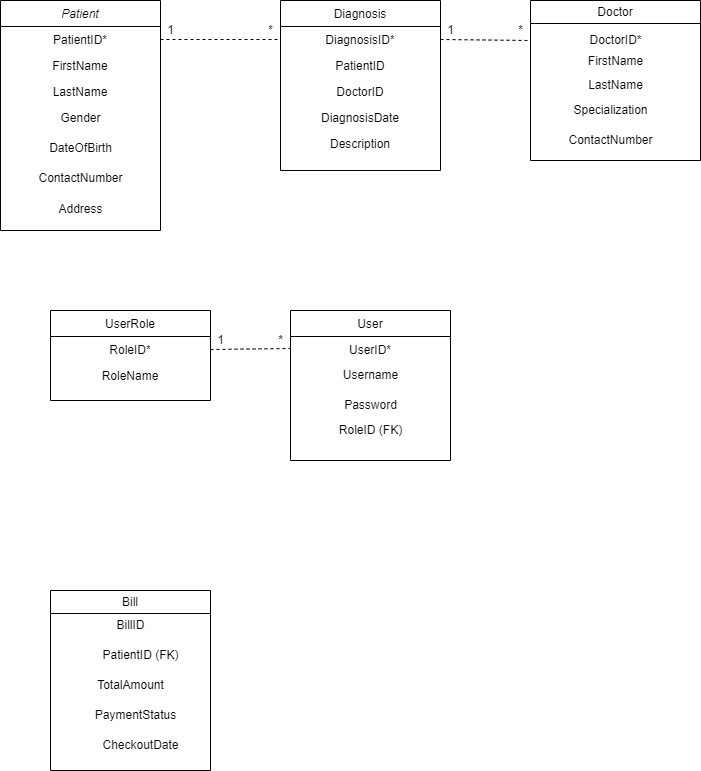
***PROJECT 1(INTERN)***

***Q1)***

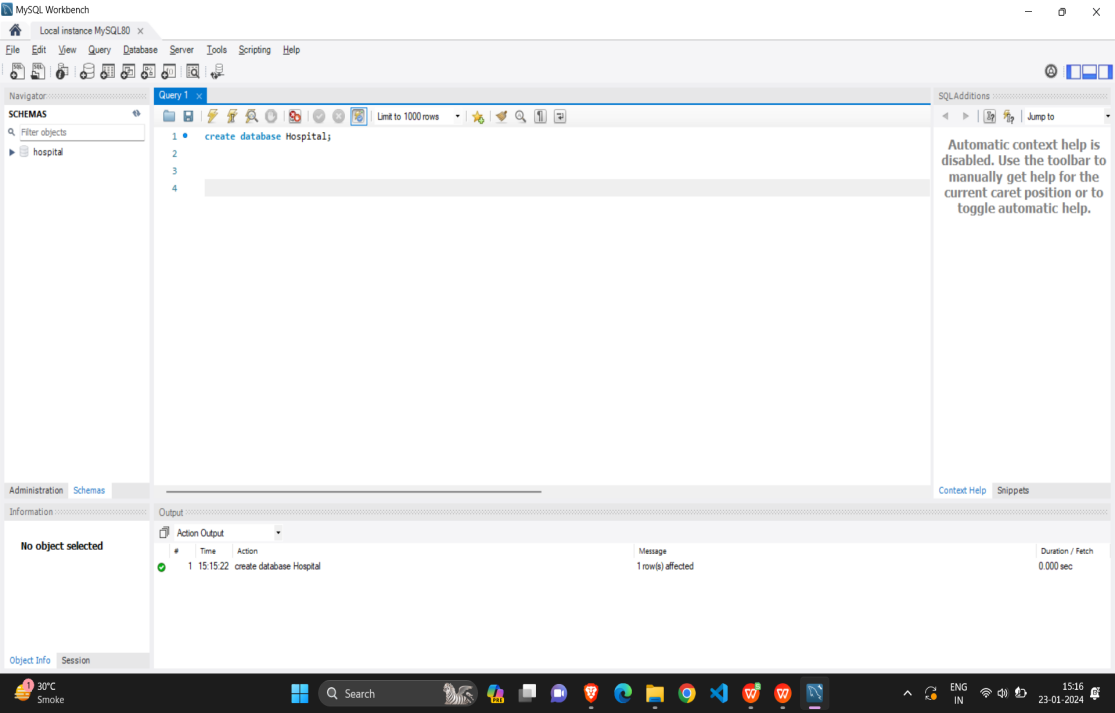
******

Normalization is the process of organizing a database to reduce data redundancy and improve data integrity. The goal is to structure the tables in such a way that data anomalies, such as insertion, update, and deletion anomalies, are minimized. The most commonly used normalization forms are First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), Boyce-Codd Normal Form (BCNF), and Fourth Normal Form (4NF).

The schema already satisfies the requirements of 1NF , 2NF and 3 NF.

1. Create a DB Schema for Hospital Management System.

Queries: **create database Hospital;**



1. Define the schema along with the constraints indicating the relationships between the entities.



Queries:

**CREATE TABLE Patient (**

**PatientID INT PRIMARY KEY,**

**FirstName VARCHAR(50),**

**LastName VARCHAR(50),**

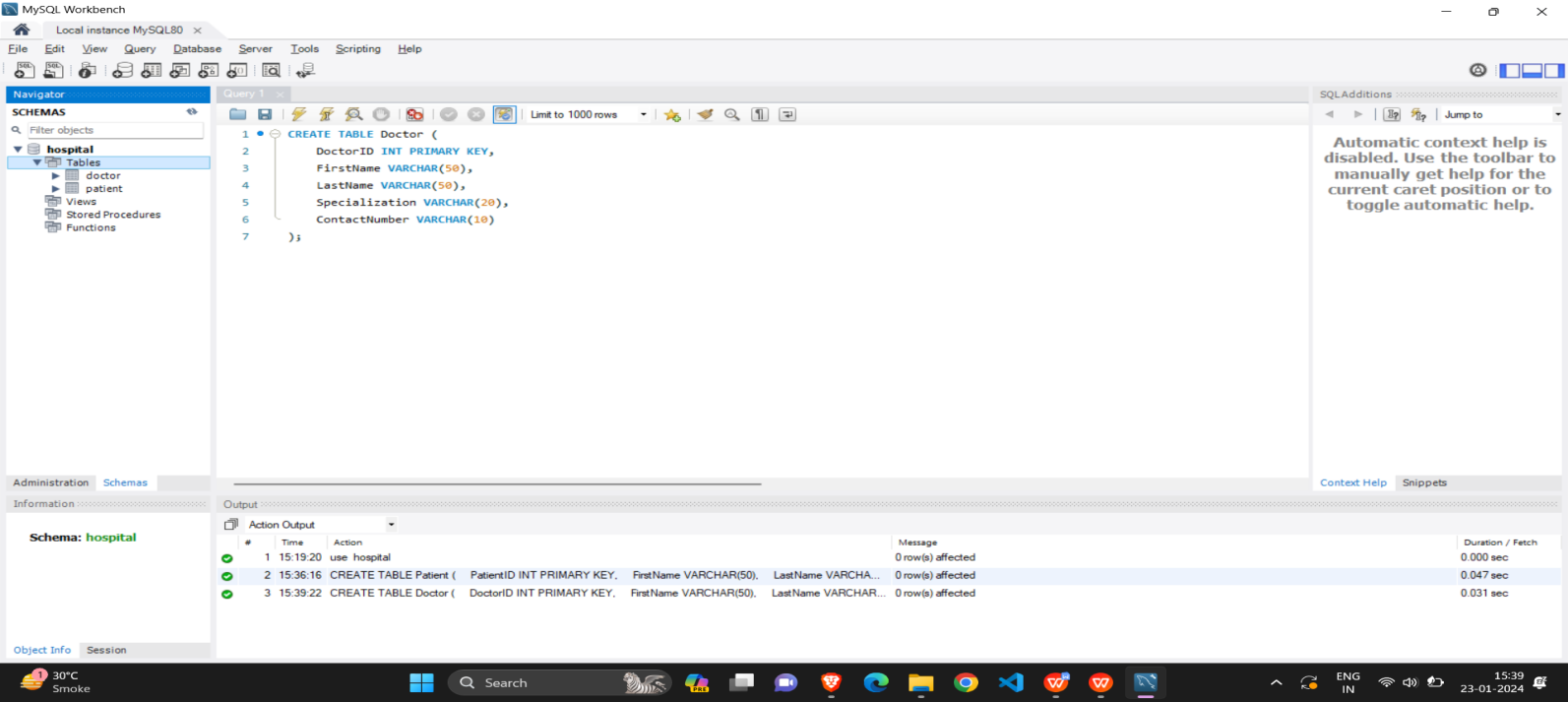
**Gender VARCHAR(10),**

**DateOfBirth DATE,**

**ContactNumber VARCHAR(20),**

**Address VARCHAR(30),**

**);**



**CREATE TABLE Doctor (**

**DoctorID INT PRIMARY KEY,**

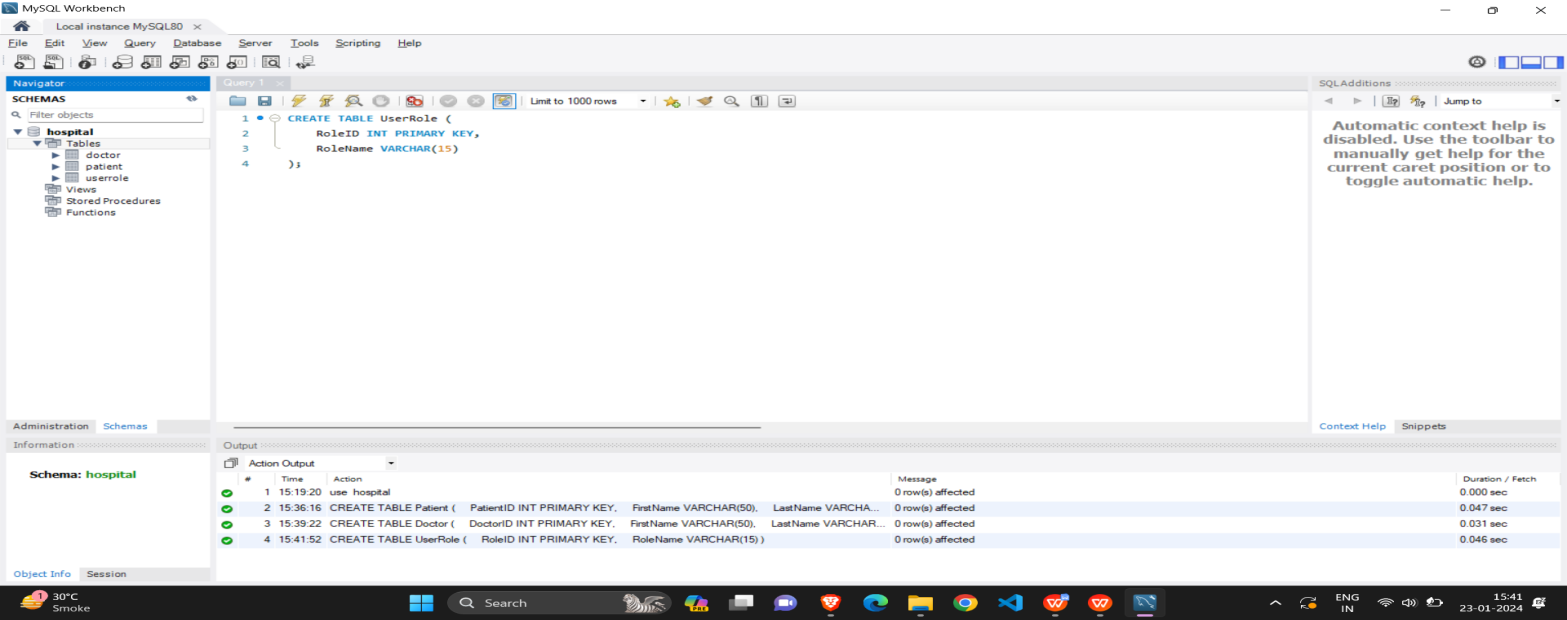
**FirstName VARCHAR(50),**

**LastName VARCHAR(50),**

**Specialization VARCHAR(20),**

**ContactNumber VARCHAR(10)**

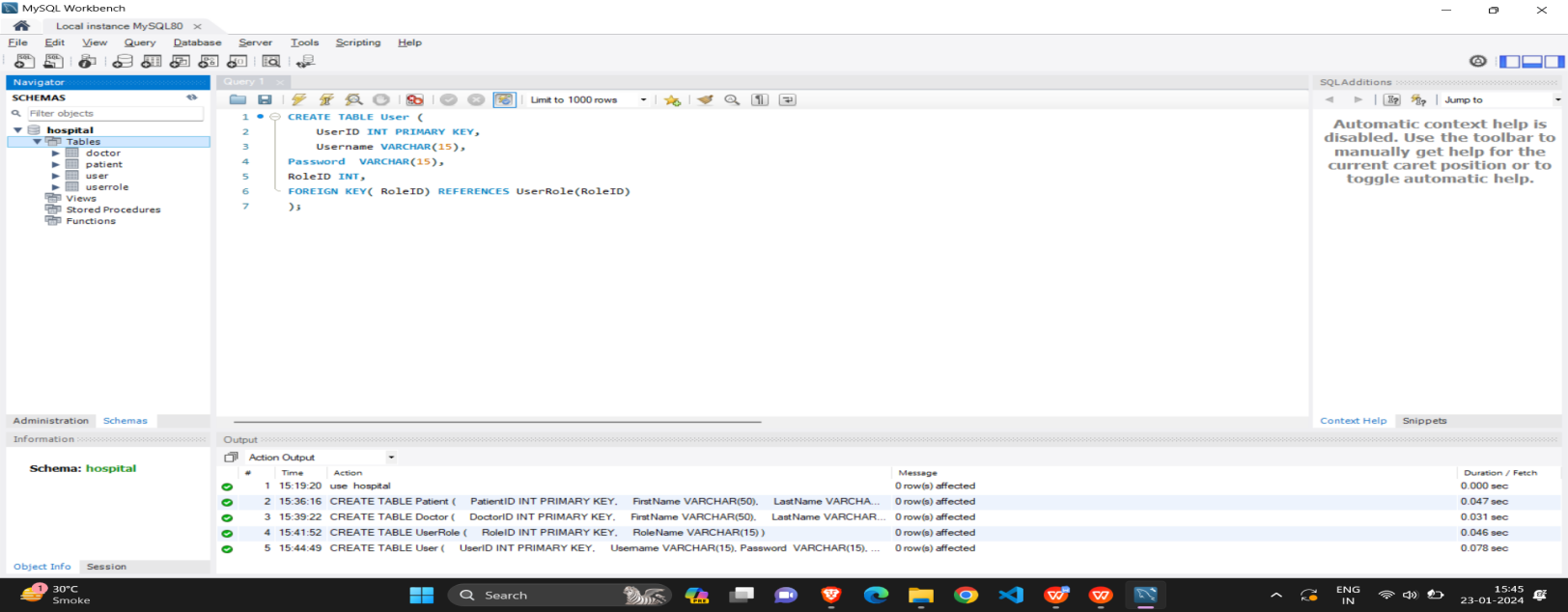
**);**

**CREATE TABLE UserRole (**

**RoleID INT PRIMARY KEY,**

**RoleName VARCHAR(15)**

**);**



**CREATE TABLE User (**

**UserID INT PRIMARY KEY,**

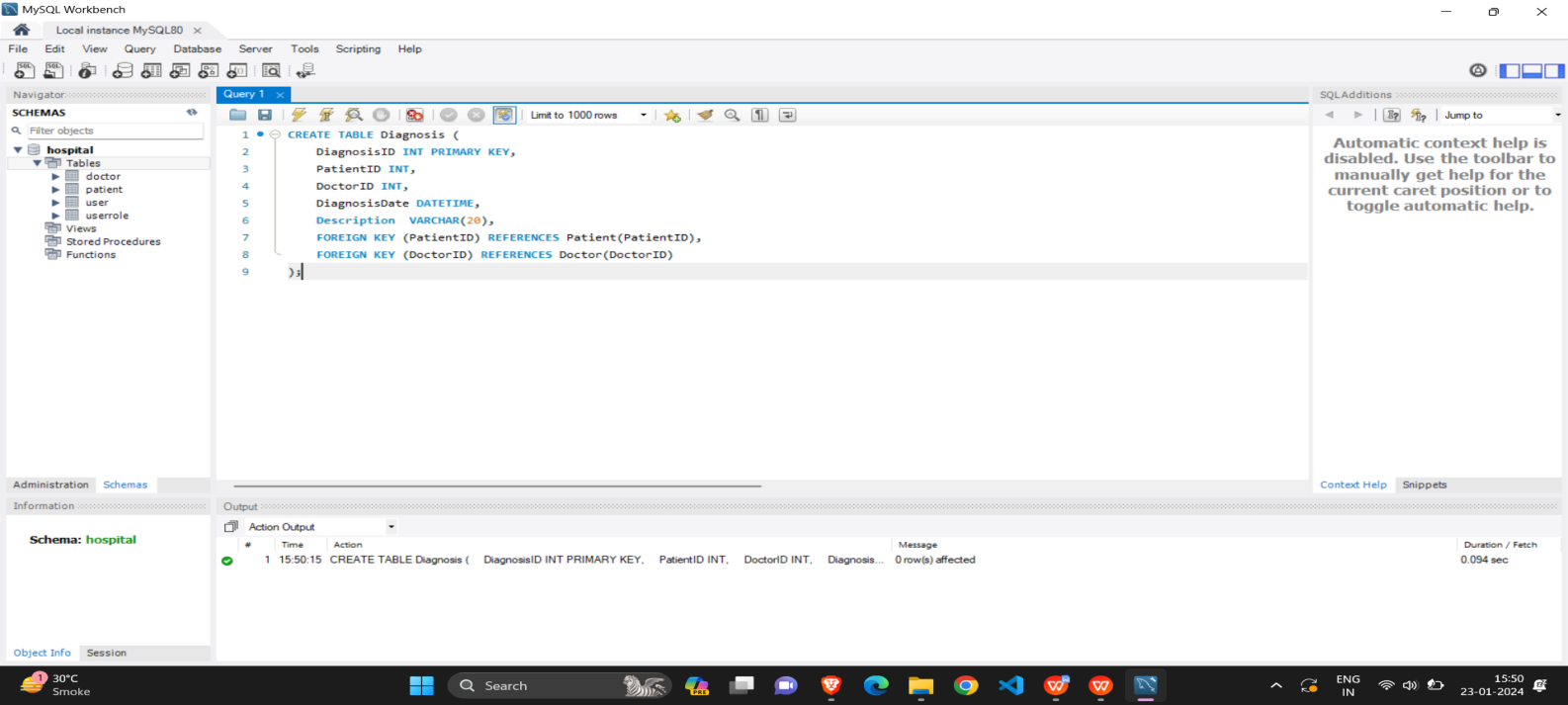
**Username VARCHAR(15),**

**Password VARCHAR(15),**

**RoleID INT,**

**FOREIGN KEY( RoleID) REFERENCES UserRole(RoleID)**

**);**



**CREATE TABLE Diagnosis (**

**DiagnosisID INT PRIMARY KEY,**

**PatientID INT,**

**DoctorID INT,**

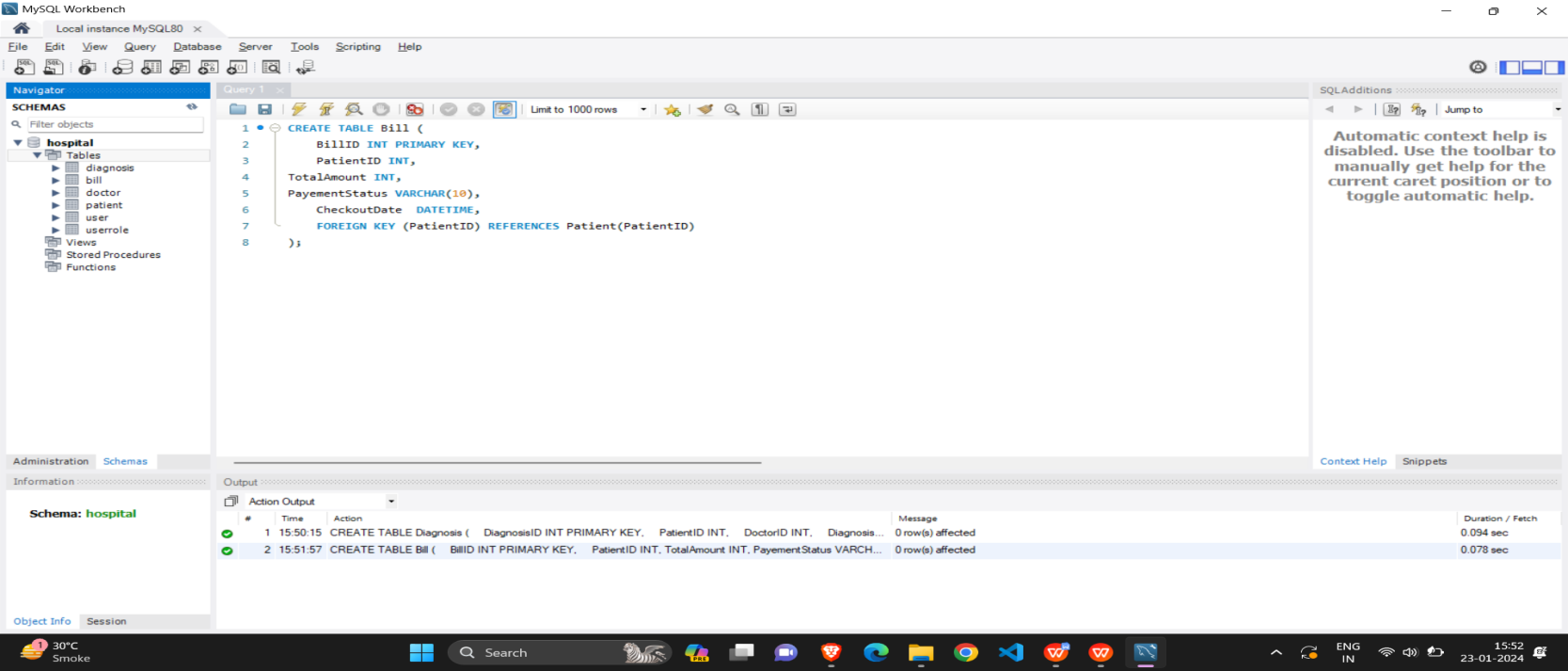
**DiagnosisDate DATETIME,**

**Description VARCHAR(20),**

**FOREIGN KEY (PatientID) REFERENCES Patient(PatientID),**

**FOREIGN KEY (DoctorID) REFERENCES Doctor(DoctorID)**

**);**



**CREATE TABLE Bill (**

**BillID INT PRIMARY KEY,**

**PatientID INT,**

**TotalAmount INT,**

**PayementStatus VARCHAR(10),**

**CheckoutDate DATETIME,**

**FOREIGN KEY (PatientID) REFERENCES Patient(PatientID)**

**);**

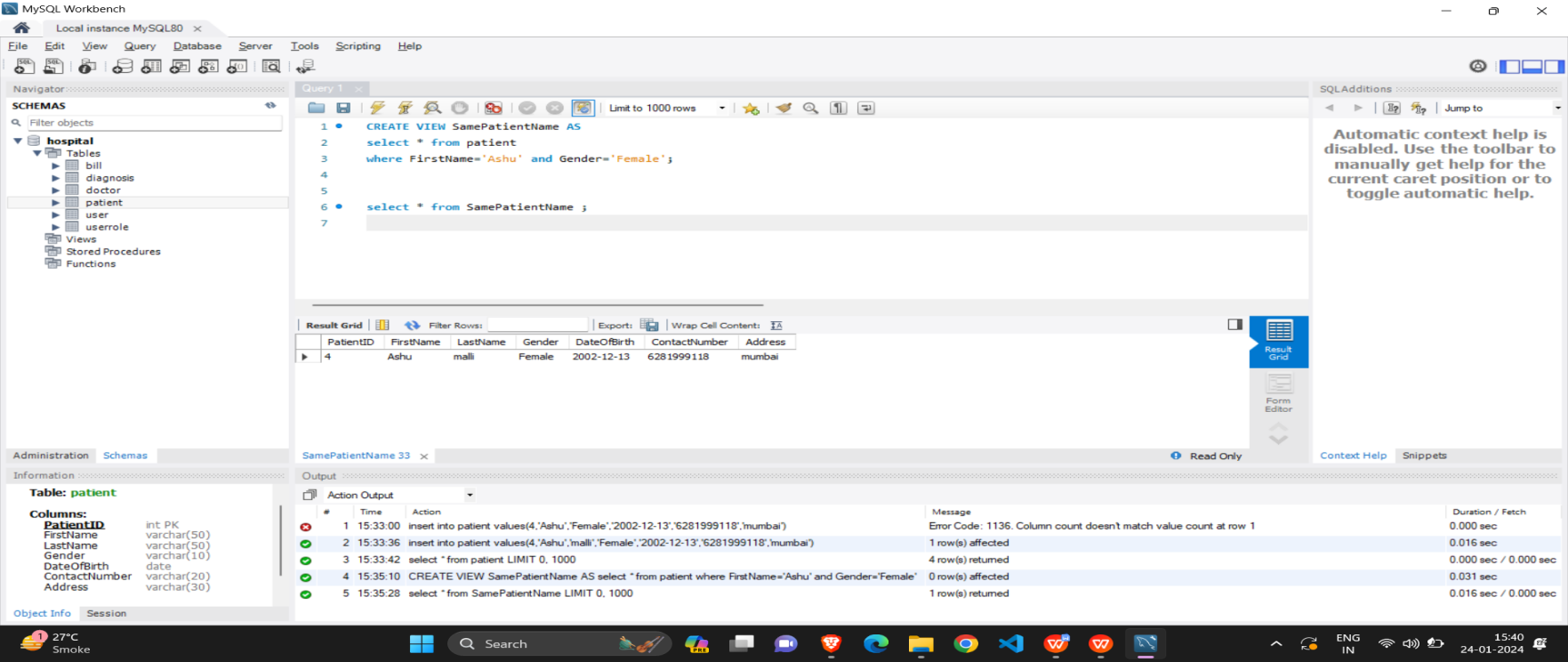
1. Be sure to make use of the database concepts like **Views, Relationships, Indexing, Stored Procedure and triggers**.

Views: ***A view in SQL is a virtual table that represents the result of a SELECT query. It does not store data itself but provides a way to encapsulate and simplify complex queries, enhance data security, and present a subset of data in a more meaningful manner.***

**CREATE VIEW SamePatientName AS**

**select \* from patient**

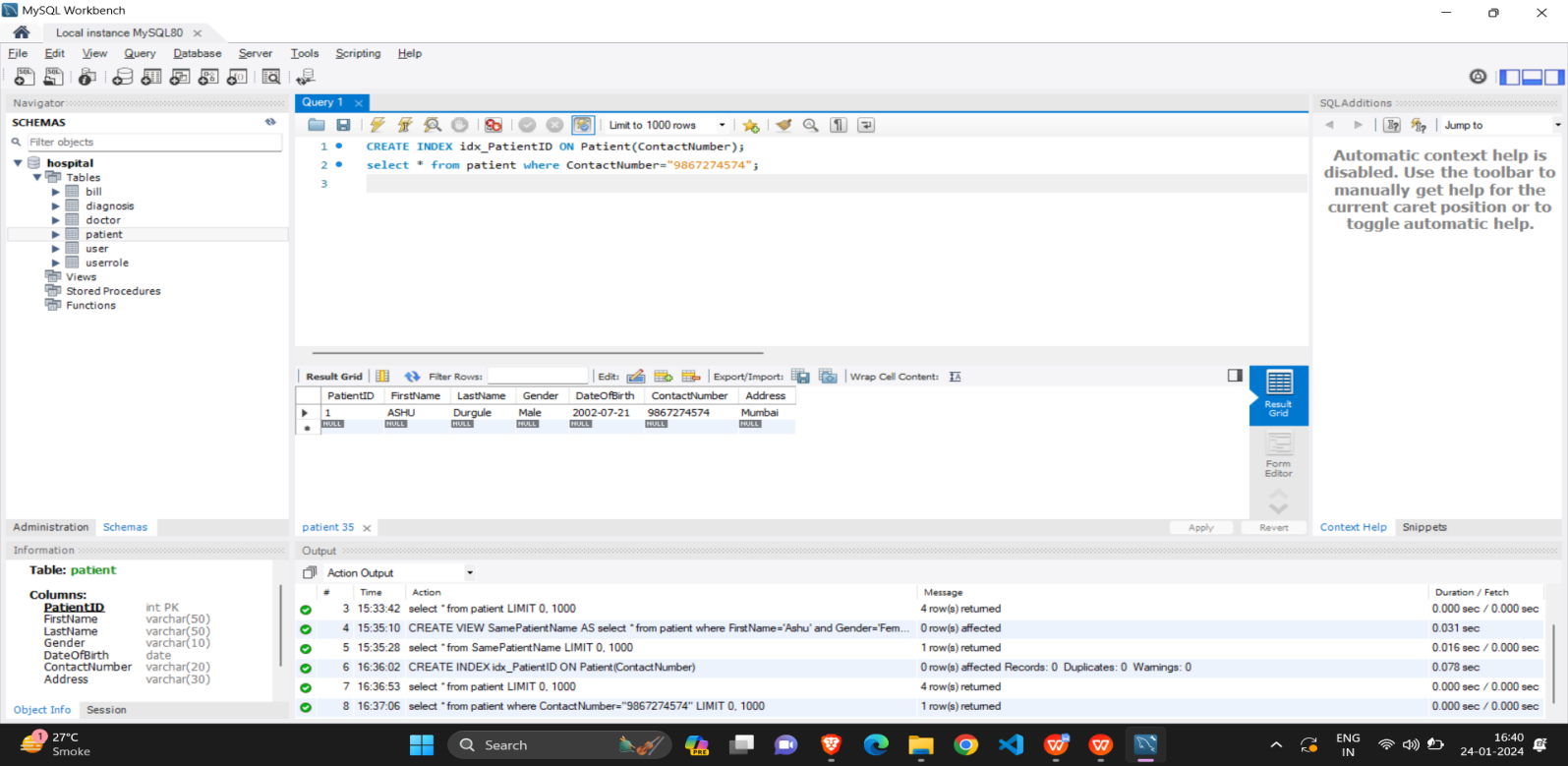
**where FirstName='Ashu' and Gender='Female';**



Indexing: ***Indexing is a database optimization technique used to improve the speed and efficiency of data retrieval operations on a database table. It involves creating a data structure, known as an index, which provides a fast and direct path to locate and access specific rows of data within a table.***

**CREATE INDEX idx\_PatientID ON Patient(ContactNumber);**

**select \* from patient where ContactNumber="9867274574";**



Relationships: ***patient to diagnosis :one to many***

***patient to bill :one to many***

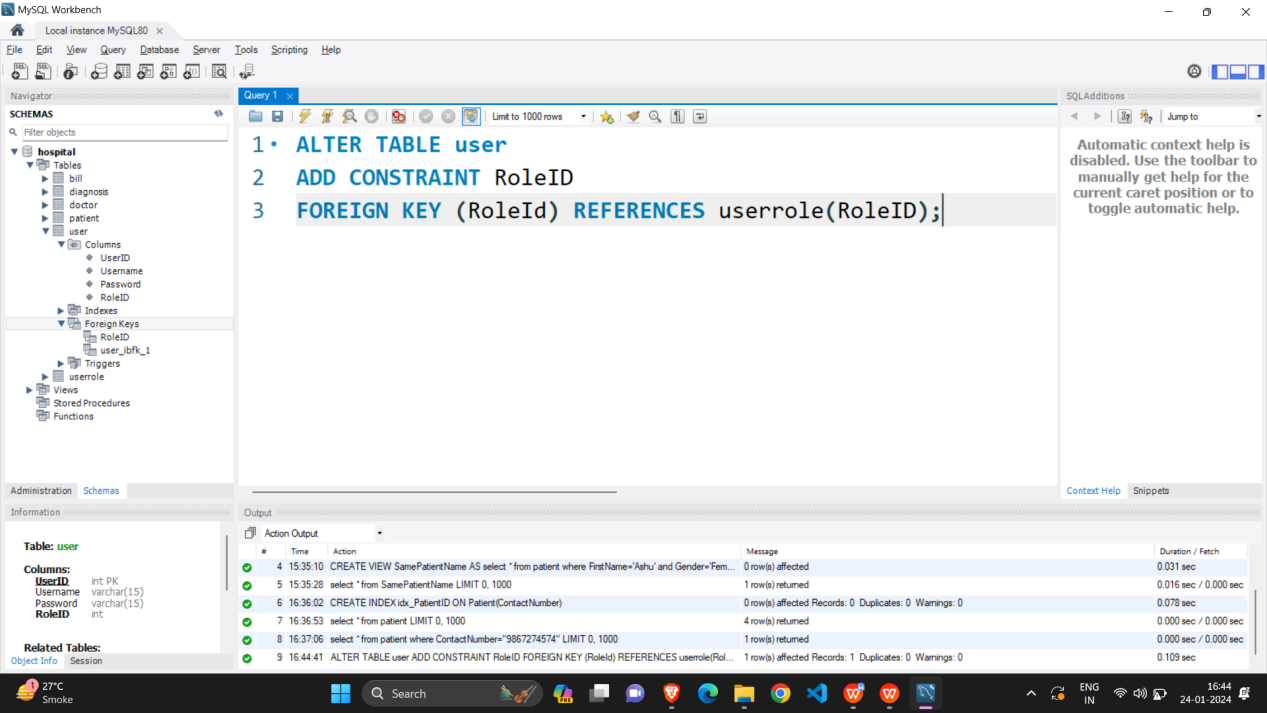
***Doctor to diagnosis :one to many***

***UserRole to User :one to many***

**ALTER TABLE user**

**ADD CONSTRAINT RoleID**

**FOREIGN KEY (RoleId) REFERENCES userrole(RoleID);**



Stored Procedure: ***A stored procedure is a precompiled collection of one or more SQL statements or procedural logic, designed to perform a specific task or set of tasks in a database. Stored procedures are stored in the database management system and can be executed repeatedly, providing a way to encapsulate and manage complex database operations, enhance performance, and promote code reusability.***

**DELIMITER //**

**CREATE PROCEDURE GetPatientInfo(IN p\_patientID INT)**

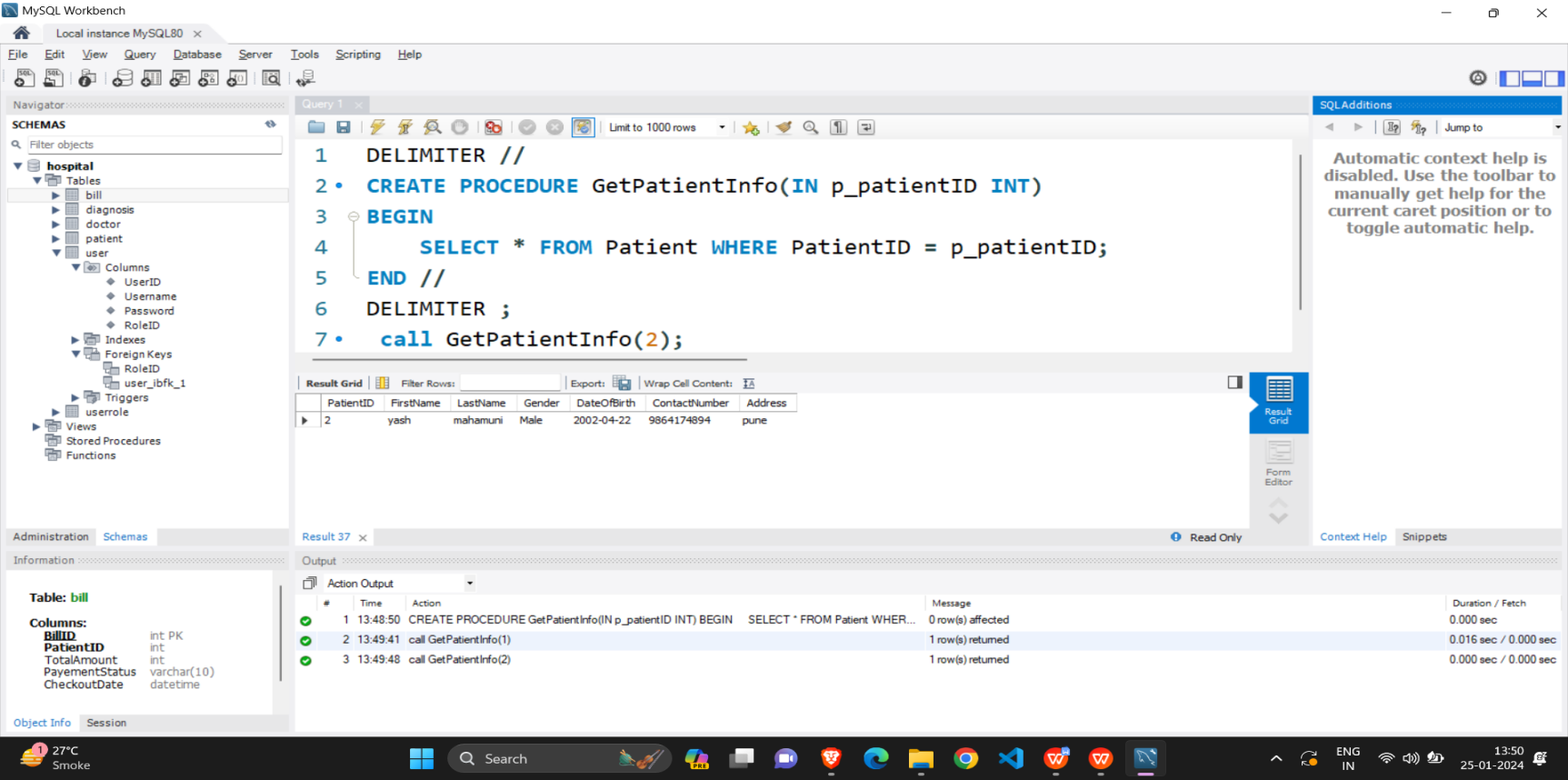
**BEGIN**

**SELECT \* FROM Patient WHERE PatientID = p\_patientID;**

**END //**

**DELIMITER ;**

**call GetPatientInfo(2);**



Triggers:  ***A trigger is a predefined set of instructions or actions in a database management system that automatically executes in response to a specific event, such as an INSERT, UPDATE, DELETE, or other database operation, occurring on a specified table or view. Triggers are used to enforce business rules, maintain data integrity, or automate tasks, providing a way to perform actions automatically without requiring explicit user intervention.***

**DELIMITER //**

**create**

**trigger trigger2 before insert**

**on patient**

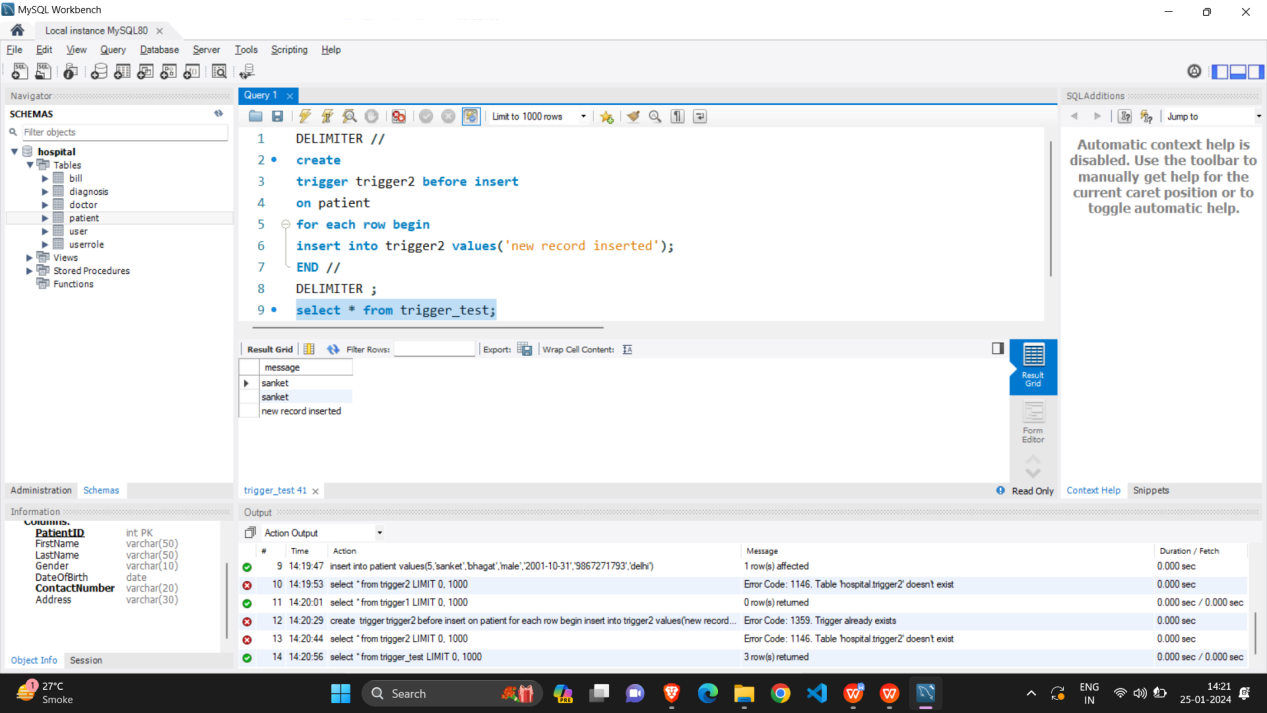
**for each row begin**

**insert into trigger2 values('new record inserted');**

**END //**

**DELIMITER ;**

**select \* from trigger\_test;**

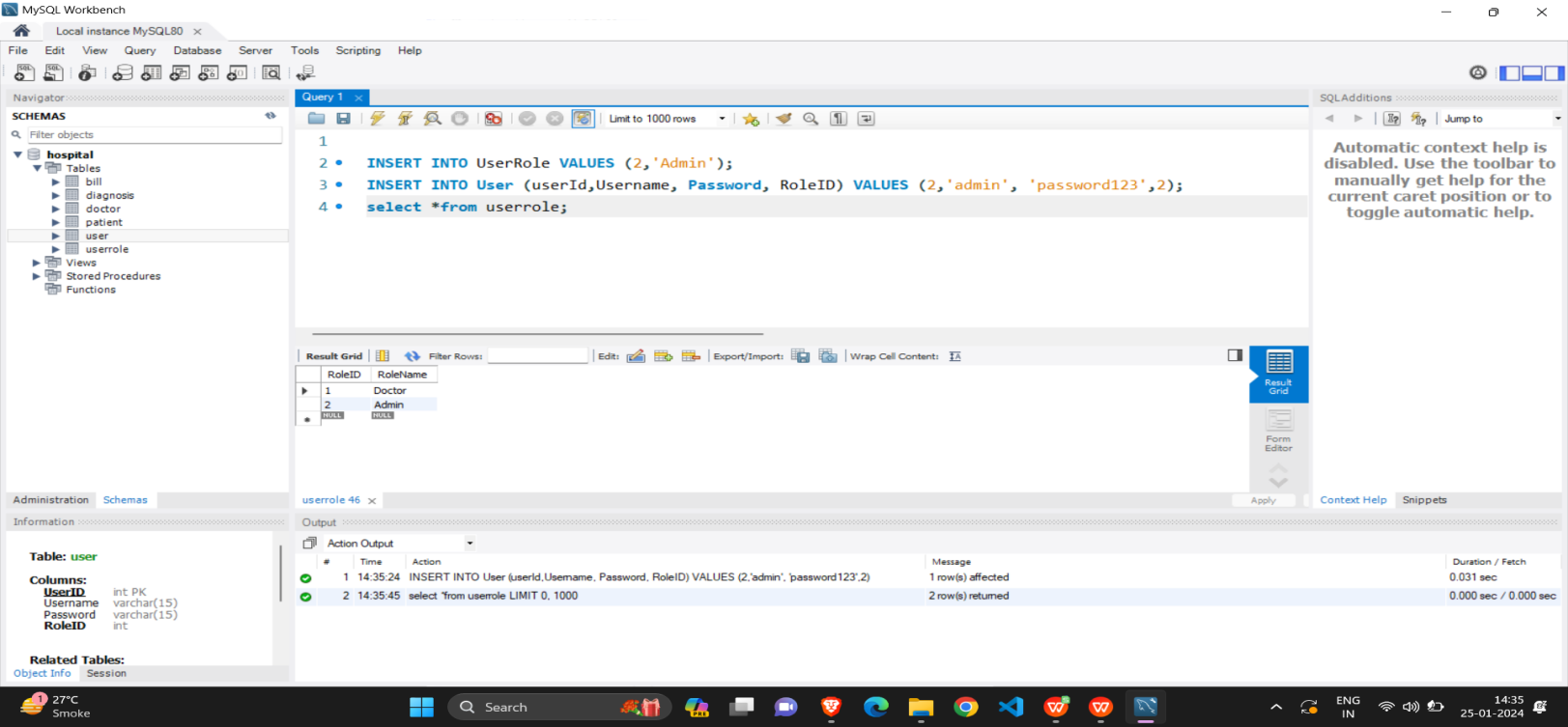


- Write necessary queries to register new user roles and personas

**INSERT INTO UserRole VALUES (2,'Admin');**

**INSERT INTO User (userId,Username, Password, RoleID) VALUES (2,'admin', 'password123',2);**

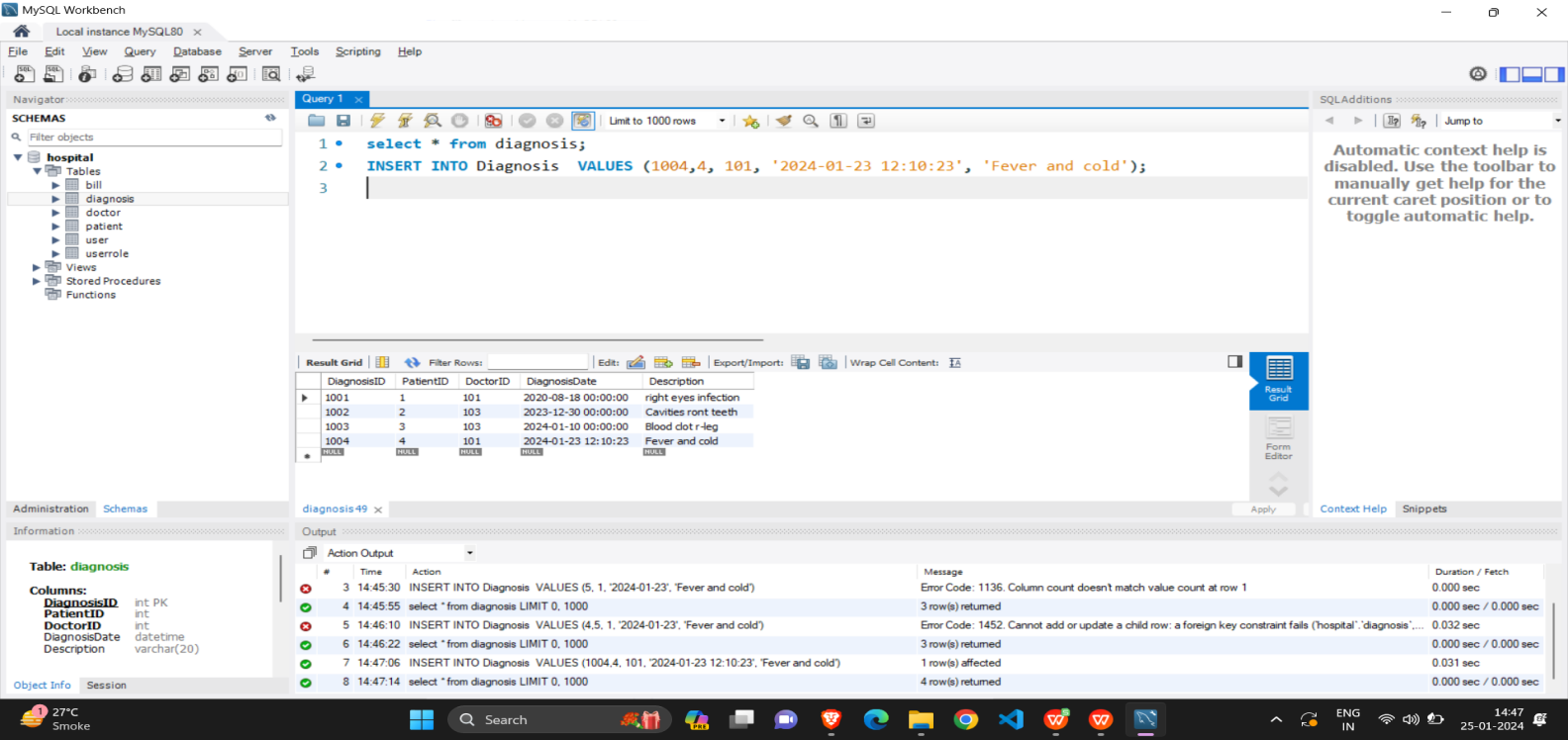
**select \*from userrole;**



- Write necessary queries to add to the list of diagnosis of the patient tagged by date.

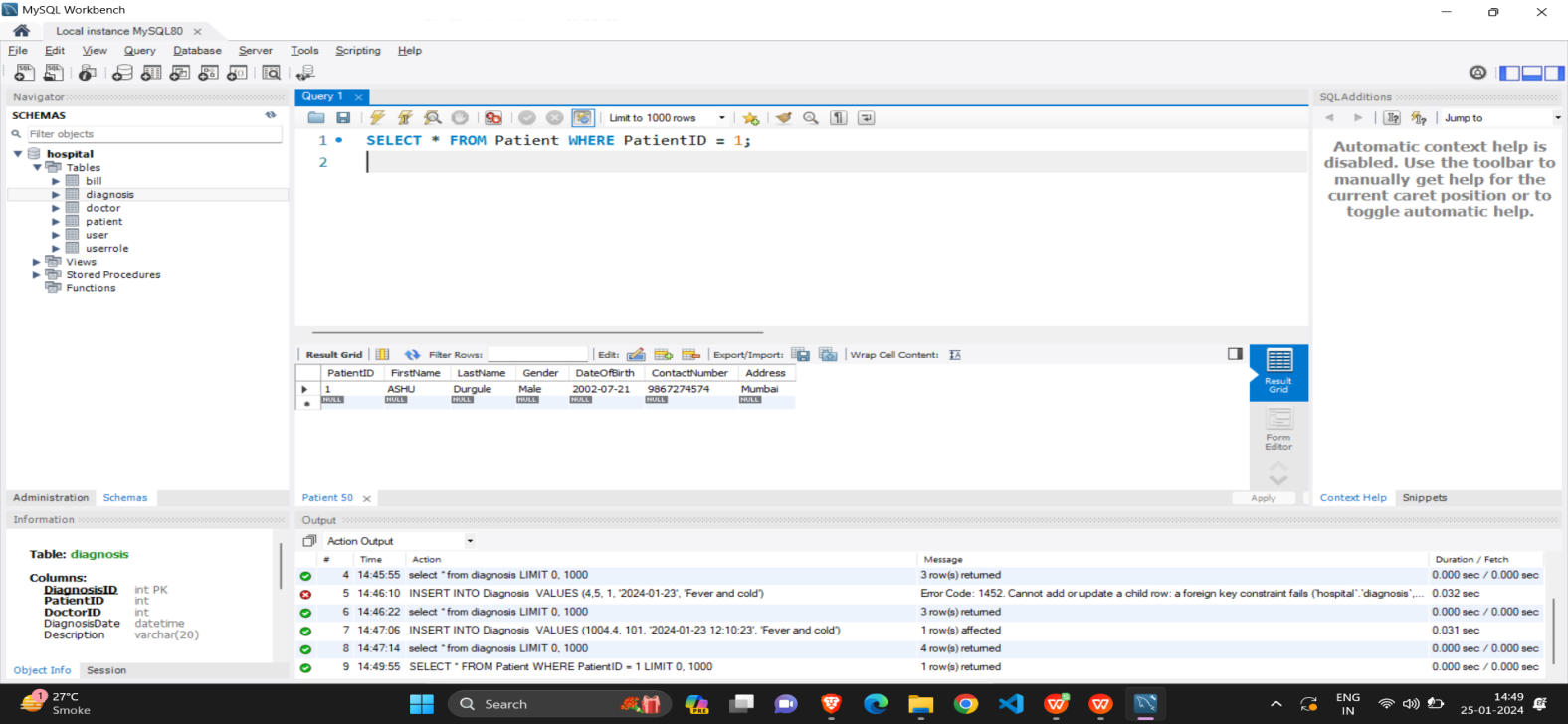
**INSERT INTO Diagnosis VALUES (1004,4, 101, '2024-01-23 12:10:23', 'Fever and cold');**

**select \* from diagnosis;**



- Write necessary queries to fetch required details of a particular patient.

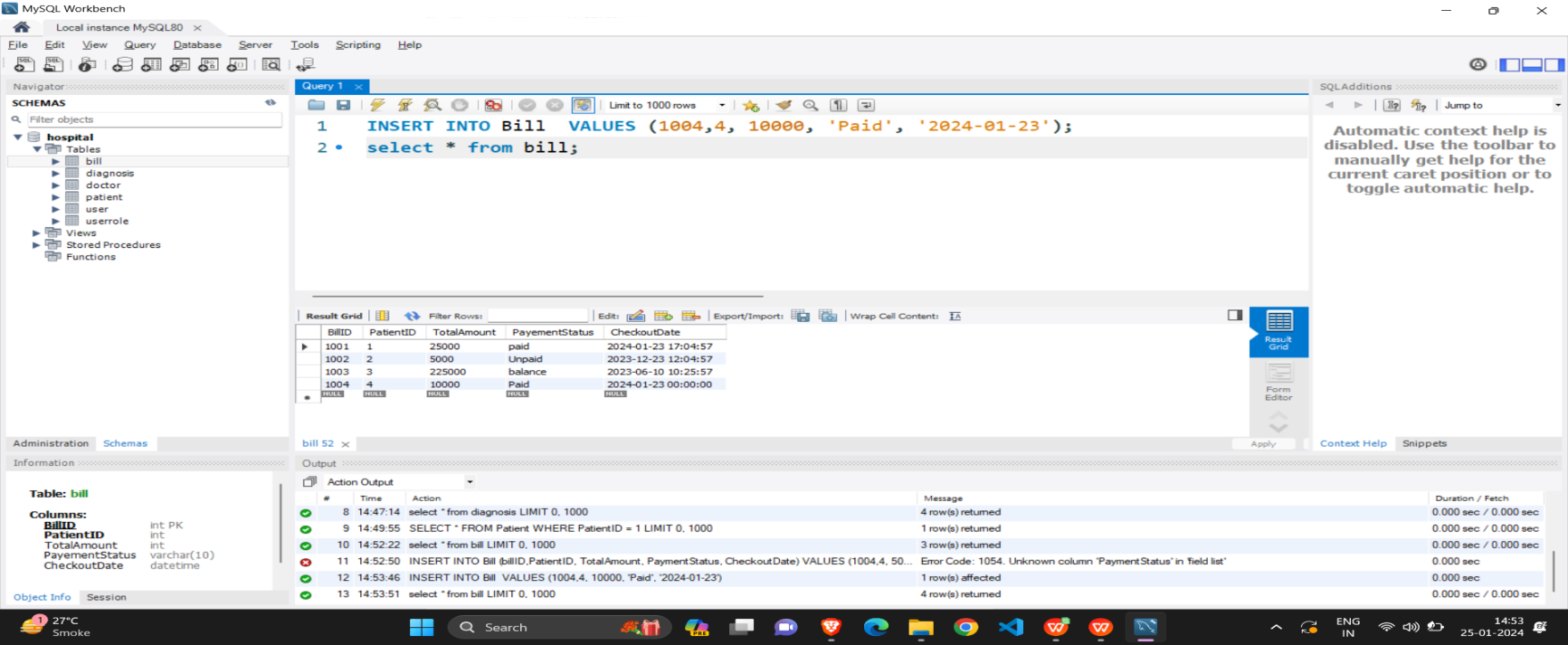
**SELECT \* FROM Patient WHERE PatientID = 1;**



- Write necessary queries to prepare bill for the patient at the end of checkout.

**INSERT INTO Bill VALUES (1004,4, 10000, 'Paid', '2024-01-23');**

**select \* from bill;**



-Write necessary queries to fetch and show data from various related tables (Joins)

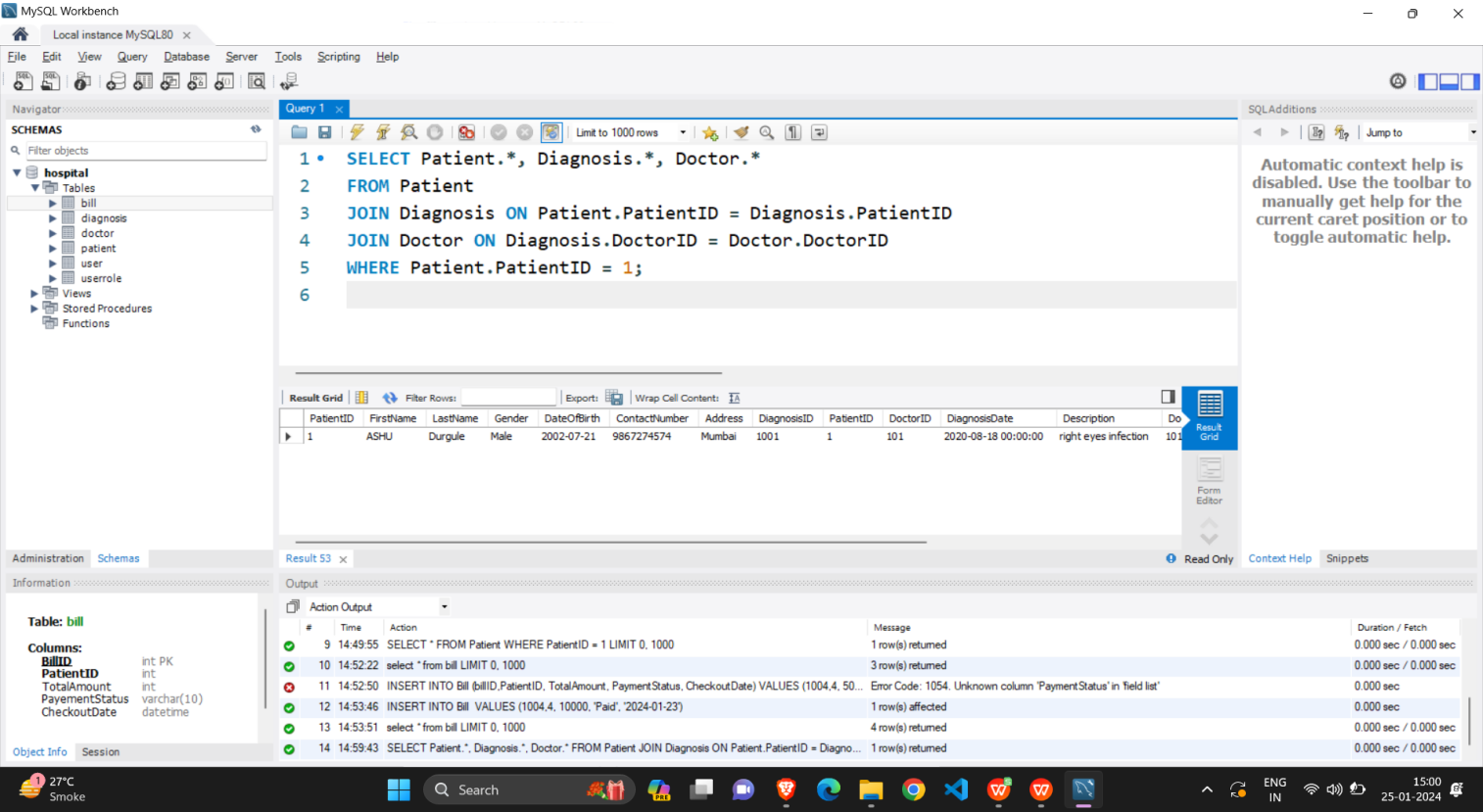
**SELECT Patient.\*, Diagnosis.\*, Doctor.\***

**FROM Patient**

**JOIN Diagnosis ON Patient.PatientID = Diagnosis.PatientID**

**JOIN Doctor ON Diagnosis.DoctorID = Doctor.DoctorID**

**WHERE Patient.PatientID = 1;**



***Q2)***

Write a report on your understanding of Rendering and Design Patterns. Mention and elaborate where a particular Rendering pattern is applicable and is well suited for which use case.

**Rendering**

Client-Side Rendering->

Client-side rendering (CSR) is the process of rendering web pages on the client using JavaScript. In this approach, the server sends the initial HTML file, but the client then uses JavaScript to dynamically update the page as needed. This allows for more interactive and responsive web pages, as the client can update specific parts of the page without needing to reload the entire page.

Advantages:

* More dynamic and interactive web applications
* Can provide a smoother and more seamless user experience
* Can reduce the need for additional server requests

Disadvantages:

* Slower initial load times
* Can be less SEO-friendly, as search engines may have difficulty indexing client-rendered content

Working of CSR: When a user requests a page, the server sends the initial HTML file, along with any required JavaScript files. The client then uses JavaScript to update the page as needed, without needing to reload the entire page.

Example:One example of a popular CSR framework is React. With React, you can write JavaScript code that updates the DOM as needed, providing a more interactive and dynamic web application.

Such as social media platforms or e-commerce websites.

Server-Side Rendering->

Server-side rendering (SSR) is the process of rendering web pages on the server and sending the fully-rendered HTML to the client. In this approach, the server generates the HTML, including any dynamic data, and sends it to the client as a complete page. The client then displays the page without any further processing.

Advantages:

* Faster initial load times
* Improved SEO optimization
* Can provide a better user experience for users with slower internet connections or less powerful devices

Disadvantages:

* Can require more server resources and maintenance
* This can result in slower subsequent page loads if the client needs to make additional server requests

Working of SSR: When a user requests a page, the server generates the HTML for that page, including any dynamic data. The fully-rendered HTML is then sent to the client, which can display the page without any further processing.

Example: popular SSR framework is Next.js. With Next.js, you can write React code and have it automatically rendered on the server, providing the benefits of SSR without having to manage the server yourself.commonly used for content-heavy websites, such as blogs or news websites, where fast initial load times and good SEO optimization are important.

Server-Side Generation->

Server-side generation (SSG) is a hybrid approach that combines the benefits of SSR and CSR. In this approach, the server generates static HTML files for each page, but also includes client-side JavaScript that can be used to update the page as needed.

Advantages:

* Fast initial load times
* Dynamic updates as needed
* Can provide a better user experience for users with slower internet connections or less powerful devices

Disadvantages:

* Can be more complex to set up and maintain

May not be suitable for applications that require real-time updates

Working of SSG: When a user requests a page, the server generates a static HTML file for that page, along with any required JavaScript files. The client can then display the page immediately, without needing to wait for any additional server requests. The client-side JavaScript can be used to update the page as needed.

Example:One example of a popular SSG framework is Gatsby. With Gatsby, you can write React code and have it automatically generated into static HTML files, providing the benefits of SSG without needing to manage the server yourself. commonly used for static websites, such as portfolios or landing pages, that require fast initial load times and some degree of interactivity. It can also be used for more complex applications that don’t require real-time updates.

Incremental static generation->

Incremental Static Regeneration, or ISR, is a concept primarily associated with Next.js, a React-based framework used for building web applications. This technique allows developers to update static pages after they have been generated, eliminating the need to rebuild the entire site. ISR is especially beneficial for large sites with numerous static pages that don’t frequently change, but where some pages might need more regular updates.

Working with example: During the initial build process, static pages are generated and served to users. Developers can specify a re-validation period for each static page. Once this period elapses, the next request to the page triggers a regeneration of the page in the background. While the page is being regenerated, the old (stale) version of the page continues to be served to users.

Upon completion of the regeneration, the old version of the page is replaced with the new one for subsequent requests. This method combines the advantages of static generation, such as performance and reliability, with the ability to update content dynamically when necessary. It is particularly useful for sites with content that changes periodically but not constantly, like e-commerce sites, blogs, or news sites.

Progressive Hydration ->

A Faster React Approach on Client Side.Progressive Hydration is introduced to reduce the hydration of server-side content on the client or to prioritize the content that needs to be hydrated when needed rather than to hydrate the whole server-side content on page load.

Example: Suppose in an E-commerce website there are (1) the Shopping Cart component, (2) the Buy Now Button component, and (3) the Product Reviews component. From a user interest perspective, the Buy Now Button component is in the center of the screen so is likely to get the user’s attention first, and the Shopping Cart component is also on-screen and related especially if that button is clicked. However, the user can’t see the Product Reviews component at all. From an optimization perspective, progressive hydration can be used to delay execution of the Product Reviews component code until it scrolls into view; maybe the user won’t scroll down to it, so why bother?

Selective Hydration->

 This lets you start hydrating your app as early as possible, before the rest of the HTML and the JavaScript code are fully downloaded. It also prioritizes hydrating the parts the user is interacting with, creating an illusion of instant hydration.

These features solve three long-standing problems with SSR in React:

* You no longer have to wait for all the data to load on the server before sending HTML. Instead, you start sending HTML as soon as you have enough to show a shell of the app, and stream the rest of the HTML as it’s ready.
* You no longer have to wait for all JavaScript to load to start hydrating. Instead, you can use code splitting together with server rendering. The server HTML will be preserved, and React will hydrate it when the associated code loads.
* You no longer have to wait for all components to hydrate to start interacting with the page. Instead, you can rely on Selective Hydration to prioritize the components the user is interacting with, and hydrate them early.

React server components->

React Server Components allow you to write UI that can be rendered and optionally cached on the server. In Next.js, the rendering work is further split by route segments to enable streaming and partial rendering, and there are three different server rendering strategies:

* [Static Rendering](https://nextjs.org/docs/app/building-your-application/rendering/server-components" \l "static-rendering-default)
* [Dynamic Rendering](https://nextjs.org/docs/app/building-your-application/rendering/server-components" \l "dynamic-rendering)
* [Streaming](https://nextjs.org/docs/app/building-your-application/rendering/server-components" \l "streaming)

There are a couple of benefits to doing the rendering work on the server, including:

* Data Fetching: Server Components allow you to move data fetching to the server, closer to your data source. This can improve performance by reducing time it takes to fetch data needed for rendering, and the number of requests the client needs to make.
* Security: Server Components allow you to keep sensitive data and logic on the server, such as tokens and API keys, without the risk of exposing them to the client.
* Caching: By rendering on the server, the result can be cached and reused on subsequent requests and across users. This can improve performance and reduce cost by reducing the amount of rendering and data fetching done on each request.
* Bundle Sizes: Server Components allow you to keep large dependencies that previously would impact the client JavaScript bundle size on the server. This is beneficial for users with slower internet or less powerful devices, as the client does not have to download, parse and execute any JavaScript for Server Components.
* Initial Page Load and [First Contentful Paint (FCP)](https://web.dev/fcp/" \t "https://nextjs.org/docs/app/building-your-application/rendering/_blank): On the server, we can generate HTML to allow users to view the page immediately, without waiting for the client to download, parse and execute the JavaScript needed to render the page.
* Search Engine Optimization and Social Network Share-ability: The rendered HTML can be used by search engine bots to index your pages and social network bots to generate social card previews for your pages.
* Streaming: Server Components allow you to split the rendering work into chunks and stream them to the client as they become ready. This allows the user to see parts of the page earlier without having to wait for the entire page to be rendered on the server.

Example: Suppose in an E-commerce website we can make all buttons as client component and all other non-interactive component as server component. Which will reduce the size of js Bundles to downloaded hence load the page faster.

Static rendering->

Static rendering refers to the process of pre-generating HTML pages during the build phase of a web application. The generated HTML pages are then served to users without the need for server-side rendering or dynamic content generation at runtime. This approach is often used to improve performance, reduce server load, and ensure fast page loads.

Example:Static rendering is useful when a route has data that is not personalized to the user and can be known at build time, such as a static blog post or a product page.

Streaming Server-Side Rendering->

Streaming Server-Side Rendering (SSR) is a technique in web development where the server starts sending HTML content to the client as it becomes available, instead of waiting for the entire HTML to be generated before sending it to the client. This can lead to faster perceived page loads, especially on slow connections, as users start seeing content sooner.Node streams allow us to stream data into the response object, which means that we can continuously send data down to the client. The moment the client receives the chunks of data, it can start rendering the contents.

Streaming SSR can be more complex depending on your application's structure, and you might want to explore frameworks like Next.js or libraries like React's react-dom/server for a more comprehensive SSR solution, especially if you're working with React or other frontend libraries.

**Design Pattern**

Compound Pattern->

The Compound Design pattern is an approach that encourages creating components or elements that can be combined or composed to build more complex and feature-rich UI elements. It promotes reusability, maintainability, and consistency across different parts of a system.

Key Principles:

* Reusability: Components are designed to be modular and reusable. Each component should have a specific responsibility and can be combined with others to create new UI elements.
* Composability: Components can be easily combined or composed to create larger, more complex structures. This promotes flexibility in design and encourages the building of diverse layouts and interfaces.
* Consistency: By using consistent building blocks, you ensure a uniform look and feel throughout your application or website.

Advantages of Compound Design:

* Modularity: Components are self-contained and can be modified or replaced without affecting other parts of the system.
* Reusability: Building blocks can be reused across different parts of the application, reducing redundancy and promoting consistency.
* Scalability: As your application grows, you can easily create new, complex components by combining existing ones.
* Maintainability: Changes to a specific component don't necessitate changes across the entire application, making maintenance more manageable.

HOC Pattern->

The Higher-Order Component (HOC) pattern is a design pattern in React that involves wrapping a component with a higher-order function to enhance its functionality or behavior. HOCs are functions that take a component and return a new component with additional props or modified behavior.

This is a simple example, and HOCs can be used for a wide range of purposes, such as:

Code Re-usability: Wrap multiple components with the same behavior without duplicating code.

Cross-Cutting Concerns: Implement features like logging, authentication, or data fetching in a modular way that can be applied to different components.

Prop Manipulation: Modify or add props before passing them down to the wrapped component.

Conditional Rendering: Control whether a component should render based on certain conditions.

State Abstraction: Manage state logic in a separate HOC to keep components focused on rendering.

It's important to note that while HOCs are a powerful pattern, they can be less explicit and may lead to complex component hierarchies. As an alternative, consider using React hooks (e.g., useEffect, useState) and render props for achieving similar functionality in a more concise and readable way, especially with the advent of React's functional components and hooks.

Hooks Pattern->

React Hooks are a powerful feature introduced in React 16.8 that allows functional components to use state and lifecycle features that were previously only available in class components. Hooks provide a more straightforward and concise way to manage state, side effects, and other React features.

Example: useState, useEffect , useContext, useReducer etc.

These are just a few examples of how React Hooks can be used in functional components. Hooks provide a more declarative and composable way to manage state and side effects in React applications. Each hook serves a specific purpose and can be combined to build complex and efficient components.

Container/Presentational Pattern->

The Container/Presentational pattern (also known as Smart/Dumb components) is a design pattern in React that separates the concerns of managing state and logic (Container components) from rendering and displaying UI elements (Presentational components). This pattern helps in maintaining a clear separation of concerns, making the codebase more modular, reusable, and easier to understand.

Container Components:

* Responsibility: Managing state, handling logic, and interacting with data sources.
* Connected to Redux: Often connected to a state management system like Redux.
* May Have Lifecycle Methods: Use lifecycle methods for tasks such as data fetching, subscriptions, or managing side effects.
* Should Not Contain Much JSX: Minimal or no JSX markup, as their primary focus is on behavior.

Presentational Components:

* Responsibility: Rendering UI elements and receiving props.
* Stateless: Should be stateless functional components whenever possible.
* Focus on Markup and Styling: Should contain minimal or no logic, with a primary focus on the structure and style of the UI.
* Receive Data via Props: Receive data and callbacks from the container components as props.

Render Props Pattern ->

The Render Props pattern is a technique in React where a component receives a function as a prop, allowing it to render its children using the provided function. This pattern is particularly useful for sharing code or behavior between components without relying on higher-order components or other abstraction layers.

Benefits of Render Props:

* Code Reusability: The logic is encapsulated in the RenderPropsComponent, making it easy to reuse across different parts of the application.
* Composability: Consumers have the flexibility to compose and customize the rendering behavior as needed.
* Separation of Concerns: The component providing the rendering logic (RenderPropsComponent) is separate from the UI rendering logic, promoting a clean separation of concerns.

The Render Props pattern is flexible and can be used for a variety of purposes. It allows for greater composability, code sharing, and customization of behavior without the need for complex higher-order components. Components utilizing the Render Props pattern often provide a way for consumers to customize the rendering behavior, making them more versatile.