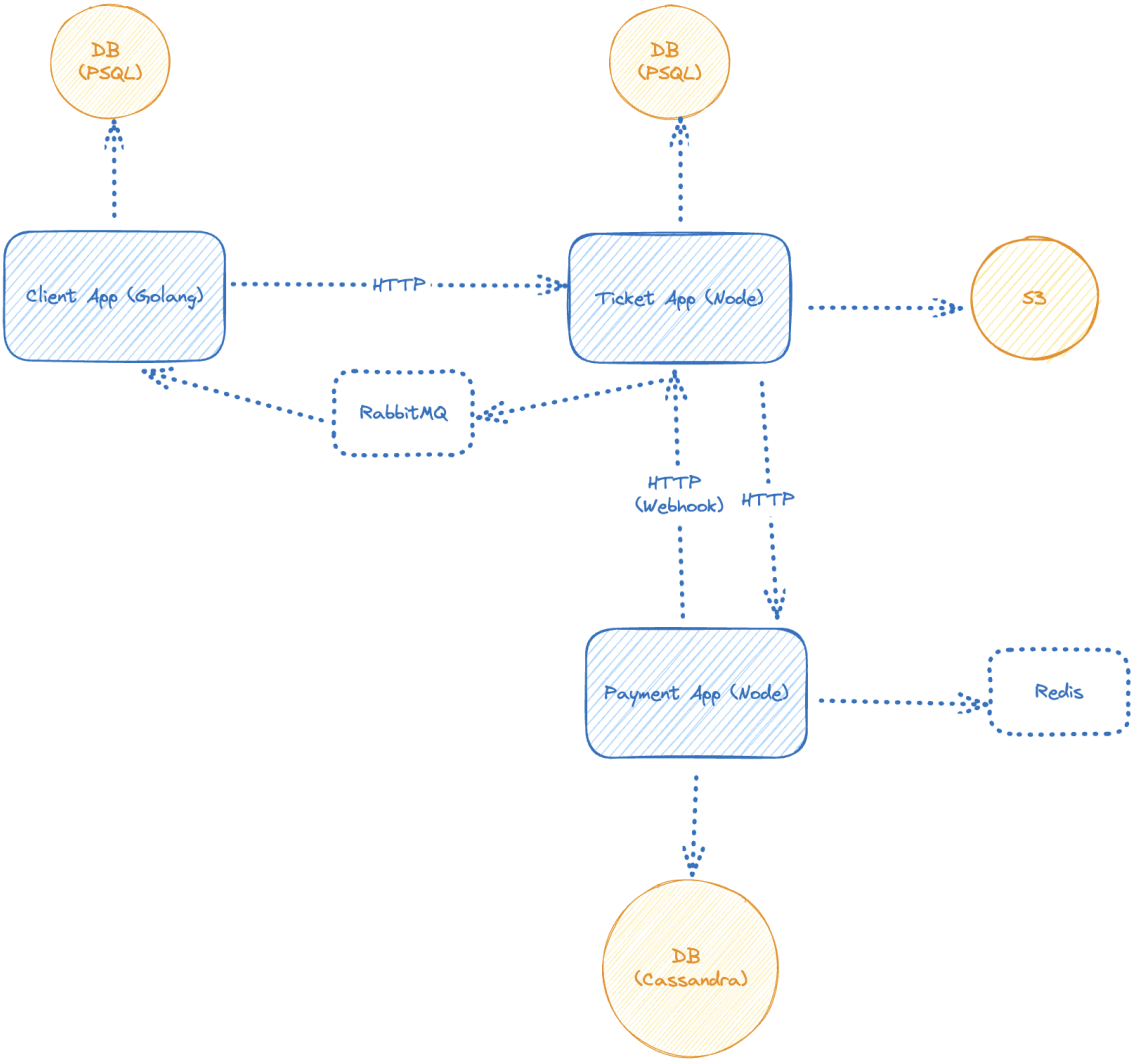
Selain itu, pengiriman email untuk link pembayaran juga bisa gagal sehingga pengguna tidak bisa membayar dan *seat* tersebut akan berada pada status “pending” untuk selamanya.

- **Ticket App**
    a. Integritas dan konsistensi data booking sehingga jika ada pemanggilan yang tidak sukses maka operasi sebelumnya dibatalkan. Seperti contoh ketika membuat booking terdapat pemanggilan ke *payment service* yang bisa saja *down*. Jika pemanggilan tersebut gagal maka *write* ke tabel booking sebelumnya harus dibatalkan juga.
    b. Penyimpanan PDF yang reliable sehingga pengguna dapat tetap mengakses PDF hasil booking melalui URL kapanpun.

2. Inter-Service

a. Pemanggilan dari Payment App ke Ticket App ketika payment sudah selesai diproses dapat menimbulkan masalah jika Ticket App *down*. Hal ini menjadi masalah karena *payment* bisa saja berhasil tetapi karena Ticket App *down* maka PDF tidak akan pernah sampai ke *user*. Oleh karena itu diperlukan mekanisme untuk menyimpan *request* yang gagal dikirim sehingga dapat dikirim ulang kembali tanpa membuat *request* lainnya *starving*.
b. Pemanggilan dari Ticket App ke Client App ketika *payment* sudah selesai dan Ticket App berhasil menerima *request* dari Payment App dapat menimbulkan masalah jika Client App *down*. Hal ini menjadi masalah karena PDF tidak akan pernah sampai ke *user*. Oleh karena itu diperlukan mekanisme untuk menyimpan *request* yang gagal dikirim sehingga ketika Client App kembali berjalan, *request* dapat diproses dan PDF dapat dikirim ke *user* dan masuk ke dalam *booking history*.

Deskripsi Solusi



1. **Client App**
   Client APP diimplementasikan menggunakan Go dengan kakas Chi Router untuk *routing*. Basis data Client diimplementasikan menggunakan PostgreSQL dengan menggunakan kakas Gorm sebagai ORM-nya. Client App dapat diakses oleh pengguna dengan memanfaatkan *endpoint* REST API yang disediakan.

2. **Ticket App**
   Secara umum, Ticket App akan diimplementasikan menggunakan Node express dengan Typescript dan basis datanya menggunakan PostgreSQL. Ticket App menggunakan REST API sehingga Client App berkomunikasi ke Ticket App dengan menggunakan *endpoint* REST API yang disediakan. Selain itu, Ticket App juga memanfaatkan *drizzle* untuk ORM

3. **Payment App**
   Payment App diimplementasikan dengan menggunakan bahasa pemrograman TypeScript dengan kakas Node.js dan Express. Basis data diimplementasikan dengan menggunakan Cassandra. *Service* dihubungkan dengan basis data melalui *driver* DataStax Cassandra untuk Node.js. Payment App dapat berkomunikasi dengan Ticket App dan pengguna lewat *endpoint* REST API yang disediakan.

4. **Spesifikasi bonus**
   Sistem antrian diimplementasikan di dalam Ticket App dengan menggunakan database PostgreSQL langsung sehingga terdapat tabel baru yaitu *booking_queue* yang menyimpan antrian dari tiket. Jika Client App me-request create booking tetapi *seat* yang di-*request* sedang di-*hold* oleh orang lain, maka *request* akan masuk ke *booking_queue*

Mitigasi Masalah Intra-Service

- **Client App**
  a. Mitigasi dilakukan dengan memasang sebuah *message queue* di dalam Client App untuk menyimpan daftar email yang akan dikirim ke pengguna. *Message queue* ini diimplementasikan dengan RabbitMQ.
- **Ticket App**
  a. Proses pembuatan booking dapat dilakukan dengan transaksi sehingga apabila terdapat satu proses atau operasi yang gagal, operasi sebelumnya dapat di-*rollback.*
  b. PDF yang dibuat disimpan pada *cloud storage* yaitu AWS S3.

Mitigasi Masalah Inter-Service

a. Mitigasi dilakukan dengan memasang sebuah *queue* di dalam Payment App untuk menyimpan *request* ke Ticket App yang gagal. *Queue* ini diimplementasikan dengan Redis.
b. Mitigasi dilakukan dengan memasang sebuah *message queue* di antara Client App dan Ticket App untuk menyimpan *request* dari Ticket App ke Client App. *Message queue* ini diimplementasikan dengan RabbitMQ.

Analisis Pemilihan Solusi

1. **Client App**
   Implementasi Client App menggunakan REST API karena memiliki kompatibilitas yang lebih luas pada *web browser*. Hal tersebut membuat Client App menjadi lebih mudah diakses oleh pengguna. Selain itu, *database* yang digunakan adalah SQL karena terdapat relasi antar data pengguna dengan riwayat booking dan juga properti ACID yang dimiliki *database* SQL membuat integritas dan konsistensi data dapat terjamin.

2. **Ticket App**
   Implementasi Ticket App menggunakan REST API karena waktu implementasi yang lebih cepat karena kesederhanaan dalam penggunaan serta pengalaman. Pengembangan, pengujian, dan debugging dapat dilakukan lebih cepat karena REST menggunakan protokol HTTP yang umum dan mudah diakses. Selain itu, untuk *use case* saat ini tidak diperlukan *real-time data streaming* karena Client App tidak perlu memproses data dari Ticket App satu per satu sehingga solusi ini sudah cukup.

   Database utama pada Ticket App menggunakan database SQL karena antar entitas-nya memiliki hubungan yang jelas. Dengan SQL DB, transaksi ACID dapat dijaga sehingga integritas dan konsistensi data dijamin. SQL juga menyediakan bahasa query yang kuat dan fleksibel untuk melakukan operasi join serta filtering.

3. **Payment App**
   Sama seperti yang lainnya, implementasi Payment App menggunakan REST API karena lebih mudah diimplementasikan dan cocok untuk *use case* yang tertulis. Selain itu, REST API juga mempunyai kompatibilitas yang lebih luas. Implementasi Payment App menggunakan basis data NoSQL karena tidak ada relasi sama sekali pada skema Payment App.

4. **Spesifikasi bonus**
   Dari segi *cost*, penggunaan PostgreSQL tidak memerlukan biaya tambahan dalam hal infrastruktur tambahan karena sistem ini terintegrasi langsung dengan database yang sudah ada. Selain itu, penggunaan *message queue* seperti *rabbitmq* kurang cocok karena antrian ada pada setiap *seat* dan jika menggunakan *rabbitmq* maka perlu membuat *queue* untuk setiap *seat* yang dapat menambah penggunaan *resource*. Dengan PostgreSQL maka persistensi data dapat dijamin dan konsistensi data juga lebih terjamin

Alasan Pemilihan Solusi Untuk Mitigasi Intra-Service

- **Client App**
  a. Penggunaan *message queue* pada Client App untuk mengirim email membuat pengiriman email dapat dilakukan secara asinkron. Selain itu, *message queue* ini juga dapat memastikan seluruh email sampai ke pengguna karena jika terjadi

kegagalan, email yang gagal akan masuk kembali ke *message queue* dan dikirim lagi kemudian.

- **Ticket App**
    a. Transaksi memungkinkan untuk melakukan serangkaian operasi sebagai satu kesatuan yang konsisten. Jika pemanggilan ke Payment App gagal maka transaksi dapat di-*abort* dan tidak akan merubah data pada basis data karena belum *commit*. Solusi ini dipilih karena merupakan solusi yang paling *straightforward* dan dapat ditangani oleh DBMS dengan baik
    b. Penggunaan *cloud storage* dibandingkan *local storage* pada *server* Ticket App karena *cloud storage* menawarkan *availability* yang lebih tinggi dan replikasi otomatis data ke berbagai zona sehingga apabila Ticket App *down*, maka *user* tetap dapat mengakses PDF. Selain itu, menyimpan pada *cloud* akan mempermudah *scaling* jika terjadi pertumbuhan data secara signifikan.

Alasan Pemilihan Solusi untuk Mitigasi Inter-Service

a. Penggunaan *queue* untuk *job webhook* memungkinkan adanya pemanggilan asinkron untuk mengeksekusi *webhook* tersebut. Selain itu, *job queue* ini juga mempunyai fungsionalitas untuk melakukan *retry* (dengan *delay exponential backoff*) secara asinkron sehingga pemanggilan *webhook* yang gagal dapat diulang tanpa mengganggu *request* yang lain.
b. Penggunaan *message queue* memungkinkan komunikasi asinkron antara Client App dan Ticket App. Hal ini memisahkan proses pengiriman request dan pemrosesan dari ketersediaan Client App, sehingga mengurangi ketergantungan pada keadaan langsung dari aplikasi klien. Apabila Client App *down*, maka Ticket App dapat tetap mengirim *request* ke *message queue* sehingga dapat diproses kembali jika Client App berjalan lagi.

Analisis Pemilihan Stack Teknologi

Berikut merupakan analisis pemilihan *tech stack* yang digunakan.

Stack Teknologi	Letak	Alasan
Bahasa Pemrograman		
TypeScript	Ticket App, Payment App	TypeScript adalah bahasa yang sangat populer dalam pemrograman *web*. TypeScript adalah bahasa yang *strongly typed* dan didukung oleh banyak *library* sehingga mempermudah *development*. Penulis juga sudah mempunyai pengalaman dalam menggunakan bahasa tersebut.
Golang	Client App	Golang adalah bahasa tersebut cukup populer dalam pemrograman *web*. Selain itu, Golang adalah bahasa yang *strongly typed*, *garbage collected*, dan mempunyai dukungan untuk pemrograman konkuren secara eksplisit sehingga mempermudah *development*. Penulis juga sudah mempunyai pengalaman dalam menggunakan bahasa tersebut.
Framework		
Node.js dan Express	Ticket App, Payment App	*Framework* ini sudah sangat populer untuk pemrograman *backend* (terutama untuk HTTP atau REST API) dan didukung oleh banyak *library* sehingga mempermudah *development*. Penulis juga sudah mempunyai pengalaman dalam menggunakan bahasa tersebut.
Chi	Client App	Kakas ini relatif lebih ringan untuk dijalankan dan dioperasikan dibandingkan kakas Golang lainnya (contoh: Gin). Selain itu, penulis juga sudah mempunyai pengalaman dalam menggunakan kakas tersebut.
***Database Management System* (DBMS)**		
PostgreSQL	Client App, Ticket App	PostgreSQL digunakan pada *service* Client App dan Ticket App. Kedua *service* tersebut menggunakan PostgreSQL yang merupakan DBMS relasional karena skema data pada kedua *service* tersebut bersifat relasional. Selain itu, jika dibandingkan dengan DBMS relasional lain seperti MySQL, PostgreSQL memiliki efisiensi yang lebih tinggi dalam pemrosesan data besar dan juga operasi *read-write*. Ticket App memiliki frekuensi tinggi pada *read* dan juga *write* sehingga PostgreSQL cocok untuk *use case* ini.

Cassandra	Payment App	Cassandra digunakan pada *service* Payment App. *Service* tersebut menggunakan Cassandra yang merupakan basis data NoSQL berbasis *key-value store* karena tidak ada relasi sama sekali untuk skema Payment App, hanya ada pasangan *key-value* saja.
Database Driver/Object Relational Mapper (ORM)		
Drizzle	Ticket App	Drizzle digunakan untuk ORM pada Ticket App karena Drizzle memiliki API yang mirip seperti sedang menulis query sehingga terdapat kesan familiaritas untuk *developer*. Selain itu, Drizzle memiliki kecepatan iterasi yang lebih cepat dibandingkan Prisma. Perubahan pada *schema* langsung ter-refleksi pada API client
GORM	Client App	GORM merupakan ORM yang paling populer digunakan untuk bahasa Go. GORM menyediakan API yang mudah dipahami dan digunakan dan memiliki struktur yang intuitif.
DataStax Cassandra Driver	Payment App	DataStax Cassandra Driver merupakan *driver* basis data yang didukung secara resmi oleh Apache Cassandra untuk bahasa pemrograman ber-*framework* Node.js. Selain itu, driver ini juga menyediakan *object mapper* (seperti pada ORM namun untuk basis data NoSQL) yang cukup intuitif.
Cloud Storage		
AWS S3	Ticket App	AWS S3 dipilih karena sudah memiliki reputasi yang baik sebagai penyimpanan data seperti *file*. Selain itu, AWS S3 dipilih karena penulis sudah berpengalaman dalam menggunakan AWS S3. Selain itu, AWS S3 juga memiliki *free plan* yang dapat digunakan.
Queue		
RabbitMQ	Client App, Ticket App	RabbitMQ dipilih dibandingkan Kafka karena sifatnya yang dapat berupa *at least once* dan *customer* (Client App) dapat mengatur kapan *message* dihapus dari *queue*. Hal ini dapat memitigasi kegagalan pemrosesan ke database ataupun proses pengiriman email. Jika salah satu dari dua hal tersebut gagal, maka *message* masih tetap ada di *queue* sehingga dapat diproses ulang dan memastikan *user* pasti mendapatkan *email* PDF-nya serta *booking history* terbaru. Rabbit MQ juga memiliki Manager UI yang dapat digunakan untuk memvisualisasikan semua *queue* dan *exchange* yang ada.
Bull (Redis)	Payment App	Bull adalah *package* (tidak seperti RabbitMQ, Kafka, dll.) yang didesain khusus untuk *framework* Node.js yang menyediakan

		fungsionalitas *queue* yang berjalan di atas Redis. Penulis memilih Bull dibandingkan *stack queue* lainnya dengan pertimbangan *use case* yang dibutuhkan cukup sederhana, yaitu sebagai *queue* yang menyimpan data dan men-*schedule request webhook* untuk menjaga reliabilitas *service* Payment App.