# Final Project Report

## CS3120 - Database Management Systems Laboratory

## Railway Reservation System

**Group 5:**
111901030 Mayank Singla
111901058 Satyam Mishra
111901005 Aditya Agarwal

---

# INDEX


| Sr no. | Title |
|---|---|
| 1. | Introduction |
| 2. | Features / Requirements of the Project |
| 3. | ER Diagram |
| 4. | Structure, Integrity, and general constraints of the Database |
| 5. | User-Defined Types and Operators |
| 6. | Functionalities of the Database |
| 7. | Utility Functions |
| 8. | Triggers |
| 9. | Database Roles |
| 10. | Views |
| 11. | Indices |
| 12. | Appendix |

# Introduction

This project aims to build a railway reservation system which is a software application that handles the entire booking data of the railway. It has been developed to override the problems prevailing in the practical manual system. It is fully based on the concept of reserving train tickets for various destinations reliably and consistently. It reduces the stress and workload of the employee who books the ticket. It also lets the travelers book and finds schedules for the trains with ease. This software can also be used by different railway companies to carry out operations in a smooth, effective, and automated manner.

# Features/Requirements of the Project

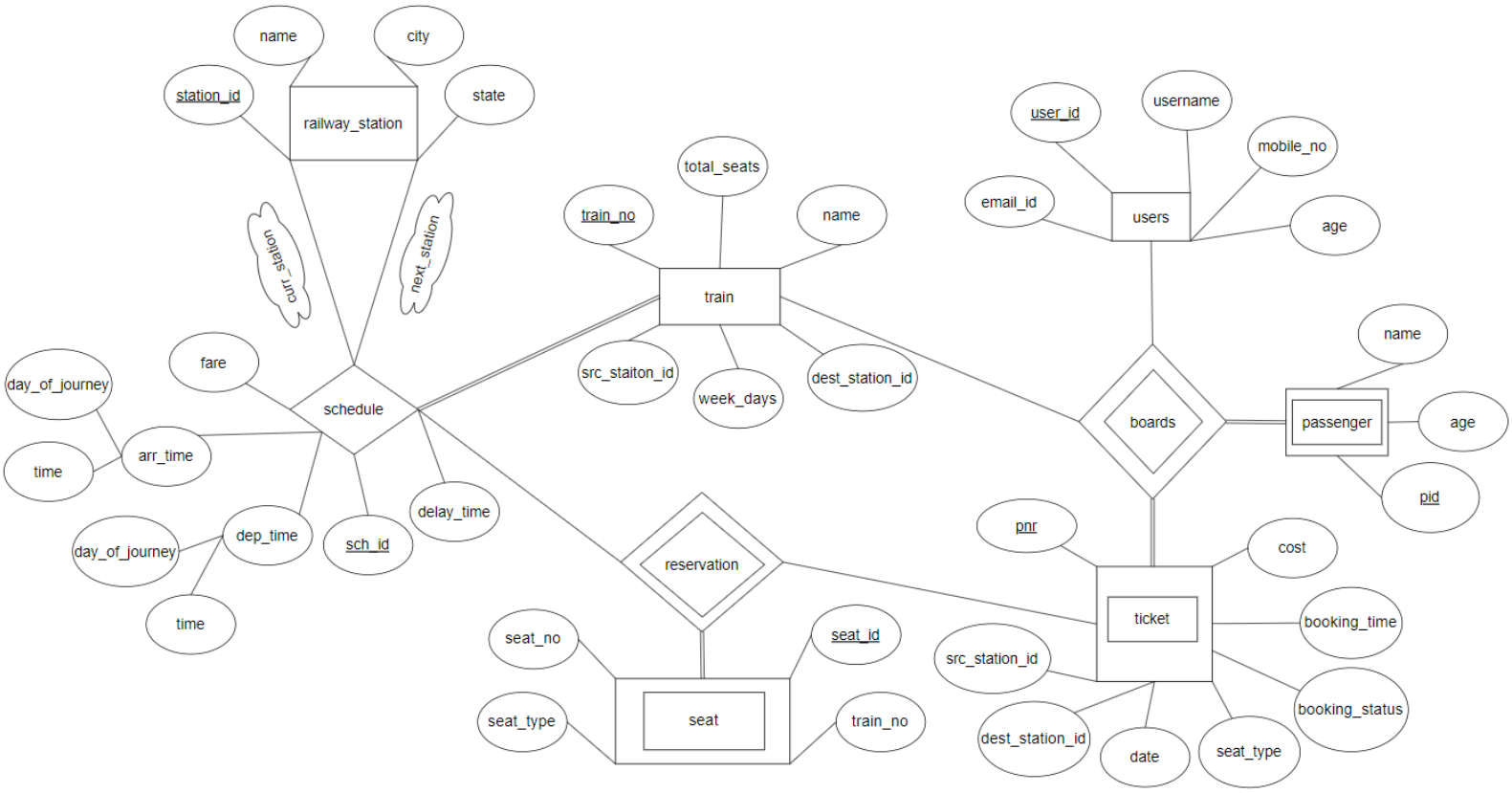
- The passengers should be able to see the actual data of available trains for some given source and destination city.
- The passengers should be able to see the schedule of a particular train on some particular day, the schedule of all the trains at a particular station, the cost it takes for going from one station to another using a particular train, the number of seats available in a particular train, and status of the ticket and the train.
- The users(customers with an account) should be able to book as well as cancel a ticket for a train.
- We should have an automated system to mark the status of a ticket as Booked whenever it is possible to allocate a seat to a passenger. It should also be done based on the priority of booking time.
- Manage the information of the ticket and we should be able to extract the details of a train as well as the customer associated with a given PNR number of a ticket.
- Only the station master role can create, update and delete the date of trains, railway stations, and schedules and update the status of the trains.
- The user could also book a ticket not only for himself but for other passengers as well who might not be the users of the system.
- It should avoid errors while entering the data and provide suitable error messages while entering invalid data.
- Maintaining separate logins for the passengers as well as users. The users will log in by first creating an account on the database.

Based on the above requirements, we

- Have formed the most relevant tables and functions that mimic the railway reservation system and are sufficient to provide the core and fundamental functionalities and facilities of the railway reservation system.
- Added the proper integrity constraints to the tables, also used several checks in the functions, procedure which ensures the proper functioning of our database.
- Introduced different roles which have different privileges over our database.

# ER Diagram

Using the requirements of the railway reservation system, the following are entity sets and relational sets.



# Structure, Integrity, and General Constraints of the Database

Here we are displaying the tables and their constraints in form of a PostgreSQL description of tables to display the results better. In the above ER diagram **board** is a weak relation and hence its table will not be required as it will be redundant.

## Users

This table will contain the users of the table i.e. those who can book tickets and those who can update the railway data. This table contains `user_id` which is the primary key for the table. We have a username, email id, and mobile number as well. Here we have added a check over the email id to follow a regular expression for a valid email id. All the other constraints implemented on other columns are taken from the PostgreSQL inbuilt constraints like age should be greater than 0.

```
railway_reservation_system=# \d users
                                    Table "public.users"
  Column   |          Type          | Collation | Nullable |                Default
-----------+------------------------+-----------+----------+---------------------------------------
 user_id   | integer                |           | not null | nextval('users_user_id_seq'::regclass)
 username  | character varying(100) |           | not null |
 email_id  | character varying(100) |           | not null |
 age       | integer                |           | not null |
 mobile_no | character varying(20)  |           | not null |
Indexes:
    "users_pkey" PRIMARY KEY, btree (user_id)
    "users_email_id_key" UNIQUE CONSTRAINT, btree (email_id)
Check constraints:
    "users_age_check" CHECK (age > 0)
    "users_email_id_check" CHECK (email_id::text ~* '^[A-Za-z0-9._%-]+@[A-Za-z0-9.-]+[.][A-Za-z]+$'::text)
Referenced by:
    TABLE "ticket" CONSTRAINT "ticket_user_id_fkey" FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
Policies:
    POLICY "users_policy"
      TO users
      USING (((username)::text = CURRENT_USER))
```

## Passenger

This table will contain the details of a passenger. A passenger need not be a user. But in our system, if someone wants to book a ticket then he needs to be a user of the database. Here `pid` is the primary key for the passenger and have age should be greater than 0. This entity is weak as it has no sense of its own. All the constraints implemented on other columns are taken from the PostgreSQL inbuilt constraints.

```
railway_reservation_system=# \d passenger;
                        Table "public.passenger"
 Column |          Type          | Collation | Nullable |      Default
--------+------------------------+-----------+----------+--------------------
 pid    | uuid                   |           | not null | uuid_generate_v4()
 name   | character varying(100) |           | not null |
 age    | integer                |           | not null |
Indexes:
    "passenger_pkey" PRIMARY KEY, btree (pid)
Check constraints:
    "passenger_age_check" CHECK (age > 0)
Referenced by:
    TABLE "ticket" CONSTRAINT "ticket_pid_fkey" FOREIGN KEY (pid) REFERENCES passenger(pid) ON DELETE CASCADE
Policies:
    POLICY "passenger_policy"
      TO users
      USING ((pid IN ( SELECT ticket.pid
   FROM ticket
  WHERE (ticket.user_id = ( SELECT users.user_id
           FROM users
          WHERE ((users.username)::text = CURRENT_USER))))))
```

## Railway Stations

This table has information about all the railway stations that are there in the database. We have `station_id` as the primary key (serial) for the station. It even has the city, state, and name of the station. All the constraints implemented on other columns are taken from the PostgreSQL inbuilt constraints.

```
railway_reservation_system=# \d railway_station;
                                  Table "public.railway_station"
   Column   |          Type          | Collation | Nullable |                      Default
------------+------------------------+-----------+----------+-----------------------------------------------------
 station_id | integer                |           | not null | nextval('railway_station_station_id_seq'::regclass)
 name       | character varying(100) |           | not null |
 city       | character varying(100) |           | not null |
 state      | character varying(100) |           | not null |
Indexes:
    "railway_station_pkey" PRIMARY KEY, btree (station_id)
    "railway_station_name_key" UNIQUE CONSTRAINT, btree (name)
Referenced by:
    TABLE "schedule" CONSTRAINT "schedule_curr_station_id_fkey" FOREIGN KEY (curr_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
    TABLE "schedule" CONSTRAINT "schedule_next_station_id_fkey" FOREIGN KEY (next_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
    TABLE "ticket" CONSTRAINT "ticket_dest_station_id_fkey" FOREIGN KEY (dest_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
    TABLE "ticket" CONSTRAINT "ticket_src_station_id_fkey" FOREIGN KEY (src_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
    TABLE "train" CONSTRAINT "train_dest_station_id_fkey" FOREIGN KEY (dest_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
    TABLE "train" CONSTRAINT "train_src_station_id_fkey" FOREIGN KEY (src_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
```

## Train

This table contains the train number and train name. Train name is unique, Here the `train_no` is the primary key. We have even stored the station id of the station where the train starts and the destination station where the whole journey of the train will end. We have also added the weekdays on which the train will run, and the total seats the train has.  source/destination are foreign keys for this table.

Train name, source/destination station id, total seats, and weekdays should be NOT NULL while input, we have also added constraints that we can only book tickets for the future. All the constraints implemented on other columns are taken from the PostgreSQL inbuilt constraints.

```
railway_reservation_system=# \d train;
                                    Table "public.train"
     Column      |          Type          | Collation | Nullable |                  Default
-----------------+------------------------+-----------+----------+-----------------------------------------
 train_no        | integer                |           | not null | nextval('train_train_no_seq'::regclass)
 name            | character varying(100) |           | not null |
 src_station_id  | integer                |           | not null |
 dest_station_id | integer                |           | not null |
 total_seats     | integer                |           | not null |
 week_days       | day_of_week[]          |           | not null |
Indexes:
    "train_pkey" PRIMARY KEY, btree (train_no)
    "train_name_key" UNIQUE CONSTRAINT, btree (name)
    "train_dest_station_id" hash (dest_station_id)
    "train_src_station_id" hash (src_station_id)
Check constraints:
    "train_total_seats_check" CHECK (total_seats >= 0)
Foreign-key constraints:
    "train_dest_station_id_fkey" FOREIGN KEY (dest_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
    "train_src_station_id_fkey" FOREIGN KEY (src_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
Referenced by:
    TABLE "schedule" CONSTRAINT "schedule_train_no_fkey" FOREIGN KEY (train_no) REFERENCES train(train_no) ON DELETE CASCADE
    TABLE "seat" CONSTRAINT "seat_train_no_fkey" FOREIGN KEY (train_no) REFERENCES train(train_no) ON DELETE CASCADE
    TABLE "ticket" CONSTRAINT "ticket_train_no_fkey" FOREIGN KEY (train_no) REFERENCES train(train_no) ON DELETE CASCADE
```

## Schedule

The schedule is a relation connecting railway stations and trains. In this table, for every train, we have a pair of station IDs and intermediate stations that lie on the route of the train from the source station to the destination station. Each entry of the table has the current station and the next station. This next station value will be NULL for the destination station. This relation has fare, days, dept-time, and arr-time as attributes. In fare, we expect to get the fare from the source station to the current station of the train and it will always be a positive value. Days will tell the running days of the train and their values must be a subset of the standard days (that has been implemented via subset operator in PostgreSQL) and the arrival and departure times will tell the arrival and departure time for the current station and train respectively. All the other constraints implemented on other columns are taken from the PostgreSQL inbuilt constraints.

NOTE: We have added the ON DELETE CASCADE on every foreign key so that on deleting any record the dependent rows also get deleted.

```
railway_reservation_system=# \d schedule;
                                  Table "public.schedule"
     Column      |     Type     | Collation | Nullable |                 Default
-----------------+--------------+-----------+----------+-----------------------------------------
 sch_id          | integer      |           | not null | nextval('schedule_sch_id_seq'::regclass)
 train_no        | integer      |           | not null |
 curr_station_id | integer      |           | not null |
 next_station_id | integer      |           |          |
 arr_time        | day_time     |           | not null |
 dep_time        | day_time     |           | not null |
 fare            | numeric(7,2) |           | not null |
 delay_time      | interval     |           | not null |
Indexes:
    "schedule_pkey" PRIMARY KEY, btree (sch_id)
    "schedule_curr_station_id" hash (curr_station_id)
    "schedule_next_station_id" hash (next_station_id)
    "schedule_train_no" hash (train_no)
Check constraints:
    "schedule_check" CHECK (arr_time <= dep_time)
    "schedule_delay_time_check" CHECK (delay_time >= '00:00:00'::interval)
    "schedule_fare_check" CHECK (fare >= 0::numeric)
Foreign-key constraints:
    "schedule_curr_station_id_fkey" FOREIGN KEY (curr_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
    "schedule_next_station_id_fkey" FOREIGN KEY (next_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
    "schedule_train_no_fkey" FOREIGN KEY (train_no) REFERENCES train(train_no) ON DELETE CASCADE
```

## Ticket

This table contains the details of the ticket booked by a user. It has `PNR` as its primary key. It even has the `cost` of the journey and the source and destination station ids for the journey and the `pid` number of the passenger traveling through this ticket. This table also contains source/destination station id, seat id, type, booking status, and time. All the fields are NOT NULL mostly  This entity is weak as it has no sense of its own.

```
railway_reservation_system=# \d ticket;
                                    Table "public.ticket"
     Column      |            Type             | Collation | Nullable |          Default
-----------------+-----------------------------+-----------+----------+---------------------------
 pnr             | uuid                        |           | not null | uuid_generate_v4()
 cost            | integer                     |           | not null |
 src_station_id  | integer                     |           | not null |
 dest_station_id | integer                     |           | not null |
 train_no        | integer                     |           | not null |
 user_id         | integer                     |           | not null |
 date            | date                        |           | not null |
 pid             | uuid                        |           | not null |
 seat_id         | integer                     |           |          |
 seat_type       | seat_type                   |           |          |
 booking_status  | booking_status              |           |          | 'Waiting'::booking_status
 booking_time    | timestamp without time zone |           |          | CURRENT_TIMESTAMP
Indexes:
    "ticket_pkey" PRIMARY KEY, btree (pnr)
    "ticket_booking_status" hash (booking_status)
    "ticket_date" btree (date)
    "ticket_dest_station_id" hash (dest_station_id)
    "ticket_pid" hash (pid)
    "ticket_seat_id" hash (seat_id)
    "ticket_seat_type" hash (seat_type)
    "ticket_src_station_id" hash (src_station_id)
    "ticket_train_no" hash (train_no)
    "ticket_user_id" hash (user_id)
Check constraints:
    "ticket_cost_check" CHECK (cost > 0)
Foreign-key constraints:
    "ticket_dest_station_id_fkey" FOREIGN KEY (dest_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
    "ticket_pid_fkey" FOREIGN KEY (pid) REFERENCES passenger(pid) ON DELETE CASCADE
    "ticket_seat_id_fkey" FOREIGN KEY (seat_id) REFERENCES seat(seat_id) ON DELETE CASCADE
    "ticket_src_station_id_fkey" FOREIGN KEY (src_station_id) REFERENCES railway_station(station_id) ON DELETE CASCADE
    "ticket_train_no_fkey" FOREIGN KEY (train_no) REFERENCES train(train_no) ON DELETE CASCADE
    "ticket_user_id_fkey" FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
Policies:
    POLICY "ticket_policy" FOR SELECT
```

# User-Defined Types and Operators

We are having the following ENUM types in our database:

- DAY_OF_WEEK
- SEAT_TYPE
- BOOKING_STATUS

```
railway_reservation_system=# \dT
          List of data types
 Schema |      Name      | Description
--------+----------------+-------------
 public | booking_status |
 public | day_of_week    |
 public | day_time       |
 public | intbig_gkey    |
 public | query_int      |
 public | seat_type      |
(6 rows)
```

We are having the following Composite type in our database:

- **DAY_TIME(day_of_journey, time)**: which stores an integer representing in which day of the journey the train is currently in and the time which will represent the time of arrival/departure of a train at a station.

We have created the following overloaded operators for user-defined types:

- **DAY_TIME <= DAY_TIME**

```
railway_reservation_system=# select (1, '12:00:00')::DAY_TIME <= (1, '02:00:00')::DAY_TIME;
 ?column?
----------
 f
(1 row)
```

- **DAY_TIME < DAY_TIME**

```
railway_reservation_system=# select (1, '01:00:00')::DAY_TIME < (1, '02:00:00')::DAY_TIME;
 ?column?
----------
 t
(1 row)
```

- **DAY_OF_WEEK @+ INT = DAY_OF_WEEK**

```
railway_reservation_system=# select 'Monday'::DAY_OF_WEEK @+ 3;
 ?column?
----------
 Thursday
(1 row)
```

- **DAY_OF_WEEK @- DAY_OF_WEEK = INT** *(Assumes LHS to be greater)*

```
railway_reservation_system=# select 'Monday'::DAY_OF_WEEK @- 'Saturday'::DAY_OF_WEEK;
 ?column?
----------
        2
(1 row)
```

# Functionalities of the Database

List of all functions and triggers and how they help in preserving the consistency of the database

## Book tickets

This procedure will take the names of all the passengers, their respective ages and their choice of seat type and the source and destination station names. It even takes the train name and date on which you want to board the train and the email id of the person booking the ticket.
It will first check if the given name, age, and seat type arrays are all of the same lengths. Then we will check if the train passed from that station on that given day. After these checks are successfully one we iterate over all the passengers and first add them to the passenger table and then to the ticket table with their reservation status as waiting. After this, we have a trigger that will get active as we insert a new entry into the ticket table it will call the function to allocate a seat.

```
railway_reservation_system=# CALL book_tickets(  -- Will get seat booked
    ARRAY['abc']::VARCHAR(100)[],
    ARRAY[37]::INT[],
    ARRAY['AC']::SEAT_TYPE[],
    'Indore_RS'::VARCHAR(100),
    'Kolkata_RS'::VARCHAR(100),
    'GIJK'::VARCHAR(100),
    '2022-05-26'::DATE,
    'abc@abc.com'::VARCHAR(100)
);
CALL
railway_reservation_system=#
```

```
railway_reservation_system=# CALL book_tickets(  -- Will not get seat booked
    ARRAY['def']::VARCHAR(100)[],
    ARRAY[55]::INT[],
    ARRAY['AC']::SEAT_TYPE[],
    'Gujarat_RS'::VARCHAR(100),
    'Jaipur_RS'::VARCHAR(100),
    'GIJK'::VARCHAR(100),
    '2022-05-26'::DATE,
    'def@def.com'::VARCHAR(100)
);
CALL
```

```
railway_reservation_system=# CALL book_tickets(  -- Will not get seat booked
    ARRAY['ghi']::VARCHAR(100)[],
    ARRAY[61]::INT[],
    ARRAY['AC']::SEAT_TYPE[],
    'Jaipur_RS',
    'Kolkata_RS',
    'GIJK',
    '2022-05-27'::DATE,
    'ghi@ghi.com'
);
CALL
railway_reservation_system=#
```

```
railway_reservation_system=# SELECT * FROM ticket;
                pnr                 | cost | src_station_id | dest_station_id | train_no | user_id |    date    |                pid                 | seat_id | seat_type | booking_status |        booking_time
------------------------------------+------+----------------+-----------------+----------+---------+------------+------------------------------------+---------+-----------+----------------+----------------------------
 a0963358-9f7d-43b0-b394-637ddc56894e | 2000 |              5 |               7 |       10 |       1 | 2022-05-26 | 2a20dee0-ab89-43f9-bc20-e72dd813627f |       1 | AC        | Booked         | 2022-04-28 22:33:32.614419
 ed20b6f4-6e6b-4bde-9a96-c8162add9369 | 2000 |              4 |               6 |       10 |       2 | 2022-05-26 | 05251dcc-17e1-48b0-a1ba-f42bf8b01ecf |         | AC        | Waiting        | 2022-04-28 22:33:57.031457
 c72e22bf-6cce-4def-aa61-fa0e93b32595 | 1000 |              6 |               7 |       10 |       3 | 2022-05-27 | d028c832-7533-4043-9326-82f2b0d33657 |         | AC        | Waiting        | 2022-04-28 22:34:26.34743
(3 rows)

railway_reservation_system=#
```

Here we can see that our trigger works fine as one seat got booked. This train has just one seat and therefore only one passenger's ticket got booked.

## Cancel Booking

This procedure will take the pnr of the ticket which we want to cancel and then it will just update the booking status for that pnr as canceled and seat_id as null. Even here a trigger will get activated and will try to allocate the canceled seat.

```
railway_reservation_system=# CALL cancel_booking('a0963358-9f7d-43b0-b394-637ddc56894e');
CALL
railway_reservation_system=#
```

```
railway_reservation_system=# SELECT * FROM ticket;
                pnr                 | cost | src_station_id | dest_station_id | train_no | user_id |    date    |                pid                 | seat_id | seat_type | booking_status |        booking_time
------------------------------------+------+----------------+-----------------+----------+---------+------------+------------------------------------+---------+-----------+----------------+----------------------------
 a0963358-9f7d-43b0-b394-637ddc56894e | 2000 |              5 |               7 |       10 |       1 | 2022-05-26 | 2a20dee0-ab89-43f9-bc20-e72dd813627f |         | AC        | Cancelled      | 2022-04-28 22:33:32.614419
 ed20b6f4-6e6b-4bde-9a96-c8162add9369 | 2000 |              4 |               6 |       10 |       2 | 2022-05-26 | 05251dcc-17e1-48b0-a1ba-f42bf8b01ecf |       1 | AC        | Booked         | 2022-04-28 22:33:57.031457
 c72e22bf-6cce-4def-aa61-fa0e93b32595 | 1000 |              6 |               7 |       10 |       3 | 2022-05-27 | d028c832-7533-4043-9326-82f2b0d33657 |       1 | AC        | Booked         | 2022-04-28 22:34:26.34743
(3 rows)

railway_reservation_system=#
```

We can see that the original ticket got canceled and the other two which were originally waiting now got booked. Thus our trigger is working fine.

## Allocate seat

In this procedure we take the pnr, train name, start and end station name of the journey. It will even take the date of the journey along with the preferred seat type.
Even here we are checking if the train crossed that station on that day.
In this procedure, we create a temporary table called a reservation. This table has all possible pairs of schedule id and seat id. And for the seat id, we are even storing its seat type and booking status. Initially the value of bookings status is false. We update this booking status by traversing over the ticket table and selecting the tuple where train number and journey date are matched and have booking type as booked. So by this, we will get all the booked seat details into the reservation table. Suppose I want to go from station A to station B then only those seats which have the booking status not as 'Booked' for all the stations from A to B are valid. After selecting such seats we can pick the seat which was booked for the largest number of stations on the total journey. This will be the seat that we will allocate to the passenger by updating the booking status for that pnr as booked. If we were not able to find any valid seat then we let the status of that passenger be left as 'waiting'.

## Get Trains

In this function, we take the source, destination stations, and date as input and then we find the source station and destination station IDs and day from the date and then we find the train which departs from the source station at the given day.

```
railway_reservation_system=# SELECT * FROM get_trains('Mumbai_RS', 'Delhi_RS', '2022-04-26');
 train_no | train_name | seats_available
----------+------------+-----------------
        3 | KMD        | 6/6
        4 | KCBMD      | 4/4
        6 | LJMPKD     | 8/8
(3 rows)
```

## Get Train Schedule

This function just takes the train name as its argument. And it returns a table consisting of tuples of the current station, next station, arrival time, and departure time sorted in the way in which the train reaches those stations.

```
railway_reservation_system=# SELECT * fROM get_train_schedule('BCDGIJKLMP');
       current_station       |        next_station         | arrival_time | departure_time | arrival_days | departure_days | delay_time
-----------------------------+-----------------------------+--------------+----------------+--------------+----------------+------------
 Bangalore_RS, Bangalore, KA | Chennai_RS, Chennai, TN     | 09:00:00     | 10:00:00       | {Monday}     | {Monday}       | 00:00:00
 Chennai_RS, Chennai, TN     | Delhi_RS, Delhi, DL         | 12:00:00     | 16:00:00       | {Monday}     | {Monday}       | 00:00:00
 Delhi_RS, Delhi, DL         | Gujarat_RS, Gandhinagar, GJ | 01:00:00     | 04:00:00       | {Tuesday}    | {Tuesday}      | 00:00:00
 Gujarat_RS, Gandhinagar, GJ | Indore_RS, Indore, MP       | 10:00:00     | 10:30:00       | {Tuesday}    | {Tuesday}      | 00:00:00
 Indore_RS, Indore, MP       | Jaipur_RS, Jaipur, RJ       | 13:00:00     | 07:30:00       | {Tuesday}    | {Wednesday}    | 00:00:00
 Jaipur_RS, Jaipur, RJ       | Kolkata_RS, Kolkata, WB     | 11:00:00     | 12:00:00       | {Wednesday}  | {Wednesday}    | 00:00:00
 Kolkata_RS, Kolkata, WB     | Lucknow_RS, Lucknow, UP     | 01:10:00     | 03:00:00       | {Friday}     | {Friday}       | 00:00:00
 Lucknow_RS, Lucknow, UP     | Mumbai_RS, Mumbai, MH       | 12:50:00     | 15:00:00       | {Friday}     | {Friday}       | 00:00:00
 Mumbai_RS, Mumbai, MH       | Palakkad_RS, Palakkad, KL   | 05:00:00     | 05:00:00       | {Saturday}   | {Sunday}       | 00:00:00
 Palakkad_RS, Palakkad, KL   |                             | 09:00:00     | 18:00:00       | {Sunday}     | {Sunday}       | 00:00:00
(10 rows)
```

## Get trains schedules at the station

Here we take the station name as input and then return the table containing train no., train name, source station, next station, the final destination of the train passing through this station, arrival time, and days, departure time and days, total seats of the train and weekdays on which this particular train starts its journey.

```
railway_reservation_system=# SELECT * FROM get_trains_schedule_at_station('Indore_RS');
```

```
train_no | train_name |       source_station        |       next_station        |    destination_station    | arrival_time | departure_time |
       arrival_days        |       departure_days       | delay_time | total_seats |         week_days
---------+------------+-----------------------------+---------------------------+---------------------------+--------------+----------------+
---------------------------+----------------------------+------------+-------------+--------------------------
       5 | DGIPK      | Delhi_RS, Delhi, DL         | Palakkad_RS, Palakkad, KL | Kolkata_RS, Kolkata, WB   | 06:00:00     | 07:00:00       |
 {Tuesday}                  | {Thursday}                 | 00:00:00   |          10 | {Monday}
       7 | IJBC       | Indore_RS, Indore, MP       | Jaipur_RS, Jaipur, RJ     | Chennai_RS, Chennai, TN   | 02:00:00     | 03:00:00       |
 {Tuesday,Wednesday,Sunday} | {Tuesday,Wednesday,Sunday} | 00:00:00   |           7 | {Monday,Tuesday,Saturday}
       9 | BCDGIJKLMP | Bangalore_RS, Bangalore, KA | Jaipur_RS, Jaipur, RJ     | Palakkad_RS, Palakkad, KL | 13:00:00     | 07:30:00       |
 {Tuesday}                  | {Wednesday}                | 00:00:00   |          24 | {Sunday}
      10 | GIJK       | Gujarat_RS, Gandhinagar, GJ | Jaipur_RS, Jaipur, RJ     | Kolkata_RS, Kolkata, WB   | 12:00:00     | 16:00:00       |
 {Friday}                   | {Friday}                   | 00:00:00   |           1 | {Thursday}
(4 rows)
```

## Get Fare

In the schedule table we have stored the cumulative fare starting from the source station of the train to the current station. So if we want a fare from station A to station B then we need to subtract the cumulative fare at A from the cumulative fare at B.
In this function, we take the name of the train and the source and destination station of the journey. Then by using two select queries we get the fare at the source and destination stations. Subtracting them will give the desired fare.

```
railway_reservation_system=# SELECT * FROM get_fare('LJMPKD', 'Jaipur_RS', 'Delhi_RS');
 get_fare
----------
     4000
(1 row)
```

## Get Passenger Details

This function will take the pnr of the passenger to display his/her details. So firstly we need to validate the pnr. So we get all the related information of the passenger by taking joins over the ticket, passenger, train, railway_station, user, and seat tables.

```
railway_reservation_system=# SELECT get_passenger_details('5753e0d5-03a4-4763-a5d6-1db617f6a271');
```

```
            get_passenger_details

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
----------------------------------------------
 (5753e0d5-03a4-4763-a5d6-1db617f6a271,2b7a68cf-95d9-475c-9c05-f311e9e3180f
,titu,31,4,KCBMD,"Chennai_RS, Chennai, TN","Delhi_RS, Delhi, DL",2022-04-25
,300,mno,mno@mno.com,9988776655,Cancelled,,)
(1 row)
```

## Number Of Seats Available From Source To Destination

This function will take the names of the train, source station, and destination station. And this will even take the date for which we want to check the availability of seats. So we will first check if the train passes through the station on the given date. Now we can iterate over the ticket table and check for those seat_ids which belong to the train on which we are querying and which have booking status as booked and which have some overlap (in terms of the stations which it covers) with the given source and destination station. So this way we can calculate those seats which are not available. Subtracting this from the total will give the vacant seats from the given source and destination station on the given train and date.

```
railway_reservation_system=# SELECT num_seats_available_from_src_to_dest('PJBK','Palakkad_RS', 'Jaipur_RS','2022-05-30');
 num_seats_available_from_src_to_dest
--------------------------------------
                                    10
(1 row)
```

## Get ticket status

In this function we take the PNR of the passenger and then first validate that the PNR exists in our database and then from the ticket table returns the current status of the ticket i.e ‘waiting’, ‘booked’ or ‘canceled’.

```
railway_reservation_system=# SELECT get_ticket_status('5753e0d5-03a4-4763-a5d6-1db617f6a271');
 get_ticket_status
-------------------
 Booked
(1 row)
```

## Get Train Status

In this query we take the name of the train and station and return the status of the train. So we simply perform the select query over the schedule table and get the status of the current status of the train.

```
railway_reservation_system=# SELECT * FROM get_train_status('KMD','Kolkata_RS') AS delay_time;
 delay_time
------------
 00:00:00
(1 row)
```

## Add railway station

In this function we take the name, city, and state of the new station we want to insert into our database.

```
CALL add_railway_station('Bangalore_RS', 'Bangalore', 'KA');
CALL add_railway_station('Chennai_RS', 'Chennai', 'TN');
```

## Add schedule

This is a very important function that can be accessed by DB admin and station master only, here we take the train name, seat number, seat types, the weekdays on which train will start its journey, the stations it will stop through, the arrival time and departure time at every station and also the day on which the train arrive or depart from the station from the starting day of journeys, at last, we take the of each station i.e the cost if we travel at the particular station from the source station.
Then we do some checking in the input data to maintain the integrity of our database

1. Array Length (seat numbers) = Array Length (seat type)
2. Array Length (arrival and departure time at the station) = Array Length (station)
3. Array Length (fare) = Array Length (station)
4. The journey length of the train is less than 7 as it follows a weekly timetable.

We insert all the data into a respective table, like a train with source, destination station, seats, weekdays into the Train table, seats with seat type into the seat table, at last, we put the data into the schedule table with train name, stations, and fare, etc…

```
CALL add_schedule(
    'PJBK'::VARCHAR(100),
    ARRAY[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]::INT[],
    ARRAY['AC', 'AC', 'AC', 'NON-AC', 'NON-AC', 'NON-AC', 'AC', 'AC', 'AC', 'AC']::SEAT_TYPE[],
    ARRAY['Monday', 'Wednesday', 'Friday']::DAY_OF_WEEK[],
    ARRAY['Palakkad_RS', 'Jaipur_RS', 'Bangalore_RS', 'Kolkata_RS']::VARCHAR(100)[],
    ARRAY[(1, '02:00:00'), (1, '05:00:00'), (1, '07:00:00'), (1, '09:00:00')]::DAY_TIME[],
    ARRAY[(1, '03:00:00'), (1, '05:30:00'), (1, '07:30:00'), (1, '09:20:00')]::DAY_TIME[],
    ARRAY[0, 1000, 4000, 5000]::NUMERIC(7, 2)[]
);
```

## Add User

In this procedure we take information from the user and add it to the database using an insert query.

```
railway_reservation_system=# CALL add_user('mno', 'mno@mno.com', 55, '9988776655');
CALL
```

```
railway_reservation_system=# SELECT * FROM USERS;
 user_id | username |   email_id    | age | mobile_no
---------+----------+---------------+-----+------------
       1 | mno      | mno@mno.com |  55 | 9988776655
```

### Update train status

In this function we take the train name, station name, and delay time of the given train at the given station then we update the delay time of this particular tuple in our schedule table into our database.

```
railway_reservation_system=# CALL update_train_status('KMD'::VARCHAR, 'Kolkata_RS'::VARCHAR, '1 hour 10 minutes 45 seconds'::INTERVAL)
CALL
railway_reservation_system=# SELECT * FROM get_train_status('KMD','Kolkata_RS') AS delay_time;
 delay_time
------------
 01:10:45
```

# Utility Functions

### Get Train Name

This function will take the train number and using the select query over the train table it will return the train name if it exists.

```
railway_reservation_system=# SELECT * FROM get_train_name(3);
 get_train_name
----------------
 KMD
```

### Get Station Name

This function will take the station id and using the select query over the station table it will return the station name if it exists.

```
railway_reservation_system=# SELECT * FROM get_station_name(3);
 get_station_name
------------------
 Delhi_RS
```

## Get Day

This function will extract the day from a given date.

```
railway_reservation_system=# SELECT * FROM get_day('2022-05-02');
 get_day
---------
 Monday
```

## Get Train Number

This function will take the train name and using the select query over the train table it will return the train number if it exists.

```
railway_reservation_system=# SELECT * FROM get_train_no('KMD');
 get_train_no
--------------
            3
```

## Get Station Id

This function will take the station id and using the select query over the station table it will return the station name if it exists.

```
railway_reservation_system=# SELECT * FROM get_station_id('Delhi_RS');
 get_station_id
----------------
              3
```

## Get User Id

This function will take the user email and using the select query over the user table it will return the station id if it exists.

```
railway_reservation_system=# select * from users;
 user_id | username |   email_id    | age | mobile_no
---------+----------+-------------+-----+------------
       1 | mno      | mno@mno.com |  55 | 9988776655
(1 row)

railway_reservation_system=# SELECT * FROM get_user_id('mno@mno.com');
 get_user_id
-------------
           1
```

## Get Schedule Ids

In this function we take the train number, source station id, and destination station id. Then we iterate over the schedule table and go over the stations in the order in which the train runs over them. While iterating over the stations we save the schedule ids into the array and return that array.

```
railway_reservation_system=# SELECT * FROM schedule WHERE train_no = 3 ORDER BY sch_id;
 sch_id | train_no | curr_station_id | next_station_id |   arr_time    |   dep_time    |  fare   | delay_time
--------+----------+-----------------+-----------------+---------------+---------------+---------+------------
      9 |        3 |               7 |               9 | (1,12:00:00) | (1,13:00:00) |    0.00 | 01:10:45
     10 |        3 |               9 |               3 | (2,14:00:00) | (3,15:00:00) | 1000.00 | 00:00:00
     11 |        3 |               3 |                 | (3,16:00:00) | (3,17:00:00) | 2000.00 | 00:00:00
(3 rows)

railway_reservation_system=# SELECT * FROM get_sch_ids(3,7,3);
 get_sch_ids
-------------
 {9,10}
```

## Validate PNR

This function will assert if the given pnr is there in the database or not.

```
railway_reservation_system=# SELECT validate_pnr('5753e0d5-03a4-4763-a5d6-1db617f6a271');
 validate_pnr
--------------

(1 row)

railway_reservation_system=# SELECT validate_pnr('5753e0d5-03a4-4763-a5d6-1db617f6a275');
ERROR:  Ticket with PNR "5753e0d5-03a4-4763-a5d6-1db617f6a275" does not exist!
```

## Get Journey At Station

This function will return the day of the journey of a given train at a given station.

```
railway_reservation_system=# SELECT * FROM get_journey_at_station(7,3);
 get_journey_at_station
------------------------
                      1
```

## Get Days At Station

This function will return the day at which the given train touches the given station.

```
railway_reservation_system=# SELECT * FROM get_days_at_station(7,3);
 get_days_at_station
---------------------
 {Monday,Thursday}
```

## Validate Train Days At Station

This function will assert if the given train passes through the given station on a given date.

```
railway_reservation_system=# SELECT * FROM validate_train_days_at_station(7,3,'2022-05-02');
 validate_train_days_at_station
--------------------------------

(1 row)

railway_reservation_system=# SELECT * FROM validate_train_days_at_station(7,3,'2022-05-03');
ERROR:  The train "KMD" does not run on the date "2022-05-03" at the station "Kolkata_RS"
```

## Get Updated Days

This function will take an array of days and add an offset to all the days in the given array.

```
railway_reservation_system=# SELECT * FROM get_updated_days(2,ARRAY['Monday']::DAY_OF_WEEK[]);
 get_updated_days
-------------------
 [0:0]={Wednesday}
```

## Pretty Station Name

This function will take the name, city, and state of the station and print it in a comma separate manner.

```
railway_reservation_system=# SELECT * FROM pretty_station_name('Kolkata_RS','Kolkata','WB');
   pretty_station_name
-------------------------
 Kolkata_RS, Kolkata, WB
(1 row)
```

# Triggers

In the railway reservation system, we want to automate the process of seat allocation for new bookings. For this purpose, we are having a **try_alloc_seat** trigger function on the **ticket** table which will try to allocate a seat whenever a new booking is made or canceled. We could have made two separate triggers for this purpose one for new booking and one for cancellation of the booking, but to avoid code repetition, we incorporate this trigger into one and make use of the special variable TG_OP which is provided by PostgreSQL. In case of a new booking, that is an INSERT operation, we are normally calling the **allocate_seat** function to try allocating a seat for this new booking if any seat is available. In case when a booking is canceled, we are taking all the tickets which still have the booking status as Waiting for the same train on the same date and sorting them in ascending order based on their booking time and calling the **allocate_seat** function for each of them to try allocating seat for those users.
We are not having any other trigger functions as of now in our database as all the input validation and error handling is done within the functions only as it makes more sense there.

```sql
-- Trigger function for automatic seat allocation
CREATE OR REPLACE FUNCTION try_alloc_seat()
RETURNS TRIGGER
LANGUAGE PLPGSQL
AS $$
DECLARE
    train_name VARCHAR(100);
    src_station_name VARCHAR(100);
    dest_station_name VARCHAR(100);
    waiting_ticket RECORD;
BEGIN
    IF TG_OP = 'INSERT' THEN
        -- Extract train_name, source/destination station names
        SELECT get_train_name(NEW.train_no)
        INTO train_name;
        SELECT get_station_name(NEW.src_station_id)
        INTO src_station_name;
        SELECT get_station_name(NEW.dest_station_id)
        INTO dest_station_name;

        -- Try allocating seat to this passenger
        CALL allocate_seat(NEW.pnr, train_name, src_station_name, dest_station_name, NEW.date, NEW.seat_type);
    ELSEIF TG_OP = 'UPDATE' AND NEW.booking_status = 'Cancelled' THEN
        -- Loop all the passengers in the waiting queue in order of booking time
        FOR waiting_ticket IN (SELECT pnr,
                                      train_no,
                                      src_station_id,
                                      dest_station_id,
                                      date,
                                      seat_type
                               FROM ticket
                               WHERE booking_status = 'Waiting'
                                   AND (date - get_journey_at_station(src_station_id, train_no) + 1) =
                               (NEW.date - get_journey_at_station(NEW.src_station_id, NEW.train_no) + 1)
                                   AND train_no = NEW.train_no
                               ORDER BY booking_time ASC)
        LOOP
            -- RAISE NOTICE '%', waiting_ticket;
            -- Extract train_name, source/destination station names
            SELECT get_train_name(waiting_ticket.train_no)
            INTO train_name;
            SELECT get_station_name(waiting_ticket.src_station_id)
            INTO src_station_name;
            SELECT get_station_name(waiting_ticket.dest_station_id)
            INTO dest_station_name;

            -- Try allocating seat to this passenger
            CALL allocate_seat(waiting_ticket.pnr, train_name, src_station_name, dest_station_name,
                waiting_ticket.date, waiting_ticket.seat_type);
        END LOOP;
    END IF;

    RETURN NEW;
END;
$$;
```

# Database Roles

We are having a total of 4 roles in our database.
This first is the **dbAdmin** which has all the privileges over the whole database and all the tables. The dbAdmin is also able to bypass RLS that we have implemented for certain tables.

Next, we have a **station_master** role which is responsible for adding/updating/deleting new railway stations and trains. This role has hence all the privileges on the tables **railway_station**, **schedule**, **train,** and **seat**. This role is the only role that has access to the procedures **add_railway_station**, **add_schedule,** and **update_train_status**. For all the other roles in the PUBLIC schema, we have revoked privileges from them for these procedures.

Next, we have the **passenger** role who does not have any special privileges on any table. This role can call the functions for which the privileges are not revoked on the public schema.

Lastly, we have the **user's** role which represents the actual users of our railway system. This role is the only role that has access to the procedures **book_tickets** and **cancel_booking**. For all the other roles in the PUBLIC schema, we have revoked privileges from them for these procedures.
This role also has all the privileges on the tables **users** and **passengers** to update his info or info about the passengers booked by this user. This role also has SELECT privileges on the table **ticket** to see the tickets booked by this user. Since a user should be able to only see/modify data related to himself only and also be able to see tickets booked by him only, we are implementing row-level security to make sure that a user is not able to access the information out other users/passengers/tickets.

We have also revoked all privileges from the roles in the PUBLIC schema for the procedure **allocate_seat** as it is only called by the trigger function and should not be called by anyone else manually. As mentioned that we are using SECURITY DEFINER, so this procedure and the trigger calling it will be invoked with SUPERUSER privileges, and hence, this procedure will be callable by triggering the event.

```
railway_reservation_system=# \c railway_reservation_system postgres
You are now connected to database "railway_reservation_system" as user "postgres".
railway_reservation_system=# SELECT * FROM users;
 user_id | username |  email_id   | age | mobile_no
---------+----------+-------------+-----+------------
       1 | abc      | abc@abc.com |  18 | 9888887777
       2 | def      | def@def.com |  25 | 9777766555
       3 | ghi      | ghi@ghi.com |  40 | 9999999999
       4 | jkl      | jkl@jkl.com |  31 | 9876543210
       5 | mno      | mno@mno.com |  55 | 9988776655
(5 rows)

railway_reservation_system=#
```

```
railway_reservation_system=# \c railway_reservation_system abc
Password for user abc:
You are now connected to database "railway_reservation_system" as user "abc".
railway_reservation_system=> SELECT * FROM users;
 user_id | username |  email_id   | age | mobile_no
---------+----------+-------------+-----+------------
       1 | abc      | abc@abc.com |  18 | 9888887777
(1 row)

railway_reservation_system=>
```

So we can see that because of row-level security the user can access only his/her details.

# Views

As explained above that we are implementing RLS and using SECURITY DEFINER and making sure that it is safe to allow users to execute the functions created. The roles who are allowed to access some tables are capable enough to view the whole table and no column needs to be hidden for those roles. So, we are only having views for reducing the workload of writing the same SELECT queries again and again.

View Name: **stations_trains**

The purpose of building this view is that it stores the schedule of all the trains at all the stations. This query is being used many times in our queries where we need to access the schedule of trains at different or at all the stations or filter them by some constraint. For this purpose, we need to do JOIN operations with three tables. Also, using this view we are bringing the data in a pretty format for better output and as required by other queries. So this view helps in writing this whole query again and again.

```sql
-- Displays schedule of all the trains at all the stations
CREATE VIEW stations_trains AS
SELECT sch.train_no,
    sch.curr_station_id,
    sch.next_station_id,
    tt.name AS train_name,
    pretty_station_name(src.name, src.city, src.state) AS source_station,
    pretty_station_name(curr.name, curr.city, curr.state) AS current_station,
    pretty_station_name(nxt.name, nxt.city, nxt.state) AS next_station,
    pretty_station_name(dest.name, dest.city, dest.state) AS destination_station,
    (sch.arr_time).time AS arrival_time,
    (sch.dep_time).time AS departure_time,
    '{' || ARRAY_TO_STRING(get_updated_days(
                    (sch.arr_time).day_of_journey,
                    tt.week_days
            ), ',', '')
        || '}' AS arrival_days,
    '{' || ARRAY_TO_STRING(get_updated_days(
                    (sch.dep_time).day_of_journey,
                    tt.week_days), ',', '')
        || '}' AS departure_days,
    sch.delay_time,
    tt.total_seats,
    tt.week_days
FROM schedule AS sch
    JOIN train AS tt ON sch.train_no = tt.train_no
    JOIN railway_station AS src ON tt.src_station_id = src.station_id
    JOIN railway_station AS curr ON sch.curr_station_id = curr.station_id
    LEFT JOIN railway_station AS nxt ON sch.next_station_id = nxt.station_id
    JOIN railway_station AS dest ON tt.dest_station_id = dest.station_id
ORDER BY sch.train_no ASC, sch.arr_time ASC;
```

# Indices

We are creating a hash index on the **train_no** column of the **seat** table as we have the query **allocate_seat** function which is the most frequent query of our database executed inside the trigger for every new booking or cancellation of a booking.
Next, we are creating hash indexes on the **train_no**, **curr_station_id,** and **next_station_id** columns of the **schedule** table and **src_station_id** and **dest_station_id** columns of the **train** table as these columns are also the most accessed fields inside many different query functions like **get_trains**, **get_train_schedule**, **get_trains_schedule_at_stationb, get_fare,** and our view **stations_trains**.
Next, we are having hash indexes on the foreign keys of the **ticket** table and BTree index on the **date** column of the **ticket** table as they are being accessed in many queries related to bookings, getting details of users and passengers,
Some query functions are **get_passenger_details**, **num_seats_available_from_src_to_dest,** and **allocate_seat.**

# Appendix

Here is the link of the google drive link which contains all the .sql files relevant to our project.
https://drive.google.com/drive/folders/1eGjdwlAq-Hw-o1kFUu0RHkSWWa7Hfi61?usp=sharing

We also took a backup of our project in form of the tar file, which is important for unprecedented times.

```
satyam@ubuntu:~/tmp$ pg_dump -d railway_reservation_system -U satyam -F t > railway_database.tar
satyam@ubuntu:~/tmp$ ls
railway_database.tar
satyam@ubuntu:~/tmp$
```

**Here is the structure of our project.**