Ho Chi Minh City University of Technology

Assignment 2 Report
Course: Database Systems

Hospital Database

Team 1 - CC03

Team members:	*Student's ID:*
Le Thanh Binh	2112897
Vu Le Binh	2152440
Tran Ngoc Minh Diep	1952610
Dinh Le Dung	2152483
Truong Huynh Anh Dung	2053622

Instructor:
Phan Trong Nhan

Ho Chi Minh City, December, 2023

Contents

1 Physical Database Design

In the previous assignment, we have already designed the Enhanced Entity Relationship Diagram (EERD) and the mapping to a relational database schema for the database of Hospital X as below:

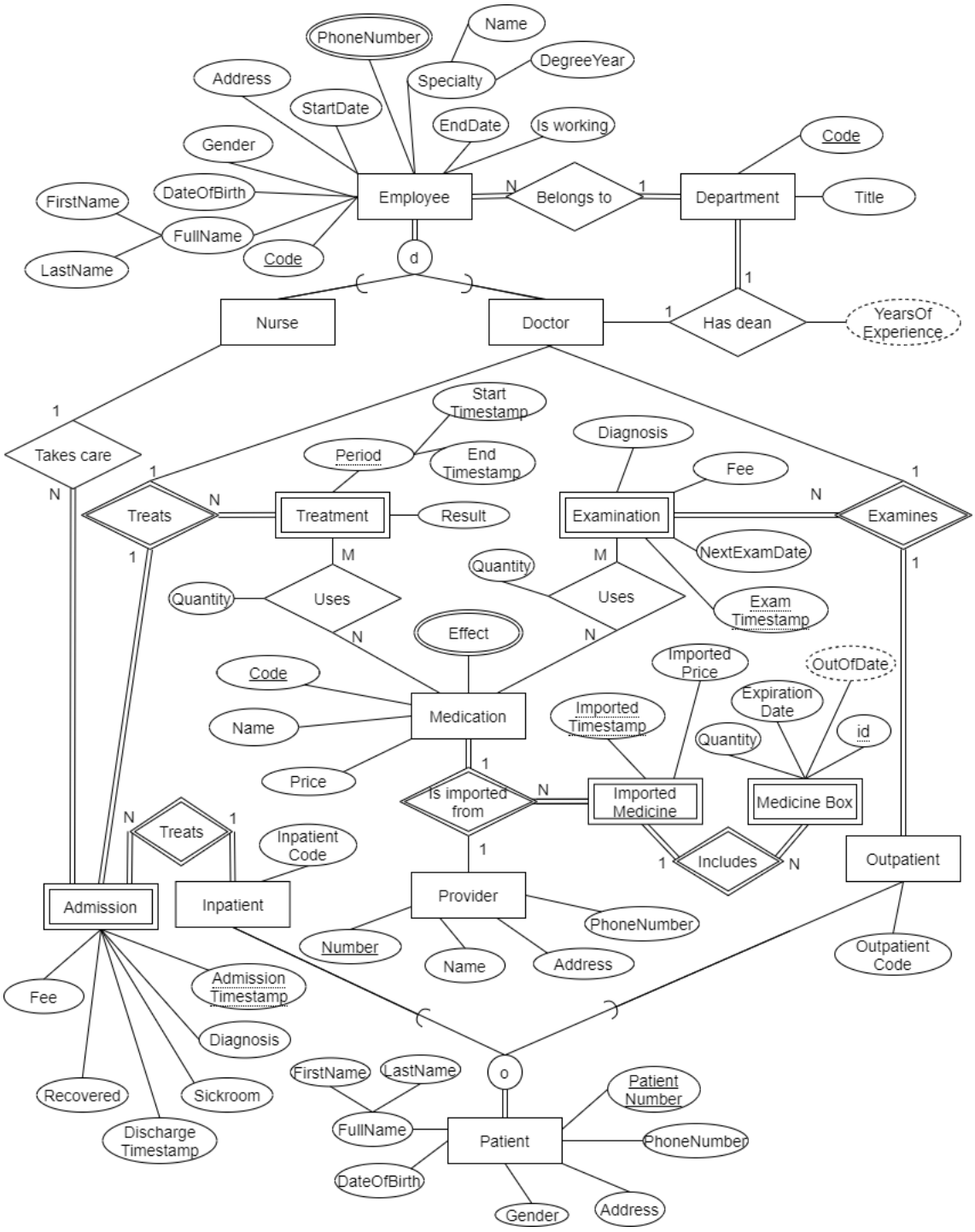


Figure 1: EERD for Hospital X

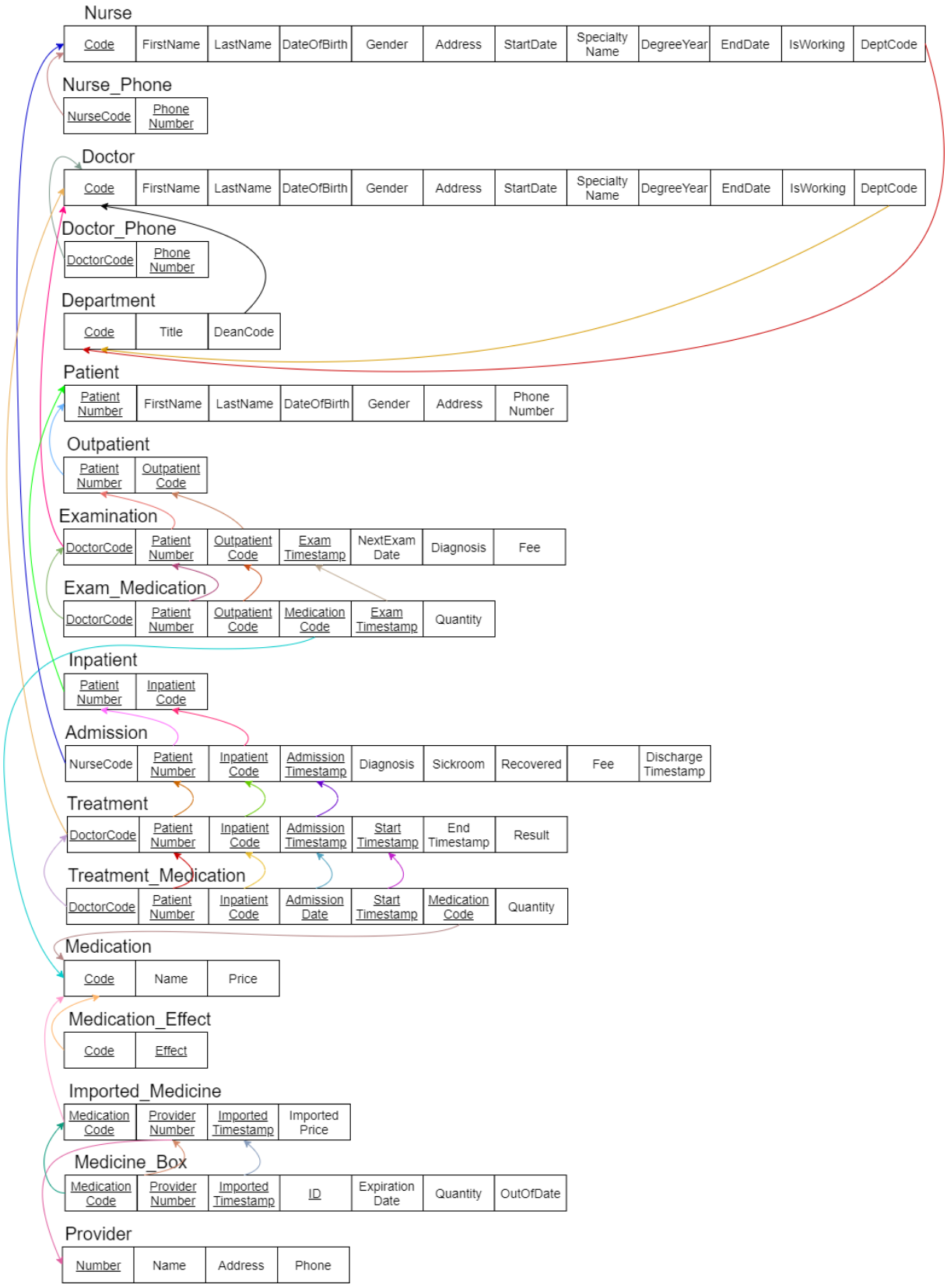


Figure 2: Relational database schema for Hospital X

Based on the EERD and mapping above, we have implement the physical database with 18 tables, including:

1. `Nurse`
2. `Nurse_Phone`
3. `Doctor`
4. `Doctor_Phone`
5. `Department`
6. `Patient`
7. `Outpatient`
8. `Examination`
9. `Exam_Medication`
10. `Inpatient`
11. `Admission`
12. `Treatment`
13. `Treatment_Medication`
14. `Medication`
15. `Medication_Effect`
16. `Provider`
17. `Imported_Medicine`
18. `Medicine_Box`

For implementation, we will use **PostgreSQL** as the DBMS to implement the physical database.

1.1 Table Nurse

The table `Nurse` contains information about the nurses who are working or used to work in the hospital. The query to create the table `Nurse` is shown as below:

```
CREATE TABLE Nurse(
  Code              CHAR(10)    NOT NULL    UNIQUE,
  Last_Name         TEXT, --include surname and middle name
  First_Name        TEXT, --given name
  Date_Of_Birth     DATE,
  Gender            CHAR        CHECK(Gender = 'F' or Gender = 'M'),
  Address           TEXT,
  Start_Date        DATE,
  Specialty_Name    TEXT,
  Degree_Year       SMALLINT    CHECK(Degree_Year BETWEEN 0 AND EXTRACT(
    YEAR FROM CURRENT_DATE)),
  End_Date          DATE,
  Is_Working        BOOL,
  Dept_Code         CHAR(2),
  PRIMARY KEY (Code)
);
```

- **Code:** This is the unique code to identify each nurse in the hospital and every nurse must have this code when they work for the hospital, so there must be `Unique` and `Not null` constraint here. Therefore, we will choose this attribute to be the primary key for the table `Nurse`. We assume that the code is a string with 10 characters, starting with "N" and following by 9 digits (for example: `N000000001`). Thus, the data type for `Code` is `Char(10)`. We choose the data type `Char` instead of `Varchar` to indicate that this unique code is a string with fixed number of characters so that any mistake relating to the inaccurate number of characters can be detected.

- **Last_Name:** The value for this column is a string containing surname and middle name of the nurse, so we will choose the data type `Text`, which is a string with variable length in case the nurse has long middle name or no middle name. For example, if a nurse's name is "Nguyen Thi A", "Nguyen Thi" will be the value for column **Last_Name**.

- **First_Name:** The value for this column is a string containing only given name of the nurse, and we also use the data type `Text` here to offer more flexibility for the string's length. For example, the value in column **First_Name** of the nurse "Nguyen Thi A" is "A".

- **Date_Of_Birth:** This column contains values on the birthdate of the nurse, so we will choose the data type `Date` here.

- **Gender:** This column contains information about the gender of the nurse. We will use "F" to denote female nurse and "M" to denote male nurse. Thus, the data type use here is `Char`, which is a single character. We also add the constraint

`CHECK(Gender = 'F' or Gender = 'M')` to check whether the input for gender is valid or not.

- **Address:** This column contains information about the address of the nurse, so we will use the data type `Text` to offer more flexibility for the string's length.
- **Start_Date:** This column contains information about the first date that the nurse started to work in the hospital, so we will use the data type `Date` here.
- **Specialty_Name:** This column contains information about name of the specialty of the nurse, and we will use the data type `Text` here.
- **Degree_Year:** This column contains information about the year in which the specialty degree of the nurse was awarded. We only need to store the year here, so we will use the data type `Smallint` instead of other data types about date and time. Besides, we also have the constraint to check if the value for degree year is between 0 and the current year, as the value for year cannot be negative or larger than the current year.
- **End_Date:** This column contains information about the date on which the nurse stops working in the hospital, so we will use the data type `Date` here. The value for this column can be null if the nurse is still working in the hospital.
- **Is_Working:** The value in this column is a flag to indicate whether the nurse is still working in the hospital at the moment or not. Therefore, the data type chosen here is `Bool`. If the nurse is currently working in the hospital, the value in this column is `True` and vice versa.
- **Dept_Code:** The value in this column is the code of the department which the nurse belongs to. We will explain this in more detail in section 1.5.

Later on, after creating the table `Department`, which contains information about the departments in the hospital, we will add foreign key `Dept_Code` in table `Nurse` which references from table `Department`.

```
1 ALTER TABLE Nurse
2 ADD FOREIGN KEY (Dept_Code) REFERENCES Department(Code)
3 ON DELETE SET NULL    ON UPDATE CASCADE;
```

We use the action `On delete set null` because in case the department is disbanded and removed from the database, the `Dept_Code` column of the nurses who used to work for that department will be set to null. Meanwhile, the action `On update cascade` is used for the case when the department change their name and their code, the `Dept_Code` of all nurses working for that department will be updated automatically.

1.2 Table Nurse_Phone

The table `Nurse_Phone` contains information about phone numbers of the nurses. A nurse may have one or many phone numbers. The query to create the table `Nurse_Phone` is shown as below:

```
CREATE TABLE Nurse_Phone(
  Nurse_Code        CHAR(10),
  Phone_Number      VARCHAR(11),
  PRIMARY KEY (Nurse_Code, Phone_Number),
  FOREIGN KEY (Nurse_Code) REFERENCES Nurse(Code)
      ON DELETE RESTRICT   ON UPDATE RESTRICT
);
```

- **Nurse_Code:** The value in this column is the unique code to identify each nurse working in the hospital with the data type `Char(10)` as we have explained in section 1.1. In this table, we do not set the unique constraint for `Nurse_Code` because one nurse can have many phone numbers, so the values for `Nurse_Code` can be duplicated.
- **Phone_Number:** This column contains phone numbers of the nurses. In Vietnam, a typical mobile phone number is 10 digits (for example: 0912345678), while a landline phone number consists of 11 digits (for example: 02812345678). Therefore, we choose the data type `Varchar(11)` to indicate that phone numbers are stored as strings with the maximum length is 11 characters.
- **Primary key:** We choose the tuple `(Nurse_Code, Phone_Number)` as the primary key for this table. `Nurse_Code` can be duplicated as one nurse may have several phone numbers. `Phone_Number` can be duplicated in case several nurses use the same landline phone number. But the tuple `(Nurse_Code, Phone_Number)` is unique.
- **Foreign key:** `Nurse_Code` is the foreign key referencing the column `Code` in table `Nurse` to indicate that these phone numbers belongs to nurses. Every `Nurse_Code` in the table `Nurse_Phone` must exist in the table `Nurse`. We use action `On delete restrict` to prevent the referenced rows in table `Nurse` from being deleted, and `On update restrict` to prevent updating the `Code` column of the referenced rows. We do not want to delete the information about a nurse completely or changing the `Nurse_Code` indiscriminately.

1.3 Table Doctor

The table `Doctor` contains information about the doctors who are working or used to work in the hospital. The query to create the table `Doctor` is shown as below:

```
CREATE TABLE Doctor(
  Code              CHAR(10)   NOT NULL   UNIQUE,
  Last_Name         TEXT, --include surname and middle name
```

```
  First_Name          TEXT, --given name
  Date_Of_Birth       DATE,
  Gender              CHAR        CHECK(Gender = 'F' or Gender = 'M'),
  Address             TEXT,
  Start_Date          DATE,
  Specialty_Name      TEXT,
  Degree_Year         SMALLINT    CHECK(Degree_Year BETWEEN 0 AND EXTRACT(
    YEAR FROM CURRENT_DATE)),
  End_Date            DATE,
  Is_Working          BOOL,
  Dept_Code           CHAR(2),
  PRIMARY KEY (Code)
);
```

- **Code:** This is the unique code to identify each doctor in the hospital and every doctor must have this code when they work for the hospital, so there must be `Unique` and `Not null` constraint here. Therefore, we will choose this attribute to be the primary key for the table `Doctor`. We assume that the code is a string with 10 characters, starting with "D" and following by 9 digits (for example: `D000000001`). Thus, the data type for `Code` is `Char(10)`. We choose the data type `Char` instead of `Varchar` to indicate that this unique code is a string with fixed number of characters so that any mistake relating to the inaccurate number of characters can be detected.

- **Last_Name:** The value for this column is a string containing surname and middle name of the doctor, so we will choose the data type `Text`, which is a string with variable length in case the doctor has long middle name or no middle name. For example, if a doctor's name is "Nguyen Van A", "Nguyen Van" will be the value for column **Last_Name**.

- **First_Name:** The value for this column is a string containing only given name of the doctor, and we also use the data type `Text` here to offer more flexibility for the string's length. For example, the value in column **First_Name** of the doctor "Nguyen Van A" is "A".

- **Date_Of_Birth:** This column contains values on the birthdate of the doctor, so we will choose the data type `Date` here.

- **Gender:** This column contains information about the gender of the doctor. We will use "F" to denote female doctor and "M" to denote male doctor. Thus, the data type use here is `Char`, which is a single character. We also add the constraint `CHECK(Gender = 'F' or Gender = 'M')` to check whether the input for gender is valid or not.

- **Address:** This column contains information about the address of the doctor, so we will use the data type `Text` to offer more flexibility for the string's length.

- **Start_Date:** This column contains information about the first date that the doctor started to work in the hospital, so we will use the data type `Date` here.
- **Specialty_Name:** This column contains information about name of the specialty of the doctor, and we will use the data type `Text` here.
- **Degree_Year:** This column contains information about the year in which the specialty degree of the doctor was awarded. We only need to store the year here, so we will use the data type `Smallint` instead of other data types about date and time. Besides, we also have the constraint to check if the value for degree year is between 0 and the current year, as the value for year cannot be negative or larger than the current year.
- **End_Date:** This column contains information about the date on which the doctor stops working in the hospital, so we will use the data type `Date` here. The value for this column can be null if the doctor is still working in the hospital.
- **Is_Working:** The value in this column is a flag to indicate whether the doctor is still working in the hospital at the moment or not. Therefore, the data type chosen here is `Bool`. If the doctor no longer works in the hospital, the value in this column is `False` and vice versa.
- **Dept_Code:** The value in this column is the code of the department which the doctor belongs to. We will explain this in more detail in section 1.5.

Later on, after creating the table `Department`, which contains information about the departments in the hospital, we will add foreign key `Dept_Code` in table `Doctor` which references from table `Department`.

```
ALTER TABLE Doctor
ADD FOREIGN KEY (Dept_Code) REFERENCES Department(Code)
ON DELETE SET NULL    ON UPDATE CASCADE;
```

We use the action `On delete set null` because in case the department is disbanded and removed from the database, the `Dept_Code` column of the doctors who used to work for that department will be set to null. Meanwhile, the action `On update cascade` is used for the case when the department change their name and their code, the `Dept_Code` of all doctors working for that department will be updated automatically.

1.4 Table Doctor_Phone

The table `Doctor_Phone` contains information about phone numbers of the doctors. A doctor may have one or many phone numbers. The query to create the table `Doctor_Phone` is shown as below:

```
CREATE TABLE Doctor_Phone(
  Nurse_Code        CHAR(10),
  Phone_Number      VARCHAR(11),
```

```
  PRIMARY KEY (Doctor_Code, Phone_Number),
  FOREIGN KEY (Doctor_Code) REFERENCES Doctor(Code)
      ON DELETE RESTRICT   ON UPDATE RESTRICT
);
```

- **Doctor_Code:** The value in this column is the unique code to identify each doctor working in the hospital with the data type `Char(10)` as we have explained in section 1.3. In this table, we do not set the unique constraint for `Doctor_Code` because one doctor can have many phone numbers, so the values for `Doctor_Code` can be duplicated.
- **Phone_Number:** This column contains phone numbers of the doctors. In Vietnam, a typical mobile phone number is 10 digits (for example: 0912345678), while a landline phone number consists of 11 digits (for example: 02812345678). Therefore, we choose the data type `Varchar(11)` to indicate that phone numbers are stored as strings with the maximum length is 11 characters.
- **Primary key:** We choose the tuple `(Doctor_Code, Doctor_Number)` as the primary key for this table. `Doctor_Code` can be duplicated as one doctor may have several phone numbers. `Phone_Number` can be duplicated in case several doctors use the same landline phone number. But the tuple `(Doctor_Code, Phone_Number)` is unique.
- **Foreign key:** `Doctor_Code` is the foreign key referencing the column `Code` in table `Doctor` to indicate that these phone numbers belongs to doctors. Every `Doctor_Code` in the table `Doctor_Phone` must exist in the table `Doctor`. We use action `On delete restrict` to prevent the referenced rows in table `Doctor` from being deleted, and `On update restrict` to prevent updating the `Code` column of the referenced rows. We do not want to delete the information about a doctor completely or changing the `Doctor_Code` indiscriminately.

1.5 Table Department

The table `Department` contains information about the departments in the hospital. The query to create the table `Department` is shown as below:

```
CREATE TABLE Department(
  Code              CHAR(2)   NOT NULL   UNIQUE,
  Title             TEXT,
  Dean_Code         CHAR(10),
  PRIMARY KEY (Code),
  FOREIGN KEY (Dean_Code) REFERENCES Doctor(Code)
     ON DELETE RESTRICT   ON UPDATE RESTRICT
);
```

- **Code:** This is the unique code which every department in the hospital must have to identify each of them, so there must be `Not null` and `Unique` constraint for this column. We will also use this code as the primary key for this table. We assume that the code for each department is the abbreviation of the name of that department with 2 characters (for example: the code for department Cardiology is CA), so we choose the data type `Char(2)` here. We choose the data type `Char` instead of `Varchar` to indicate that this unique code is a string with fixed number of characters so that any mistake relating to the inaccurate number of characters can be detected.

- **Title:** This is the column contains the title (or name) of the department. We will use the data type `Text` as the string with flexible length for this column.

- **Dean_Code:** This column contains the code of the doctor who is the dean of that department. We do not put `Not null` constraint here in case a new department has not got a dean yet. The code of doctor is a string with 10 characters as we have explained in section 1.3, so the data type here is `Char(10)`. This is also the foreign key referencing the primary key in table `Doctor` to maintain the dean relationship between `Doctor` and `Department`. We use action `On delete restrict` to prevent the referenced rows in table `Doctor` from being deleted, and `On update restrict` to prevent updating the `Code` column of the referenced rows. We will prevent the information about the current dean of the department from being removed entirely from the database, and prevent changing the code of that dean indiscriminately.

1.6 Table Patient

Table `Patient` contains demographic information about the patients in the hospital The query to create the table `Patient` is shown as below:

```
CREATE TABLE Patient(
  Patient_Number      INT       GENERATED ALWAYS AS IDENTITY,
  Last_Name           TEXT,
  First_Name          TEXT,
  Date_Of_Birth       DATE,
  Gender              CHAR      CHECK(Gender = 'F' or Gender = 'M'),
  Address             TEXT,
  Phone_Number        VARCHAR(11),
  PRIMARY KEY (Patient_Number)
);
```

- **Patient_Number:** This is the unique number to identify each patient in the hospital. We use the data type `Int` and the clause `Generate always as identity` to make PostgreSQL generate the number automatically in ascending order. For example, the `Patient_Number` of the first patient going to the hospital will be 1, and that of the second patient will be 2, and so on. This column is also chosen to

be the primary key for table `Patient`. If the user try to insert or update values into the column `Patient_Number`, the DBMS will raise error.

- **Last_Name:** The value for this column is a string containing surname and middle name of the patient, so we will choose the data type `Text`, which is a string with variable length in case the patient has long middle name or no middle name. For example, if a patient's name is "Vo Van G", "Vo Van" will be the value for column **Last_Name**.
- **First_Name:** The value for this column is a string containing only given name of the patient, and we also use the data type `Text` here to offer more flexibility for the string's length. For example, the value in column **First_Name** of the patient "Vo Van G" is "G".
- **Date_Of_Birth:** This column contains values on the birthdate of the patient, so we will choose the data type `Date` here.
- **Gender:** This column contains information about the gender of the patient. We will use "F" to denote female patient and "M" to denote male patient. Thus, the data type use here is `Char`, which is a single character. We also add the constraint `CHECK(Gender = 'F' or Gender = 'M')` to check whether the input for gender is valid or not.
- **Address:** This column contains information about the address of the patient, so we will use the data type `Text` to offer more flexibility for the string's length.
- **Phone_Number:** This column contains phone number of the patient. In Vietnam, a typical mobile phone number is 10 digits (for example: 0912345678), while a landline phone number consists of 11 digits (for example: 02812345678). Therefore, we choose the data type `Varchar(11)` to indicate that phone numbers are stored as strings with the maximum length is 11 characters.

1.7 Table Outpatient

Table `Outpatient` contains information about the outpatients who receive examination from the doctors without being admitted to the hospital. The query to create the table Patient is shown as below:

```
1 CREATE TABLE Outpatient(
2   Patient_Number      INT         NOT NULL   UNIQUE,
3   Outpatient_Code     CHAR(11)    NOT NULL   UNIQUE    CHECK (
    Outpatient_Code LIKE 'OP%'),
4   PRIMARY KEY (Patient_Number, Outpatient_Code),
5   FOREIGN KEY (Patient_Number) REFERENCES Patient(Patient_Number)
6       ON DELETE RESTRICT    ON UPDATE RESTRICT
7 );
```

- **Patient_Number:** This is the unique number to identify all patients in the hospital as we have explain in section 1.6. Every outpatient must have this number, and each outpatient has 1 and only 1 `Patient_Number`, so we put the constraints `Not null` and `Unique` here.

- **Outpatient_Code:** The value in this column is the unique code to identify each outpatient in the hospital, so there must be `Not null` and `Unique` constraint here. The unique code for an outpatient is a string with 11 characters, starting with "OP" and followed by 9 digits (for example: "OP000000001"). Therefore, we will choose the data type `Char(11)` to specify the string with fixed number of characters so that any mistake relating to the inaccurate number of characters can be detected. We also add the check constraint to verify that the input value for `Outpatient_Code` is in the right form (starting with "OP").

- **Primary key:** We choose the tuple `(Patient_Number, Outpatient_Code)` to be the primary key for this table because `Patient_Number` is unique for each patient and `Outpatient_Code` is unique for each outpatient.

- **Foreign key:** `Patient_Number` is the foreign key referencing the primary key in table `Patient` to indicate that an outpatient is also a patient. Every value for `Patient_Number` in the table `Outpatient` must exist in the table `Patient`. We use action On delete restrict to prevent the referenced rows in table `Patient` from being deleted, and On update restrict to prevent updating the `Patient_Number` column of the referenced rows. We do not allow removing all the information about the patient from the database and updating the automatically-generated `Patient_Number`.

1.8 Table Examination

Table `Examination` contains information about the examinations that the outpatients have with the doctors in the hospital. The query to create the table `Examination` is shown as below:

```
CREATE TABLE Examination(
  Doctor_Code          CHAR(10)    NOT NULL,
  Patient_Number       INT         NOT NULL,
  Outpatient_Code      CHAR(11)    NOT NULL,
  Exam_Timestamp       TIMESTAMP   NOT NULL,
  Next_Exam_Date       DATE,
  Diagnosis            TEXT,
  Fee                  INT         CHECK(Fee >= 0),
  PRIMARY KEY (Doctor_Code, Patient_Number, Outpatient_Code,
   Exam_Timestamp),
  FOREIGN KEY (Doctor_Code) REFERENCES Doctor(Code)
      ON DELETE RESTRICT    ON UPDATE RESTRICT,
  FOREIGN KEY (Patient_Number, Outpatient_Code)
      REFERENCES Outpatient(Patient_Number, Outpatient_Code)
      ON DELETE RESTRICT    ON UPDATE RESTRICT
```

```
15 );
```

- **Doctor_Code:** This column contains the code of the doctor who examines the outpatient. Every examination must have a doctor, so we put `Not null` constraint here. We will not put `Unique` constraint here since one doctor can examine for many outpatients. The data type for `Doctor_Code` is `Char(10)` as we have explain in section 1.3.
- **Patient_Number:** This is the unique number to identify all patients in the hospital as we have explain in section 1.6. Each examination must involve one patient, in particular: outpatient, so there must be `Not null` constraint here. We will not put the `Unique` constraint here because one outpatient can have many examination with one or many doctors, so the values in `Patient_Number` column can be duplicated.
- **Outpatient_Code:** The value in this column is the unique code to identify each outpatient in the hospital as we have explained in section 1.7. The reason for choosing `Not null` constraint but not choosing `Unique` constraint is the same as column `Patient_Number`.
- **Exam_Timestamp:** The value for this column is the date and time of the examination, so we choose the data type `Timestamp` here. For example, if an outpatient has an examination with a doctor on 8^{th} January 2022 at 07:05:25, the value for `Exam_Timestamp` would be `'2022-01-08 07:05:25'`. We have to record the date and time of each examination, so we put the `Not null` constraint here.
- **Next_Exam_Date:** This column contains the date on which the outpatient need to be re-examined according to the indication of the doctor, so we use the data type `Date` here. Some patients do not need to be re-examinated, so the value in this column can be null and we will not put the `Not null` constraint here.
- **Diagnosis:** This column contains the string indicating the diagnosis that the doctor has made, so we choose the data type `Text` here so that the string's length can be flexible.
- **Fee:** The value in this column is the money the outpatient has to pay for this examination (excluding the fee for medication). The smallest unit for Vietnamese currency is dong (VND) and the values for VND are non-negative integers, so we use the data type `Int` with the constraint `Check(Fee >= 0)`.
- **Primary key:** We choose the tuple `(Doctor_Code, Patient_Number, Outpatient_Code, Exam_Timestamp)` to be the primary key for the table `Examination`, because in the EERD, we have design `Examination` as a weak entity type that depends on `Doctor` and `Outpatient`, with the partial key `Exam_Timestamp` to distinguish the examinations within the same doctor and outpatient. So the primary key of `Examination` will be the combination of the primary key of `Doctor`, the primary key of `Outpatient` and the partial key `Exam_Timestamp`.

- **Foreign key:** There are 2 tables which the table `Examination` references to, namely `Doctor` and `Outpatient`.
  - `Doctor_Code` is the foreign key referencing the primary key in table `Doctor` to indicate that a doctor involves in this examination. Every value for `Doctor_Code` in the table `Examination` must exist in the table `Doctor`. We use action On delete restrict to prevent the referenced rows in table `Doctor` from being deleted, and On update restrict to prevent updating the primary key of the referenced rows. We do not allow removing all the information about the doctor from the database or changing `Doctor_Code` because there are some cases in which we need to retrieve the information of a doctor who involved in a particular examination but he or she no longer works in the hospital.
  - `(Patient_Number, Outpatient_Code)` is the foreign key referencing the primary key in table `Outpatient` to indicate that an outpatient has this examination. Every value for `(Patient_Number, Outpatient_Code)` in the table `Examination` must exist in the table `Outpatient`. We use action On delete restrict to prevent the referenced rows in table `Outpatient` from being deleted, and On update restrict to prevent updating the `Patient_Number` and `Outpatient_Code` columns of the referenced rows. We do not allow removing all the information about the outpatient from the database, or updating the automatically-generated `Patient_Number` and the unique code `Outpatient_Code`.

1.9 Table Exam_Medication

Table `Exam_Medication` contains information about the medication used in each examination of the outpatients. The query to create the table `Exam_Medication` is shown as below:

```
CREATE TABLE Exam_Medication(
  Doctor_Code          CHAR(10)    NOT NULL,
  Patient_Number       INT         NOT NULL,
  Outpatient_Code      CHAR(11)    NOT NULL,
  Exam_Timestamp       TIMESTAMP   NOT NULL,
  Medication_Code      CHAR(10),
  Quantity             INT         CHECK(Quantity >= 0),
  PRIMARY KEY (Doctor_Code, Patient_Number, Outpatient_Code,
   Medication_Code, Exam_Timestamp),
  FOREIGN KEY (Doctor_Code, Patient_Number, Outpatient_Code,
   Exam_Timestamp)
    REFERENCES Examination(Doctor_Code, Patient_Number, Outpatient_Code
   , Exam_Timestamp)
    ON DELETE RESTRICT    ON UPDATE RESTRICT
);
```

- **Doctor_Code:** This column contains the code of the doctor who examines the outpatient. Every examination must have a doctor, so we put `Not null` constraint here. We will not put `Unique` constraint here since one doctor can examine for many outpatients. The data type for `Doctor_Code` is `Char(10)` as we have explain in section 1.3.

- **Patient_Number:** This is the unique number to identify all patients in the hospital as we have explain in section 1.6. Each examination must involve one patient, in particular: outpatient, so there must be `Not null` constraint here.

- **Outpatient_Code:** The value in this column is the unique code to identify each outpatient in the hospital as we have explained in section 1.7. The reason for choosing `Not null` constraint is the same as column `Patient_Number`.

- **Exam_Timestamp:** The value for this column is the date and time of the examination. The data type `Timestamp` and the constraint `Not null` have been explained in section 1.8.

- **Medication_Code:** This is the unique code of the medication used in this examination. The data type of this column is `Char(10)`, which will be explained in more detail in section 1.14.

- **Quantity:** The value for this column is the quantity of the medication used. This value is a non-negative integer, so we choose the data type `Int` with the check constraint to ensure that `Quantity` is greater or equal to 0. For example, if 10 pills of Paracetamol are used in a particular examination, the value for `Quantity` column will be 10.

- **Primary key:** We choose the tuple `(Doctor_Code, Patient_Number, Outpatient_Code, Medication_Code, Exam_Timestamp)` as the primary key for the table `Exam_Medication`. The value of each single attribute in this tuple can be duplicated, but the value of each tuple is unique.

- **Foreign key:** The tuple `(Doctor_Code, Patient_Number, Outpatient_Code, Exam_Timestamp)` is the foreign key referencing the primary key in table `Examination` to maintain the relationship between `Exam_Medication` and `Examination`. We use action On delete restrict to prevent the referenced rows in table `Examination` from being deleted, and On update restrict to prevent updating the primary key of the referenced rows. We will prevent the information about examination of outpatients from being removed entirely from the database, and make the primary key of the referenced rows fixed.

Later on, after creating the table `Medication`, which contains information about the medications in the hospital, we will add foreign key `Medication_Code` which references to table `Medication`.

```
ALTER TABLE Exam_Medication
ADD FOREIGN KEY (Medication_Code) REFERENCES Medication(Code)
ON DELETE RESTRICT   ON UPDATE RESTRICT;
```

We use the action `On delete restrict` to prevent removing the information about the medication which has been used from the database, and `On update restrict` to prevent changing the identifying code of the medication indiscriminately.

1.10 Table Inpatient

Table `Inpatient` contains information about the inpatients who are admitted to the hospital, receiving treatments from doctors and being taken care by nurses. The query to create the table Inpatient is shown as below:

```
CREATE TABLE Inpatient(
  Patient_Number      INT         NOT NULL   UNIQUE,
  Inpatient_Code      CHAR(11)    NOT NULL   UNIQUE    CHECK (
   Inpatient_Code LIKE 'IP%'),
  PRIMARY KEY (Patient_Number, Inpatient_Code),
  FOREIGN KEY (Patient_Number) REFERENCES Patient(Patient_Number)
      ON DELETE RESTRICT    ON UPDATE RESTRICT
);
```

- **Patient_Number:** This is the unique number to identify all patients in the hospital as we have explain in section 1.6. Every inpatient must have this number, and each inpatient has 1 and only 1 `Patient_Number`, so we put the constraints `Not null` and `Unique` here.
- **Inpatient_Code:** The value in this column is the unique code to identify each inpatient in the hospital, so there must be `Not null` and `Unique` constraint here. The unique code for an inpatient is a string with 11 characters, starting with "IP" and followed by 9 digits (for example: "IP000000001"). Therefore, we will choose the data type `Char(11)` to specify the string with fixed number of characters so that any mistake relating to the inaccurate number of characters can be detected. We also add the check constraint to verify that the input value for `Inpatient_Code` is in the right form (starting with "IP").
- **Primary key:** We choose the tuple `(Patient_Number, Inpatient_Code)` to be the primary key for this table because `Patient_Number` is unique for each patient and `Inpatient_Code` is unique for each inpatient.
- **Foreign key:** `Patient_Number` is the foreign key referencing the primary key in table `Patient` to indicate that an inpatient is also a patient. Every value for `Patient_Number` in the table `Inpatient` must exist in the table `Patient`. We use action On delete restrict to prevent the referenced rows in table `Patient` from being deleted, and On update restrict to prevent updating the `Patient_Number` column of

the referenced rows. We do not allow removing all the information about the patient from the database and updating the automatically-generated `Patient_Number`.

1.11 Table Admission

Table `Admission` contains information about each time the inpatient is admitted to the hospital. The query to create the table `Admission` is shown as below:

```
CREATE TABLE Admission(
  Patient_Number      INT         NOT NULL,
  Inpatient_Code      CHAR(11)    NOT NULL,
  Admission_Timestamp TIMESTAMP   NOT NULL,
  Nurse_Code          CHAR(10),
  Diagnosis           TEXT,
  Sick_Room           CHAR(6),
  Recovered           BOOL        DEFAULT FALSE,
  Fee                 INT         CHECK(Fee >= 0),
  Discharge_Timestamp TIMESTAMP,
  PRIMARY KEY (Patient_Number, Inpatient_Code, Admission_Timestamp),
  FOREIGN KEY (Patient_Number, Inpatient_Code) REFERENCES Inpatient(
    Patient_Number, Inpatient_Code)
      ON DELETE RESTRICT    ON UPDATE RESTRICT,
  FOREIGN KEY (Nurse_Code) REFERENCES Nurse(Code)
      ON DELETE RESTRICT    ON UPDATE RESTRICT
);
```

- **Patient_Number:** This is the unique number to identify all patients in the hospital as we have explain in section 1.6. Each admission must involve one patient, in particular: inpatient, so there must be `Not null` constraint here. We will not put the `Unique` constraint here because one inpatient can have many admissions if he or she is admitted to the hospital many times, so the values in `Patient_Number` column can be duplicated.

- **Inpatient_Code:** The value in this column is the unique code to identify each outpatient in the hospital as we have explained in section 1.10. The reason for choosing `Not null` constraint but not choosing `Unique` constraint is the same as column `Patient_Number`.

- **Admission_Timestamp:** The value for this column is the date and time of the admission, so we choose the data type `Timestamp` here. For example, if an inpatient is admitted on 22^{th} January 2022 at 06:30:55, the value for `Admission_Timestamp` would be `'2022-01-22 06:30:55'`. We have to record the date and time of each admission, so we put the `Not null` constraint here.

- **Nurse_Code:** This column contains the code of the nurse who takes care of the inpatient. Every inpatient must be taken care of by a nurse, so we put `Not null` constraint here. We will not put `Unique` constraint here since one nurse can take

care many inpatients. The data type for `Nurse_Code` is `Char(10)` as we have explain in section 1.1.

- **Diagnosis:** This column contains the string indicating the diagnosis that the doctor has made, so we choose the data type `Text` here so that the string's length can be flexible.
- **Sick_room:** This column contains the information about the room where the inpatient stays during the time he or she is admitted to the hospital. We assume that the format for the sick room is a string with 6 characters, the first 2 characters are the building code, the third character is a hyphen, the last 3 characters are the room number. For example, the sick room for the inpatient 1 is "B1-404". Therefore, we will use the data type `Char(6)` here.
- **Recovered:** This column is a flag to indicate whether the inpatient has recovered or not, so we use the data type `Bool`. When first being admitted to the hospital, the inpatient is ill so he or she is not recovered yet. Therefore, by default, the value for `Recovered` column is `False`.
- **Fee:** The value in this column is the money the inpatient has to pay during the time he or she stays at the hospital (excluding the fee for medication). The smallest unit for Vietnamese currency is dong (VND) and the values for VND are non-negative integers, so we use the data type `Int` with the constraint `Check(Fee >= 0)`.
- **Discharge_Timestamp**: The value for this column is the date and time when the inpatient is discharged from the hospital after he or she has recovered, so we use the data type `Timestamp`. The `Discharge_Timestamp` can be null if the inpatient has not recovered.
- **Primary key:** We choose the tuple `(Patient_Number, Inpatient_Code, Admission_Timestamp)` to be the primary key for the table `Examination`. One inpatient can have many admissions, and at a particular timestamp there may be many inpatients being admitted, but each tuple of `(Patient_Number, Inpatient_Code, Admission_Timestamp)` is unique.
- **Foreign key:** There are 2 tables which the table `Table Admission` references to, namely `Nurse` and `Inpatient`.
  - `Nurse_Code` is the foreign key referencing the primary key in table `Nurse` to indicate that a nurse takes care of the inpatient in this admission. Every value for `Nurse_Code` in the table `Admission` must exist in the table `Nurse`. We use action On delete restrict to prevent the referenced rows in table `Nurse` from being deleted, and On update restrict to prevent updating the `Nurse` column of the referenced rows. We do not allow removing all the information about the nurse from the database or changing `Nurse_Code` because there are some

cases in which we need to retrieve the information of a nurse who took care of a particular inpatient but he or she no longer works in the hospital.

– `(Patient_Number, Inpatient_Code)` is the foreign key referencing the primary key in table `Inpatient` to indicate that an inpatient has this examination. Every value for `(Patient_Number, Inpatient_Code)` in the table `Admission` must exist in the table `Inpatient`. We use action On delete restrict to prevent the referenced rows in table `Outpatient` from being deleted, and On update restrict to prevent updating the `Patient_Number` and `Outpatient_Code` columns of the referenced rows. We do not allow removing all the information about the inpatient from the database, or updating the automatically-generated `Patient_Number` and the unique code `Outpatient_Code`.

1.12 Table Treatment

Table Treatment contains information about the treatments that the inpatients receive from the doctors in the hospital during the time they are admitted. The query to create the table Treatment is shown as below:

```
CREATE TABLE Treatment(
  Doctor_Code          CHAR(10)    NOT NULL,
  Patient_Number       INT         NOT NULL,
  Inpatient_Code       CHAR(11)    NOT NULL,
  Admission_Timestamp  TIMESTAMP   NOT NULL,
  Start_Timestamp      TIMESTAMP,
  End_Timestamp        TIMESTAMP,
  Result_              TEXT,
  PRIMARY KEY (Doctor_Code, Patient_Number, Inpatient_Code,
   Admission_Timestamp, Start_Timestamp),
  FOREIGN KEY (Doctor_Code) REFERENCES Doctor(Code)
      ON DELETE RESTRICT    ON UPDATE RESTRICT,
  FOREIGN KEY (Patient_Number, Inpatient_Code, Admission_Timestamp)
      REFERENCES Admission(Patient_Number, Inpatient_Code,
   Admission_Timestamp)
      ON DELETE RESTRICT    ON UPDATE RESTRICT
);
```

- **Doctor_Code:** This column contains the code of the doctor who treats the inpatient. Every treatment must have a doctor, so we put `Not null` constraint here. We will not put `Unique` constraint here since one doctor can treat many inpatients. The data type for `Doctor_Code` is `Char(10)` as we have explain in section 1.3.
- **Patient_Number:** This is the unique number to identify all patients in the hospital as we have explain in section 1.6. Each admission must involve one patient, in particular: inpatient, so there must be `Not null` constraint here. We will not put the `Unique` constraint here because one inpatient can have many admissions if he or she is admitted to the hospital many times, so the values in `Patient_Number` column can be duplicated.

- **Inpatient_Code:** The value in this column is the unique code to identify each outpatient in the hospital as we have explained in section 1.10. The reason for choosing `Not null` constraint but not choosing `Unique` constraint is the same as column `Patient_Number`.
- **Admission_Timestamp:** The value for this column is the date and time of the admission, so we choose the data type `Timestamp` here. For example, if an inpatient is admitted on 22^{th} January 2022 at 06:30:55, the value for `Admission_Timestamp` would be `'2022-01-22 06:30:55'`. We have to record the date and time of each admission, so we put the `Not null` constraint here.
- **Start_Timestamp** The value for this column is the date and time when the treatment starts, so we choose the data type `Timestamp` here.
- **End_Timestamp** The value for this column is the date and time when the treatment ends, so we choose the data type `Timestamp` here.
- **Result_:** This column contains information about the result of the treatment, so we use data type `Text` to specify the string with flexible length.
- **Primary key:** We choose the tuple `(Doctor_Code, Patient_Number, Inpatient_Code, Admission_Timestamp, Start_Timestamp)` as the primary key for the table `Treatment`. The value of each single attribute in this tuple can be duplicated, but the value of each tuple is unique.
- **Foreign key:** There are 2 tables which the table `Table Treatment` references to, namely `Doctor` and `Admission`.
  - `Doctor_Code` is the foreign key referencing the primary key in table `Doctor` to indicate that a doctor involves in this treatment. Every value for `Doctor_Code` in the table `Treatment` must exist in the table `Doctor`. We use action On delete restrict to prevent the referenced rows in table `Doctor` from being deleted, and On update restrict to prevent updating the `Doctor` column of the referenced rows. We do not allow removing all the information about the doctor from the database or changing `Doctor_Code` because there are some cases in which we need to retrieve the information of a doctor who involved in a particular treatment but he or she no longer works in the hospital.
  - `(Patient_Number, Inpatient_Code, Admission_Timestamp)` is the foreign key referencing the primary key in table `Admission` to indicate that this treatment belongs to a particular admission. Every value for `((Patient_Number, Inpatient_Code, Admission_Timestamp)` in the table `Treatment` must exist in the table `Admission`. We use action On delete restrict to prevent the referenced rows in table `Admission` from being deleted, and On update restrict

to prevent updating the primary key of the referenced rows. We do not allow removing all the information about the inpatient from the database, while making the primary key of the referenced rows fixed.

1.13 Table Treatment_Medication

The table `Treatment_Medication` contains information about the medication used in each treatment of the inpatients. The query to create the table `Treatment_Medication` is shown as below:

```
CREATE TABLE Treatment_Medication(
  Doctor_Code          CHAR(10)    NOT NULL,
  Patient_Number       INT         NOT NULL,
  Inpatient_Code       CHAR(11)    NOT NULL,
  Admission_Timestamp  TIMESTAMP   NOT NULL,
  Start_Timestamp      TIMESTAMP,
  Medication_Code      CHAR(10),
  Quantity             INT,
  PRIMARY KEY (Doctor_Code, Patient_Number, Inpatient_Code,
   Admission_Timestamp, Start_Timestamp, Medication_Code),
  FOREIGN KEY (Doctor_Code, Patient_Number, Inpatient_Code,
   Admission_Timestamp, Start_Timestamp)
      REFERENCES Treatment(Doctor_Code, Patient_Number, Inpatient_Code,
    Admission_Timestamp, Start_Timestamp)
      ON DELETE RESTRICT    ON UPDATE RESTRICT
);
```

- **Doctor_Code:** This column contains the code of the doctor who treats the inpatient. Every treatment must have a doctor, so we put `Not null` constraint here. We will not put `Unique` constraint here since one doctor can treat many inpatients. The data type for `Doctor_Code` is `Char(10)` as we have explain in section 1.3.

- **Patient_Number:** This is the unique number to identify all patients in the hospital as we have explain in section 1.6. Each admission must involve one patient, in particular: inpatient, so there must be `Not null` constraint here. We will not put the `Unique` constraint here because one inpatient can have many admissions if he or she is admitted to the hospital many times, so the values in `Patient_Number` column can be duplicated.

- **Inpatient_Code:** The value in this column is the unique code to identify each outpatient in the hospital as we have explained in section 1.10. The reason for choosing `Not null` constraint but not choosing `Unique` constraint is the same as column `Patient_Number`.

- **Admission_Timestamp:** The value for this column is the date and time of the admission, so we choose the data type `Timestamp` here. For example, if an inpatient is admitted on 22^{th} January 2022 at 06:30:55, the value for `Admission_Timestamp`

would be `'2022-01-22 06:30:55'`. We have to record the date and time of each admission, so we put the `Not null` constraint here.

- **Start_Timestamp** The value for this column is the date and time when the treatment starts, so we choose the data type `Timestamp` here.
- **Medication_Code:** This is the unique code of the medication used in this treatment. The data type of this column is `Char(10)`, which will be explained in more detail in section 1.14.
- **Quantity:** The value for this column is the quantity of the medication used. This value is a non-negative integer, so we choose the data type `Int` with the check constraint to ensure that `Quantity` is greater or equal to 0. For example, if 30 pills of Paracetamol are used in a particular treatment, the value for `Quantity` column will be 30.
- **Primary key:** We choose the tuple `(Doctor_Code, Patient_Number, Inpatient_Code, Medication_Code, Admission_Timestamp, Start_Timestamp)` as the primary key for the table `Treatment_Medication`. The value of each single attribute in this tuple can be duplicated, but the value of each tuple is unique.
- **Foreign key:** The tuple `(Doctor_Code, Patient_Number, Inpatient_Code, Admission_Timestamp, Start_Timestamp)` is the foreign key referencing the primary key in table `Treatment` to maintain the relationship between `Treatment_Medication` and `Treatment`. We use action On delete restrict to prevent the referenced rows in table `Treatment` from being deleted, and On update restrict to prevent updating the primary key of the referenced rows. We will prevent the information about treatment of inpatients from being removed entirely from the database, and make the primary key of the referenced rows fixed.

Later on, after creating the table `Medication`, which contains information about the medications in the hospital, we will add foreign key `Medication_Code` which references to table `Medication`.

```
1 ALTER TABLE Treatment_Medication
2 ADD FOREIGN KEY (Medication_Code) REFERENCES Medication(Code)
3 ON DELETE RESTRICT   ON UPDATE RESTRICT;
```

We use the action `On delete restrict` to prevent removing the information about the medication which has been used from the database, and `On update restrict` to prevent changing the identifying code of the medication indiscriminately.

1.14 Table Medication

The table `Medication` contains information about the medications used in the hospital. The query to create the table Medication is shown as below:

```
CREATE TABLE Medication(
  Code              CHAR(10)    NOT NULL    UNIQUE,
  Name_             TEXT,
  Price             INT         CHECK(Price >= 0),
  PRIMARY KEY (Code)
);
```

- **Code:** This is the unique code to identify each medication in the hospital, so there must be `Not null` and `Unique` constraint here. This `Code` is also the primary key for the table `Medication`. We assume that this unique code is a string with 10 characters, the first 7 characters represent the Anatomical therapeutic chemical code (ATC) which is used to classify medication, the 8th character is a hyphen, the last 2 characters are two digits to denote the dose in a unit of that medication. For example, the medication called Paracetamol 500 mg has the `Code` of "N02BE01_01".
- **Name_:** This is the column contains the name of the medication, for example "Paracetamol 500 mg". We will use the data type Text as the string with flexible length for this column.
- **Price:** The value in this column is the price for the smallest unit of the medication. For example, if a small box of Paracetamol 500 mg contains 10 packs, each pack contains 10 pills, then the value in `Price` column is the price for one pill. Besides, this is the price that the hospital sell to the patients, not the original imported price. The smallest unit for Vietnamese currency is dong (VND) and the values for VND are non-negative integers, so we use the data type `Int` with the constraint `Check(Fee >= 0)`.

1.15 Table Medication_Effect

Table `Medication_Effect` contains information about the effect of each medication. The query to create the table `Medication_Effect` is shown as below:

```
CREATE TABLE Medication_Effect(
  Code              CHAR(10)    NOT NULL,
  Effect            TEXT,
  PRIMARY KEY (Code, Effect),
  FOREIGN KEY (Code) REFERENCES Medication(Code)
      ON DELETE RESTRICT   ON UPDATE RESTRICT
);
```

- **Code**: This is the unique code to identify each medication in the hospital, with the data type `Char(10)` as we have explained in section 1.14. When storing the effect of a medication, we must know which type of medication that effect belongs to, so there must be `Not null` constraint here. One medication can have many effects, so the values for `Code` column can be duplicated.

- **Effect:** The value of this column is the string indicating the effect of a particular medication, such as "relieve pain" or "reduce fever". We choose the data type `Text` to specify the string with flexible length.
- **Primary key:** We choose the tuple `(Code, Effect)` as the primary key for this table. `Code` can be duplicated as one medication may have several effects. `Effect` can be duplicated in case several medications have the same effect. But the tuple `(Code, Effect)` is unique.
- **Foreign key:** `Code` is the foreign key referencing the column with the same name in table `Medication` to indicate that this effect belongs to which medication. Every `Code` in the table `Medication_Effect` must exist in the table `Medication`. We use action `On delete restrict` to prevent the referenced rows in table `Medication` from being deleted, and `On update restrict` to prevent updating the `Code` column of the referenced rows. We do not want to delete the information about a medication completely or changing the primary key indiscriminately.

1.16 Table Provider

The table `Provider` contains information about the providers who supply medications for the hospital. The query to create the table `Provider` is shown as below:

```
CREATE TABLE Provider(
  Number_          INT    NOT NULL    UNIQUE,
  Name_            TEXT,
  Address          TEXT,
  Phone_Number     VARCHAR(11),
  PRIMARY KEY (Number_)
);
```

- `Number_`: This is the unique number to identify each provider who supplies medications for the hospital, so there must be `Not null` and `Unique` constraints here. The column `Number_` is also the primary key for the table `Provider`. We assume that the values for `Number_` are integers, so we choose the data type `Int` here. For example, the first medication provider of the hospital will be mark as 1, the next is number 2, and so on.
- **Name_:** This is the column contains the name of the provider, for example "Sanofi Vietnam". We will use the data type Text as the string with flexible length for this column.
- **Address:** This column contains information about the address of the provider, so we will use the data type `Text` to offer more flexibility for the string's length.
- **Phone_Number:** This column contains phone number of the provider. In Vietnam, a typical mobile phone number is 10 digits (for example: 0912345678), while a

landline phone number consists of 11 digits (for example: 02812345678). Therefore, we choose the data type `Varchar(11)` to indicate that phone numbers are stored as strings with the maximum length is 11 characters.

1.17 Table Imported_Medicine

The table `Imported_Medicine` contains information about each type of medication being imported from which provider at different time. The query to create the table `Imported_Medicine` is shown as below:

```
CREATE TABLE Imported_Medicine(
  Medication_Code     CHAR(10)    NOT NULL,
  Provider_Number     INT         NOT NULL,
  Imported_Timestamp  TIMESTAMP   NOT NULL,
  Imported_Price      INT         CHECK(Imported_Price >= 0),
  PRIMARY KEY (Medication_Code, Provider_Number, Imported_Timestamp),
  FOREIGN KEY (Medication_Code) REFERENCES Medication(Code)
      ON DELETE RESTRICT   ON UPDATE RESTRICT,
  FOREIGN KEY (Provider_Number) REFERENCES Provider(Number_)
      ON DELETE RESTRICT   ON UPDATE RESTRICT
);
```

- **Medication_Code:** This is the unique code to identify each medication in the hospital, presented by a string with 10 characters as we have explained in section section 1.14. Thus, we choose the data type `Char(10)` here. When the hospital imports the medication, they must know which type of medication they are provided, so there must be `Not null` constraint here.
- **Provider_Number**: This is the unique number to identify each provider supplying medications for the hospital, with the data type `Int` as we have explained in section 1.16. When the hospital imports the medication, they must know who is the provider, so there must be `Not null` constraint here.
- **Imported_Timestamp:** The value for this column is the date and time the hospital imports medication from the provider, so we use the data type `Timestamp` here. We have to record the timestamp when the medication is imported, so `Not null` constraint is used here.
- **Imported_Price:** The value in this column is the price for the smallest unit of the medication. For example, if a small box of Paracetamol 500 mg contains 10 packs, each pack contains 10 pills, then the value in `Price` column is the price for one pill. Besides, this is the price that the hospital buy medication from the provider. The smallest unit for Vietnamese currency is dong (VND) and the values for VND are non-negative integers, so we use the data type `Int` with the constraint `Check(Fee >= 0)`.

- **Primary key:** We choose the tuple (`Medication_Code, Provider_Number, Imported_Timestamp`) as the primary key for table `Imported_Medicine`. One type of medication can be imported from many providers, one provider can supply many medications, and at a particular time there are many medications imported from many providers. But each tuple of (`Medication_Code, Provider_Number, Imported_Timestamp`) is unique.
- **Foreign key:** There are 2 tables which the table `Imported_Medicine` references to, namely `Medication` and `Provider`.
  - `Medication_Code` is the foreign key referencing the primary key in table `Medication` to indicate which medication is imported. Every value for `Medication_Code` in the table `Imported_Medicine` must exist in the table `Medication`. We use action On delete restrict to prevent the referenced rows in table `Medication` from being deleted, and On update restrict to prevent updating the primary key of the referenced rows. We do not allow removing all the information about the medication from the database or changing the primary key in `Medication` table indiscriminately.
  - `Provider_Number` is the foreign key referencing the primary key in table `Provider` to indicate who provides that medication at that time. Every value for `Provider_Number` in the table `Imported_Medicine` must exist in the table `Provider`. We use action On delete restrict to prevent the referenced rows in table `Provider` from being deleted, and On update restrict to prevent updating the primary key of the referenced rows because in some cases we need to retrieve the information about a provider who provided a type of medication at a particular time, but this provider no longer supplies medications for the hospital.

1.18 Table Medication_Box

Table `Medication_Box` contains information about the medicine boxes that are imported by a provider at a particular time. Every package in a medicine box is the same type, has the same production date and expiration date. The query to create the table `Medication_Box` is shown as below:

```
CREATE TABLE Medicine_Box(
  Medication_Code     CHAR(10)    NOT NULL,
  Provider_Number     INT         NOT NULL,
  Imported_Timestamp  TIMESTAMP   NOT NULL,
  ID                  INT         NOT NULL,
  Expiration_Date     DATE,
  Quantity            INT,
  Out_Of_Date         BOOL        DEFAULT FALSE,
  PRIMARY KEY (Medication_Code, Provider_Number, Imported_Timestamp, ID
   ),
```

```
10   FOREIGN KEY (Medication_Code, Provider_Number, Imported_Timestamp)
11       REFERENCES Imported_Medicine(Medication_Code, Provider_Number,
     Imported_Timestamp)
12       ON DELETE RESTRICT   ON UPDATE RESTRICT
13 );
```

- **Medication_Code:** This is the unique code to identify each medication in the hospital, presented by a string with 10 characters as we have explained in section section 1.14. Thus, we choose the data type `Char(10)` here. When the hospital imports the boxes of medication, they must know which type of medication they are provided, so there must be `Not null` constraint here.
- **Provider_Number**: This is the unique number to identify each provider supplying medications for the hospital, with the data type `Int` as we have explained in section 1.16. When the hospital imports the boxes of medication, they must know who is the provider, so there must be `Not null` constraint here.
- **Imported_Timestamp:** The value for this column is the date and time the hospital imports medication from the provider, so we use the data type `Timestamp` here. We have to record the timestamp when this medicine box is imported, so `Not null` constraint is used here.
- **ID:** This is the partial key to identify each medicine box of the same type, imported from the same provider at the same timestamp. For example, on 19th October 2019, at 15:15:23, the hospital imports 3 big boxes of Paracetamol from the provider STADA. So the ID of the first box is 1, the second box is 2 and the third box is 3. At another timestamp, the hospital imports 2 boxes of Aspirin from provider Agimexpharm, then the ID of the first box is 1 and that of the second box is 2.
- **Expiration_Date:** The value for this column is the date when the medications in that box expire, so we choose the data type `Date` here.
- **Out_Of_Date:** The value for this column is a flag to indicate whether the medications in that box expire or not. The hospital will buy the medications that are still usable from the providers, so by default, the flag `Out_Of_Date` is set to `False`.
- **Primary key:** We choose the tuple `(Medication_Code, Provider_Number, Imported_Timestamp, ID)` as the primary key for the table `Medication_Box`. The value of each single attribute in this tuple can be duplicated, but the value of each tuple is unique.
- **Foreign key:** `(Medication_Code, Provider_Number, Imported_Timestamp)` is the foreign key referencing the primary key in table `Imported_Medicine` to indicate that this medicine box belongs to which importing time. Every value for `(Medication_Code, Provider_Number, Imported_Timestamp)` in the table `Medicine_Box` must exist in the table `Imported_Medicine`. We use action On delete

restrict to prevent the referenced rows in table `Imported_Medicine` from being deleted, and On update restrict to prevent updating the primary key of the referenced rows. We do not allow removing all the information about the medication importing time from the database, while making the primary key of the referenced rows fixed.

2 Store Procedure / Function / SQL

2.1 Increase Inpatient Fee to 10% for all the current inpatients who are admitted to hospital from 01/09/2020.

```
UPDATE Admission
SET Fee = Fee * 1.1
WHERE Discharge_Timestamp IS NULL
    AND Recovered = FALSE
    AND Admission_Timestamp >= '2020-09-01 00:00:00';
```

This UPDATE statement is increasing the admission fee by 10% for all current inpatients that meet certain criteria. Let me explain it in detail:

`UPDATE Admission`

This specifies we are updating existing records in the Admission table.

`SET Fee = Fee * 1.1`

This sets the Fee column to the current Fee value multiplied by 1.1, which is a 10% increase.

`WHERE Discharge_Timestamp IS NULL AND Recovered = FALSE`

The WHERE clause specifies two criteria - only update records where:

1. Discharge_Timestamp is NULL - The patient has not been discharged yet
2. Recovered is FALSE - The patient treatment is still ongoing, they are a current inpatient

`AND Admission_Timestamp >= '2020-09-01 00:00:00'`

An additional filter that the admission must have occurred on or after Sep 1st, 2020. So only current, unrecovered patients admitted after that date will be updated.

2.2 Select all the patients (outpatient and inpatient) of the doctor named 'Nguyen Van A'.

```
SELECT Patient.patient_number, first_name, last_name, date_of_birth,
   gender, Address, Phone_Number,
outpatient.outpatient_code, inpatient.inpatient_code
FROM Patient
left outer join outpatient on Patient.patient_number = outpatient.
   patient_number
```

```
left outer join inpatient on Patient.patient_number = inpatient.
    patient_number
WHERE Patient.patient_number in(
  select patient_number
  from treatment
  where doctor_code in(
    select code
    from doctor
    where last_name ='Nguyen Van' and first_name = 'A'
  )
  union
  select patient_number
  from examination
  where doctor_code in(
    select code
    from doctor
    where last_name ='Nguyen Van' and first_name = 'A'
  )
)
;
```

This SQL query is used to select all patients (both outpatients and inpatients) that are associated with the doctor "Nguyen Van A". Let me explain it step-by-step:

`SELECT Patient.patient_number, first_name, last_name, etc...`

This selects all the columns we want from the Patient table for the matched patients.

`LEFT OUTER JOIN outpatient LEFT OUTER JOIN inpatient`

This joins the outpatient and inpatient tables so we know which patients are outpatients and/or inpatients.

`WHERE Patient.patient_number IN (...)`

This filters the patients returned to only those inside the subquery. This subquery first finds the doctor code for "Nguyen Van A" in the Doctor table. It then finds all patients associated with that doctor code in either the Treatment or Examination tables, and combines them with UNION.

2.3 Write a function to calculate the total medication price a patient has to pay for each treatment or examination. Input: Patient ID Output: A list of payment of each treatment or examination

```
CREATE or replace FUNCTION get_patient_cost(p_no INTEGER)
RETURNS TABLE (
    patient_number_ INT,
    outpatient_code CHAR(11),
    exam_doctor Char(10),
    exam_timestamp timestamp,
    exam_medication Char(10),
```

```
    exam_medication_unit_price INT,
    exam_medication_quantity INT,
    inpatient_code CHAR(11),
    treat_doctor char(10),
    admission_timestamp timestamp,
    treatment_start timestamp,
    treatment_medication Char(10),
    treatment_medication_unit_price INT,
    treatment_medication_quantity int,
    total_price int
)
language plpgsql
AS $$
declare
  examRow record;
  examCur cursor for(
    select * from Exam_Medication, Medication
    WHERE Patient_Number = p_no and Exam_Medication.Medication_Code =
   Medication.Code
  );
  treatRow record;
  treatCur cursor for(
    select * from Treatment_Medication, Medication
    WHERE Patient_Number = p_no and Treatment_Medication.
   Medication_Code = Medication.Code
  );
BEGIN
  open examCur;
  loop
    fetch examCur into examRow;
    exit when not found;
    patient_number_ := examRow.Patient_Number;
    outpatient_code := examRow.Outpatient_Code;
    exam_doctor := examRow.Doctor_Code;
    exam_timestamp := examRow.Exam_Timestamp;
    exam_medication := examRow.Medication_Code;
    exam_medication_unit_price := examRow.Price;
    exam_medication_quantity := examRow.Quantity;
    total_price := exam_medication_unit_price *
   exam_medication_quantity;
    inpatient_code := null;
    treat_doctor := null;
    admission_timestamp := null;
    treatment_start := null;
    treatment_medication := null;
    treatment_medication_unit_price := null;
    treatment_medication_quantity := null;
    return next; --insert a row into return table
  end loop;
  close examCur;
```

```
55
56    open treatCur;
57    loop
58      fetch treatCur into treatRow;
59      exit when not found;
60      patient_number_ := treatRow.Patient_Number;
61      inpatient_code := treatRow.Inpatient_Code;
62      treat_doctor := treatRow.Doctor_Code;
63      admission_timestamp := treatRow.Admission_Timestamp;
64      treatment_start := treatRow.Start_Timestamp;
65      treatment_medication := treatRow.Medication_Code;
66      treatment_medication_unit_price := treatRow.Price;
67      treatment_medication_quantity := treatRow.Quantity;
68      total_price := treatment_medication_unit_price *
     treatment_medication_quantity;
69      outpatient_code := null;
70      exam_doctor := null;
71      exam_timestamp := null;
72      exam_medication := null;
73      exam_medication_unit_price := null;
74      exam_medication_quantity := null;
75      return next; --insert a row into return table
76    end loop;
77    close treatCur;
78  END;
79  $$;
```

The overall purpose is to get a cost breakdown of all medications prescribed for a patient, across their examinations and treatments.

Function definition and returns:

- `CREATE OR REPLACE FUNCTION` - Allows recreating the function
- `get_patient_cost` - Function name
- `p_no INTEGER` - Input parameter representing patient number
- `RETURNS TABLE` - Returns a set of rows rather than single value
- Table contains cost details columns for ease of analysis

Declarations:

- `examRow` and `treatRow`: Record variables to store a row fetched from cursor
- `examCur` and `treatCur`
  - Declare ref cursor to loop through resultsets
  - Cursor queries join Exam_Medication and Treatment_Medication tables to their respective Medication tables

  - So medication details are available in fetched rows

Cursor exam logic:

- Open `examCur`
- Start a loop and fetch into `examRow` variable
- Exit loop if no more rows
- For each row:
  - Extract details like exam date, prescribed medication etc.
  - Calculate total price
  - Insert the row into return table using `RETURN NEXT`
- Close cursor after loop

Cursor treat logic:
Same as above, but fetches treatment details instead of exam details.

- Join Treatment_Medication to Medication
- Fetch treatment rows
- Insert rows into return table

Key advantage:

- Avoid complex SQL query with multiple unions
- Handle row processing and insertion using cursors

2.4 Write a procedure to sort the doctor in increasing number of patients he/she takes care in a period of time. Input: Start date, End date Output: A list of sorting doctors.

```
1 create or replace PROCEDURE sort_doctor(startDate DATE, endDate DATE)
2 LANGUAGE plpgsql
3 AS $$
4 DECLARE
5     doctorRow record;
6     doctorCur cursor(startDate DATE, endDate DATE) for(
7       select dcode, count(*) as numP from(
8         (
9           select doctor.code as dcode, patient.patient_number as pnum,
   examination.exam_timestamp as visit
```

```
10          from doctor, examination, patient
11          where doctor.code = examination.doctor_code and patient.
    patient_number = examination.patient_number
12          and examination.exam_timestamp >= (startDate + time '00:00:00
    ')
13          and examination.exam_timestamp <= (endDate + time '23:59:59')
14        )
15        union
16        (
17          select doctor.code as dcode, patient.patient_number as pnum,
    treatment.admission_timestamp as visit
18          from doctor, treatment, patient
19          where doctor.code = treatment.doctor_code and patient.
    patient_number = treatment.patient_number
20          and treatment.admission_timestamp >= (startDate + time '
    00:00:00')
21          and treatment.admission_timestamp <= (endDate + time '
    23:59:59')
22        )
23      ) as SortDoc
24      group by dcode
25      order by count(*) ASC
26    );
27 begin
28   delete from sort;
29   open doctorCur(startDate DATE, endDate DATE);
30   loop
31     fetch doctorCur into doctorRow;
32     exit when not found;
33     insert into sort(doctor_code, number_of_patients) values (doctorRow
    .dcode, doctorRow.numP);
34   end loop;
35   close doctorCur;
36 end;
37 $$;
```

Procedure details:

1. Declaration:
   - `doctorRow`: A record variable to store information about each doctor.
   - `doctorCur`: A cursor to iterate through the doctor data.
   - The cursor is defined with a subquery that combines data from two tables:
     - `examination`: Contains information about examinations conducted by doctors.
     - `treatment`: Contains information about treatments administered by doctors.
     - The subquery filters the data based on the provided `startDate` and `endDate`.

   - It aggregates the data by doctor code (`dcode`) and counts the number of patients (`numP`).
   - The results are ordered by the number of patients in ascending order.

2. Begin block:

   - Deletes all existing entries from the `sort` table.
   - Opens the `doctorCur` cursor with the provided `startDate` and `endDate`.
   - Enters a loop that iterates through the results returned by the cursor.
     - Fetches the next doctor record into the `doctorRow` variable.
     - Exits the loop if no more records are found.
     - Inserts a new row into the `sort` table for the current doctor, including their `doctor_code` and `number_of_patients`.
   - Closes the `doctorCur` cursor.

Overall, this procedure accomplishes the following:

- It retrieves data about examinations and treatments conducted by doctors within a specific period.
- It groups the data by doctor and counts the number of patients each doctor saw.
- It sorts the doctors by the number of patients in ascending order.
- It stores the sorted list of doctors in the `sort` table.

Points to note:

- The cursor uses the `union` operator to combine data from two separate tables.
- The `time '00:00:00'` and `time '23:59:59'` additions ensure that the time range includes the entire day.
- The `order by count(*) ASC` clause sorts the data by the number of patients in ascending order.
- The procedure first clears the `sort` table before inserting new data.

3 Building Applications

3.1 NextAuth.js

NextAuth.js is a library designed to simplify the authentication process in Next.js applications, using providers as diverse as Google, Facebook, GitHub, and many others. NextAuth.js's operating principle revolves around simplifying authentication in Next.js

applications by providing a flexible and scalable authentication solution. When working with NextAuth.js, a credential provider is a module or service responsible for handling authentication credentials. This can include handling various authentication methods like email/password, social logins, etc. In this assignment, I will use NextAuth creditial provider. User's creditial is then verify with the database for any further actions.

Some of the key benefits of using NextAuth.js:

- Uses tokens for authentication, which improves security and avoids storing sensitive credentials on the client side. The token is signed and validated, ensuring the security of the authentication process.
- Automatically manages sessions and cookies, helping maintain user authentication across requests. This reduces the burden on developers and creates a smooth user experience.
- Designed to work well, integrating with Next.js API routes.
- Provides custom hooks and events, allowing you to add custom logic to the authentication process, helping to flexibly and extend the functionality of NextAuth.js to your specific needs.

Core Concepts Of NextAuth.js:

- Providers - login-specific authentication configurations
- Credentials - how you authenticate by providing a username and password
- When a user authenticates, a user object is created and stored in the session. This object contains information about the user

  NextAuth provides Middleware. Next.js 12 has introduced Middleware. It is a way to run logic before accessing any page, even when they are static. Middleware allows you to run code before a request is completed. Then, based on the incoming request, you can modify the response by rewriting, redirecting, modifying the request or response headers, or responding directly.
  Middleware runs before cached content and routes are matched. This goes well with NextAuth, that ask user to sign in before geting to any routes. In this assignment, i will guard all accessible routes using Middleware.

3.2 Zod

Zod is a TypeScript-first schema declaration and validation library designed to be as developer-friendly as possible. The goal is to eliminate duplicative type declarations.

Zod schemas act as a model or blueprint for the shape of data you expect to receive or work with in your application. You define what fields are required, what types they should be, nesting structures like objects and arrays, custom validation logic, and more.

Some of the key benefits of using Zod:

- Automatic TypeScript type generation from schemas
- Validating data by defining allowed primitive types, shapes, arrays etc.
- Useful error messages when data fails validation
- Small file size and no dependencies

Value for database interactions that Zod provides:

- Input Validation - Before inserting or updating data in a database, you can use Zod to validate that the shape and types of that data match what that table/document expects. This avoids errors from bad data getting written.
- Output Validation - When querying and returning data from a database, you can parse it through a Zod schema to validate the structure before using it in your application and get complete TypeScript types.
- Serialization/Deserialization - The parsing in Zod acts like a serializer/deserializer into a consistent, typed format for storing data in databases or transmitting across the network.
- Domain Modeling - Zod schemas provide a central place to model the domains and data shapes in your codebase, keeping logic consistent and enabling reusable validation.
- Error Handling - On validation failure, Zod provides easy to understand errors including field paths and expected formats - great for consistent error handling.

Core Concepts Of Zod:

- Schemas define permitted structure and validations
- Parsing converts untyped input $\rightarrow$ typed output data
- Validation checks data and throws easy to handle errors
- Inference automatically generates TypeScript types

Ecosystem Integrations

- Express middleware for request/response validation
- String parsing from query strings and JSON requests
- Database adapters like Prisma and Mongoose
- Bundling optimized validations with Esbuild/Vite
- GUI schema builders like @colinhacks/zod-builder

Zod will be a great tool for client-side validation as user inputs data. It will alert the user before the form is sent to the server-side if there is any errors. Errors here are not refer to the correctness of the input, but it refers to the conditions or the type that is assigned to that output!

3.3 User authentication

All route of the page is guarded by NextAuth middleware, thus when a user try to enter any route of the page, they will be redirected to the login page:

```ts
export { default } from "next-auth/middleware";

export const config = { matcher: ["/", "/patients/(.*)",  "/doctors/(.*)"] };
```

Middleware.ts

Manager's acccount is stored in the database in table `dba_account`:

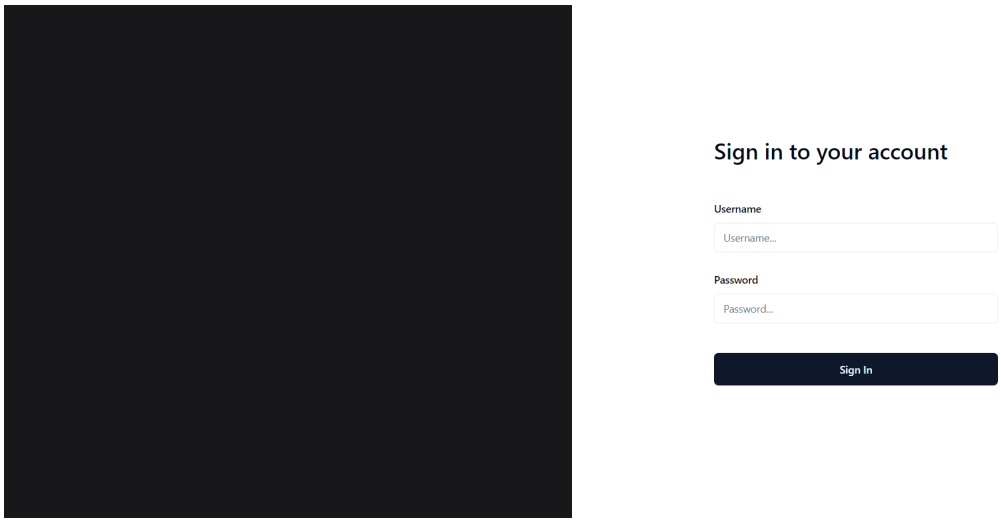
dba_account

#	user_id	username	password
1	1	dbmanage	$2a$06$9Q54VN..MSb2RwaeCCGIXOzoWuFsn7JVtJgIUdsAeXpE79YUIIVVu

dba_account table

Password is not plain text. It is hashed using Blowfish algorithm. This password is generated using `pgcrypto` extension of PostgreSQL as manager register account. We don't implement register function here. We use this table for account verification.

This is our group signin page



Sigin page

As an user sign in, it is first validate in the client-side by Zod. We limit username from 1 - 20 character(s) and limit username from 1 - 50 character(s).

```
const formSchema = z.object({
  username: z.string().min(1, {
    message: "*Username must be at least 1 character.",
  }).max(20, {
    message: "*Username must be at most 20 characters.",
  }),
  password: z.string().min(1, {
    message: "*Password must be at least 1 character.",
  }).max(50, {
    message: "*Username must be at most 50 characters.",
  }),
})
```

Figure 3: Zod schema

Here is one of the error shown as user input invalid (empty username input)

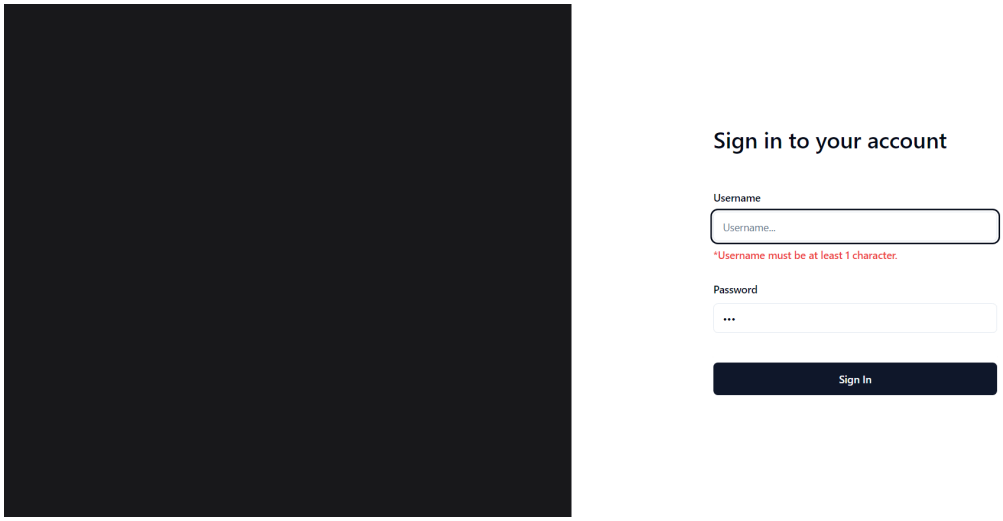


Figure 4: Zod schema

When the user finished the form, the input data is then send to the database for verification. The user's credential is verified via PostgreSQL. The query include: username and password hash comparision.

When user is authenticated, a session is set to indicate success authentication. Here is the session

```
async authorize(credentials, req) {
  const pool = new Pool({
    connectionString: process.env.DATABASE_URL,
  }); // connect to db
  const sql = `
  SELECT * FROM dba_account
  WHERE username = '${credentials?.username}'
  AND password = crypt('${credentials?.password}', password);
  `;
  const { rows } = await pool.query(sql); // run sql
  await pool.end();
  var user;
  if (rows.length > 0) {
    user = {
      id: rows[0].user_id,
      name: rows[0].username,
      email: "",
    };
  } else user = null;
  if (user) return user; else return null;
},
```

Figure 5: Authenticate function

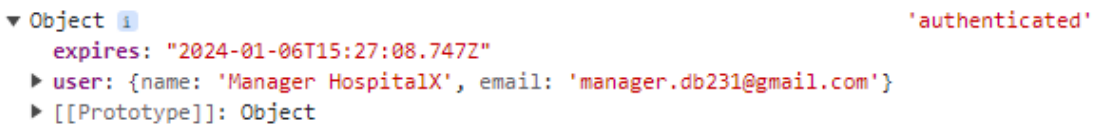


Figure 6: Login session

Session is provided to all router of the page. User is the redirected to the homepage, with the avatar.

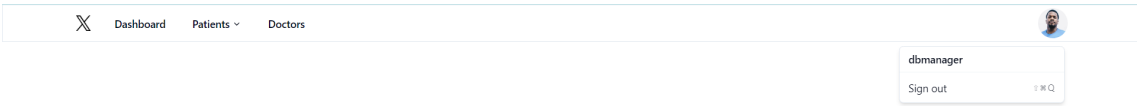


Figure 7: Login successful

If user want to logout, the just need to click on the avatar, then click logout button to logout.

3.4 Search information

To search for patient information, Manager fist enters patient's phone number and patient name. We choose this to be on the search because phone number for each person is unique! And it is also easy to remember. In case patient doesn't have phone number, it is also allow to search by patient name
To open search page, hover on patients then select find patient

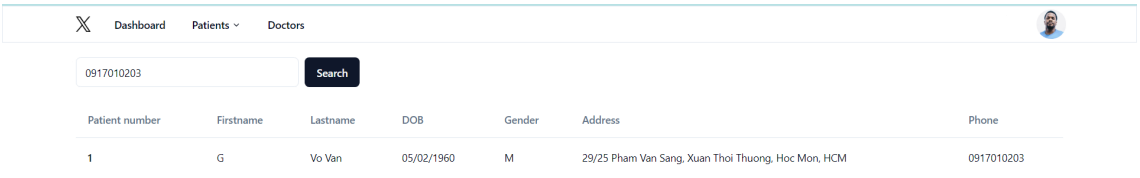


Figure 8: Find patient

Manager just need to input patient's phone or patient's name to the input and click Search. If user existed, patient information is showed up as a row.

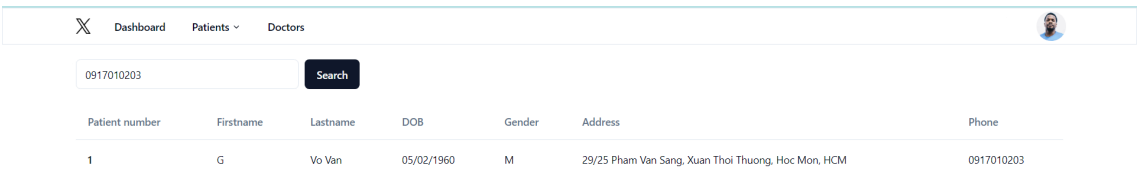


Figure 9: Patient list

```
sql = `SELECT * FROM patient WHERE phone_number = '${phone}'
OR CONCAT(last_name, ' ', first_name) ILIKE '${phone}'
`;
```

Figure 10: Search query

Here is the sql for searching patient by name of phone number:

We use `ILIKE` in order to query without caring lower or uppercase letter because it ignores the case of the letters.

To even view more information, click on the corresponding row. Manager is redirected to a page where full information of patient. It includes: Patient information, admissions, treatments, examinations and medications,...

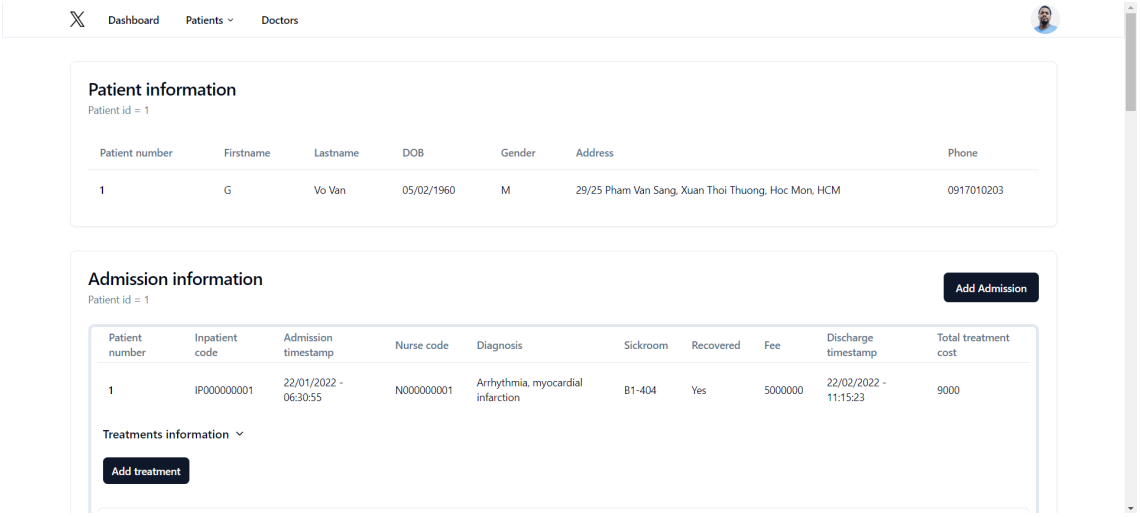


Figure 11: Patient page, first half

Notice that the information page of patient is called via URL, with parameter ID. We use this ID to retrieve the data.

Let's go on details of the patient information page. Firstly is patient information. The row here is same is the one on search information page:

This data is called via sql:

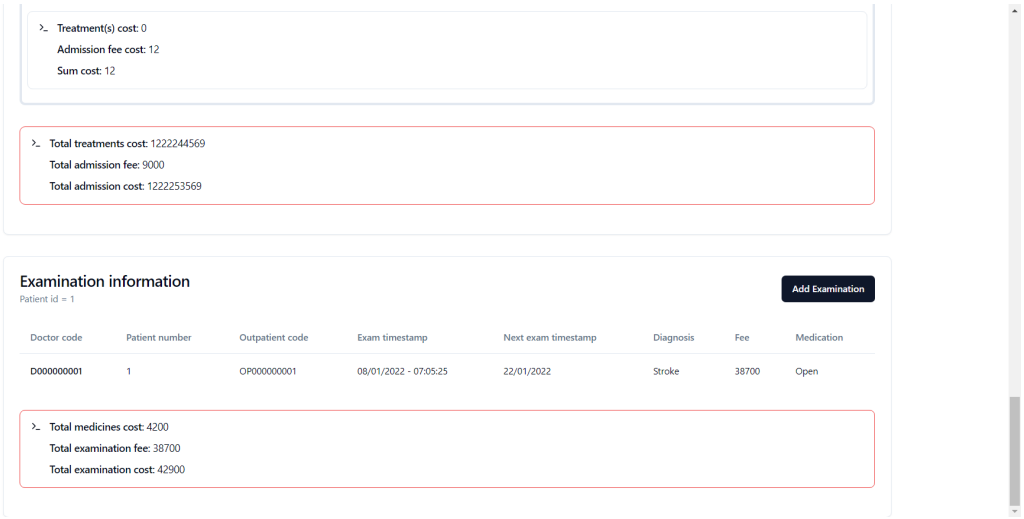


Figure 12: Patient page, second half

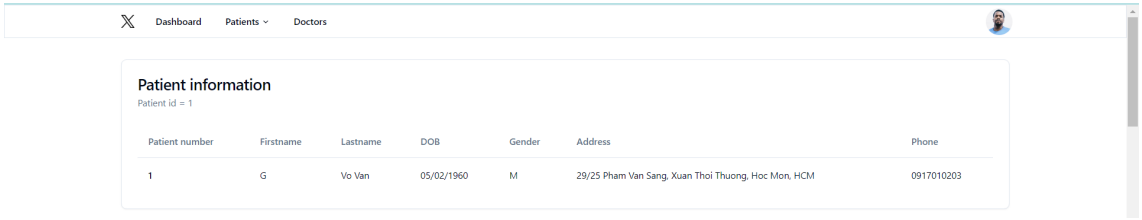


Figure 13: Patient information table

```
sql = `SELECT * FROM patient WHERE patient_number = ${id}`;
```

Figure 14: Select patient information base on ID

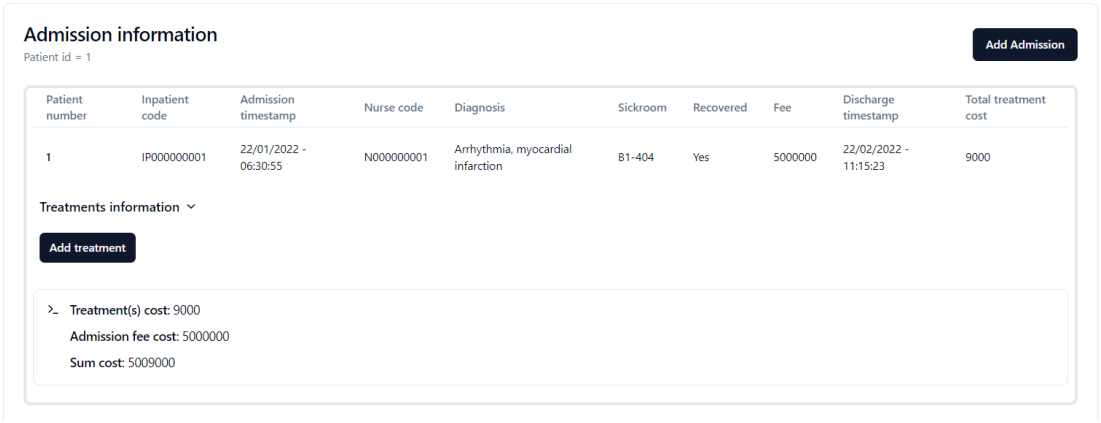


Figure 15: Admission section

Next, we come to admission section of an patient, if this patient is an inpatient, list of admission will be displayed here.

Admission list is called via:

```
  SELECT
    a.*,
    tm.total_value
FROM
    admission a
LEFT JOIN (
    SELECT
        tm.patient_number,
        tm.admission_timestamp,
        SUM(tm.quantity * m.price) AS total_value
    FROM
        treatment_medication tm
    JOIN
        medication m ON tm.medication_code = m.code
    GROUP BY
        tm.patient_number,
        tm.admission_timestamp
) tm ON a.patient_number = tm.patient_number
    AND a.admission_timestamp = tm.admission_timestamp
WHERE
    a.patient_number = ${id};
```

Figure 16: Select admission base on ID

For each admission, there is also a list of treatments.

Treatment list is called via:

For each treatment, there is a column called medication. Click this to view the list of medication associated with that treatment.

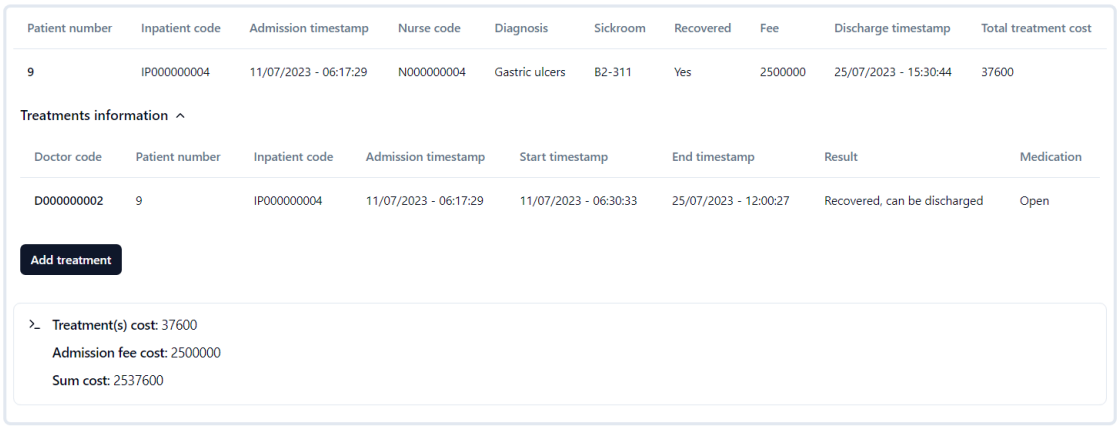


Figure 17: Treatment section

```
SELECT
  *
FROM
  treatment t
JOIN
  admission a ON t.inpatient_code = a.inpatient_code
             AND t.admission_timestamp = a.admission_timestamp
WHERE
  a.patient_number = ${id};
```

Figure 18: Select treatments, display base on ID, admission

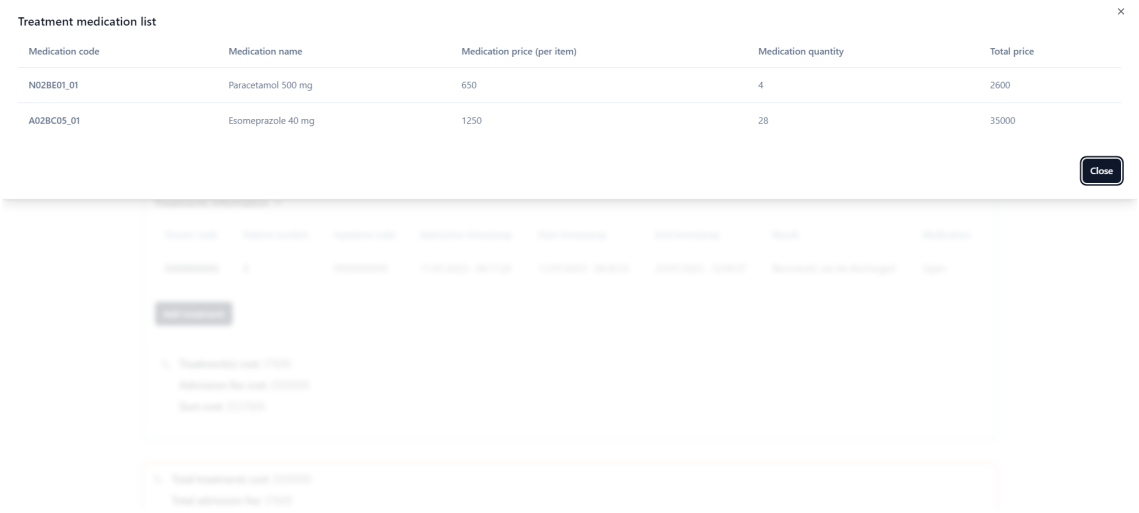


Figure 19: Treatment medication section

```
SELECT
  tm.quantity,
  tm.start_timestamp,
  tm.admission_timestamp,
  m.code,
  m.name_,
  m.price,
  m.price * tm.quantity AS total_value
FROM
    (
        SELECT
            t.inpatient_code,
            t.admission_timestamp,
            t.doctor_code,
            t.start_timestamp,
            t.end_timestamp,
            t.result_,
            tm.medication_code,
            tm.quantity
        FROM
            treatment t
        JOIN
            admission a ON t.inpatient_code = a.inpatient_code
                        AND t.admission_timestamp = a.admission_timestamp
        JOIN
            treatment_medication tm ON t.inpatient_code = tm.inpatient_code
                                    AND t.admission_timestamp = tm.admission_timestamp
                                    AND t.start_timestamp = tm.start_timestamp
        WHERE
            a.patient_number = ${id}
    ) tm
JOIN
    medication m ON tm.medication_code = m.code;
```

Figure 20: Select treatment medication base on ID, treatment

Treatment medication list is call via sql:

Next, we come to examination section of an patient, if this patient is an outpatient, list of examination will be displayed here.

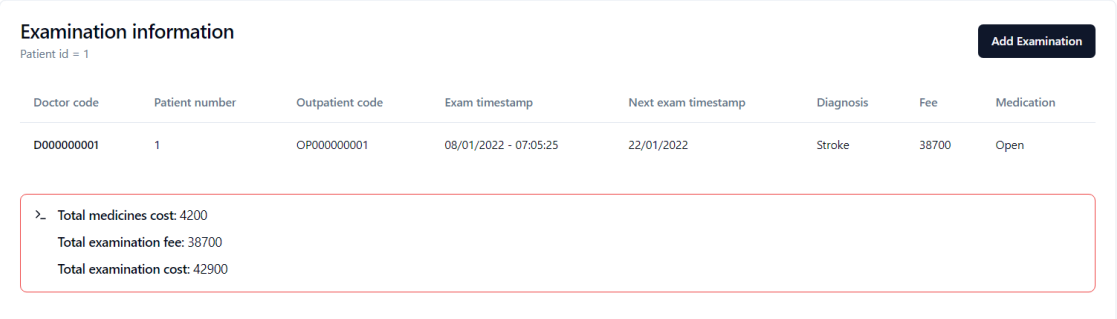


Figure 21: Examination section

Examination list is call via sql:

```
SELECT * FROM examination where patient_number = ${id}
```

Figure 22: Select examination via ID

For each examination, there is a column called medication. Click this to view the list of medication associated with that examination.

Examination medication list is call via sql:

3.5 Add new patient:

To add a new patient, hovers on patients tab. Then click Add Patient.

Manager is then redirect to a page to fill in patient's information. This include:

Figure 23: Examination medication section

```
SELECT
  em.*,
  m.name_,
  m.price,
  em.quantity * m.price AS total_value
FROM
  exam_medication em
JOIN
  medication m ON em.medication_code = m.code
WHERE
  em.patient_number = ${id};
```

Figure 24: Select examination medication via ID, examination

Figure 25: Add patient page

- Firstname, Lastname
- Address
- Phone number
- Date of birth
- Gender

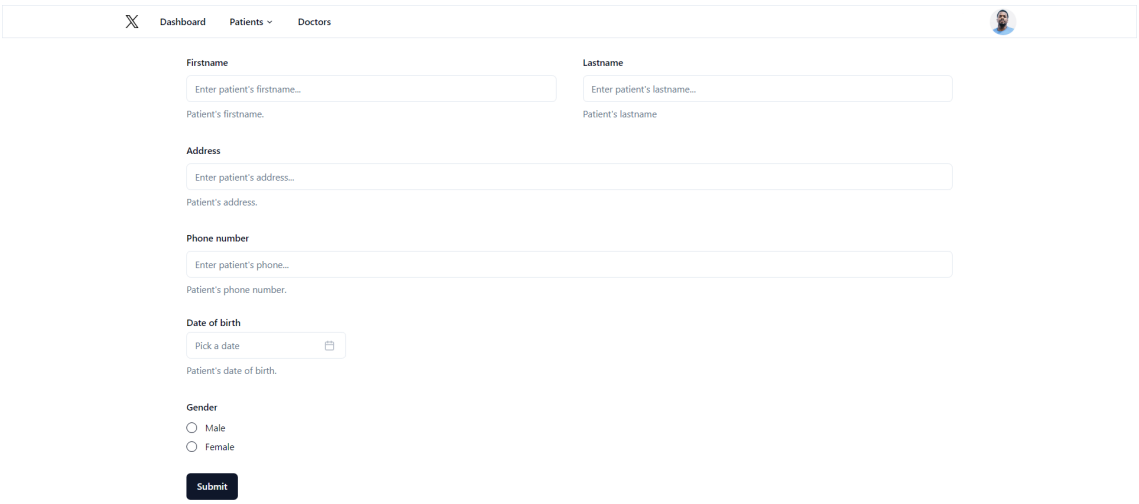


Figure 26: Add patient form

User fill in information. When the form is subnmitted, it is first validate on the client-side by Zod. Here is example of an error submission:

After verified by Zod, the data is sent via POST request:

If all the information is valid. It is devied into 2 cases, patient have phone number:

It then revalidate one more time. This time it is not revalidate due to the valid of the data, but this time it is to the correctness of the data. (check if phone number already existed).

If phone number is sent empty, it write to the database.

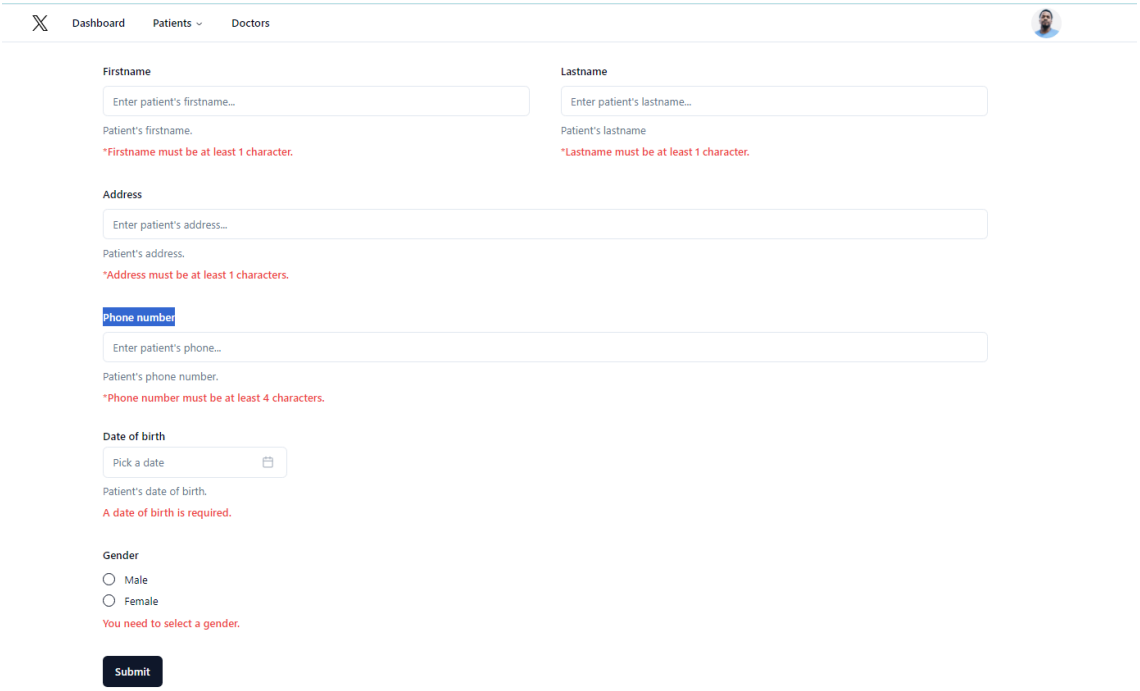


Figure 27: Add patient Zod verification

```
const response = await fetch('/api/patient/add', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    firstname,
    lastname,
    address,
    phone,
    dob,
    gender
  })
});
```

Figure 28: Send form data

```
if (body.phone !== "") {
  const checkphoneSql = `
    SELECT * FROM patient WHERE phone_number = '${body.phone}'
  `
  const { rows: phones } = await pool.query(checkphoneSql);
  if (phones.length === 0) {
    const sql =`
    INSERT INTO Patient(First_Name, Last_Name, Date_Of_Birth, Gender, Address, Phone_Number)
    VALUES(
      '${body.firstname}',
      '${body.lastname}',
      '${body.dob}',
      '${body.gender === 'male' ? 'M' : 'F'}',
      '${body.address}',
      '${body.phone}'
    );
    `
    const { rows } = await pool.query(sql);

    const sql1 =`
    SELECT patient_number FROM patient WHERE phone_number = '${body.phone}'
    `
    const { rows: patient_id } = await pool.query(sql1);


    return Response.json({ res: "success", id: patient_id[0].patient_number }, { status: 200 });
  } else return Response.json({ res: "fail", id: phones[0].patient_number }, { status: 200 });
}
```

Figure 29: Case have phone number

```
const sql =`
INSERT INTO Patient(First_Name, Last_Name, Date_Of_Birth, Gender, Address, Phone_Number)
VALUES(
  '${body.firstname}',
  '${body.lastname}',
  '${body.dob}',
  '${body.gender === 'male' ? 'M' : 'F'}',
  '${body.address}',
  '${body.phone}'
);
`
const { rows } = await pool.query(sql);

const sql1 =`
SELECT MAX(patient_number) AS highest_patient_number
FROM patient;
`
const { rows:maxid } = await pool.query(sql1);

await pool.end();

return Response.json({ res: "success", id: maxid[0].highest_patient_number }, { status: 200 });
```

Figure 30: Case empty phone number

Both will finally return a new id for the patient.

If add successful, a dialog is shown as following. Click 'Continue' to go to that patient information page. The id it is redirected to is taken from the response. If click 'Cancel', user is remain as this page.

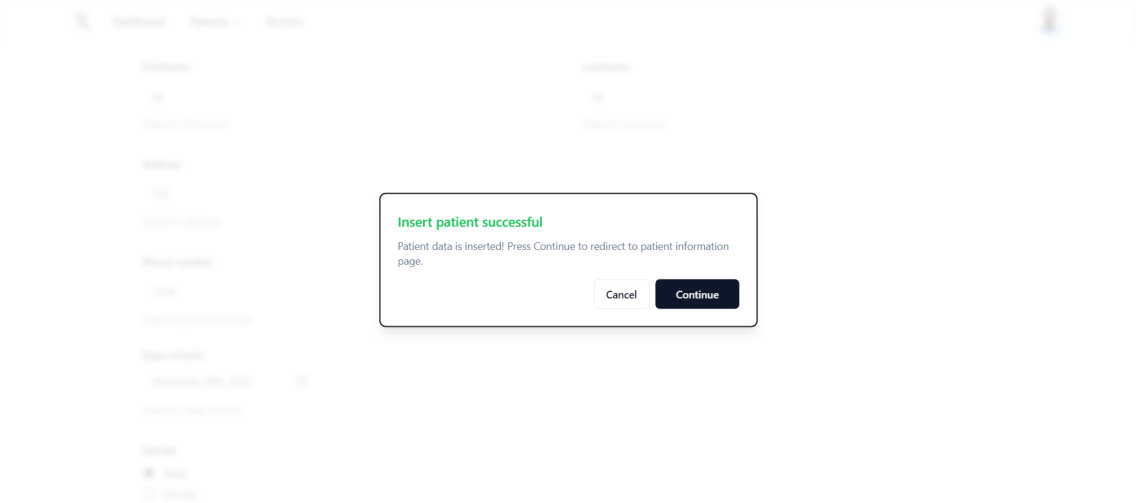


Figure 31: Add patient successful

After click 'Continue', manager is redirect to patient's information page. In here, manager can add patient admission, treatment and examination.

When click on add admission, manager is then redirect to a page to fill in patient's admission information. This include:

- Inpatient code
- Admission timestamp
- Nurse Code
- Diagnosis
- Sickroom
- Recovered
- Fee
- Discharge timestamp

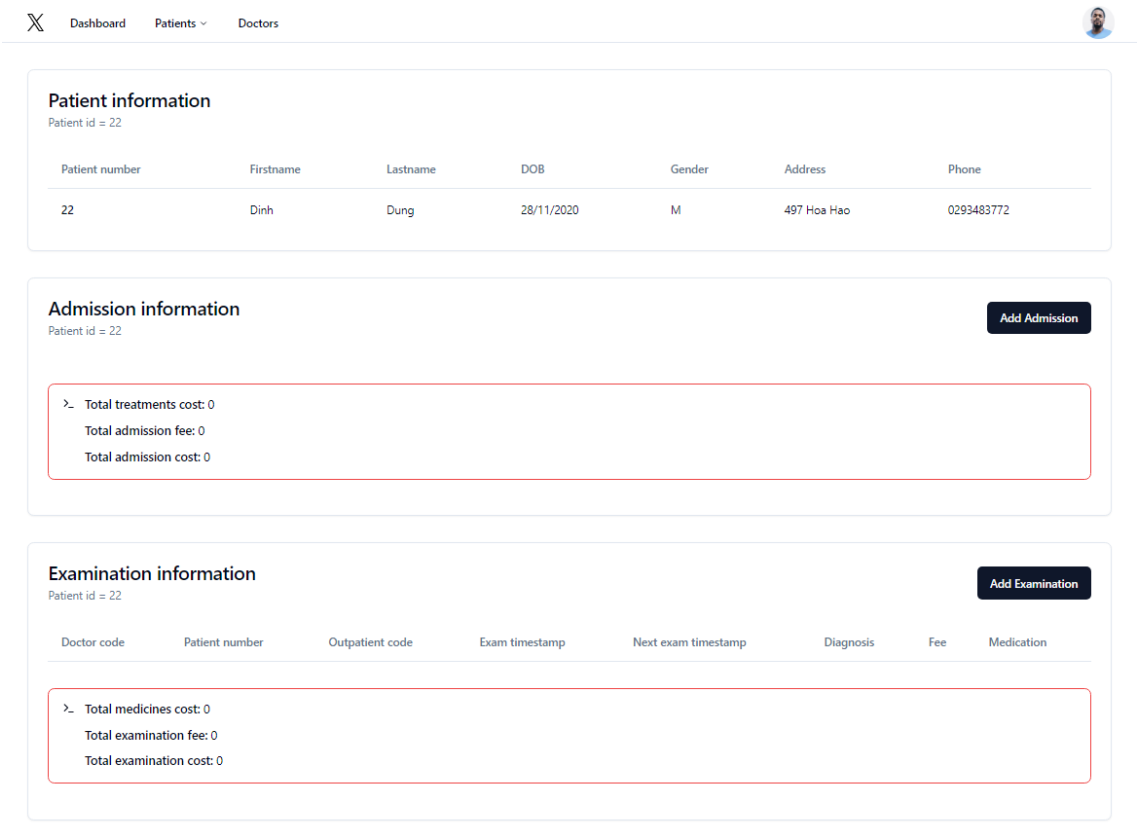


Figure 32: Add patient page

It will check: Is this person have inpatient code, if yes, use it. If not, generate a new code, base on the most recent code and add one unit.

User fill in information. When the form is subnmitted, it is first validate on the client-side by Zod. Here is example of an error submission:

If all the information is valid. It then revalidate one more time. This time it is not revalidate due to the valid of the data, but this time it is to the correctness of the data.

- Nursecode existence
- Admission timestamp is before Discharge timestamp
- This admission is registered

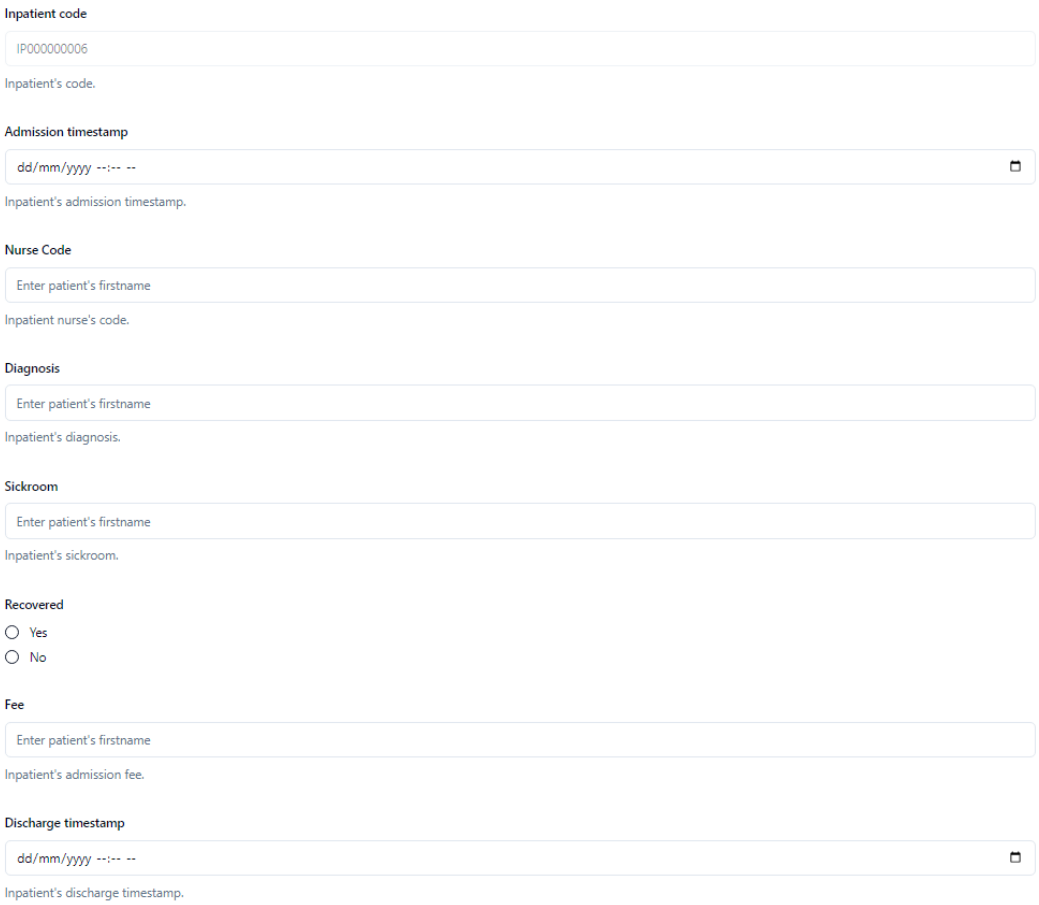


Figure 33: Add admission page

```
const ipcodeSql =`SELECT inpatient_code FROM inpatient where patient_number = ${id}`
const { rows: ipcode } = await pool.query(ipcodeSql);

if (ipcode.length === 0) { // new patient
  const heighestIPSql = `
  SELECT MAX(CAST(SUBSTRING(inpatient_code FROM 3) AS INTEGER)) AS ip_code
  FROM inpatient
  `
  const { rows: heighestID } = await pool.query(heighestIPSql);

  return Response.json({ state: "new", ip: "", maxip: heighestID[0].ip_code }, { status: 200 });

}

const IPcodeSql = `
SELECT inpatient_code AS ip_code
FROM inpatient
WHERE patient_number = ${id};
`
const { rows: IPcode } = await pool.query(IPcodeSql);

await pool.end();

return Response.json({ state: "old", ip: IPcode[0].ip_code, maxip: 0 }, { status: 200 });
```

Figure 34: Check having IP code

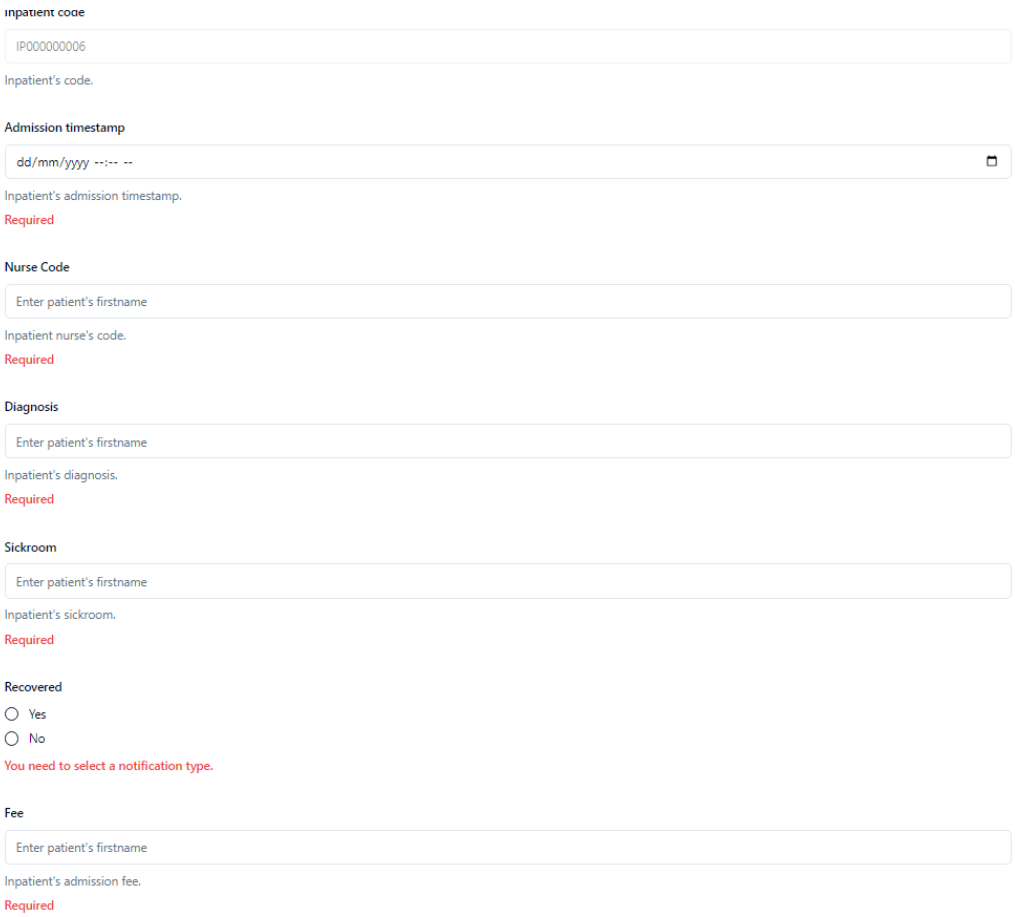


Figure 35: Zod client-side verfication

```
function isStartDateBeforeEndDate(startDateString: string, endDateString: string) {
  const startDate = new Date(startDateString);
  const endDate = new Date(endDateString);
  return startDate <= endDate;
}


export async function POST(req: Request){
  const body = await req.json()
  const pool = new Pool({
    connectionString: process.env.DATABASE_URL,
  });

  const nurseSql =`
  SELECT code FROM nurse WHERE code = '${body.nursecode}'
  `
  const { rows: nurses } = await pool.query(nurseSql);

  if (nurses.length === 0)
    return Response.json({ res: "fail", warning: "Nurse code doesn't exist!" }, { status: 200 });

  if (!isStartDateBeforeEndDate(body.admissiontime, body.dischargetime)) {
    return Response.json({ res: "fail", warning: "Admission timestamp is after Discharge timestamp!" }, { status: 200 });
  }

  const checkValidSql =`
  SELECT * FROM admission WHERE patient_number = ${body.id} AND admission_timestamp = '${body.admissiontime}';
  `
  const { rows: checkValid } = await pool.query(checkValidSql);

  if (checkValid.length > 0)
    return Response.json({ res: "fail", warning: "You already booked an admission at this time!" }, { status: 200 });
```

Figure 36: Revalidate in server side

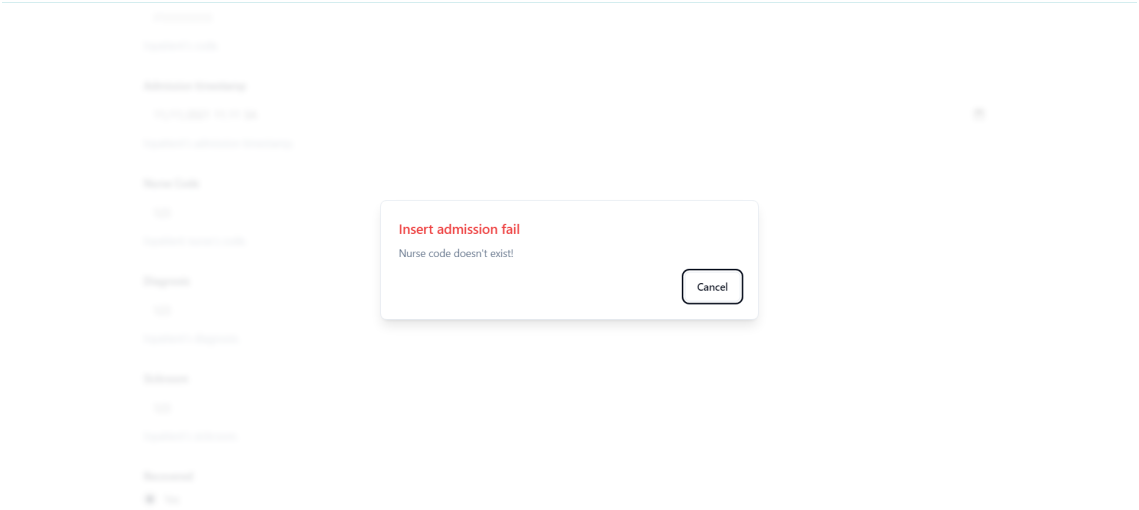


Figure 37: Error when enter non-existing nurse code

```
const ipcodeSql =`SELECT inpatient_code FROM inpatient where patient_number = ${body.id}`
const { rows: ipcode } = await pool.query(ipcodeSql);

if (ipcode.length === 0) { // new patient
  const heighestIPSql = `
  INSERT INTO Inpatient(Patient_Number, Inpatient_Code)
  VALUES(
    ${body.id},
    '${body.ipcode}'
  );
  `
  const { rows: heighestID } = await pool.query(heighestIPSql);

  return Response.json({ state: "new", ip: "", maxip: heighestID[0].ip_code }, { status: 200 });

}
```

Figure 38: Add IP code to `inpatient_table`

If all information is valid, it then add this new record. If this is a new inpatient then it add the IP code the `inpatient_table`.

If all information is valid, manager is alerted. Click 'Continue' to go to that patient information page. If click 'Cancel', user is remain as this page.

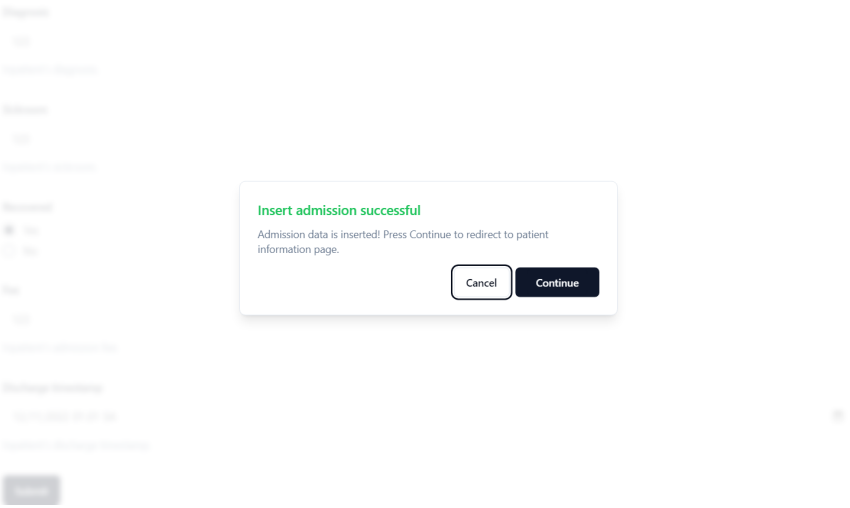


Figure 39: Valid diaglog show up

When redirect back to patient information page, a new admission is add to the table.

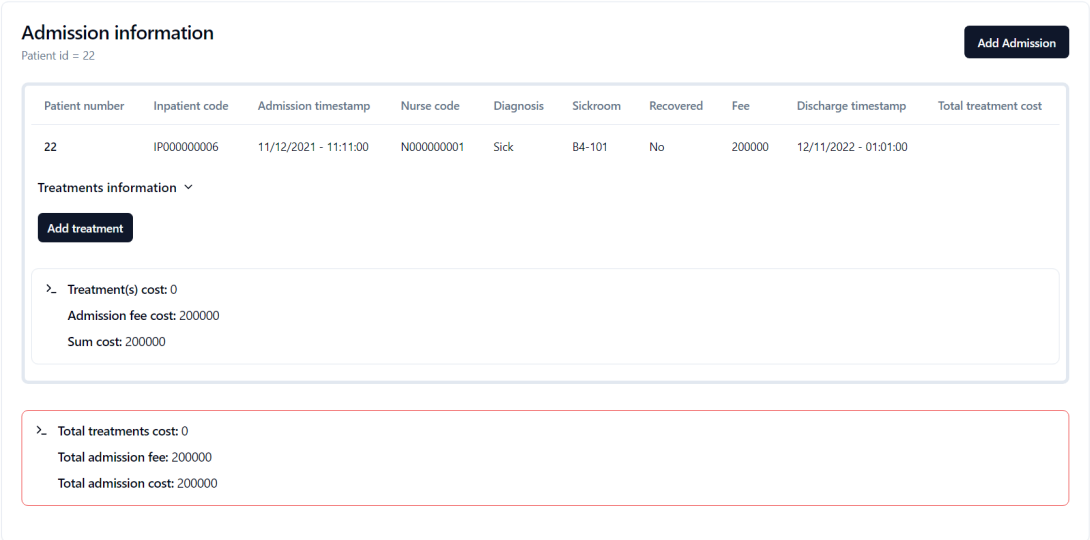


Figure 40: A new admission line

When click on add treatment, manager is then redirect to a page to fill in patient's treatment information. This include:

- Inpatient code
- Admission timestamp
- Doctor Code
- Result
- Start timestamp
- End timestamp

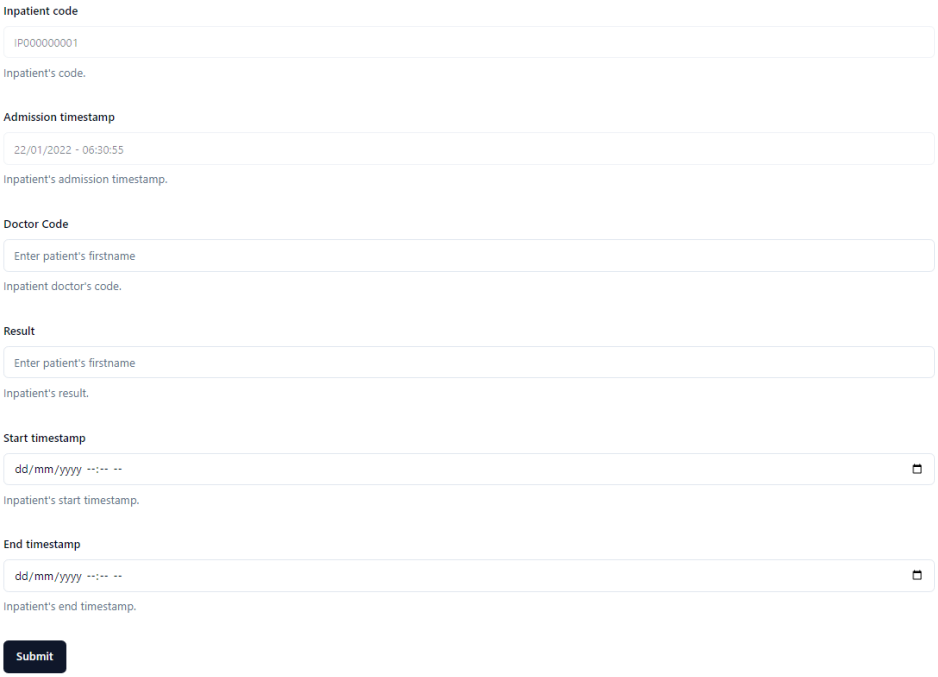


Figure 41: Add admission page

User fill in information. When the form is subnmitted, it is first validate on the client-side by Zod. Here is example of an error submission:

If all the information is valid. It first retrieve IP code for display.

It then revalidate one more time. This time it is not revalidate due to the valid of the data, but this time it is to the correctness of the data.

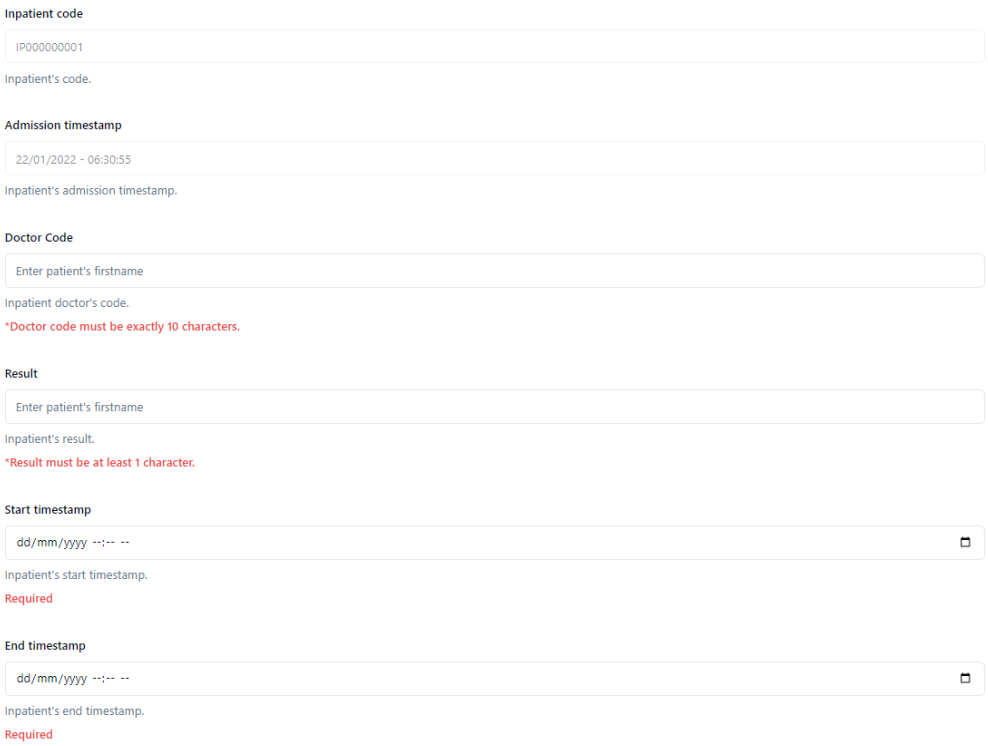


Figure 42: Zod client-side verfication

```
const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
});

const IPcodeSql = `
SELECT inpatient_code AS ip_code
FROM inpatient
WHERE patient_number = ${id};
`;
const { rows: IPcode } = await pool.query(IPcodeSql);

await pool.end();

return Response.json({ ip: IPcode[0].ip_code }, { status: 200 });
};
```

Figure 43: Retrieve IP code for display

- Existed doctor code
- Admission timestamp is before Start timestamp
- Start timestamp is before End timestamp

```
function isStartDateBeforeEndDate( startDateString: string,  endDateString: string) {
  const startDate = new Date(startDateString);
  const endDate = new Date(endDateString);
  return startDate <= endDate;
}

export async function POST(req: Request) {
  const body = await req.json();
  const pool = new Pool({
    connectionString: process.env.DATABASE_URL,
  });

  const doctorSql = `
  SELECT code FROM doctor WHERE code = '${body.doctorcode}'
  `;
  const { rows: doctors } = await pool.query(doctorSql);

  if (doctors.length === 0)
    return Response.json(
      { res: "fail", warning: "Doctor code doesn't exist!" },
      { status: 200 }
    );

  if (!isStartDateBeforeEndDate(body.admissiontime, body.starttime)) {
    return Response.json(
      { res: "fail", warning: "Admission timestamp is after Start timestamp!" },
      { status: 200 }
    );
  }

  if (!isStartDateBeforeEndDate(body.starttime, body.endtime)) {
    return Response.json(
      { res: "fail", warning: "Start timestamp is after End timestamp!" },
      { status: 200 }
    );
  }
```

Figure 44: Revalidate correctness

It also check if treatment is booked by this user at this time

If all information is valid, it will be written in the database: It also check if treatment is booked by this user at this time

```
const checkValidSql = `
SELECT * FROM treatment WHERE patient_number = ${body.id}
AND admission_timestamp = '${body.admissiontime}' AND start_timestamp = '${body.starttime}' AND doctor_code = '${body.doctorcode}';
`;
const { rows: checkValid } = await pool.query(checkValidSql);

if (checkValid.length > 0)
  return Response.json(
    { res: "fail", warning: "You already booked a treatment at this time!" },
    { status: 200 }
  );
```

Figure 45: Check if treatment is booked

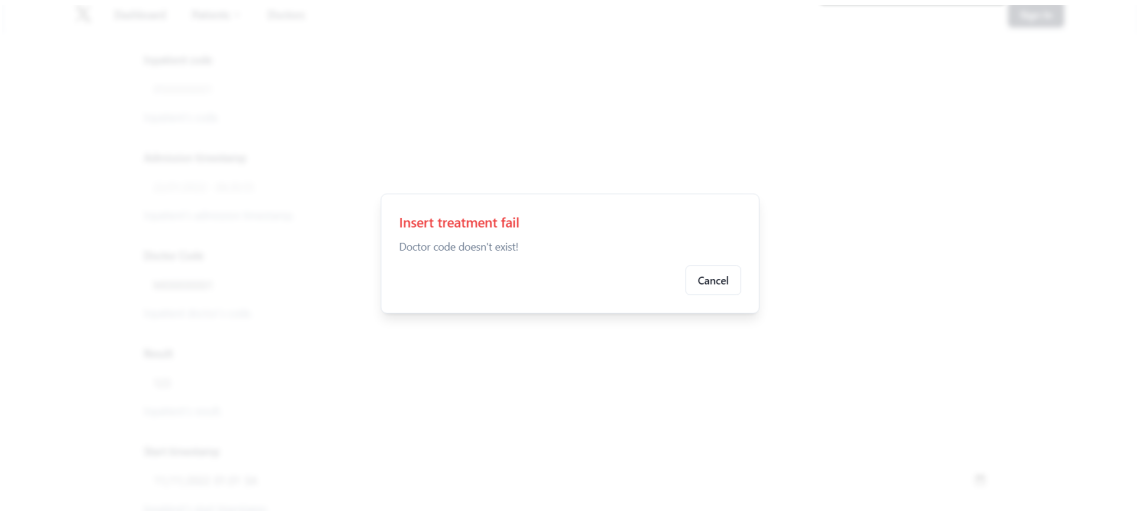


Figure 46: Error when enter non-existing nurse code

```
const sql = `
INSERT INTO Treatment(Doctor_Code, Patient_Number, Inpatient_Code, Admission_Timestamp, Start_Timestamp, End_Timestamp, Result_)
VALUES(
  '${body.doctorcode}',
  ${body.id},
  '${body.ip}',
  '${body.admissiontime}',
  '${body.starttime}',
  '${body.endtime}',
  '${body.result}'
);
`;
const { rows } = await pool.query(sql);

await pool.end();

return Response.json({ res: "success", warning: "" }, { status: 200 });
```

Figure 47: Write data to the database

If all information is valid, manager is alerted. Click 'Continue' to go to that patient information page. The redirected ID is taken from the response. If click 'Cancel', user is remain as this page.

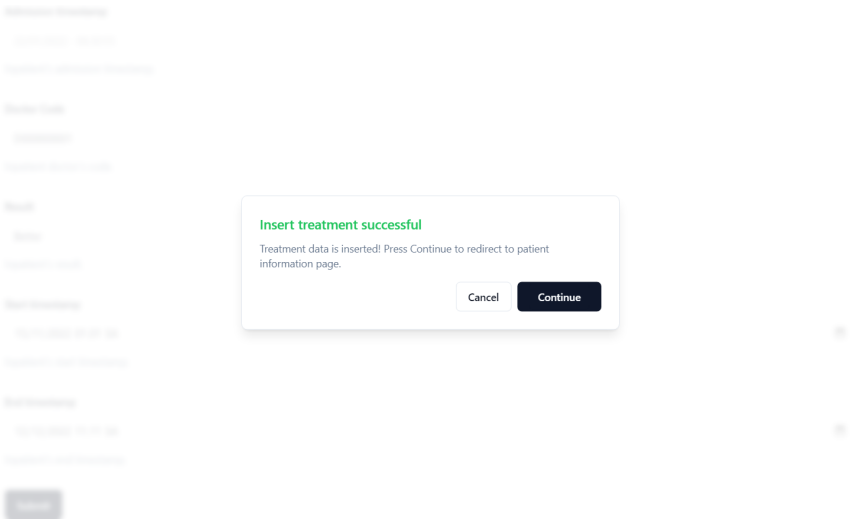


Figure 48: Valid diaglog show up

When redirect back to patient information page, a new treatment is add to the table.

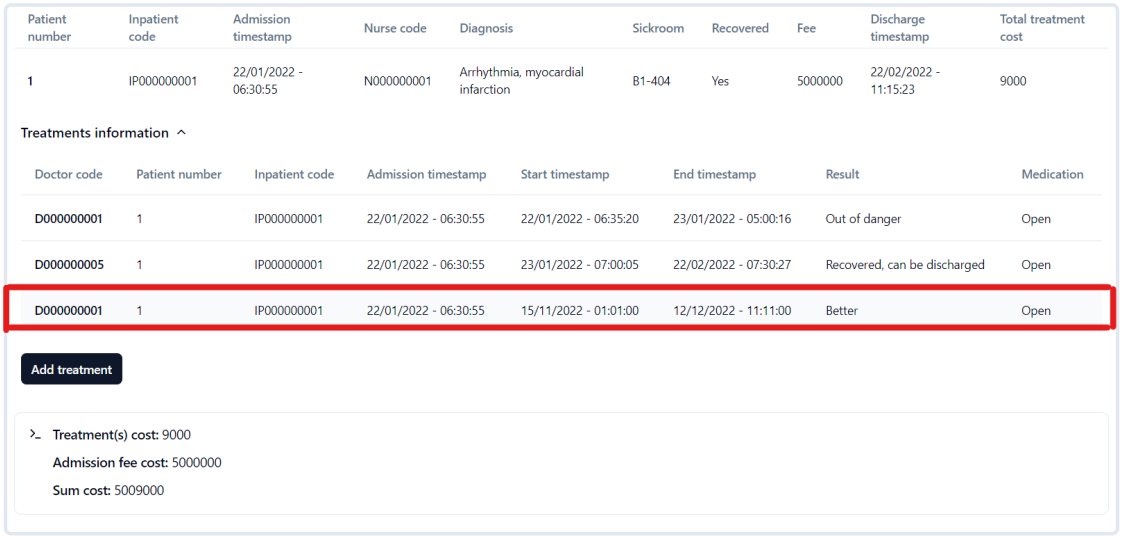


Figure 49: A new admission line

When click on add examination, manger is then redirect to a page to fill in patient's examination information. This include:

- Outpatient code

- Doctor Code
- Exam timestamp
- Diagnosis
- Fee
- Next exam date

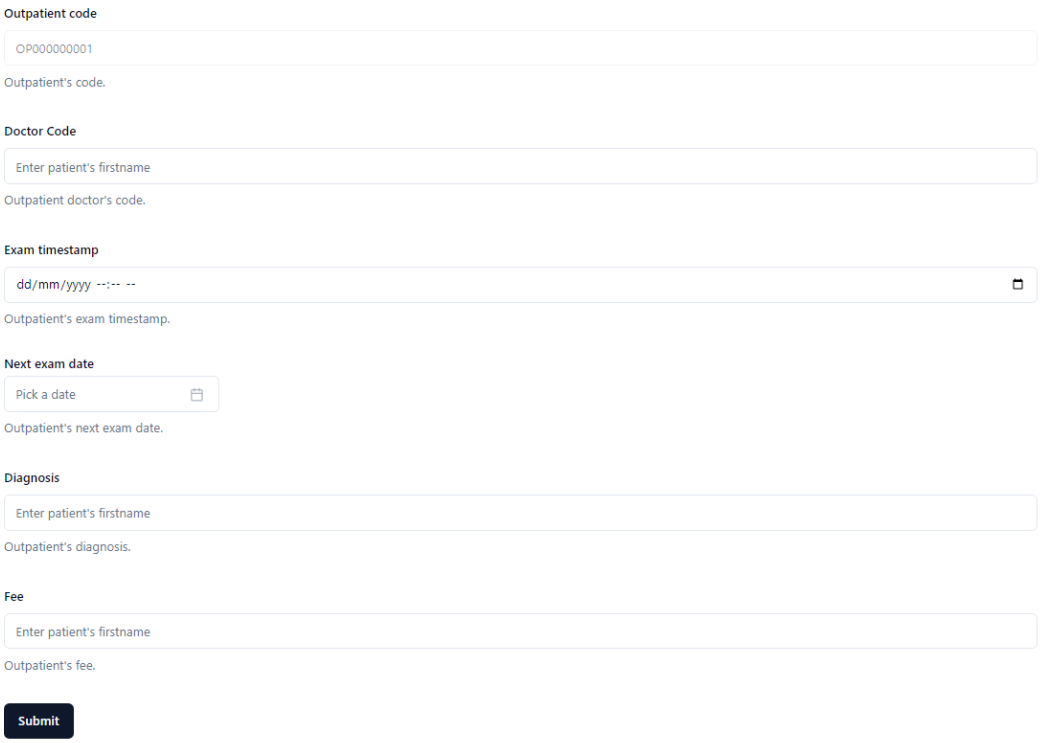


Figure 50: Add admission page

User fill in information. When the form is subnmitted, it is first validate on the client-side by Zod. Here is example of an error submission:

If all the information is valid, it retrieve OP code. If outpatient have one, display it, else generate a new one

If all the information is valid. It then revalidate one more time. This time it is not revalidate due to the valid of the data, but this time it is to the correctness of the data.

- Valid doctor code

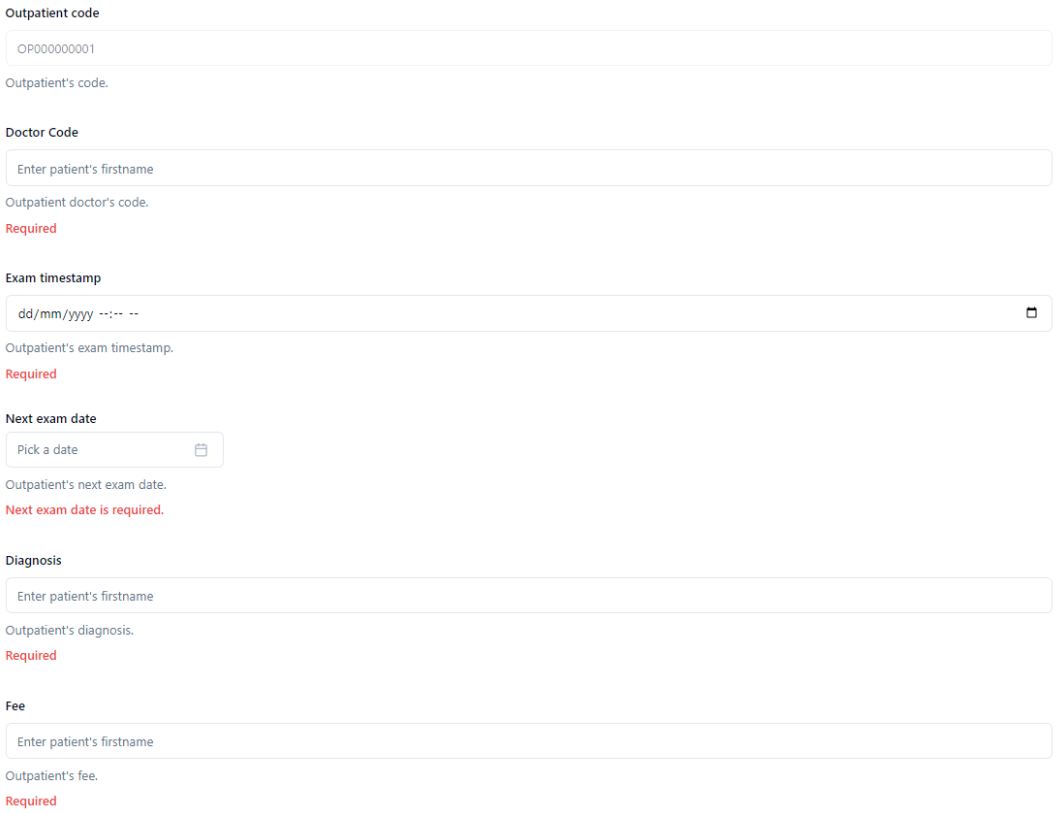


Figure 51: Zod client-side verfication

```
const opcodeSql =`SELECT outpatient_code FROM outpatient where patient_number = ${id}`
const { rows: opcode } = await pool.query(opcodeSql);

if (opcode.length === 0) { // new patient
  const heighestOPSql = `
  SELECT MAX(CAST(SUBSTRING(outpatient_code FROM 3) AS INTEGER)) AS op_code
  FROM outpatient
  `
  const { rows: heighestID } = await pool.query(heighestOPSql);

  return Response.json({ state: "new", ip: "", maxip: heighestID[0].op_code }, { status: 200 });
}

const IPcodeSql = `
SELECT outpatient_code AS ip_code
FROM outpatient
WHERE patient_number = ${id};
`
const { rows: IPcode } = await pool.query(IPcodeSql);

await pool.end();

return Response.json({ state: "old", ip: IPcode[0].ip_code, maxip: 0 }, { status: 200 });
```

Figure 52: Generate OP code

- Exam timestamp is before Next exam date
- Is examination is booked already by that user

```
function isStartDateBeforeEndDate(startDateString: string, endDateString: string) {
  const startDate = new Date(startDateString);
  const endDate = new Date(endDateString);
  return startDate <= endDate;
}

export async function POST(req: Request){
  const body = await req.json()
  const pool = new Pool({
    connectionString: process.env.DATABASE_URL,
  });

  const doctorSql =`
  SELECT code FROM doctor WHERE code = '${body.doctorcode}'
  `
  const { rows: doctors } = await pool.query(doctorSql);

  if (doctors.length === 0)
    return Response.json({ res: "fail", warning: "Doctor code doesn't exist!" }, { status: 200 });

  if (!isStartDateBeforeEndDate(body.examtime, body.nextexamdate)) {
    return Response.json({ res: "fail", warning: "Exam timestamp is after Next exam date!" }, { status: 200 });
  }

  const checkValidSql =`
  SELECT * FROM examination WHERE patient_number = ${body.id} AND exam_timestamp = '${body.examtime}' AND doctor_code = '${body.doctorcode}';
  `
  const { rows: checkValid } = await pool.query(checkValidSql);

  if (checkValid.length > 0)
    return Response.json({ res: "fail", warning: "You already booked an examination at this time!" }, { status: 200 });
```

Figure 53: Revalidate on server side

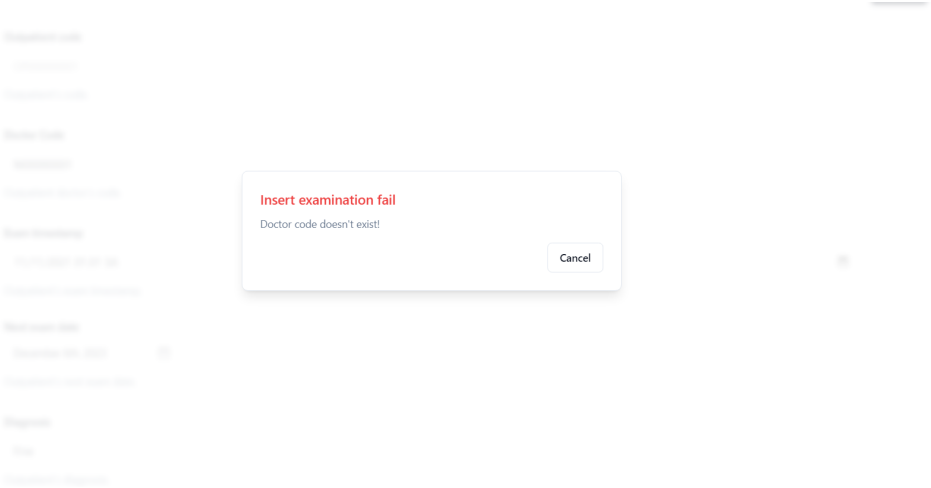


Figure 54: Error when enter non-existing nurse code

If all information is valid, is OP code not exist, add first. Then add record to the table:

```
const opcodeSql =`SELECT outpatient_code FROM outpatient where patient_number = ${body.id}`
const { rows: opcode } = await pool.query(opcodeSql);

if (opcode.length === 0) { // new patient
  const heighestOPSql = `
  INSERT INTO OutPatient(Patient_Number, Outpatient_Code)
  VALUES(
    ${body.id},
    '${body.opcode}'
  );
  `
  const { rows: heighestID } = await pool.query(heighestOPSql);
}


const sql =`
INSERT INTO Examination(Doctor_Code, Patient_Number, Outpatient_Code, Exam_Timestamp, Next_Exam_Date, Diagnosis, Fee)
VALUES(
  '${body.doctorcode}',
  ${body.id},
  '${body.opcode}',
  '${body.examtime}',
  '${body.nextexamdate}',
  '${body.diagnosis}',
  ${body.fee}
);
`
const { rows } = await pool.query(sql);

await pool.end();

return Response.json({ res: "success", warning: ""}, { status: 200 });
```

Figure 55: Write data to OP if not existed first!

If all information is valid, manager is alerted. Click 'Continue' to go to that patient information page. The redirect page ID is taken from the response. If click 'Cancel', user is remain as this page.

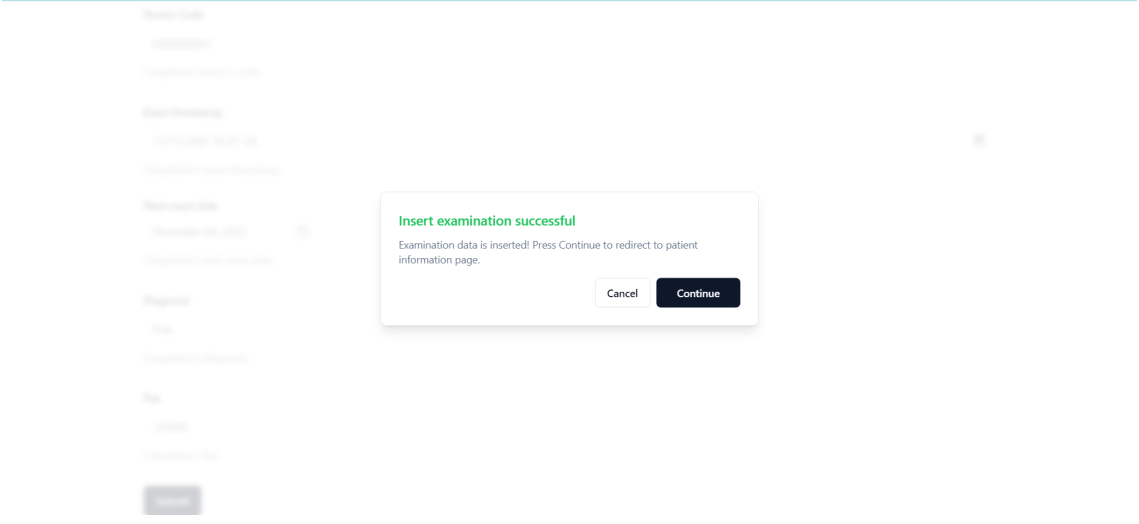


Figure 56: Valid diaglog show up

When redirect back to patient information page, a new examination is add to the table.

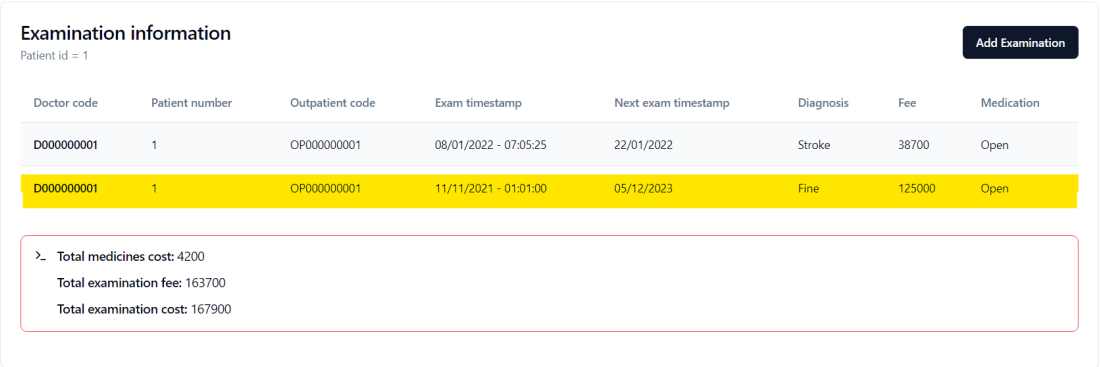


Figure 57: A new admission line

3.6 List details of all patients which are treated by a doctor:

To view the doctor, click 'Doctor' on the navigation bar. Manager is redirected to the page to search for doctor. He/She can be found base on the doctor code.

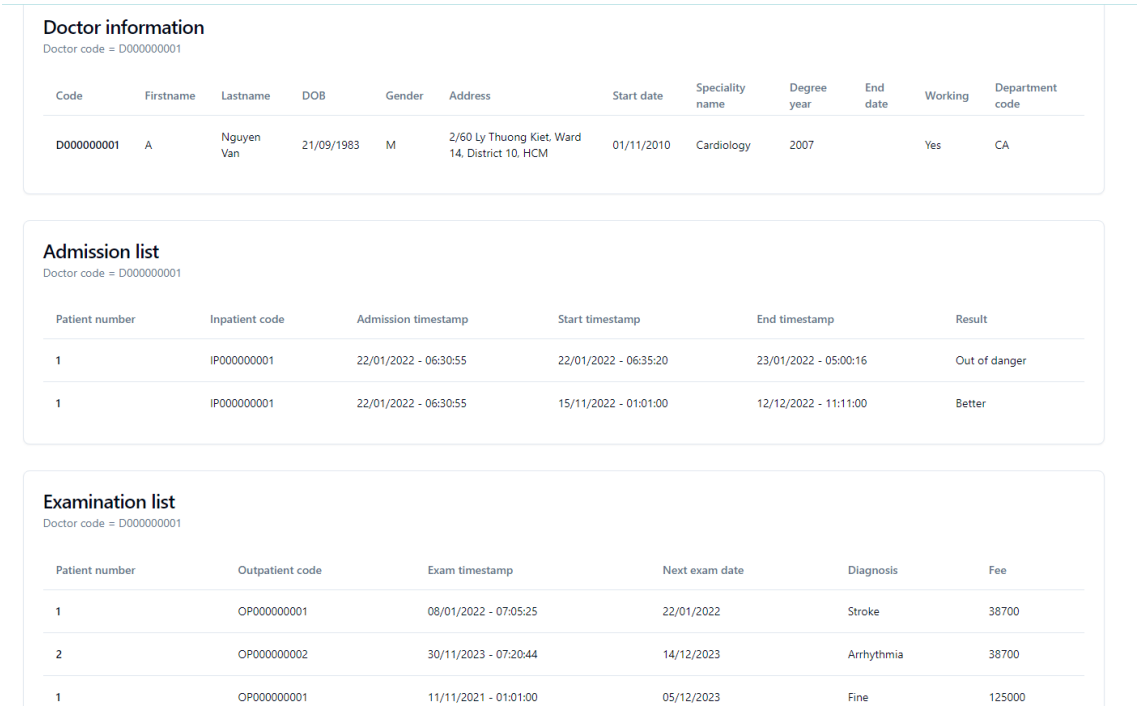
Figure 58: A new admission line

To view doctor information, click on the corresponding line. User will be redirected to that doctor's information page.

Doctor information is found by ID.

The page cover 3 sections: Doctor information, admission list and examination list. In the first section, the information in the previous page is shown here. It includes:

- Code
- Firstname
- Lastname DOB
- Gender
- Address



Doctor information

Doctor code = D000000001

Code	Firstname	Lastname	DOB	Gender	Address	Start date	Speciality name	Degree year	End date	Working	Department code
D000000001	A	Nguyen Van	21/09/1983	M	2/60 Ly Thuong Kiet, Ward 14, District 10, HCM	01/11/2010	Cardiology	2007		Yes	CA

Admission list

Doctor code = D000000001

Patient number	Inpatient code	Admission timestamp	Start timestamp	End timestamp	Result
1	IP000000001	22/01/2022 - 06:30:55	22/01/2022 - 06:35:20	23/01/2022 - 05:00:16	Out of danger
1	IP000000001	22/01/2022 - 06:30:55	15/11/2022 - 01:01:00	12/12/2022 - 11:11:00	Better

Examination list

Doctor code = D000000001

Patient number	Outpatient code	Exam timestamp	Next exam date	Diagnosis	Fee
1	OP000000001	08/01/2022 - 07:05:25	22/01/2022	Stroke	38700
2	OP000000002	30/11/2023 - 07:20:44	14/12/2023	Arrhythmia	38700
1	OP000000001	11/11/2021 - 01:01:00	05/12/2023	Fine	125000

Figure 59: A new admission line

```
const sql =`SELECT * FROM doctor where code = '${id}'`
const { rows: doctor } = await pool.query(sql);
```

Figure 60: Find doctor by ID.

- Start date
- Speciality name
- Degree year
- End date
- Working
- Department code

Information of doctor admission, examination is retrieve from the database:

```
const treatmentSql =`select * from Treatment where doctor_code = '${id}'`
const { rows: treatments } = await pool.query(treatmentSql);

const examinationSql =`select * from Examination where doctor_code = '${id}
const { rows: examinations } = await pool.query(examinationSql);


await pool.end();

// return Response.json({ hello: now }, {status : 200});

return Response.json({ doctor, treatments, examinations }, { status: 200 })
```

Figure 61: Retrieve doctors' admissions, examinations

Next is doctor's admission list. It includes information of inpatient treated by this doctor. To view more information on the patient, click on the corresponding row. Manager is redirected to corresponding information page. Finally is is doctor's examination list. It includes information of outpatient examined by this doctor. To view more information on the patient, click on the corresponding row. Manager is redirected to corresponding information page.

3.7 Payment report:

To view patient payment report, first, manager have to go to that patient's information page. Firstly, on the navigation bar, hover on 'Patients', then click on find patient.

A patient can be found base on the phone number. After enter the phone number and click search, a row of detail of that patient is shown.

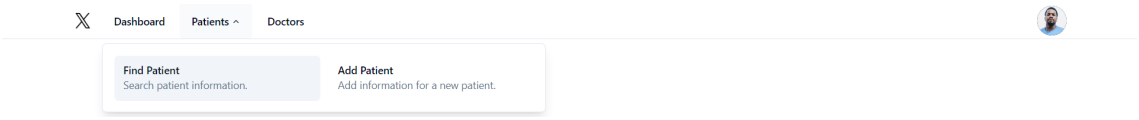


Figure 62: A new admission line

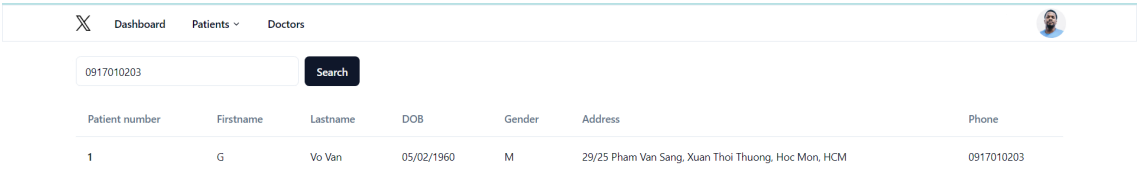


Figure 63: A new admission line

To view information, click on the corresponding row. Manager is redirect to the corresponding page.

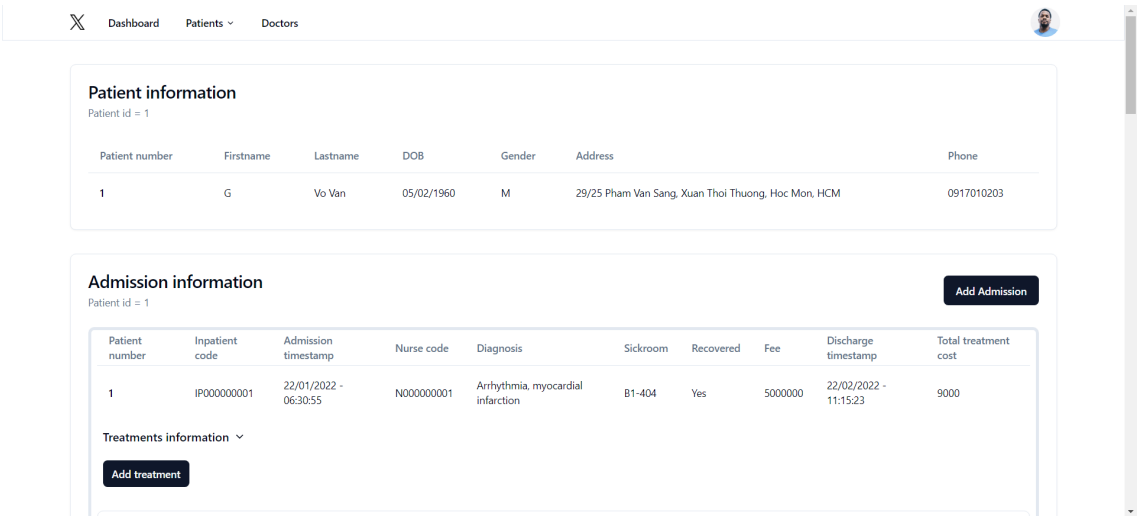


Figure 64: A new admission line

On this page, for each admission, the sum treatments cost (sum of all medications) along with admission fee is shown. There is also a row 'Sum cost'. It is equal to sum of treatment(s) cost and admission fee cost.

To retrieve the medication price we take quantity $\times$ price and display as a new column. In that sql we retrieve sum of all medication:

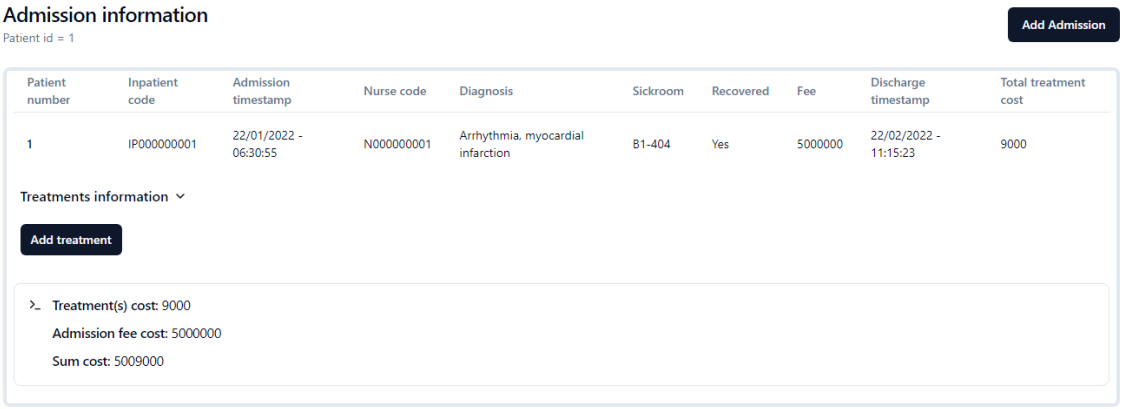


Figure 65: A new admission line

```sql
SELECT
  tm.quantity,
  tm.start_timestamp,
  tm.admission_timestamp,
  m.code,
  m.name_,
  m.price,
  m.price * tm.quantity AS total_value
FROM
    (
        SELECT
            t.inpatient_code,
            t.admission_timestamp,
            t.doctor_code,
            t.start_timestamp,
            t.end_timestamp,
            t.result_,
            tm.medication_code,
            tm.quantity
        FROM
            treatment t
        JOIN
            admission a ON t.inpatient_code = a.inpatient_code
                        AND t.admission_timestamp = a.admission_timestamp
        JOIN
            treatment_medication tm ON t.inpatient_code = tm.inpatient_code
                                    AND t.admission_timestamp = tm.admission_timestamp
                                    AND t.start_timestamp = tm.start_timestamp
        WHERE
            a.patient_number = ${id}
    ) tm
JOIN
    medication m ON tm.medication_code = m.code;
```

Figure 66: Treatment medication price

In the end of each treatment we display the fee and sum of medication price. We also take sum of them.

In the end of admission section, total of all treatments cost and total of all admission fee cost is shown here. There is also a total admission cost row, which is equal the sum of two above.

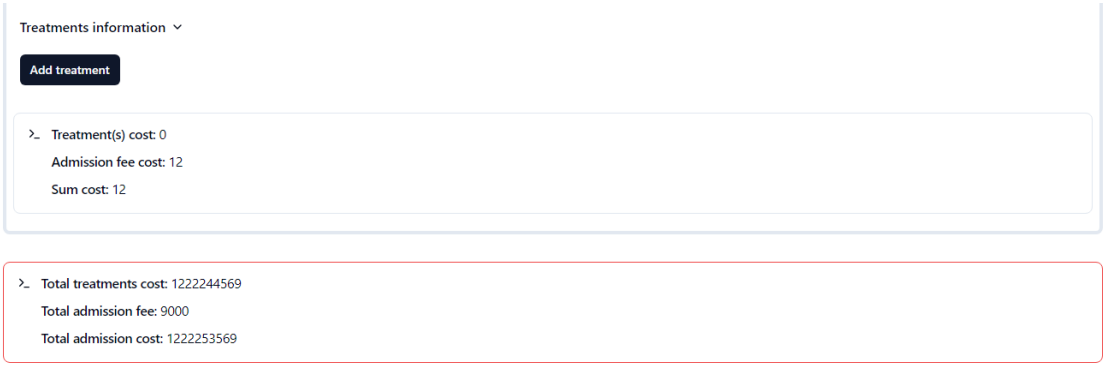


Figure 67: A new admission line

The total fee price and medication price is retrieve using this sql:

For examination section, in the end of this section, there is total medicines cost (sum of medication of all examination) and total examination fee (sum of all examination fee). In the end, there is a row total examination cost, which is the sum of total medicines cost and total examination fee

To retrieve the medication price we take quantity $\times$ price and display as a new column.

To retrieve the sum of medication price and sum of medication price via:

Finally there is a total payment section, which is sum of all payment

We take the total sum by add all the sums retrieve in the section above!

```
  SELECT
    SUM(a.fee) AS total_fee,
    SUM(COALESCE(tm.total_value, 0)) AS total_value
FROM
    admission a
LEFT JOIN (
    SELECT
        tm.patient_number,
        tm.admission_timestamp,
        SUM(tm.quantity * m.price) AS total_value
    FROM
        treatment_medication tm
    JOIN
        medication m ON tm.medication_code = m.code
    GROUP BY
        tm.patient_number,
        tm.admission_timestamp
) tm ON a.patient_number = tm.patient_number
    AND a.admission_timestamp = tm.admission_timestamp
WHERE
    a.patient_number = ${id};
```

Figure 68: Total price

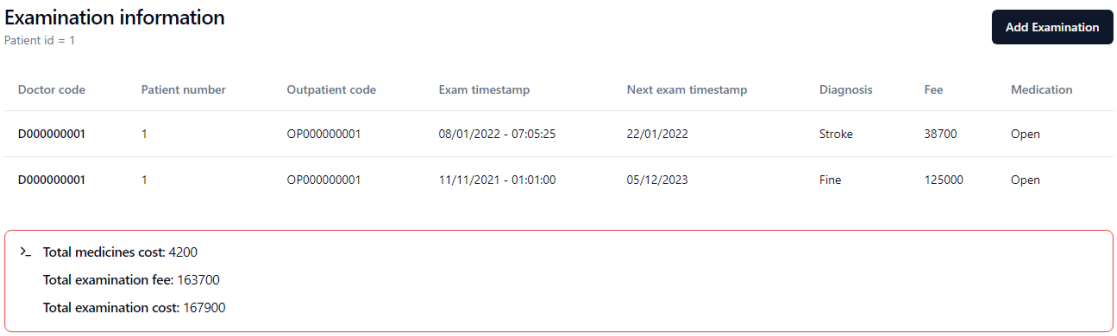


Figure 69: A new admission line

```
SELECT
  em.*,
  m.name_,
  m.price,
  em.quantity * m.price AS total_value
FROM
  exam_medication em
JOIN
  medication m ON em.medication_code = m.code
WHERE
  em.patient_number = ${id};
```

Figure 70: Examination medication price

```
 SELECT
    SUM(em.quantity * m.price) AS total_value
FROM
    exam_medication em
JOIN
    medication m ON em.medication_code = m.code
WHERE
    em.patient_number = ${id};
```

Figure 71: Sum examination medication price

```
    SELECT
    SUM(fee) AS total_fee
FROM
    examination
WHERE
    patient_number = ${id};
    `
```

Figure 72: Sum examination fee price

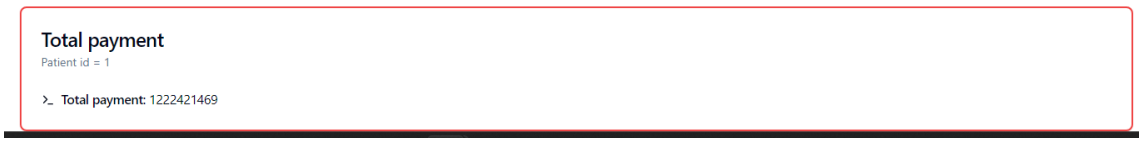


Figure 73: A new admission line

4 Database Management

4.1 Indexing efficiency

In this section, we will prove one use case of indexing efficiency when dealing with table containing large number of records. We do the test with the table **Imported_Medicine** by creating 10 million records. After that, we will use the following query to retrieve the information about the imported medicine from the provider with **Provider_Number = 2343**:

```
1    SELECT * FROM Imported_Medicine WHERE Provider_Number = 2343;
```

As we can see from figure Query result before indexing, the total query time is 1 second 12 milliseconds.

Then, we will check how many blocks are needed to store the table **Imported_Medicine**:

```
1 SELECT relname AS "relation",
2        pg_relation_size(C.oid) / 8192 AS "blocks",
3        pg_size_pretty(pg_relation_size(C.oid)) AS "size"
4 FROM pg_class C
5 LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
6 WHERE relname = 'Imported_Medicine' ;
```

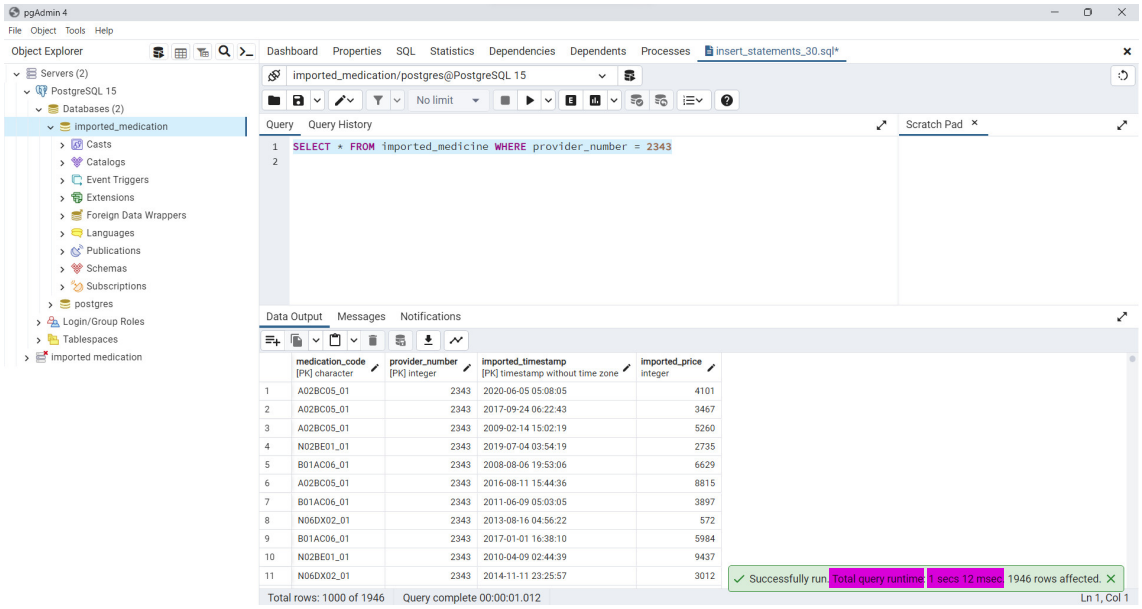


Figure 74: Query result before indexing

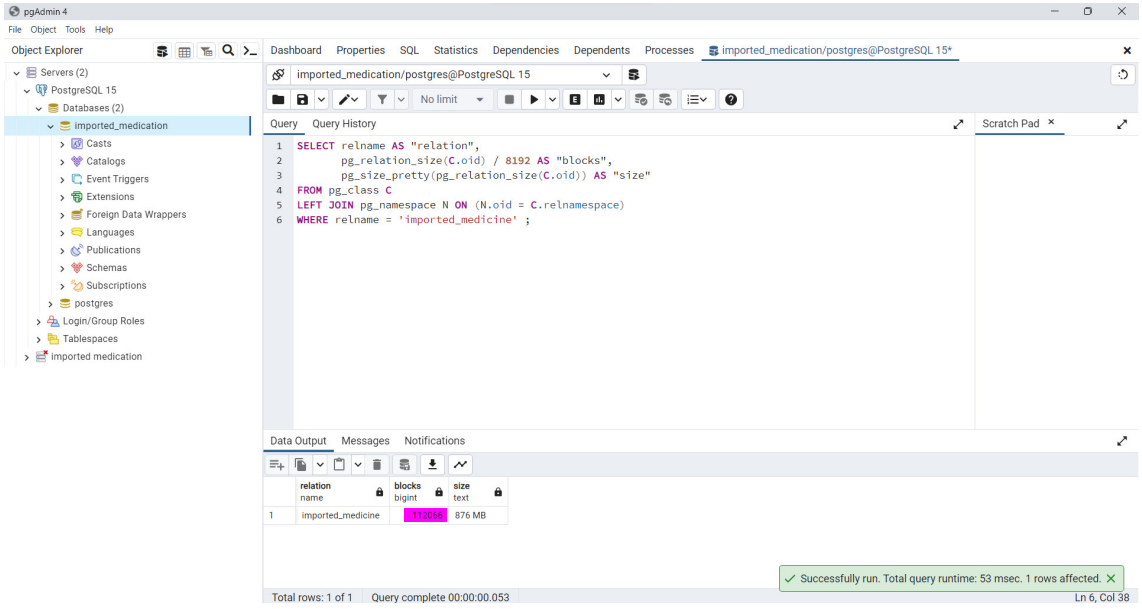


Figure 75: Number of blocks to store table **Imported_Medicine**

As we can see from the figure Number of blocks to store table **Imported_Medicine**, there are 112066 blocks needed to store this table.

After that, we will run the explain analyze for the select query:

```
1 EXPLAIN ANALYZE SELECT  * FROM Imported_Medicine WHERE Provider_Number
    = 2343;
```

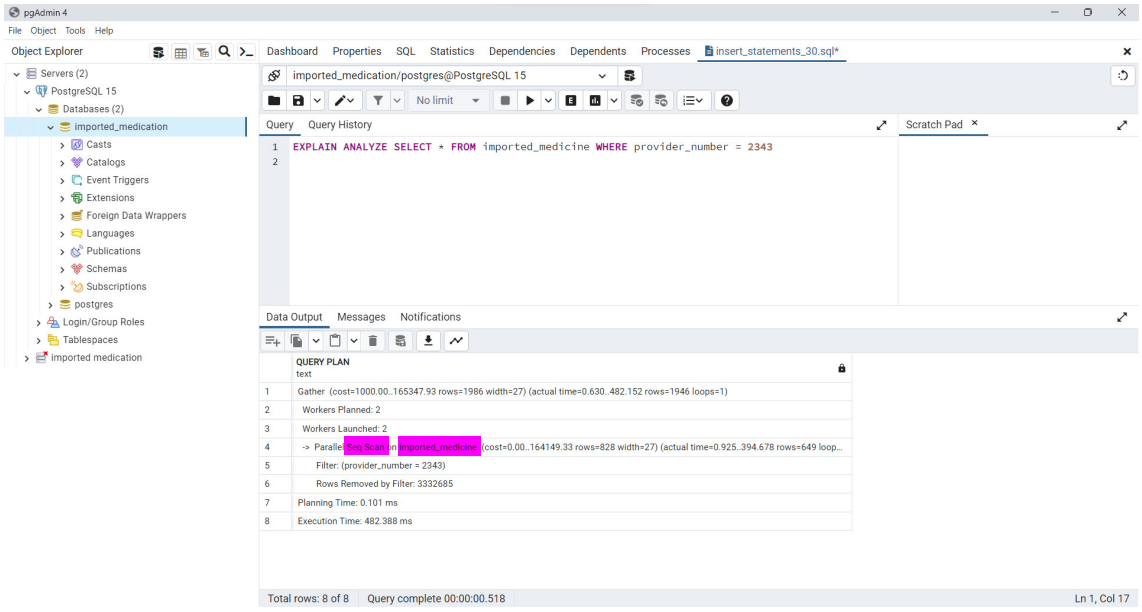


Figure 76: Explain analyze before indexing

As we can see from the figure Explain analyze before indexing, to retrieve the result for the query, the DBMS use Seq Scan, which means it scans the entire table stored on disk. Therefore, when doing the linear search on this table, the number of block accesses in average will be $\lceil 112066 \div 2 \rceil = 56033$ (block accesses).

Now, we will apply the indexing on the column **Provider_Number**:

```
CREATE INDEX idx_provider_number ON Imported_Medicine(Provider\_Number)
    ;
```

After creating the index, we will run the select query again.

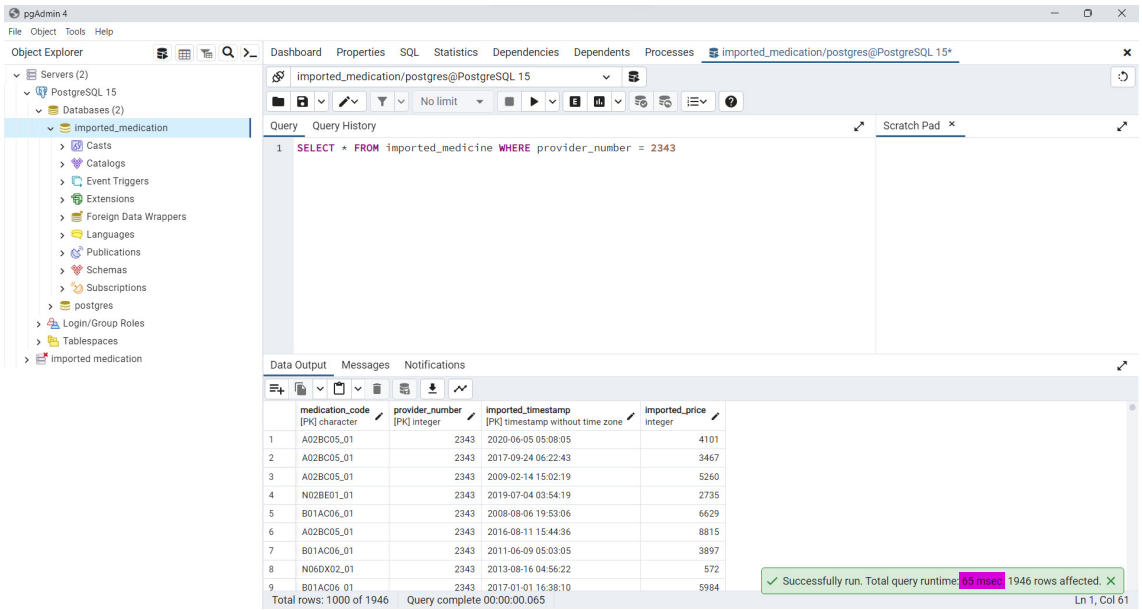


Figure 77: Query result after indexing

As we can see from the figure Query result after indexing, the total query runtime now is only 65 milliseconds, much faster than before indexing.

Then, we will run the explain analyze for the select query after indexing:

```
EXPLAIN ANALYZE SELECT  * FROM Imported_Medicine WHERE Provider_Number
    = 2343;
```

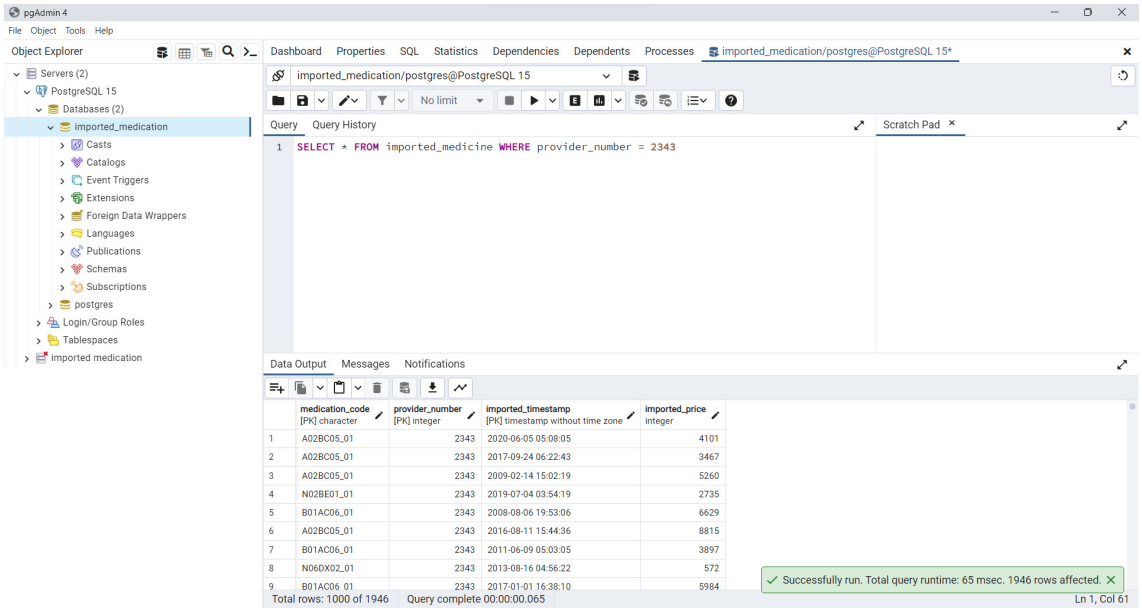


Figure 78: Explain analyze after indexing

As we can see from the figure Explain analyze after indexing, the number of heap blocks now is 1921, and the DBMS do the Bitmap Heap Scan, which is the sequential search on the Bitmap index file. Therefore, when doing the linear search on this index file, the number of block accesses in average will be $\lceil 1921 \div 2 \rceil = 961$ (block accesses).

We can see that after indexing, the number of block accesses is much smaller (961 ¡ 56033). To sum up, indexing is efficient when dealing with large-sized table.

4.2 A use case of database security: SQL Injection

- **What is SQL injection:** SQL injection is an attack technique in which an attacker inserts or "injects" malicious SQL statements into SQL queries executed by a web application. This is one of the most common and dangerous attack methods in the field of web security. Attackers use SQL injection to perform unauthorized actions in the database or obtain sensitive information.

- **Example of SQL injection:**

```
SELECT * FROM users WHERE username = '$_POST[username]' AND
password = '$_POST[password]'
```

Website query command line

```
' OR '1'='1'; --
```

Input from attacker

```
SELECT * FROM users WHERE username = '' OR '1'='1'; --' AND
password = '...'
```

Query after input

As a result, the condition '1'='1' is always true, so the application will return all records in the users table, without checking the password. This allows attackers to obtain user account information easily.

- **Why is SQL injection dangerous:**

1. Unauthorized Access to Data: Successful SQL injection attacks can grant unauthorized access to sensitive data stored in the database. Attackers can retrieve, view, and potentially download sensitive information such as usernames, passwords, credit card numbers, or personal information. They can also modify or delete data in unexpected ways, leading to data loss or corruption.

For an example:

```
UPDATE users SET password = 'new_password' WHERE username = 'b'
And password = 'old_password'
```

Password change function in an application

```
new_password = attacker_password
b = victim_username
old_password = ' OR '1'='1'--
```

Input from attacker

```
1    UPDATE users SET password = 'attacker_password' WHERE username
   = 'victim_username'And password = '' OR '1'='1'--'
2
```

Query after input

With this input from the attacker, because '1'='1' is always true so the condition for the password is always satisfied, the '–' at the end is a comment mark, causing the rest of the query to be ignored so that it can secure the query can execute without error. As a result, the victim lost their account since the password was changed by the attacker

2. User Authentication Bypass: By injecting malicious SQL code into login forms, attackers may bypass authentication mechanisms, allowing them to log in as any user, including administrators, without knowing the correct credentials.

3.Database and Server Compromise: SQL injection can lead to full compromise of the underlying server. Attackers may execute system commands, install malware, or gain control over the server, extending the scope of the attack beyond the database.

4.Cross-Site Scripting : SQL injection can be used in conjunction with other attacks, such as Cross-Site Scripting. An attacker may inject malicious code into a website, which is then executed by unsuspecting users, leading to further compromises

- **How to prevent SQL injection:**

  1.Check and Filter Input Data: Before using user data, check and filter it to ensure that it does not contain special characters that could be used to inject malicious SQL code.
  This is referred as client-side validation. It plays a crucial role in enhancing the user experience, improving data quality, and reducing server load in web applications

  - Immediate User Feedback: Client-side validation provides immediate feedback to users as they interact with the user interface. This instant response helps users identify and correct errors in their input without having to wait for a round trip to the server.
  - Reduced Server Load: By validating user input on the client side, unnecessary requests to the server with invalid data can be minimized. This reduces the server load and conserves resources, leading to improved overall application performance and scalability, particularly in scenarios with a large number of users.

  In this case we use Zod as a schema declaration and validation library. In order to send to the input data to the server-side, it is first validated by Zod. If the data is invalid with the declared type, it is warned to the user!

Let's take an example: Given add new patient scenario. User have to input this form:

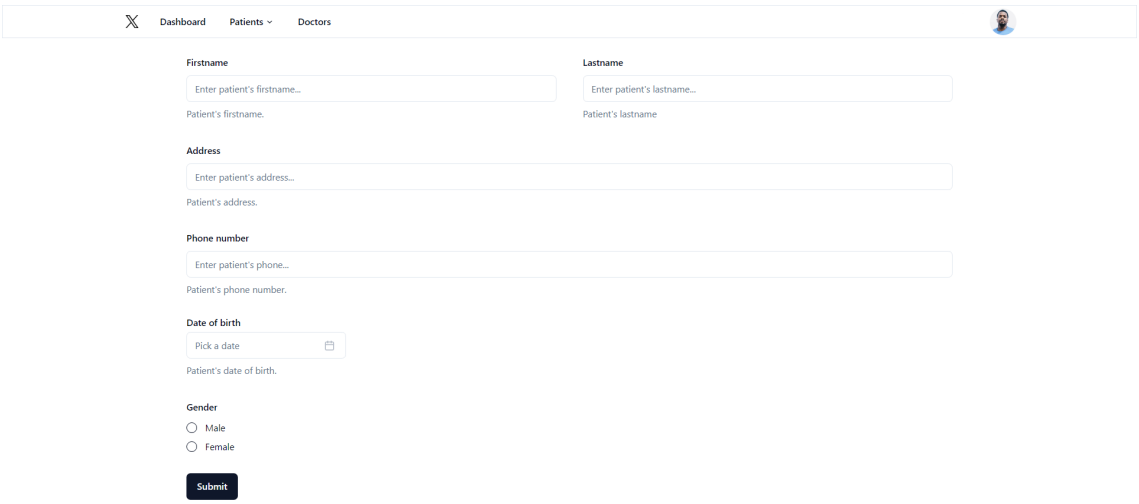


Figure 79: Add patient case

The schema for this form is:

When user not to follow the schema, warning error is given:

While client-side validation offers some advantages, it also comes with certain uncertainties and limitations. Here are some points to consider regarding the uncertainty of client-side validation:

- Bypassing Client-Side Validation: One of the main uncertainties is that client-side validation can be bypassed easily. Since the validation logic is executed on the client side, a user with malicious intent can disable or modify the client-side scripts to submit invalid data directly to the server.
- Dependence on Browser Environment: Client-side validation relies on the user's browser environment. Different browsers may interpret and execute JavaScript differently, leading to inconsistencies in validation behavior.

2.Using framework: frameworks like Hibernate or Java Persistence API often provide automated mechanisms to prevent SQL injection. It is recommended using these framework instead of manipulating SQL directly.

```
const formSchema = z.object({
  firstname: z.string().min(1, {
    message: "*Firstname must be at least 1 character.",
  }).max(20, {
    message: "*Firstname must be at most 20 characters"
  }),
  lastname: z.string().min(1, {
    message: "*Lastname must be at least 1 character.",
  }).max(20, {
    message: "*Lastname must be at most 20 characters"
  }),
  address: z.string().min(1, {
    message: "*Address must be at least 1 characters.",
  }).max(200, {
    message: "*Address must be at most 200 characters"
  }),
  phone: z.string().max(11, {
    message: "*Phone must be at most 11 characters"
  }),
  dob: z.date({
    required_error: "A date of birth is required.",
  }),
  gender: z.enum(["male", "female"], {
    required_error: "You need to select a gender.",
  })
});
```

Figure 80: Add patient case

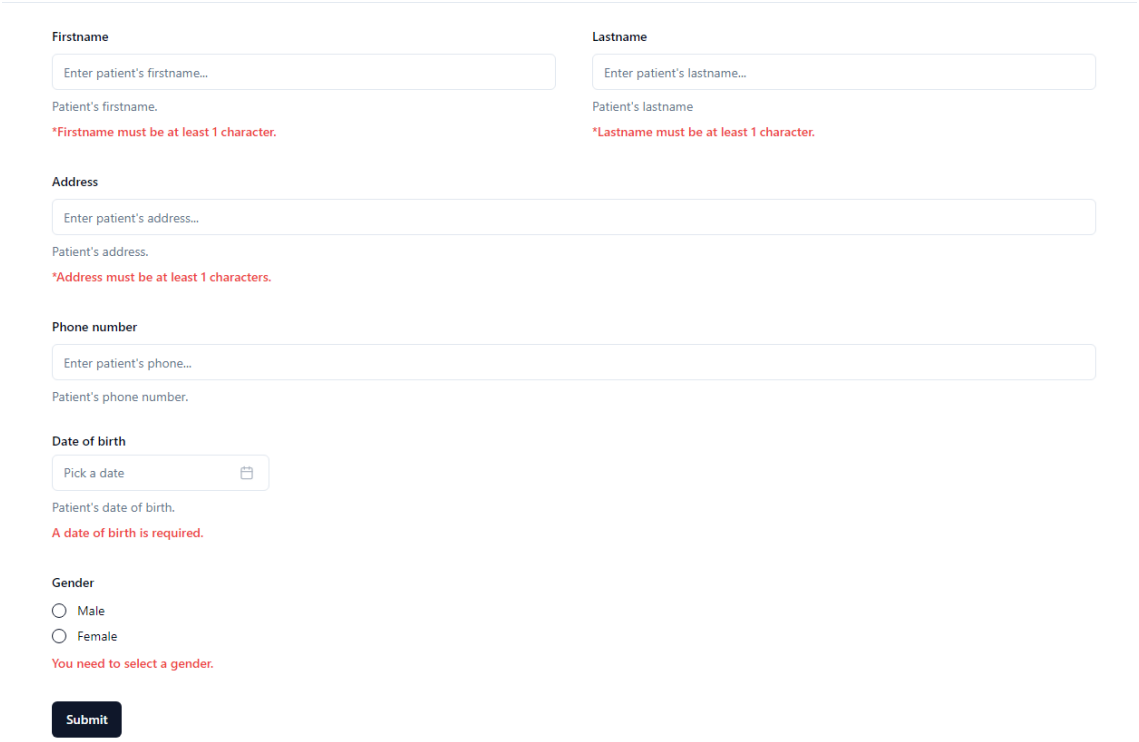


Figure 81: Not follow the schema warning

3.Using Parameterized Queries: Server-side validation is crucial for security. It prevents malicious users from bypassing client-side validation or tampering with requests before they reach the server. Unlike client-side validation, which occurs in the user's browser, server-side validation is performed on the server, providing a more secure and robust approach.

One common use case for server-side validation involves the use of prepared statements when interacting with a database. Prepared Statement uses parameters (placeholders) in SQL statements instead of embedding user values directly. This helps prevent SQL injection by not allowing user data to hijack SQL syntax. It is reffered as server-side validation.

But our team use Neon Serverless Postgres, with Pool connection. But sadly it is not yet support Prepared Statement. It is state in the documentation `https://neon.tech/docs/connect/connection-pooling#connection-pooling-notes-and-limitations`. We sadly found this out when our team almost done this assignment!

But our team manage to have the Prepared Statement and run it on the desktop version of PostgreSQL. Here is the original query:

Connection pooling notes and limitations

Neon uses PgBouncer in *transaction mode*, which does not support Postgres features such as prepared statements or LISTEN/NOTIFY. For a complete list of limitations, refer to the "*SQL feature map for pooling modes*" section in the pgbouncer.org Features documentation.

Figure 82: Neon Serverless Postgres haven't supported Prepared Statement

```
const sql =`
INSERT INTO Patient(First_Name, Last_Name, Date_Of_Birth, Gender, Address, Phone_Number)
VALUES(
  '${body.firstname}',
  '${body.lastname}',
  '${body.dob}',
  '${body.gender === 'male' ? 'M' : 'F'}',
  '${body.address}',
  '${body.phone}'
);
```

Figure 83: Original query

Now we change it to prepared statement. Given manager want to insert a new patient record: `('Dinh', Dung', 2003-11-22', 'M', '497 Hoa Hao', '0123456789')`. The corresponding sql for that is:

Query | Query History

```
1  PREPARE insert_patient AS
2      INSERT INTO Patient(First_Name, Last_Name, Date_Of_Birth, Gender, Address, Phone_Number)
3      VALUES($1, $2, $3, $4, $5, $6)
4      RETURNING patient_number;
5
6  -- Execute the prepared statement with parameters
7  EXECUTE insert_patient(
8      'Dinh',
9      'Dung',
10     '2003-11-22',
11     'M',
12     '497 Hoa Hao',
13     '0123456789'
14 );
15
16 -- Deallocate the prepared statement (optional, but recommended for cleanup)
17 DEALLOCATE insert_patient;
```

Figure 84: Corresponding Prepare Statement

Run the sql and display all rows gives:

Query | Query History

```
1  PREPARE insert_patient AS
2      INSERT INTO Patient(First_Name, Last_Name, Date_Of_Birth, Gender, Address, Phone_Number)
3      VALUES($1, $2, $3, $4, $5, $6)
4      RETURNING patient_number;
5
6  -- Execute the prepared statement with parameters
7  EXECUTE insert_patient(
8      'Dinh',
9      'Dung',
10     '2003-11-22',
11     'M',
12     '497 Hoa Hao',
13     '0123456789'
14 );
15
16 -- Deallocate the prepared statement (optional, but recommended for cleanup)
17 DEALLOCATE insert_patient;
18 SELECT * FROM patient
```

Data Output | Messages | Notifications

	patient_number [PK] integer	first_name text	last_name text	date_of_birth date	gender character	address text	phone_number character varying (11)
1	1	Dinh	Dung	2003-11-22	M	497 Hoa Hao	0123456789

Figure 85: Insert to table via Prepare Statement