MyTravel.com – Blockchain based ticket management system

Assignment Report

Leveraging Hyperledger Fabric

(Group-3) Ansh Makwe, Divyansh Chaurasia

(241110010, 241110022)

Date of Submission:

18th April 2025

0.1 Acknowledgment

We express our gratitude towards Professor Dr. Angshuman Karmakar Sir for giving us this opportunity to work on this assignment. Also, we express our gratitude towards Er. Sumit Lahiri Sir for guiding us throughout the development of this assignment and product.

Contents

Chapter 1

Introduction

1.1 Context and Motivation

With the rapid digitalization of services, the travel industry has been focusing on secure and transparent ticketing systems that handle reservations, payments, cancellations, and refunds. Traditional centralized ticket-booking platforms (like popular aggregator websites) typically depend on a single database, which can lead to trust issues, potential single points of failure, and difficulties with audit transparency.

Blockchain technology, and in particular Hyperledger Fabric, offers a decentralized approach that maintains a shared, tamper-proof ledger across multiple participants. This ensures:

- Transparency and auditability of all transactions.
- Strong security measures (digital identities, cryptographic operations).
- Near-real-time updates for seat availability and finality of ticket booking.
- Automated refund and dynamic pricing logic enforced by chaincode (smart contracts).

Hence, the main motivation behind this assignment is to develop a **Blockchain-based Ticket Management System** that ensures both functional and non-functional requirements are met for a modern, secure, and transparent travel booking platform.

1.2 Assignment Goals and Requirements

The key objectives of this assignment can be summarized as follows:

1. Develop a robust chaincode (Hyperledger Fabric smart contract) that manages user and provider registration, travel option listings, seat availability, ticket booking, refunds, dynamic pricing, and cancellations.

2. Provide a minimal or slightly enhanced web-based UI so that customers (passengers) and providers (transport companies) can easily interact with the blockchain network.

3. Ensure data privacy so that customers cannot see each other's bookings and providers cannot see sensitive customer information (phone numbers, etc.).

4. Demonstrate *block-based finality*, so that each booking is only considered "confirmed" after the transaction is recorded in the ledger and at least two additional blocks have been appended (simulating multi-block confirmation).

5. Conform to non-functional requirements, including preventing overbooking and double-booking, robust availability for 1000+ daily users, and avoiding concurrency issues.

1.3 Document Overview

This report is structured into the following chapters:

- **Chapter 1: Introduction** – Provides motivation, overview, and objectives.
- **Chapter 2: Hyperledger Fabric Network Setup** – Explains the single organization network structure (one Org, two peers, one orderer).
- **Chapter 3: System Architecture** – Describes the overall system, including the chaincode, backend server, and front-end UI.
- **Chapter 4: Requirements Mapping** – Explains how the functional and non-functional requirements, along with minimum deliverables and bonus points, are satisfied.
- **Chapter 5: Chaincode and Business Logic Implementation** – Provides a deep dive into chaincode functions and the logic for dynamic pricing, refunds, seat selection, etc.
- **Chapter 6: Frontend and Backend Implementation** – Details the React front-end, Express-based backend, and how they integrate with the Fabric network.
- **Chapter 7: Deployment Instructions** Step-by-step guide on how to run this product.
- **Chapter 8: Testing, Demonstration, and Results** – Shows testing procedures, how tickets are booked, cancelled, confirmed, etc.
- **Chapter 9: Limitations and Future Enhancements** – Discusses system constraints and potential improvements.

- **Chapter 10: Conclusion and References**
- **Chapter 11: Responsibilities Distribution**

Chapter 2

Hyperledger Fabric Network Setup

2.1 Overview of the Chosen Network Topology

Hyperledger Fabric is a permissioned blockchain framework allowing modular networks with configurable components. For this assignment, we set up:

- **One organization (Org1)** that hosts:
  1. **Peer0** – The anchor peer for the organization, receives all ledger updates and endorses transactions.
  2. **Peer1** – Another endorsing peer that also holds the ledger.
- **One ordering node** – Handles transaction ordering into blocks.
- **Certificate Authority** (CA) – Issues and manages digital identities (X.509 certificates) for customers, providers, and admin.

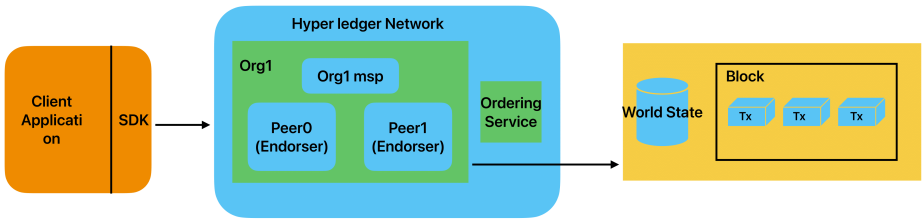

Figure 2.1: My Tickets Architecture

2.2 Key Configuration Steps

2.2.1 Certificate Authority

A Fabric CA server is used to:

- Enroll an organization admin identity (admin of Org1).
- Register and enroll new identities (new customers, new providers).

2.2.2 Peer and Orderer Setup

1. **Peer0.org1.example.com** – Anchor peer for Org1, endorses transactions and maintains the ledger.
2. **Peer1.org1.example.com** – Additional endorsing peer for high availability.
3. **Orderer.example.com** – Processes transactions from all peers, ensures correct block ordering before distribution to peers.

2.2.3 Channel Creation and Chaincode Deployment

A single channel named `mychannel` is created, and the chaincode named `stake` is installed and instantiated on Peer0 and Peer1 of Org1. This chaincode encapsulates all logic for:

- Customer Registration
- Provider registration
- Travel option creation
- Ticket booking, cancellation, rescheduling
- Payment simulation and dynamic pricing, etc.

Once deployed, the chaincode is ready to accept transactions from the client application(s).

Chapter 3

System Architecture

3.1 High-Level Overview

The platform is composed of:

1. **Front-end (React.js)** – An enhanced UI for both *Customers* and *Providers*.
2. **Back-end (Node.js / Express)** – Provides RESTful endpoints that proxy requests to the Hyperledger Fabric network. Manages identity wallets, chaincode invocations, and query evaluations.
3. **Chaincode (stake.js)** – The smart contract on Hyperledger Fabric that enforces business rules for ticket booking, refunds, seat availability, and more.
4. **Wallet & Identity Management** – Maintains user credentials for each new identity. Each user has a unique enrollment certificate stored in a local filesystem wallet.
5. **Blockchain Ledger** – Maintained by the peers, storing all transactions. Only *confirmed* transactions finalize seat allocations and payments.

3.2 High Level Data Flow Summary of Crucial Features

3.2.1 User Registration Flow

- A new user (customer or provider) signs up on the front-end.
- The Node.js server calls Fabric CA to register and enroll the new identity, generating the certificate and key.
- Chaincode `registerCustomerDynamic` or `registerProviderDynamic` is invoked to create an on-ledger record for that user.

3.2.2 Booking Flow

1. A customer searches for travel options in the front-end, specifying `source`, `destination`, `date`, etc.
2. The server queries the chaincode function `listTravelOptionsSorted`, filters by availability, price, or rating.
3. The customer selects a seat from the available seats.
4. The chaincode function `bookTicket` verifies seat availability, subtracts balance from the user, credits the provider, and marks the seat as "PENDING_CONFIRMATION", which gets confirmed after two block confirmations.

3.2.3 Block Confirmation Flow

- The *autoConfirmScheduler*, a scheduled process checks if at least two more blocks have been appended since the booking.
- Once satisfied, it calls `confirmTicket`, setting the ticket status to “CONFIRMED” in the ledger.
- Also, checks if there is a travel option that departs within two minutes. If this travel option has pending tickets and there are vacant seats, they are assigned confirmed status, and if tickets cannot be confirmed, the respective tickets are cancelled and refunds are processed according to business rules, which will be discussed later.

3.2.4 Cancellation and Refund Flow

- A user or provider calls `cancelTicket` or `cancelTravelListing`.
- For `cancelTicket` The chaincode determines refund amounts based on how close the departure date is and returns funds to the customer’s balance, subtracting from the provider’s balance.
- For `cancelTravelListing` full refund is processed.

3.2.5 Add Travel Option

- A provider calls `addTravelOption`
- The provider mentions the source, destination, departure date, departure time, transport mode, seat capacity, and base price. When this call goes to the chaincode a ledger entry is made for this new travel option.

Chapter 4

Requirements Mapping

4.1 Functional Requirements

The assignment listed key user-centric requirements, including:

1. **Ticket Booking (Source to Destination)**
   Implemented in `bookTicket` with seat-level selection.

2. **Cancel an Existing Ticket**
   Handled via `cancelTicket`, with partial refunds if close to departure.

3. **Change Date of Travel (Reschedule)**
   `rescheduleTicket` chaincode function cancels the old ticket and books a new one, applying a penalty.

4. **Show 3+ Options**
   The chaincode queries all available providers for bus, train, plane, or user-defined modes and displays them if a count of three is avaialble. The UI displays them sorted by *price*, *provider rating*, etc.

5. **Provider Onboarding and Deletion**
   New providers are registered and can be removed. If a provider is deleted, all associated travel listings are canceled, with refunds.

6. **Sorting by Provider Rating, Mode, Price**
   `listTravelOptionsSorted` returns a curated list that can be sorted or filtered in multiple ways.

7. **Dynamic Pricing**
   Implemented in `bookTicket` with occupancy-based formula. This will be explained thoroughly in the next chapter.

8. **Booking Details Verification**
   The ledger is the source of truth. The ticket ID encodes booking details, verified on the front-end with QR code scanning (`TicketVerification.jsx`).

9. **Update Passenger and Provider Details**
   `updateCustomerDetails` and `updateProviderDetails` allow changing name, contact, or marking the profile anonymous (only for customer).

4.2 Non-Functional Requirements

4.2.1 Consistency, Robustness, Transparency

- Consistency: Endorsement policies and chaincode logic ensure each transaction is validated prior to commit.
- Robustness and Transparency: Full transaction history is on the immutable ledger; tickets are confirmed only after ledger finality (two subsequent blocks).
- Also, the dynamic pricing details are displayed in the ticket for transparency.

4.2.2 Scalability and Performance

- Our product is indeed scalable for 1000+ daily active users with 2500+ bookings.
- Detailed Testing and Analysis will be discussed in further slides.

4.2.3 Double Booking and Overbooking Prevention

Double Booking Prevention

- **Per-seat existence check:**
  - On every `bookTicket(...)` the chaincode iterates existing tickets for that `travelOptionId` via `getStateByPartialCompositeKey(ticketPrefix, [travelOptionId])`.
  - It throws an error if any non-cancelled ticket already has the requested `seatNumber`.
- **In-memory `bookedSeats` guard:**
  - After loading `travelOptionData`, we maintain a `bookedSeats` array.
  - Before creating a new ticket we check `if (bookedSeats.includes(requestedSeatNumber))throw ....`
- **MVCC concurrency control:**

  - Fabric's MVCC ensures that if two transactions try to book the same seat simultaneously, only one can commit.
  - The loser sees a version conflict and must retry, preventing duplicate writes.

- **Atomic composite-key uniqueness:**
  - Each ticket's composite key embeds `travelOptionId`, `customerId`, and a timestamp.
  - This guarantees each write is distinct and doesn't overwrite another.

4.2.4 Over Booking Prevention

- **`availableSeats` counter:**
  - At booking start we assert `if (availableSeats <= 0)throw ....`
  - On success we decrement `availableSeats` by 1 *within the same transaction.*
- **Capacity never goes negative:**
  - Because the check and decrement occur atomically, `availableSeats` can never drop below zero.
- **Auto-confirmation with capacity enforcement:**
  - As departure approaches, a scheduler invokes `autoConfirmTicketsForTravelOption` .
  - It confirms up to the remaining `availableSeats`, and *cancels (and refunds)* any extra pending tickets.

4.2.5 Show 3+ Options

- The chaincode queries all available providers for bus, train, plane, or user-defined modes and displays them (if a count of three is available). The UI has an option to display them sorted by *price*, *provider rating*, etc.
- However, in our Business logic we have not forced a provider to add three plus options.

4.3 Minimum Deliverables

How chaincode handles the requirements-

1. **User Creation** – `registerCustomerDynamic`

2. **Provider Creation** – `registerProviderDynamic`
3. **Invisible Bookings** – Each user sees only their own bookings.
4. **Ticket Creation** – `bookTicket` with seat selection.
5. **Payment + Ledger Update** – Payment simulation is done via updating `balance` fields in chaincode, and tickets are confirmed after two block confirmations.
6. **Ticket Deletion** – `cancelTicket`.
7. **List Self Tickets** – We store tickets of a user within the user data, the user can view them in the My Tickets tab from the frontend.
8. **List Transport Options** – `listTravelOptionsSorted` or `listTravelOptions`.
9. **Provider Info Privacy** – No personal information is accessible by any other entity.
10. **Customer Info Privacy** – `Name` and `contact` can be hidden if anonymous. No personal information is accessible by any other entity
11. **Minimal Web UI** – Provided via React + Express endpoints.
12. **IBM Hyperledger Fabric** – Our chaincode is built for Fabric v2.5 test network.
13. **Authenticating service providers** – We use a dummy verify GSTIN Number button in provider registration.
14. **Tackling Bots** – We use a dummy maths captcha for addressing this.
15. **Verifying Booking Details** – We give an option to download the QR code of the ticket to the user, that can be uploaded on our web DApp to verify booking details.

4.4 Extra (Bonus) Pointers

- **Filtering by Price, Avalaibilty and Provider** – Achieved by optional parameters in `listTravelOptionsSorted`.
- **System Deployed Online** – Deployed on AWS. Public link
- **Extra Useful Features** –

1. **Ease of Ticket Verification -** The user can download the QR code of a ticket that can be uploaded to our website to confirm the validity of that ticket for a particular seat number (partly inspired from metro).

2. **Seat Selection for Customer -** For very travel option we consider a grid of 10 columns, and we display a grid of seats and the user can select a particular seat to book a ticket for (if available). If that particular seat is vacant, then it is displayed in green colour, and if it is not, it is displayed in red colour, and the customer cannot select it.
3. **Rating Option -** A customer can rate a provider, but the catch is that he/she cannot rate a provider before the journey that has commenced.
4. **Automatic Ticket Confirmation -** Apart from handling double booking and overbooking, we smartly handle the scenario where a pending ticket is confirmed if someone with a confirmed ticket in that travel option cancels their ticket. This is done by the auto confirm scheduler, which works on iterating through the pending tickets of a travel option ID.
5. **Persistence -** We provide two scripts to start the system, `first.sh` and `second.sh`. `first.sh` will be used for deploying chaincode initially, and if there is some update in chaincode, then to redploy it `second.sh` will be used.

Aspect	**first.sh**	**second.sh**
Network teardown	Calls `network.sh down` to destroy containers, volumes, and artifacts.	Omits teardown; leaves network running.
Network bring-up	Calls `network.sh up -s couchdb createChannel -ca -c mychannel`.	Skips network up/channel creation.
Chaincode install & approve	Uses `deployCC ... -cci initLedger` to package, install, commit *and* initialize ledger.	Uses `deployCC` without `-cci initLedger`; does not re-initialize.
Wallet & ledger state	Removes `wallets` and starts with empty world-state.	Preserves wallets and ledger state for ongoing data access.
Version bump	Typically `-ccv 1` for initial release.	Increments `-ccv` (e.g. from 1 to 2) to trigger lifecycle upgrade.
Downtime impact	Full network restart ⇒ longer downtime.	Only chaincode reload ⇒ minimal downtime.
Use case	Initial blockchain setup and data seeding.	Rolling out a new chaincode version on a live network.

Key points:

- **Data preservation:** Upgrades preserve existing identities and on-chain data; first-time wipes state.

  - **Efficiency:** Reduces deployment time after chaincode update.

To ensure that ledger data and user identities survive chaincode redeployments and container restarts, we employed the following persistence strategies:

- **Docker Volumes:** Each peer and the orderer use named Docker volumes to store their ledger (blocks and world state). Example:

```
volumes:
  peer0.org1.example.com:
  peer1.org1.example.com:
  orderer.example.com:
```

  These volumes are not removed during an upgrade, so the blockchain history remains intact.

- **CouchDB Volumes:** CouchDB instances backing each peer's state database also use named volumes (`couchdb0`, `couchdb1`), preserving all indexed JSON documents across restarts.

Chapter 5

Chaincode and Business Logic Implementation

5.1 Chaincode Structure (stake.js)

Our chaincode is implemented in JavaScript (Node.js) and extends the `Contract` class from `fabric-contract-api`. Every record is stored as JSON and indexed by composite keys. The following sections provide an in-depth look at each functional block of the chaincode.

5.1.1 Entity Data Model

We use composite keys to differentiate records:

- `"customer"`: Each customer record is indexed with a composite key of the format `customercustomerID` .
- `"provider"`: Provider records are stored using keys like `providerproviderID` .
- `"travelOption"`: Travel options are indexed by keys `travelOptiontravelOptionID` . These objects include source, destination, departure details, seat capacities, available seats, and base price.
- `"ticket"`: Tickets use composite keys in the form `tickettravelOptionID, customerID, timestamp` . Tickets also store booking timestamps, seat numbers, pricing breakdown and status.

An example customer record:

```
{
  "customerId": "uniqueClientID",
  "wallet": "sameAsCustomerID",
  "name": "Alice Smith",
```

```
  "contact": "alice@example.com",
  "balance": 1000,
  "bookings": ["ticketCompositeKey1", "ticketCompositeKey2"]
}
```

5.1.2 Core Functions Overview

The chaincode exposes functions covering the entire booking lifecycle along with account and travel option management:

- **Initialization and Registration:**
  - `initLedger`: Optionally pre-populates the ledger with dummy data.
  - `registerCustomerDynamic`: Registers a new customer using their client ID, saving basic details and initializing a wallet with a default balance.
  - `registerProviderDynamic`: Registers a new service provider with details such as name, contact, service type, rating and an initial balance.
- **Profile Updates and Wallet Management:**
  - `updateCustomerDetails` and `updateProviderDetails`: Update the profiles for customers and providers, with an option to mark the profile as anonymous (null contact).
  - `depositFunds`: Deposits funds into a customer's wallet after validating the deposit amount.
- **Travel Option Management:**
  - `addTravelOption`: Allows a provider to add a new travel listing. It checks for duplicate active listings and deducts a fixed fee from the provider's balance.
  - `deleteTravelOption`: Deletes a travel option if there are no active bookings.
  - `cancelTravelListing`: Permits providers to cancel an entire travel option. This cancels all active tickets under that option and processes refunds accordingly.
  - `listTravelOptions` and `listTravelOptionsSorted`: Enable querying of travel options by source, destination, and additional filters like price range, provider, and seat availability. When sorting by rating, the provider's details are dynamically attached.
- **Booking and Payment Mechanism:**

  - `bookTicket`: Core functions for ticket booking. They validate seat availability (including through both direct state checks and the bookedSeats array), calculate dynamic pricing based on occupancy, deduct a fixed platform fee of 5 currency units, update customer and provider balances, and create the ticket record.
  - **Dynamic Pricing:** The dynamic price is computed as:

$$\text{dynamicPrice} = \text{basePrice} \times \left(1 + \frac{\text{bookedCount}}{\text{seatCapacity}} \times 0.5\right)$$

    capped to a maximum of $1.5 \times$ basePrice. The final cost charged is:

$$\text{finalCost} = \text{dynamicPrice} + 5$$

- **Ticket Confirmation and Block-Based Logic:**
  - `confirmTicket`: Increments a ticket's confirmation count. Once the count reaches a threshold (set here as 1 or 2 blocks), the ticket status is updated to `CONFIRMED`.
  - `autoConfirmTicketsForTravelOption`: Automatically confirms or cancels pending tickets based on available seats within 2 hours of departure. Extra tickets beyond the available seats are cancelled with full refunds.
- **Cancellation, Rescheduling, and Refunds:**
  - `cancelTicket`: Cancels an active ticket (with status `PENDING_CONFIRMATION` or `CONFIRMED`). Refunds are processed based on the time difference from departure:
    * If the cancellation is made at least 48 hours before departure, a full refund is issued.
    * Between 24 and 48 hours, an 80% refund is applied.
    * Within 24 hours, no refund is given.
  - `rescheduleTicket`: Cancels the old ticket (processing the corresponding refund) and books a new ticket with a new travel option. The new booking uses updated dynamic pricing.
- **Account Deletion:**
  - `deleteCustomer`: Cancels all active tickets for a customer, processes refunds, and deletes the customer record.
  - `deleteProvider`: Cancels all travel options registered by the provider, refunds active bookings, and then deletes the provider record.

- **Additional Query Functions:**
  - `getCustomerTickets`: Retrieves all tickets booked by the customer.
  - `getCustomerDetails` and `getProviderDetails`: Return account details.
  - `getTicketDetails`: Fetches details for a specific ticket.
  - `getProviderTravelOptions`: Returns all travel options for the logged-in provider.
  - `getAllTravelOptions`: Returns all travel options on the ledger.
  - `rateProvider`: Allows a customer to rate the provider after travel, updating the provider's average rating.

5.1.3 Block Confirmation and Auto-Confirmation

Tickets are initially marked as `PENDING_CONFIRMATION`. The chaincode uses:

- `confirmTicket`: Manually increments ticket confirmations until a defined threshold is met.
- `autoConfirmTicketsForTravelOption`: Automates confirmation (or cancellation with refunds) based on:
  - The number of pending tickets versus the available seats.
  - A simulated block confirmation mechanism (e.g., checking block timestamps) to determine if the ticket should be auto-confirmed.

5.1.4 Rescheduling Logic

The `rescheduleTicket` function ensures that:

- The current (old) ticket is cancelled and the corresponding seat is released.
- A refund is computed based on the time difference from departure.
- A new ticket is booked for the new travel option with updated pricing.
- The new ticket maintains a reference to the old ticket.

5.1.5 What if Departure Date is Already Passed?

- **Customers' Perspective:**
  1. `ListTravelOptions` : will jot show tickets with departure time $<$ current time
  2. Customer will also not able to book a ticket if departure time $<$ current time
- **Provioders' Perspective:**

1. `AddTravelOptions:` Customer will not be able to add travel option with departure time < current time
2. `CancelTravelListing:` If a provider cancels a travel listing after departure date, no refund will be processed
3. `DeleteProvider:` If a provider deletes its profile, no refund will be processed for the listing with departure time < current time

Chapter 6

Frontend and Backend Implementation

6.1 Backend (Node/Express) Layer

The backend is built using Node.js and Express, and it exposes a RESTful API to facilitate all blockchain interactions. It serves as the interface between client-side requests and the chaincode functions, as well as handling user authentication via Firebase. The backend components include identity enrollment, registration, ticket booking and management, query operations, and automated block and time–based scheduling of ticket confirmations.

6.1.1 Key SDK Scripts and Chaincode Interaction

The backend integrates a wide range of SDK scripts that interact with the chaincode on the Hyperledger Fabric network. Key scripts include:

- **Identity Enrollment and Registration**
  - `enrollAdmin.js` and – Enroll the admin identities for Org1 and store it in the wallet.
  - `registerUser.js` – Register test users (appUser for Org1).
  - `serverRegisterCustomerDynamic.js` and `serverRegisterProviderDynamic.js` – Dynamic registration endpoints that call chaincode functions (`registerCustomer` and `registerProvider`) for new customers and providers.
- **Ticket Booking and Management**
  - `serverBookTicket.js` – Invokes the `bookTicket` chaincode function to book tickets. It also integrates with a pending ticket registry and records the booking block number for later confirmation.

  - `serverCancelTicket.js` – Cancels an active ticket and handles refunds according to the cancellation policy.
  - `serverRescheduleTicket.js` – Cancels an old ticket (with partial or full refund based on time from departure) and books a new ticket for a different travel option.
  - `serverConfirmTicket.js` – Submits a transaction to confirm a ticket after simulated block–based confirmations.

- **Travel Option Queries and Listings**
  - `serverListTravelOptions.js` – Retrieves travel options based on source and destination.
  - `serverListTravelOptionsSorted.js` (and variant `serverListTravelOptionsSorted1.js`) – Returns sorted and filtered travel options based on criteria such as price range, provider, departure date, and seat availability.
  - `serverGetAllTravelOptions.js` – Retrieves all travel options stored on the ledger.

- **Account and Profile Management**
  - `serverUpdateCustomerDetails.js` – Updates a customer profile with new name, contact details, and an option to mark the profile as anonymous.
  - `serverUpdateProviderDetails.js` – Updates provider profile details.
  - `serverDeleteCustomer.js` and `serverDeleteProvider.js` – Remove customer or provider accounts by canceling all active bookings and processing refunds.

- **Rating and Provider Details**
  - `serverRateProvider.js` – Allows customers to rate a provider after travel. It validates the travel date and then updates the provider's average rating.
  - `getCustomerDetails.js` and `getProviderDetails.js` – Query chaincode for detailed account information.
  - `serverGetCustomerTickets.js` and `serverGetTicketDetails.js` – Retrieve the list of tickets for a customer or details of a specific ticket.
  - `serverGetProviderTravelOptions.js` – Retrieves all travel options created by a provider.

- **Block Confirmation and Auto-Confirmation**

- `serverAutoConfirmTickets.js` – Automatically confirms tickets for a travel option that is within 2 hours of departure. This uses both block–based and time–based evaluation.
- `autoConfirmScheduler.js` and `blockEventListener.js` – Implement a scheduler that periodically checks the blockchain (using QSCC) and triggers ticket confirmations based on block height and departure time. A pending ticket registry (via `pendingTicketRegistry.js`) is used to track bookings waiting for confirmation.

6.1.2 Express Server and REST Endpoints

The `server.js` file orchestrates the Express application, defines all the REST endpoints, and configures middleware for JSON parsing and Cross-Origin Resource Sharing (CORS). Key endpoints include:

- `/enrollall` (GET): Enrolls admin identities and registers test users.
- `/registercustomerdynamic` (POST): Registers a new customer by invoking `serverRegisterCustomerDynamic.js`.
- `/registerproviderdynamic` (POST): Registers a new provider by invoking `serverRegisterProviderDynamic.js`.
- `/bookticket` (POST): Books a ticket by calling `serverBookTicket.js` and then records the booking block in a pending ticket registry.
- `/cancelticket` (POST): Cancels a ticket and processes the appropriate refund.
- `/rescheduleticket` (POST): Reschedules a ticket by cancelling the existing one and booking a new ticket.
- `/listtraveloptions` (GET): Lists travel options based on source and destination.
- `/listtraveloptionssorted` (GET): Lists travel options with sorting and filtering criteria.
- `/confirmticket` (POST): Confirms a ticket when enough block confirmations have been achieved.
- `/depositfunds` (POST): Deposits funds into a customer wallet.
- `/getcustomerdetails` (GET): Retrieves details for a customer identity.
- `/getcustomertickets` (GET): Retrieves all tickets for the logged–in customer.
- Provider-specific endpoints such as:

  - `/getprovidertraveloptions` (GET)
  - `/getproviderdetails` (GET)
  - `/addtraveloption` (POST)
  - `/deletetraveloption` (POST)
  - `/canceltravellisting` (POST)

- `/rateprovider` (POST): Submits a provider rating.
- Account deletion endpoints: `/deletecustomer` and `/deleteprovider`.

6.1.3 Authentication and Additional Services

Authentication is handled separately via Firebase in the `serverAuth.js` route. This module:

- Provides signup and login endpoints for both customers and providers.
- Manages password hashing (via bcrypt) and stores user profiles in Firestore.

The backend also launches an auto-confirm scheduler (via `autoConfirmScheduler.js`) and a block event listener (via `blockEventListener.js`) to continuously monitor the chain for pending confirmations. A persistent pending ticket registry (`pendingTicketRegistry.js`) is used to track the ticket booking state across process restarts.

6.2 Frontend (React) Components Overview

The UI is built entirely in React and uses React Router for navigation between screens. The application also integrates with Firebase for authentication and a custom backend via API calls.

6.2.1 Role and Authentication Screens

- **RoleSelection.jsx**
  Displays the initial interface allowing visitors to choose a role as `User` or `Provider`. It also triggers system enrollment (e.g., network initialization).

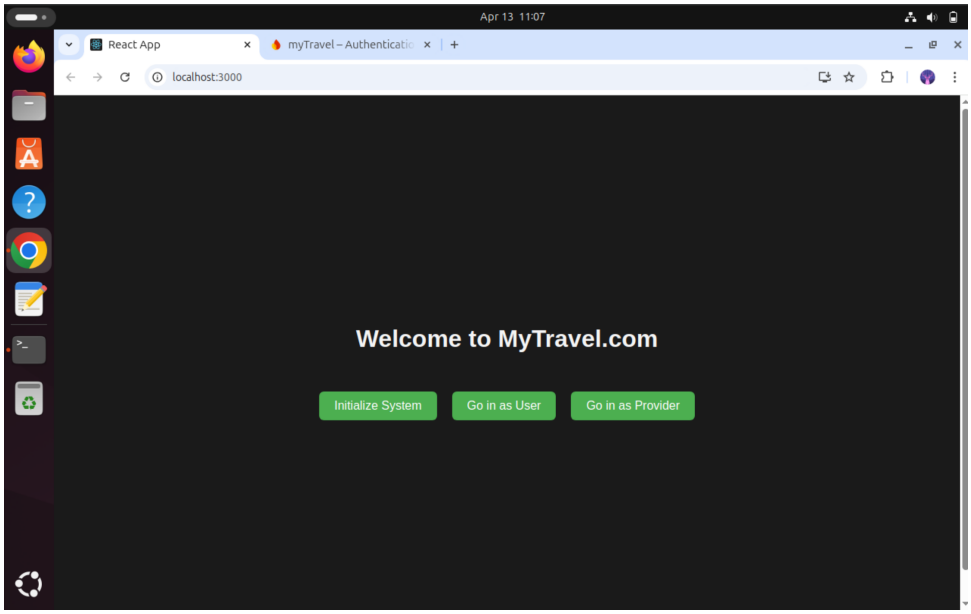

Figure 6.1: Home Screen

- **Login.jsx**
  Provides login forms where users are authenticated through Firebase. On successful sign in, it calls methods to update user details and navigates to role-specific home screens.

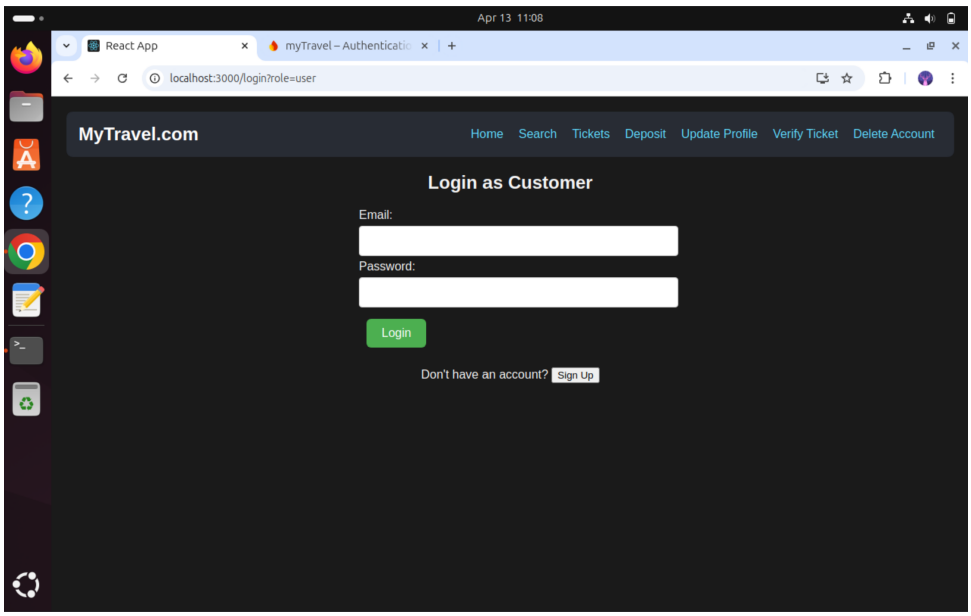

Figure 6.2: Login Screen

- **SignUp.jsx**
  Handles registration for both customers and providers. It includes additional provider validations (e.g., GSTIN verification) and a simple captcha before calling registration endpoints via the API.

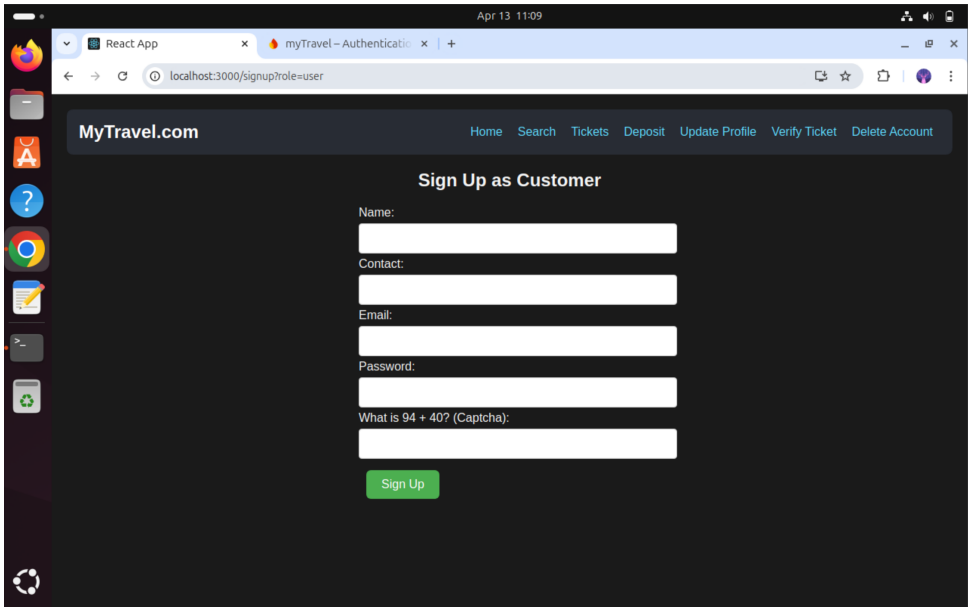

Figure 6.3: Sign up Screen

6.2.2 Home and Dashboard Screens

- **UserHome.jsx**
  Displays the customer dashboard with details such as `balance` and name. Provides navigation for travel searches, depositing funds, viewing tickets, and updating profile details.

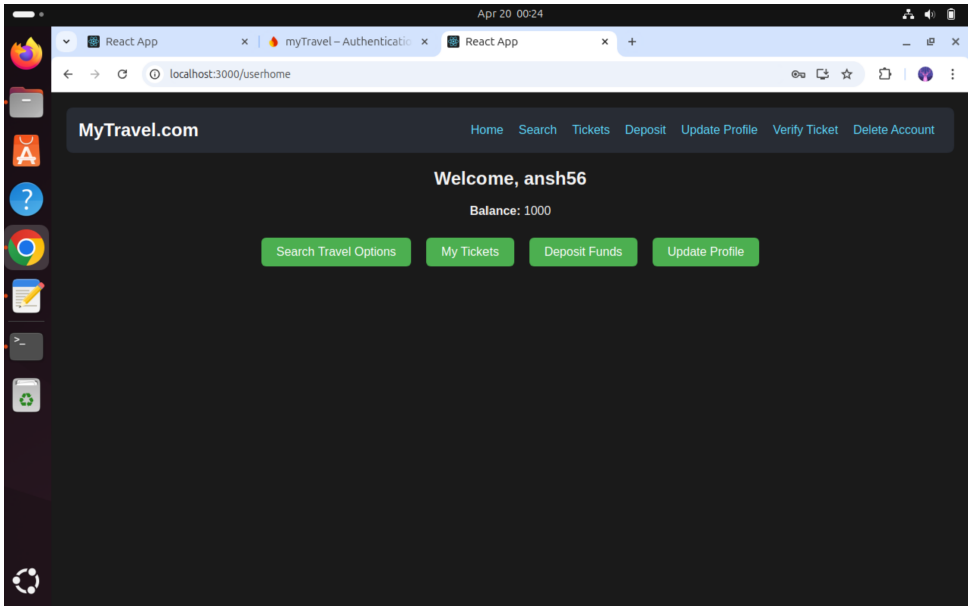

Figure 6.4: User Home Screen

- **ProviderHome.jsx**
  Displays the provider's dashboard listing key details like provider name, balance,

rating, and total number of ratings. It includes links to add new travel options, view existing options, or update the provider profile.

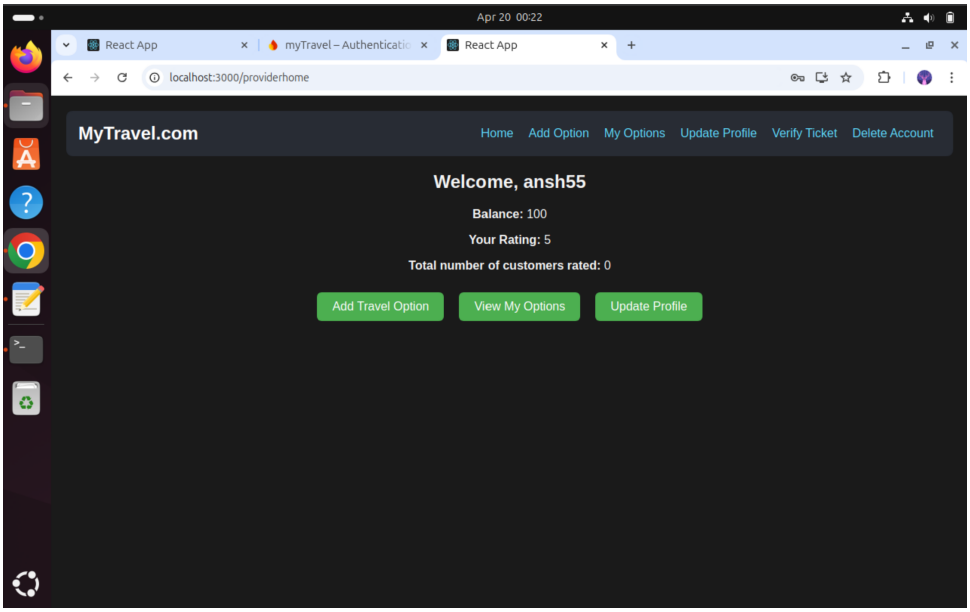

Figure 6.5: Provider Home Screen

6.2.3 Travel Option and Booking Screens

- **SearchTravel.jsx**
  Allows users to search for travel options by specifying parameters like source, destination, departure date, price range, transport mode, and optionally filtering by service provider. The screen displays the available options and routes the user to book a ticket.

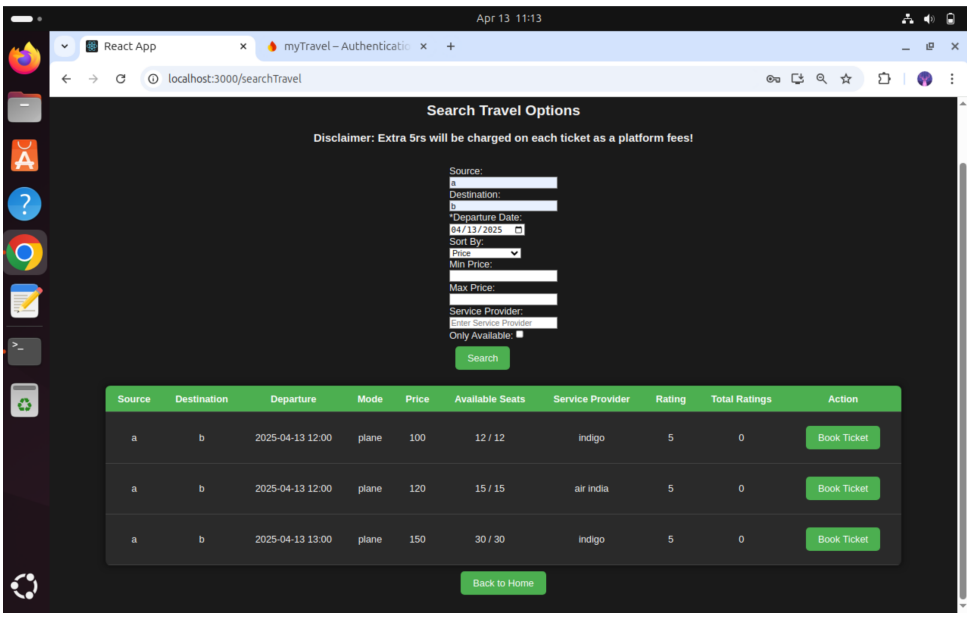

Figure 6.6: Search Travel Options Screen

- **SeatSelection.jsx**
  Provides a grid-based seat selection interface for a chosen travel option. Booked seats are visually marked as unavailable, and users can select a seat to confirm their booking. Booking is finalized via an API call.

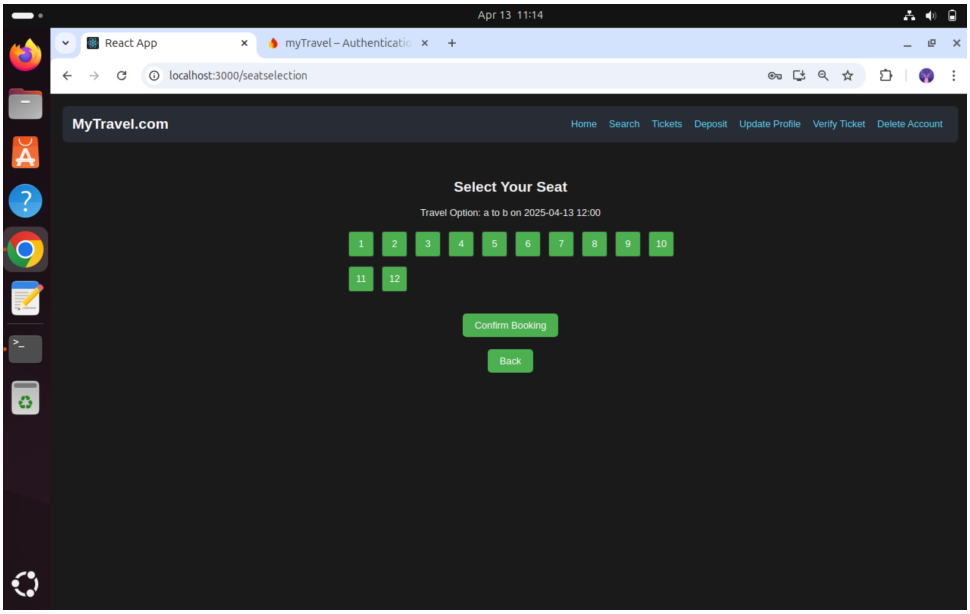

Figure 6.7: Seat Selection Screen

- **MyTickets.jsx**
  Lists all booked tickets for the customer, displaying status such as `PENDING_CONFIRMATION`, `CONFIRMED`, or `CANCELLED`. It also provides options to cancel, reschedule, print tickets, or rate a provider.

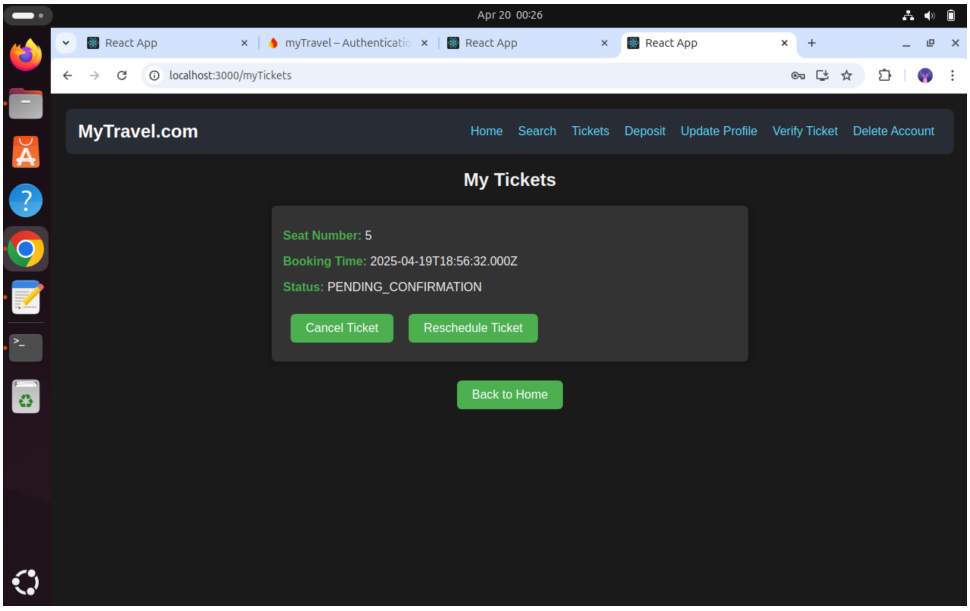

Figure 6.8: My Tickets Screen

- **DepositFunds.jsx**
  Allows users to deposit funds into their accounts. The screen accepts an amount, calls the appropriate API endpoint, and updates the user's balance.

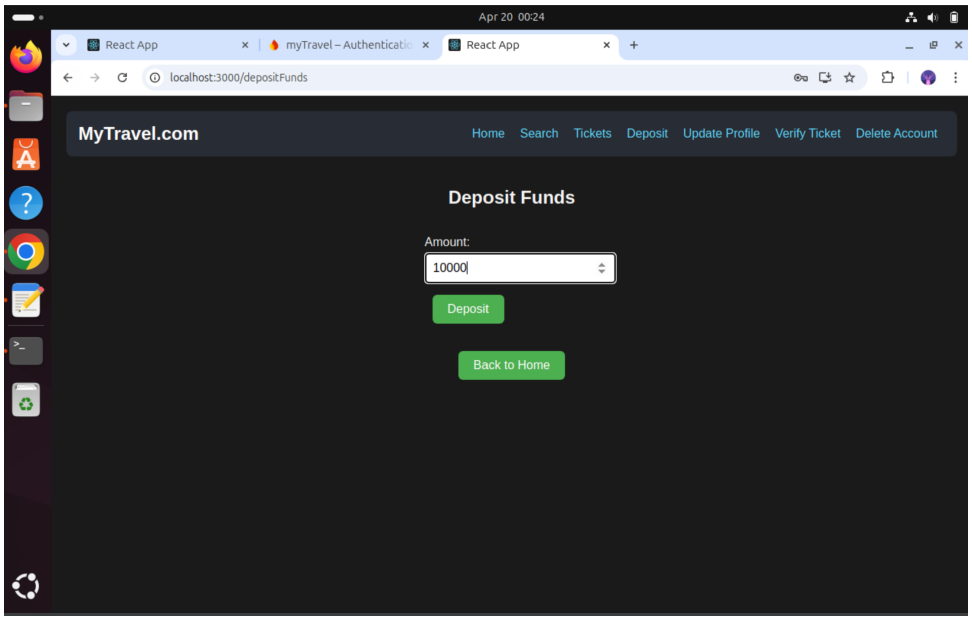

Figure 6.9: Deposit Funds Screen

6.2.4 Provider Functionality Screens

- **AddTravelOption.jsx**
  Enables providers to add a new travel option by entering details such as source, destination, departure time, transport mode, seat capacity, and base price. A small fee is charged per listing.

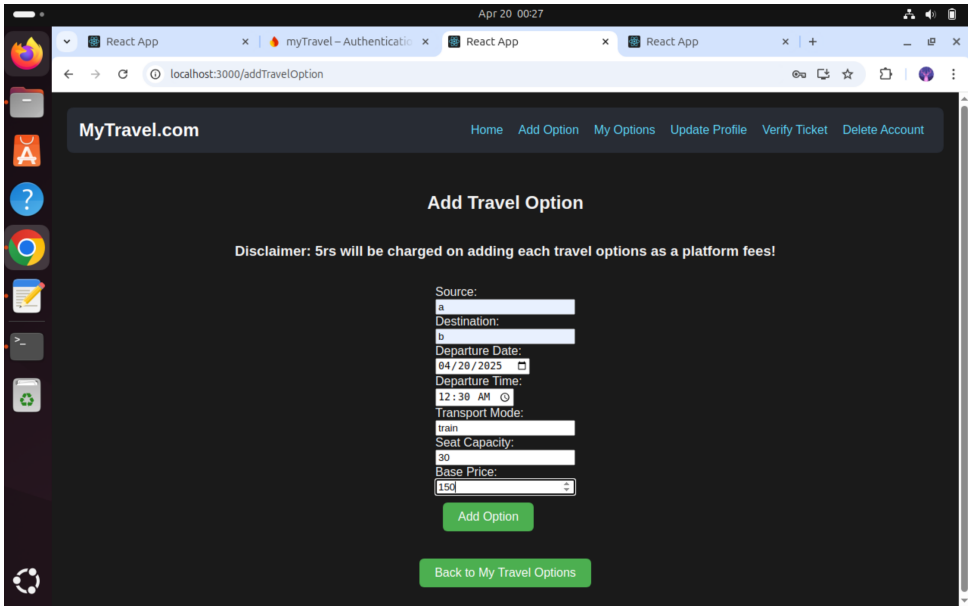

Figure 6.10: Add Travel Options Screen

- **ProviderTravelOptions.jsx**
  Displays all travel options posted by the logged-in provider. Providers can cancel entire travel listings (triggering refunds) from this screen.

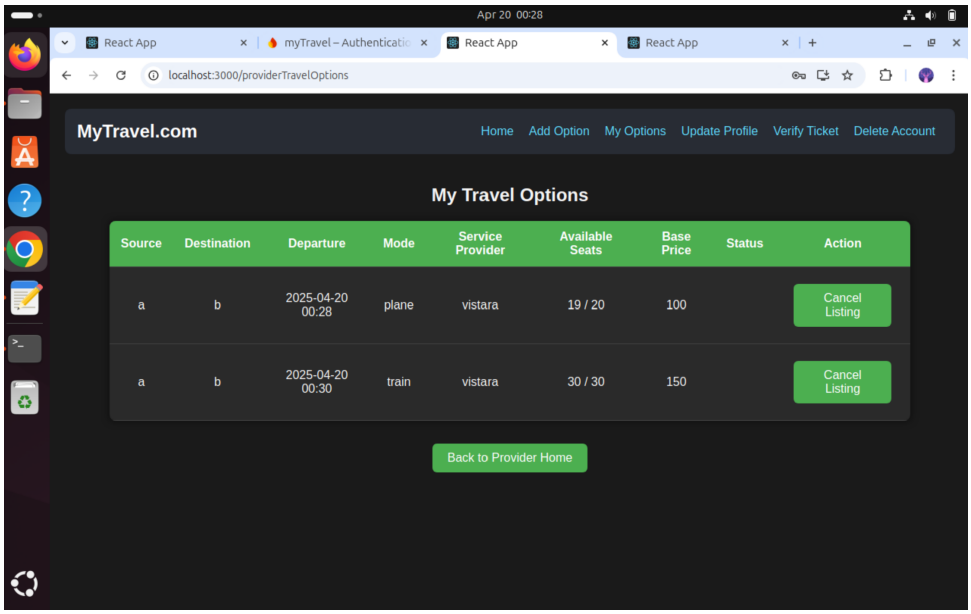

Figure 6.11: Provider Travel Options Screen

6.2.5 Ticket Management and Post-Booking Screens

- **RescheduleTicket.jsx**
  Provides the user with alternative travel options when rescheduling a ticket. After filtering the results to exclude the current selection, it navigates to a rescheduling seat selection screen.

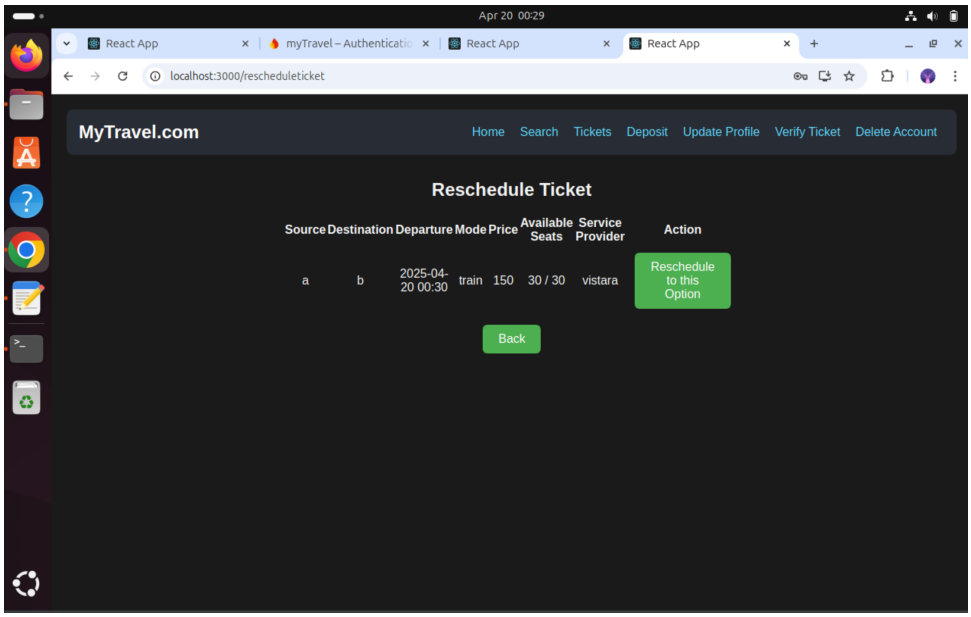

Figure 6.12: Reschedule Ticket Screen

- **RescheduleSeatSelection.jsx**
  Handles seat selection specifically for tickets undergoing rescheduling.

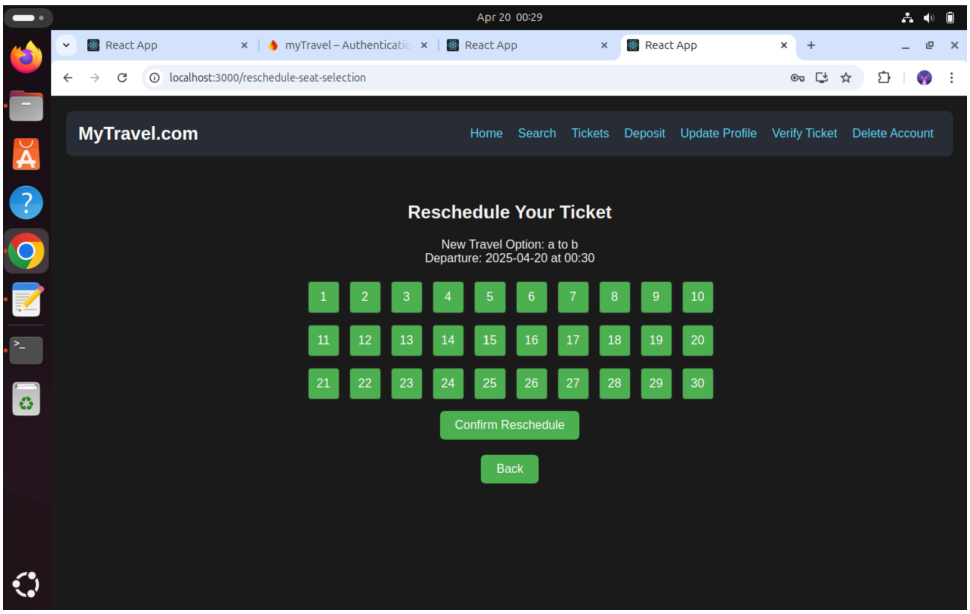

Figure 6.13: Reschedule Ticket Seat Selection Screen

- **RateProvider.jsx**
  Lets users rate the service provider after their travel is completed. It submits the rating along with the related ticket information.

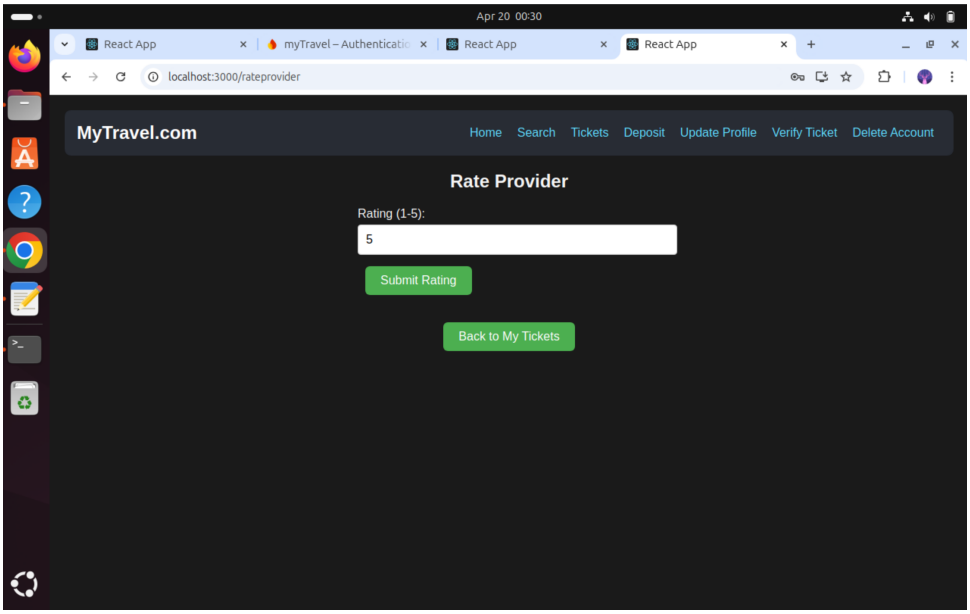

Figure 6.14: Rate Provider Screen

- **PrintableTicket.jsx**
  Generates a printable view of a ticket along with a QR code. The QR code can be downloaded or printed for offline verification.

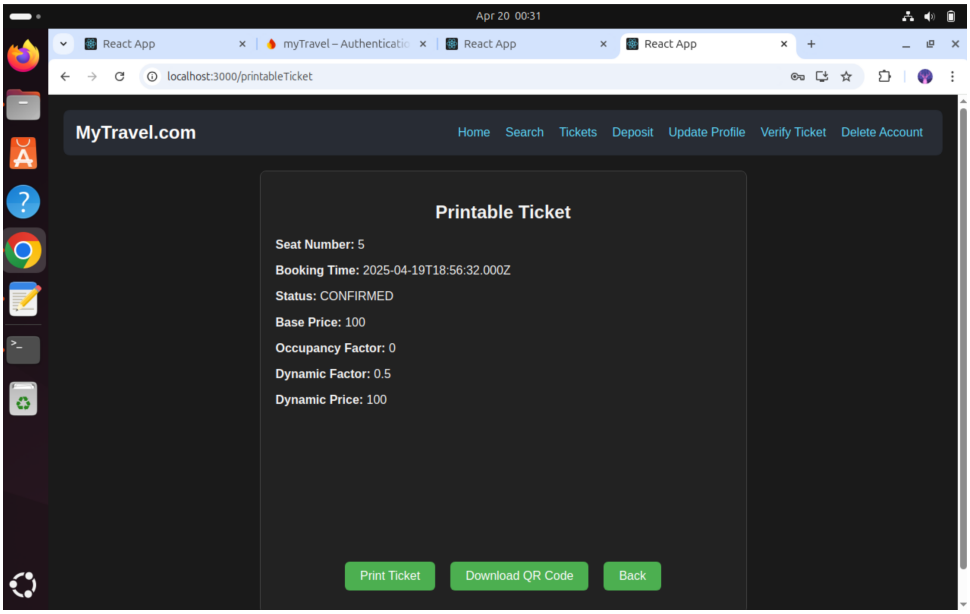

Figure 6.15: Printable Ticket Screen

- **TicketVerification.jsx**
  Allows ticket verification by scanning a QR code from an image upload. It extracts the ticket ID and verifies its status via a backend API.

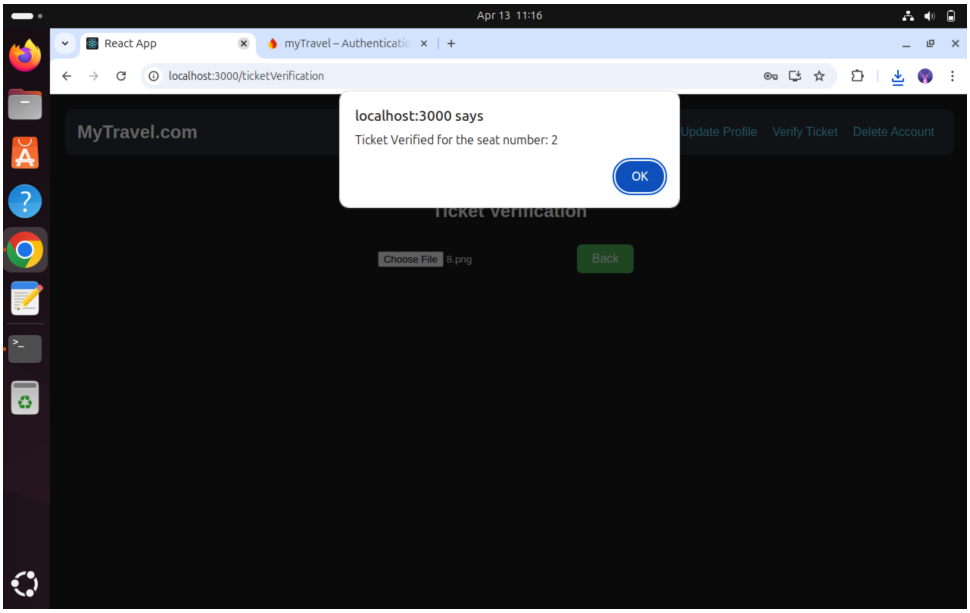

Figure 6.16: Ticket Verification Screen

6.2.6 Profile and Account Management Screens

- **UpdateCustomerDetails.jsx**
  Provides customers with a form to update profile details, including name and contact. Users can also choose to update their anonymity preferences.

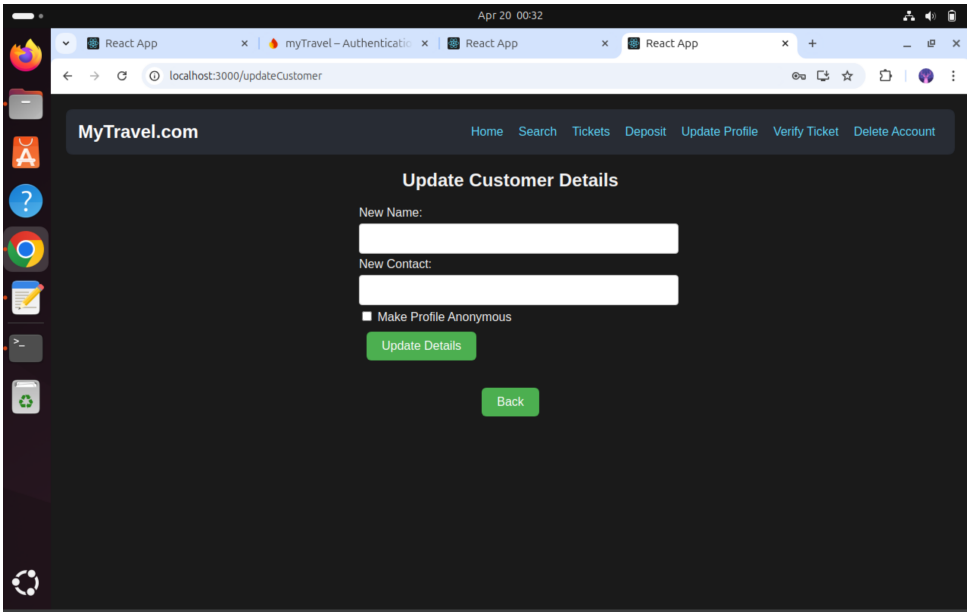

Figure 6.17: Update Customer Details Screen

- **UpdateProviderDetails.jsx**
  Similar to customer updates but tailored for providers. It allows updates to provider name and contact details.

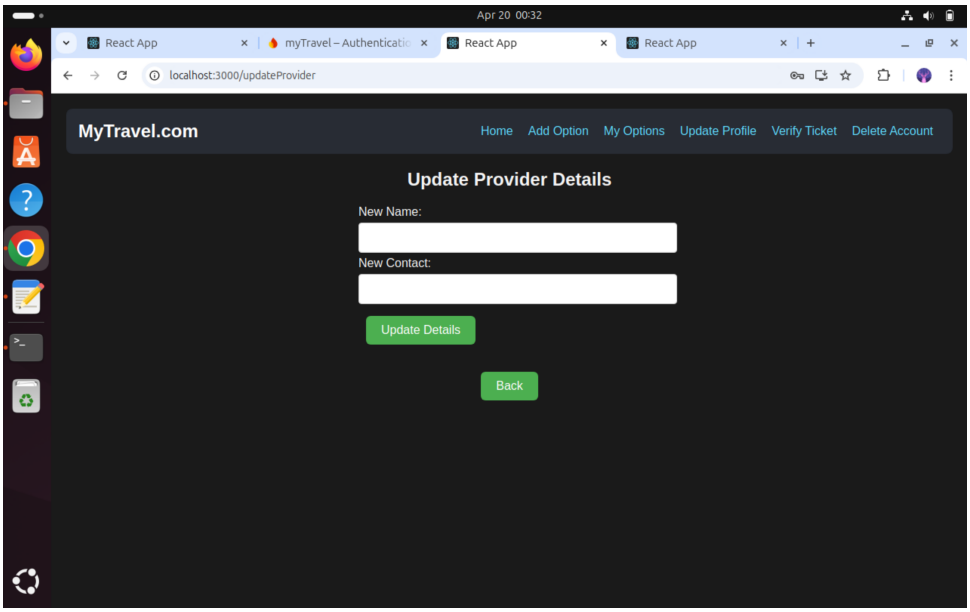

Figure 6.18: Update Provider Details Screen

- **DeleteAccount.jsx**
  (Routing available; implementation not detailed in the provided source.)
  Offers the functionality to delete user or provider accounts.

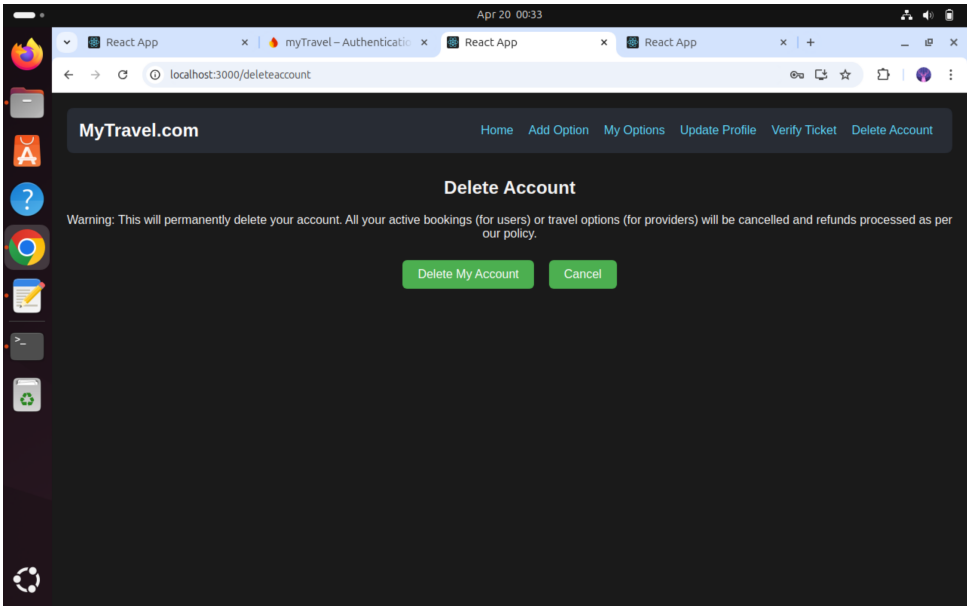

Figure 6.19: Delete Account Screen

6.2.7 Supporting Components and Routing

- **Navbar.jsx**
  Displays a navigation bar with role-specific links. It provides entry points to various screens such as home, search, tickets, and profile updates.

- **ProtectedRoute.jsx**
  A higher-order component to ensure only authenticated users (customers or providers) can access certain routes. It redirects unauthorized users to the login page.

- **Loader.jsx**
  Renders a spinner and loader animation during asynchronous operations or page transitions.

- **App.js**
  Centralizes routing logic (using React Router). It sets up all routes, handles role-based navigation, and wraps the application in an authentication context provider.

- **api.js**
  Configures the Axios instance for API calls. It sets the base URL dynamically, allowing the application to interact with the backend server.

6.2.8 Dynamic Pricing Display

When a ticket is booked, the chaincode returns a *pricingBreakdown* object:

```
{
```

```
  basePrice: <number>,
  occupancyFactor: <number>,
  dynamicFactor: 0.5,
  dynamicPrice: <number>
}
```

This is displayed in the front-end (e.g., `PrintableTicket.jsx`) for user clarity and operational transparency.

6.3 Security and Privacy

- **Certificate-based identities:** Each user has an X.509 certificate from the Fabric CA.
- **Privacy controls:** `isAnonymous` flags can hide name and contact data from chaincode state.
- **Access control:** The chaincode checks `ctx.clientIdentity.getID()` to ensure a user can only see or modify their own data.

Chapter 7

Deployment Instructions

7.1 Prerequisites

1. Install Docker and Docker Compose.
2. Install Node.js (v14+ recommended) and npm.
3. Install Hyperledger Fabric binaries and samples.
4. Set up Firebase for authentication and Firestore for user data storage.

7.2 Fabric Network Setup

- Clone the Hyperledger Fabric `test-network` repository.
- Navigate to the `sdk` directory (e.g., `/home/vboxuser/sec-try/fabric-samples/sdk`) and run:

```
./first.sh javascript
```

7.3 Network Down and Docker Cleanup

If the network is already running and you need to bring it down before re-deploying:

- Run the following command to stop the network:

```
sudo ./networkDown.sh
```

- Clean up Docker containers:

```
sudo docker container prune -f
```

7.4 Subsequent Upgrades

Keep the network running, avoid networkDown.sh so that wallets, ledger state and peers remains intact

7.4.1 Deploy upgraded chaincode version using:

- Run the following command to stop the network:

```
./second.sh javascript
```

7.5 Backend Deployment

- In a separate terminal navigate to the backend folder (e.g., `/home/vboxuser/sec-try/fabric-sam`
- Install dependencies:

```
npm install
```

- Launch the server:

```
nodemon server.js
```

7.6 Frontend Deployment

- Navigate to the client folder (e.g., `/home/vboxuser/sec-try/fabric-samples/client`).
- Install dependencies:

```
npm install
```

- Start the development server:

```
npm start
```

- Once the client starts, the My Ticket website will be accessible at `localhost:3000`.

Chapter 8

Testing Results

For testing normal endpoints, normal node scripts were used to call the endpoint,s and their behaviour was observed to confirm that they are working correctly.

8.1 Introduction

Here, we describe the methodology and scripts used to validate the system's ability to handle 2500 booking transactions in a single day under realistic workload conditions. We employ a two-phase approach:

1. **Data Preparation** (`loadTestSetup.js`): bulk-register users, providers, and travel options; export JSON fixtures.
2. **Load Execution** (`testing.js`): drive booking requests via K6, collect latency and success metrics.

8.2 Test Objectives

- **Throughput:** 2 500 booking operations within 24 h
- **Concurrency:** up to 10 virtual users (VUs)
- **Latency SLA:** 95th-percentile http_req_duration $<$ 6000 ms
- **Reliability:** 99% successful bookings (i.e. HTTP 200)

8.3 Environment

- **Backend API:** `http://localhost:8000`
- **Load Test Machine:** Ubuntu 20.04, 4 vCPUs, 8 GB RAM

- **Tooling:**
  - Node.js 14+, `axios`, `fs`
  - K6 v0.34.1

8.4 Data Preparation Script (`loadTestSetup.js`)

Purpose

Populate the ledger with:

- 500 customers (`cust1@test.com` ... `cust500@test.com`)
- 500 providers (`prov1@test.com` ... `prov500@test.com`)
- 5 travel options per provider, randomized among 10 major Indian cities

Outputs three JSON files for downstream load testing.

8.5 Load Execution Script (`testing.js`)

8.5.1 Purpose

Drive booking requests against the `/bookticket` endpoint, selecting random customer and travel option, pacing with ≈1 request/s per VU. **IMP - During the testing our Auto-Confirm Scheduler was made to run every 5 minutes.**

8.5.2 K6 Configuration

Listing 8.1: K6 Options

```
export let options = {
  scenarios: {
    bookings: {
      executor: 'shared-iterations',
      vus: 10,
      iterations: 3200,
      maxDuration:'40m'
    }
  },
  thresholds: {
    'http_req_duration': ['p(95)<6000']
  }
};
```

8.5.3 Test Logic

Listing 8.2: Booking Function

```
export default function () {
  const cust = customers[Math.floor(Math.random()*customers.length)];
  const opt = opts[Math.floor(Math.random()*opts.length)];
  const seat = Math.ceil(Math.random()*opt.seatCapacity);

  let res = http.post(
    `http://localhost:8000/bookticket?email=${encodeURIComponent(cust)}`,
    JSON.stringify({ travelOptionId: opt.travelOptionId, seatnumber: seat }),
    { headers: { 'Content-Type': 'application/json' } }
  );
  check(res, { 'status is 200': r => r.status===200 });
  sleep(1);
}
```

8.6 Test Observations

From the K6 terminal output (Fig 8.1 , Fig 8.2 and Fig 8.3) (3 200 total iterations), we observed that we were able to achieve:

- **Total requests:** 3 200 (avg. 2.37 req/s)
- **Succeeded:** 3 047 (95.21%)
- **Failed:** 153 (4.78%)
- **Latency (ms):**
  - $P_{90} = 5\,030$
  - $P_{95} = 5\,690$
  - max = 9 460
- **Failure reasons:**
  - MVCC read-conflict (concurrent ledger writes, the testing script sent a request to book two tickets from different users with exactly the same travel option ID and seat number, one got successful, and the other one threw this error, which would result in booking being unsuccessful for these extraneous users).
  - Insufficient balance
  - Seat already booked (duplicate seat requests)

– And other business logic that would lead to unsuccessful bookings.

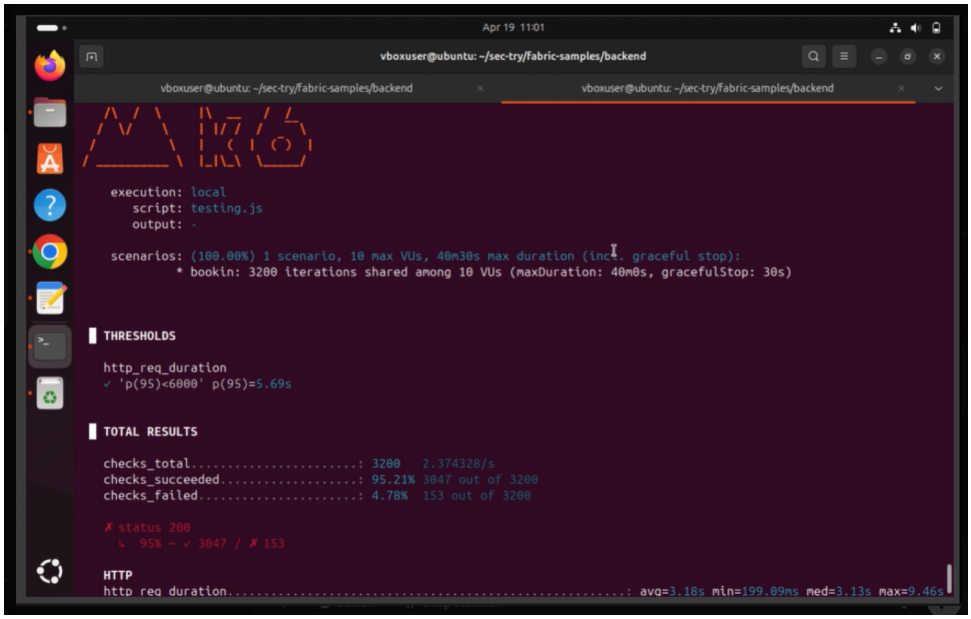

Figure 8.1: Testing Terminal Output 1

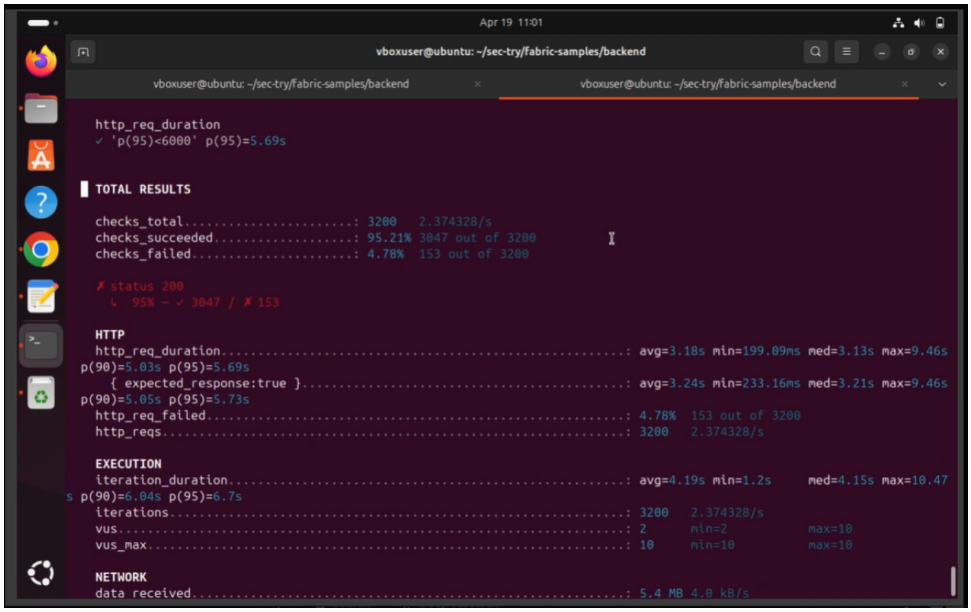

Figure 8.2: Testing Terminal Output 2

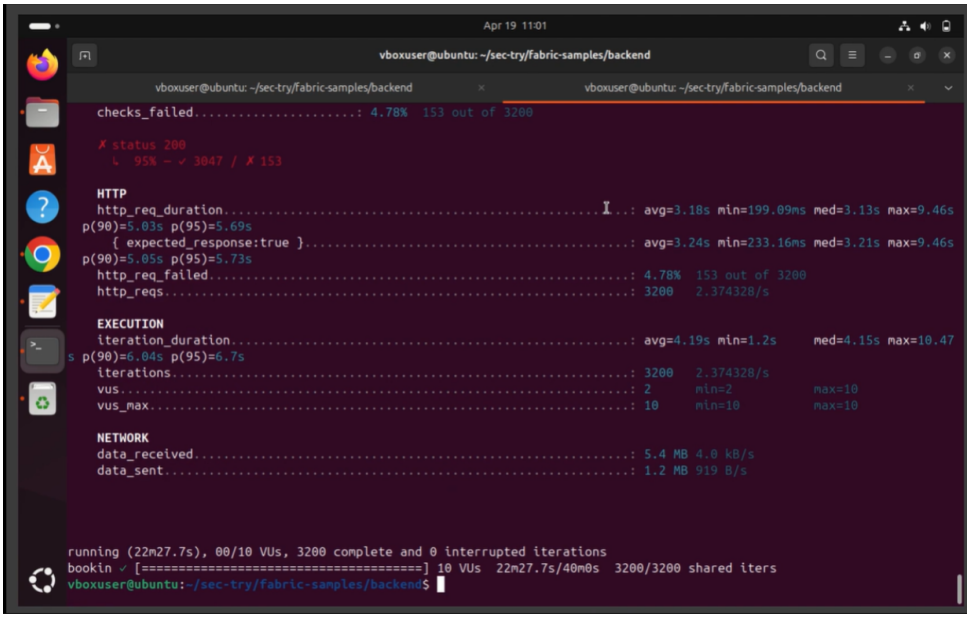

Figure 8.3: Testing Terminal Output 3

The screen recording of the terminal output when this test script was running can be found here.

8.7 Had it been a one day Scenario

$$\text{Observed sustained rate} = 2.37 \text{ req/s}$$

$$\text{Seconds per day} = 86\,400 \text{ s}$$

$$\text{Total requests per day} = 2.37 \times 86\,400 \approx 205{,}000 \text{ req/day}$$

$$\text{Success rate} = 95\%$$

$$\text{Successful bookings per day} = 0.95 \times 205{,}000 \approx 195{,}000 \text{ bookings/day}$$

Chapter 9

Limitations and Future Enhancements

9.1 Current Limitations

1. **Scalability to Very High Loads** – The chaincode is not heavily optimized for extremely large queries. If we had 1 million daily bookings, we might need advanced indexing or partitioning.

2. **Latency** - The test on our laptop showed high latency; a better machine might provide better latency, but this shows a direction of potential optimization. One could use dynamic load balancers.

3. **Simplicity of the Payment Model** – We store `balance` fields in the ledger for each user. In production, we might integrate a real payment gateway or use a token-based approach.

9.2 Potential Future Enhancements

- **Multi-Org Deployment** – Allow different providers from different organizations to manage their own peers, increasing trust boundaries.

- **Advanced Dynamic Pricing Strategies** – Incorporate real-time demand predictions, surge pricing, or AI-driven seat pricing.

- **Microservice Payment Integrations** – Connect to a real-time payment gateway, bridging off-chain fiat transactions with on-chain booking confirmations.

- **Comprehensive Analytics** – Providers may want dashboards of seat utilization, revenue, or user demographics (with user consent).

Chapter 10

Conclusion and References

10.1 Conclusion

This assignment successfully implements a **Blockchain-based Ticket Management System** using Hyperledger Fabric. Through a single organization setup with two peers and one orderer, we have demonstrated:

- Secure and transparent ledger-based seat allocation.
- Role-based operations for *customers* (ticket purchase, cancellation) and *providers* (adding routes, canceling listings).
- *Dynamic pricing* influenced by seat occupancy rates.
- *Refund logic* that factors in the remaining time before departure.
- *Block-based finality* to confirm tickets after at least two subsequent blocks.
- A fully functional front-end (React) and a Node/Express back-end that orchestrates chaincode calls and identity management.
- Other essential and extra functionalities.

10.2 References

1. Hyperledger Fabric Documentation: https://hyperledger-fabric.readthedocs.io/
2. Node.js Official Site: https://nodejs.org/
3. React.js Official Docs: https://reactjs.org/

Chapter 11

Responsibilities Distribution

11.1 Workload Distribution

We have evenly distributed the workload among us, as shown in Table 11.1. We have assigned each member a specific responsibilities to ensure efficient collaboration and completion of tasks.

241110010 Ansh Makwe	**241110022 Divyansh Chaurasia**
Enroll admin and authentication	Register user and provider
Update customer/provider details	Get customer/provider details
Deposit balance	Get customer tickets
Delete customer/provider	Book ticket and cancel ticket
Reschedule ticket and cancel ticket	Get ticket details and auto-confirm ticket
Get provider details and travel options	Add travel option/delete travel option
Cancel travel listing	List and filter travel options
Rate provider	Dynamic pricing for tickets
Build user/provider home page	Build login page with Firebase auth integration
Register user/provider page	Form page for adding travel options
Page to book ticket/list tickets	Reschedule ticket interface
Cancel ticket interface	Block confirmation logic integration
Preventing overbooking	Preventing double booking and

Table 11.1: Workload Distribution Between Team Members

— End of Report —