

## ABSTRACT

In the dynamic realm of real estate management, technological advancements play a pivotal role in streamlining processes and enhancing efficiency. This study presents the development of a web application leveraging the MERN (MongoDB, Express.js, React.js, Node.js) stack, aimed at revolutionizing real estate management practices.

Traditional real estate management systems often suffer from inefficiencies and lack of scalability due to fragmented data handling and manual processes. This web application addresses these challenges by integrating modern technologies to create a cohesive platform for property listing, management, and transactions.

We hypothesize that the implementation of the MERN stack in our web application will significantly improve user experience, streamline property management workflows, and facilitate seamless communication between stakeholders. By leveraging React.js for the frontend, Node.js for backend logic, Express.js for routing, and MongoDB for database management, we anticipate a more agile and scalable solution compared to conventional systems.

This web application aims to redefine real estate management practices by harnessing the power of modern web technologies, paving the way for a more efficient and transparent real estate ecosystem.

This application supports uploading of pictures of real estate properties, creating an account for uploading and contacting the owner. It also features Map to show the exact location by tracking the Latitude and Longitude of the location.

We have also harnessed the latest technologies, using various libraries and frameworks to build this application to enhance User Experience.

Admin can add any location for real estate property, view details and delete user, realtor, and property information (product). As the website is user-friendly, it contains both simple search for any general user and administrator who can upload the details of the property. The owner of the property can register in this website securely as the password is hashed securely. The details of the property can be easily added and modified.

## TABLE OF CONTENTS

<b>CERTIFICATE</b> -----	ii
<b>DECLARATION</b> -----	iii
<b>ACKNOWLEDGEMENTS</b> -----	iv
<b>ABSTRACT</b> -----	v
Chapters (Title)	Page Numbers
<b>Chapter 1: Introduction</b> -----	1
1.1 Introduction	
1.2 Objectives	
1.3 Scope of Project	
<b>Chapter 2: Literature Review</b> -----	6
2.1 Literature Survey	
2.2 Real-Estate Transaction Stages	
<b>Chapter 3: System Design</b> -----	7
3.1 System Requirements	
3.2 System Design and Table Structure	
<b>Chapter 4: Flow Charts and DFD</b> -----	9
4.1 System Flow Charts	
4.2 Data Flow Diagrams	
4.3 Entity Relationship Diagram	
<b>Chapter 5: Coding Implementation – Client and Server</b> -----	12
<b>Chapter 6: Testing</b> -----	22
6.1 System Testing	
6.2 Testing Methods	
<b>Chapter 7: Output Result &amp; Conclusion</b> -----	27
7.1 Output of the Application / Results	
7.2 Conclusion of the Application	
7.2 Future Scope of the system	
<b>Bibliography</b> -----	31

# **Chapter 1: INTRODUCTION**

## **1.1 Introduction**

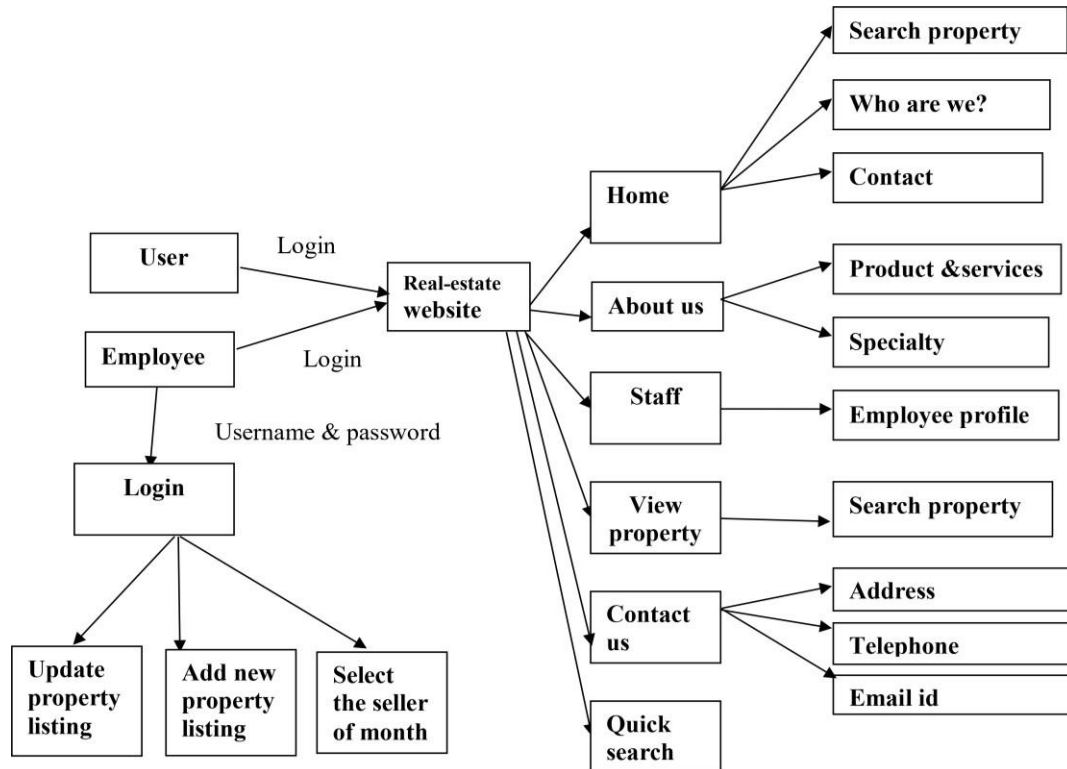
Modern technology has been industrialized to the extent that even searching for land and apartments is made possible over the internet. The process of searching for a dream house or apartment can be done over the internet. Customers looking to buy or rent a new home, apartment or any other property can search over the internet.

Customer looking to buy or rent a new home, apartment or any other property can search over the internet, while sitting at their home. Now customers will not have to go to the agents personally they can search for their desired home or apartment of a reasonable rate which suits their pocket. Customers looking to buy or rent a new home, apartment or any other property can search over the internet. Customer can search for desired plot or house in a particular area.

They can specify the number of rooms, bathroom they need and then can make the Search. They can also describe what all facilities they want near their house like schools, malls, garages, shops etc. They can search for an apartment or plot which suits their budget. They can also specify the amount by entering the cost. Now customers will not have to go to the agents personally, they can search for their desired home or apartment at a reasonable rate which suits their pocket. And they can also have the view of apartment online, as the pictures are being uploaded.

This is an online web-based application and independent software product.

### Block Diagram:



### 1.2 Objectives:

The major objective of our real estate website is the convenience it offers.

- By sitting back at home, customers looking to buy a new home, apartment or any other property can search the desired property by several clicks of mouse buttons.
- Customers do not have to go to the agents personally, they can save their time by doing an online search.
- User friendly website for easily adding and deleting property details and location information.
- To integrate all information which customers' needs on the website.
- To allow the user to enter the desired amount for which he wishes to spend.
- To allow the customers to have a pictorial view of the apartment he wishes to buy.
- To overcome the limitations being faced during the manual system of handling the entire records of the company manager and to manage the work schedule effectively and efficiently as well.

### 1.3 Scope of the Project:

- To design a real estate website of a particular agency using information collected through the internet.
- To design a website which integrates all the information needed by the user.
- To allow the user to search plot & land according to his needs as the user can specify the location where he wants the land.
- Quickly and easily manages property listings.

## Chapter 2: LITERATURE REVIEW

### 2.1 Literature Review

Past studies concerning the relationship between the Internet and real estate can be classified into a few dominating themes. One direction is how the Internet has become and continues to be a very important tool for marketing real estate and related services. Rodriquez, Lipscomb and Yancey (1996) identified four different types of real-estate related sites, including those that offered both real estate for sale and real estate services, and provided an extensive list of these sites.

A growing body of literature looks at the effect of the Internet on retail sales, property and service (e.g., Mander, 1996; Wheaton, 1996; Schwarz, 1997; Borsuk, 1999; Hemel and Schmidt, 1999; Baen, 2000; and Miller, 2000). For example, Baen and Guttery (1997) examined how the Internet threatened the traditional relationship among licensees, real estate buyers and sellers, and how these developments would create savings for real estate consumers.

In the old days when we want to purchase a property we can't directly communicate with the owners. We must contact the help of mediators, but the mediators take lot of amount and it is also time-consuming process. In older days the property dealing procedure consisted of many steps like finding agent, appoint correct meeting time, location and so on. Up till now there was no Security in Online Real Estate System, Registration form improves the security by limiting user.

### 2.2 Real Estate transaction Stages

Real estate agents and firms are essentially market intermediaries, connecting buyers and sellers and facilitating the real estate transaction process. Traditionally, real estate sales can be divided into five stages: property listing, buyer search, property evaluation, negotiation and execution/closing. With the development and popularity of information technology, each stage of this process has been affected profoundly.

**Stage 1: Property listing.** In the past, real estate agents listed houses and entered them into a Multiple Listing Service (MLS) database. The MLS is today an online network of properties listed for sale and supported by the NAR. In effect, the MLS created a cartel-like role in managing information and virtually ensures that the agent will have a pivotal role in the real estate transaction.

**Stage 2: Buyer search.** Although potential buyers can search for homes on their own through browsing newspaper advertisements or call owners directly, most prospective buyers generally seek homes through agents that have access to MLS listings.

**Stage 3: Property evaluation.** Traditionally, after buyers find a property of interest, an agent often arranges for a walk-through showing or has access to a house through a lock-box. Now, it is possible to conduct a "virtual" walk-through online.

**Stage 4: Negotiation and agreement.** Negotiating the purchase agreement successfully can be considered as the most challenging task for a real estate agent. It involves advice regarding price, offers and counteroffers, and contract contingencies.

**Stage 5: Execution/closing.** The customary role of the escrow agent or attorney in directing the closing is also changing. The Electronic Signatures in Global and National Commerce Act or E-Sign legislation passed on October 1, 2000, significantly removed impediments for business and government transactions to be conducted electronically.

## Chapter 3: Design and Implementation

### 3.1 System Requirements:

#### Server Requirements:

- **Node.js:** The server-side code will be written in Node.js, so the server should have Node.js installed. Recommended Node.js version is the LTS (Long Term Support) version.
- **MongoDB:** The application will use MongoDB as the database, so the server should have MongoDB installed. MongoDB server can be hosted locally or on a cloud platform like MongoDB Atlas.

#### Client Requirements:

- **Web Browser:** The client-side code will be written using React.js, so users should have a modern web browser that supports JavaScript.
- **Internet Connection:** Users need a stable internet connection to access the web application.

#### Hardware Requirements:

- **Server Hardware:** The server should have sufficient CPU, RAM, and storage resources based on the expected traffic and workload of the application. For small to medium-scale applications, a standard virtual private server (VPS) with at least 2 CPU cores, 2 GB RAM, and SSD storage is recommended.
- **Client Hardware:** Since it's a web application, clients only need devices capable of running modern web browsers. This includes desktops, laptops, tablets, and smartphones.

#### Security Requirements:

- **Authentication:** The application should support user authentication and authorization to ensure that only authorized users can access certain features and data.
- **Data Encryption:** Sensitive data transmitted between the client and server should be encrypted using SSL/TLS to prevent eavesdropping and data tampering.

#### Scalability Requirements:

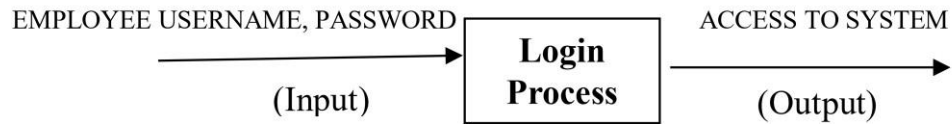
- **The architecture should be designed to scale horizontally or vertically to handle increasing user loads and data volumes.**
- **Load Balancing:** If the application is expected to have high traffic, a load balancer should be used to distribute incoming requests across multiple server instances.
- **Database Sharding:** As the data grows, sharding or partitioning the database may be necessary to improve performance and scalability.

#### Functional Specifications/ Requirements:

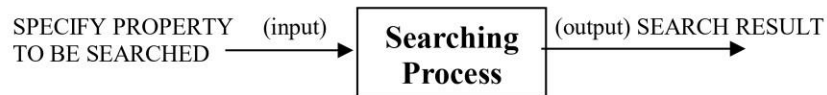
- **Manage listings, realtors, contact inquiries and website users via admin.**
- **Display listings in app with pagination.**
- **Form info should go to database and notify realtor(s) with an email.**
- **Frontend register/login to track inquiries.**
- **Only registered Users can upload property listing.**

## DESCRIPTION OF 'INPUT TO' AND 'OUTPUT OF' OF PROCESSES

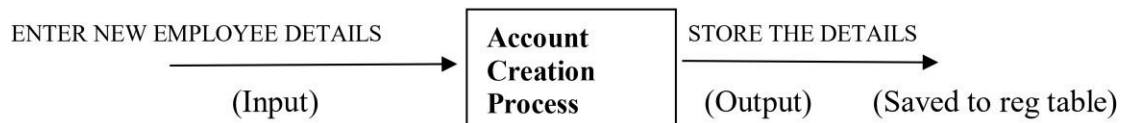
### Employee Login Process



### Property Searching Process



### Registering New User Process



### Update property listing process



### Updating New Property Process



## 3.2 System Design

The real estate web application will be built using the MERN (MongoDB, Express.js, React.js, Node.js) stack, providing seamless user experience and efficient backend operations.

### Frontend:

The front end, developed with React.js, will offer a responsive and intuitive interface. Components such as property listings, search filters, maps, and user profiles will facilitate property browsing. State management will be handled using Redux or React Context API, ensuring smooth handling of application state. React Router will manage client-side routing, enabling seamless navigation between pages without page reloads. Axios or Fetch API will handle HTTP requests to the backend API for fetching property data and user authentication. User authentication will be secured with JWT tokens, with sessions managed for persistent user logins. The UI will be designed to be responsive, catering to various devices.

### Backend:

The backend will consist of an Express.js server providing a RESTful API for CRUD operations. MongoDB will serve as the database, storing property data and user accounts, with Prisma ORM ensuring schema validation and data manipulation. Middleware functions will handle user authentication and authorization, verifying JWT tokens and protecting routes. Multer middleware will facilitate file uploads like property images. Error handling middleware will catch and handle errors gracefully, while security measures like input validation and password hashing will be implemented.

### Deployment:

The application can be hosted on cloud platforms like Github Pages with separate environments for development, testing, and production. CI/CD pipelines will automate testing and deployment processes, ensuring code changes are thoroughly tested before deployment. Monitoring tools like Prometheus and Grafana can be set up to track application performance and user behavior, while logging will record application events for debugging purposes.

This streamlined architecture ensures scalability, security, and maintainability, providing a robust foundation for real estate web application.

### 3.2.1 System Design Primary Objective:

In general, the following design objectives should be kept in mind:

- **Practicality:** The system must be stable and can be operated by people with an average IQ.
- **Efficiency:** This involves accuracy, timeliness, and comprehensiveness of the system output.
- **Cost:** It is desirable to aim for a system with a minimum cost subject to the condition that it must satisfy all the requirements.
- **Flexibility:** The system should be modifiable depending on the changing needs of the user.
- **Security:** This is a very important aspect of the design and should cover areas of hardware reliability, fallback procedures, physical security of data and provision for detection of fraud and abuse.



### 3.2.2 System Design Activities:

Several development activities are carried out during structured design. They are database design, implementation planning, system test preparation, and system interface specification and user documentation.

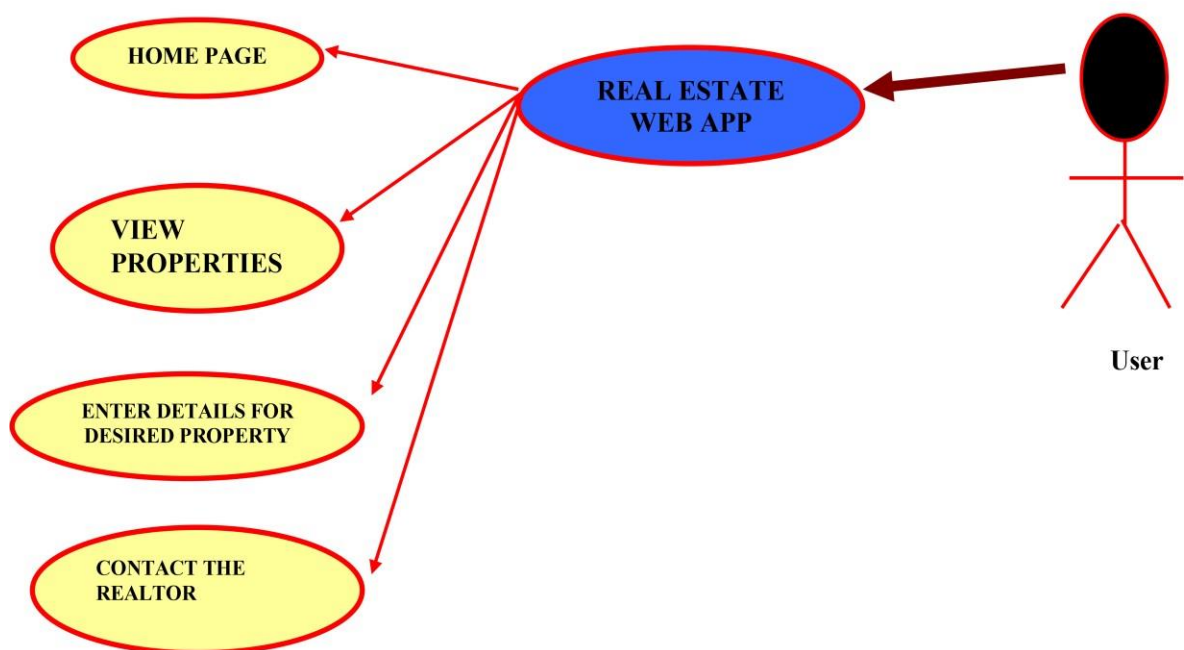
- Database Design: This activity deals with the design of the physical database. A key is to determine how the access paths are to be implemented.
- Program Design: In conjunction with database design is a decision on the programming language to be used and the flowcharting, coding, and the debugging procedure prior to conversion. The operating system limits the programming languages that will run of the system.
- System and Program Test Preparation: Each aspect of the system has separate test requirements. System testing is done after all programming and testing is completed. The test cases cover every aspect of the proposed system, actual operations user interface and so on.

### 3.2.3 Use case Diagram:

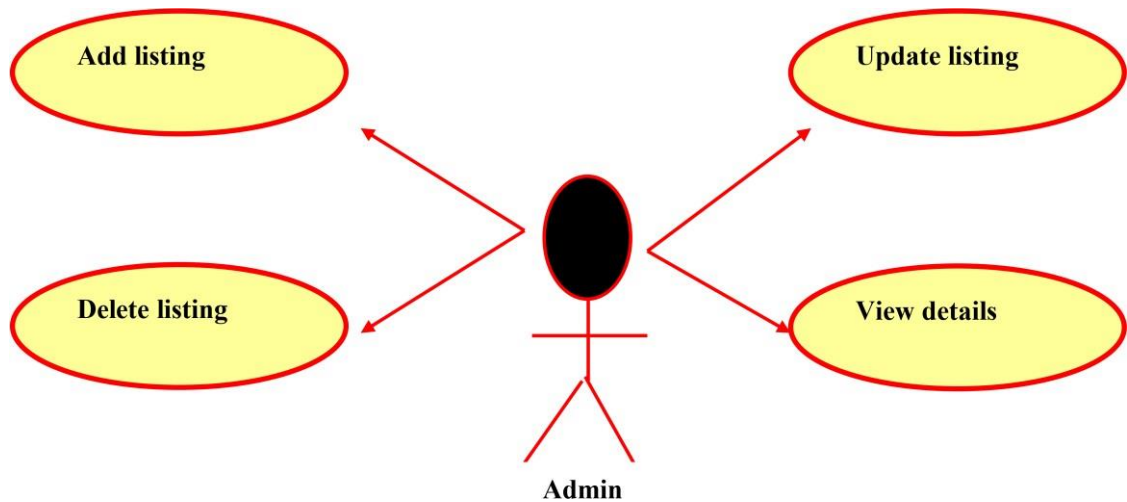
A use case diagram is a diagram which consists of a set of use cases and actors enclosed by system boundary, and association between use cases and actors. Use-cases diagram is especially important in organizing, modeling the behavior of the system.

A use case is a set of scenarios tied together by a common user goal. A scenario is a sequence of steps describing the interaction between a user and system.

Use case Diagram for User:



Use Case for Admin listing properties:



### Table Structures

#### CATEGORY

Category ID	Category Name	Categorydescription
Auto number	Text	text

#### EMPLOYEE

Empid	Fname	Lname	Password	Areacode	Phone	Email	Gender	Username	Datestart
Auto number	Text	text	Text	text	Text	text	text	text	text

#### LIST PROPERTY

Listgid	Desclong	Avaliability	Acres	Sqrfeet	Numbath	Numbed	Streetnum	Street	Zip
Auto no	Memo	Text	text	number	number	number	text	text	text

Catergoryid	Schooldist	Price	Access	Empid	Proprtyid
number	text	currency	currency	Auto no	Auto no

#### PROPERTY

Propertyid	Propertyname	Propertydescription
Auto number	Text	Text

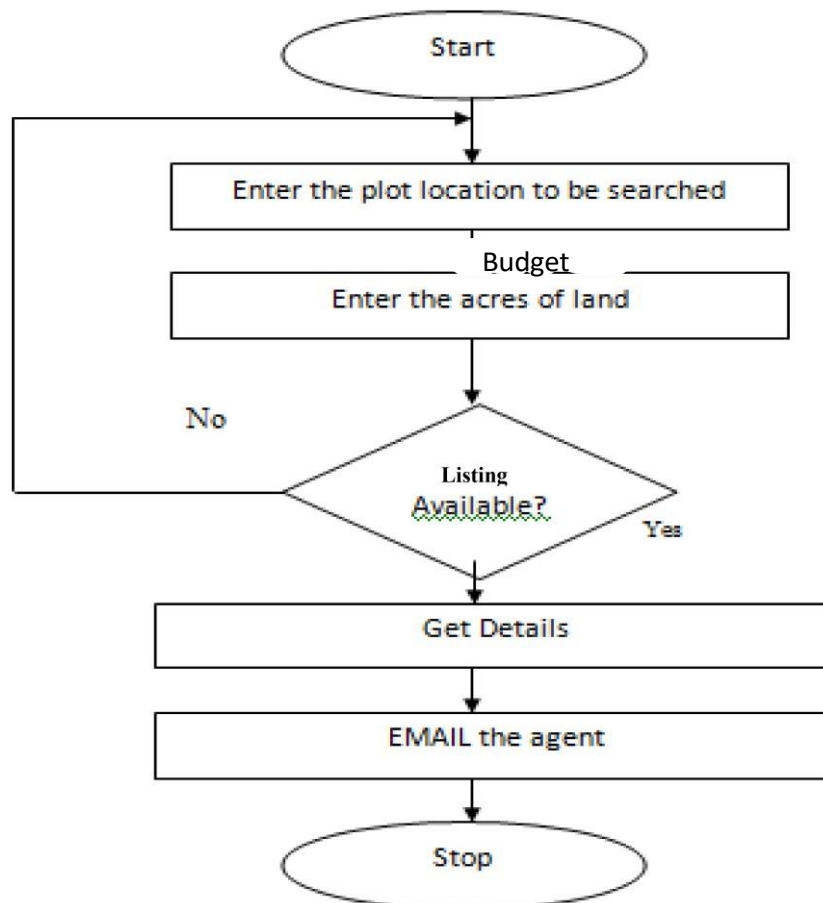
#### CONTACT

Emailed	subject	query
Text	Text	Text

## Chapter 4: Flow Chart and Data Flow Diagrams

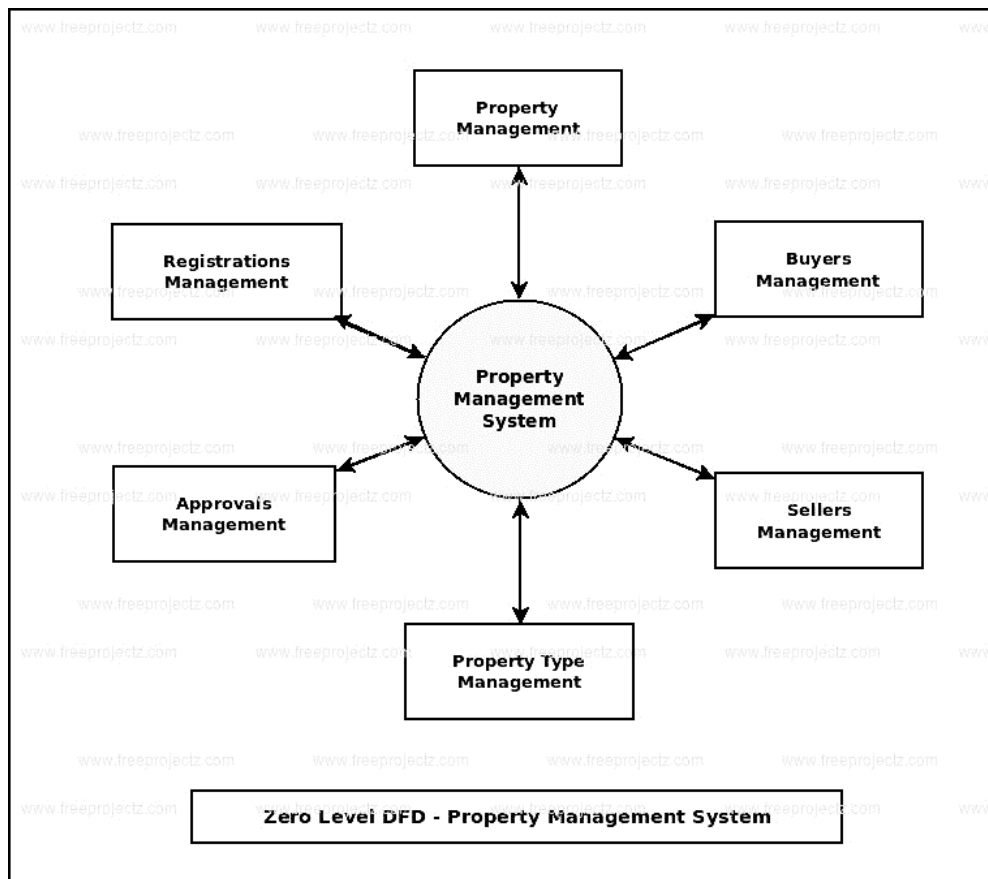
### 4.1 Flow Chart:

#### USER SEARCH FOR PROPERTY

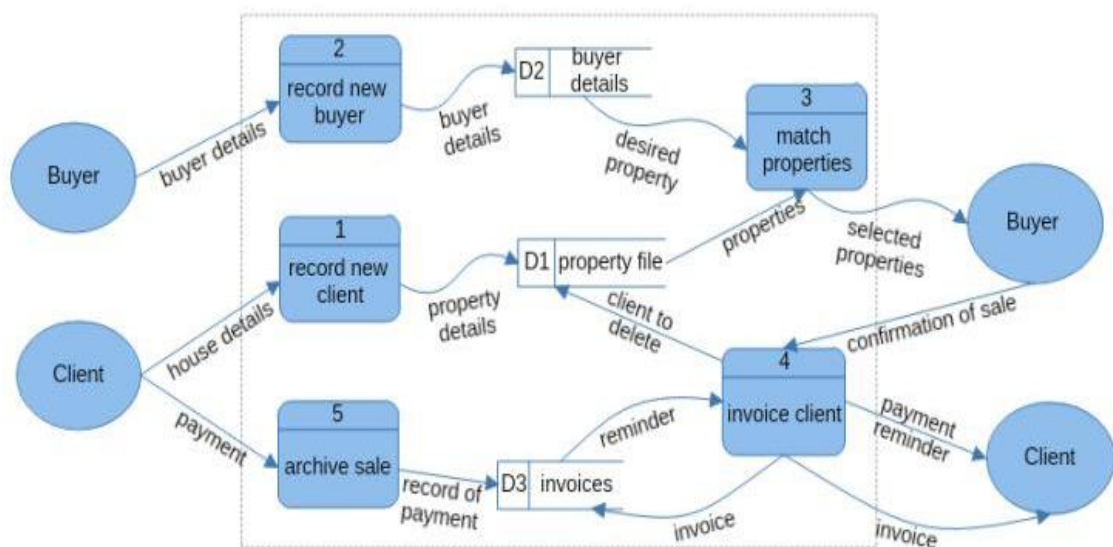


### 4.2 Data Flow Diagrams

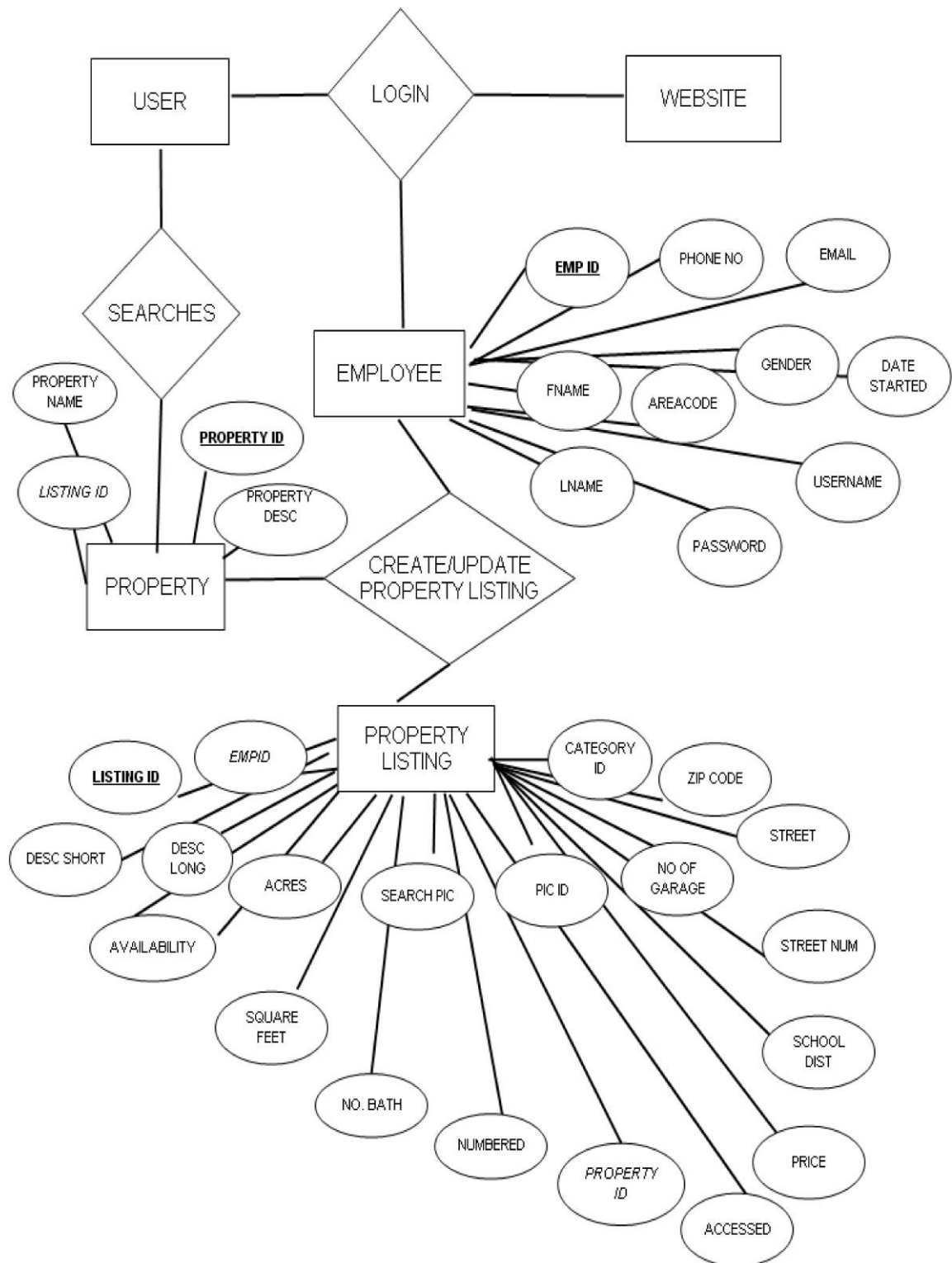
A graphical tool used to describe and analyze the movement of data through a system, manual or automated including the process, stores of data, and delays in the system. Data Flow Diagrams are the central tool and the basis from which other components are developed. The transformation of data from input to output, through processes, may be described logically and independently of the physical components associated with the system. The DFD is also known as a data flow graph or a bubble chart.



## Estate Agency Data Flow Diagram (Level 1)



### 4.3 Entity Relationship Diagram



## Chapter 5: Coding Implementation

Main implementation code of the Front End and Back End.

### 5.1 Front End of the Application (Client Side):

#### App.jsx (ReactJS Code)

```
import HomePage from “./routes/homePage/homePage”;
import { createBrowserRouter, RouterProvider } from “react-router-dom”;
import ListPage from “./routes/listPage/listPage”;
import { Layout, RequireAuth } from “./routes/layout/layout”;
import SinglePage from “./routes/singlePage/singlePage”;
import ProfilePage from “./routes/profilePage/profilePage”;
import Login from “./routes/login/login”;
import Register from “./routes/register/register”;
import ProfileUpdatePage from “./routes/profileUpdatePage/profileUpdatePage”;
import NewPostPage from “./routes/newPostPage/newPostPage”;
import { listPageLoader, profilePageLoader, singlePageLoader } from “./lib/loaders”;
function App() {
  const router = createBrowserRouter([
    {
      path: “/”,
      element: <Layout />,
      children: [
        {
          path: “/”,
          element: <HomePage />,
        },
        {
          path: “/list”,
          element: <ListPage />,
          loader: listPageLoader,
        },
        {
          path: “/:id”,
          element: <SinglePage />,
          loader: singlePageLoader,
        },
      ],
    },
    {
      path: “/login”,
      element: <Login />,
    },
    {
      path: “/register”,
      element: <Register />,
    },
  ]),
}
```

```

    {
      path: "/",
      element: <RequireAuth />,
      children: [
        {
          path: "/profile",
          element: <ProfilePage />,
          loader: profilePageLoader
        },
        {
          path: "/profile/update",
          element: <ProfileUpdatePage />,
        },
        {
          path: "/add",
          element: <NewPostPage />,
        },
      ],
    },
  ],
});

return <RouterProvider router={router} />;
}

export default App;
homePage.jsx

import { useContext } from "react";
import SearchBar from "../../components/searchBar/SearchBar";
import "../../homePage.scss";
import { AuthContext } from "../../context/AuthContext";

function HomePage() {

  const { currentUser } = useContext(AuthContext)

  return (
    <div className="homePage">
      <div className="textContainer">
        <div className="wrapper">
          <h1 className="title">Find Real Estate & Get Your Dream Place</h1>
          <p>
            Find your dream home from our list of properties. Developed by Ritam. Project
            Guide – Sartaj Sir.
          </p>
          <SearchBar />
          <div className="boxes">
            <div className="box">
              <h1>16+</h1>

```

```

        <h2>Years of Experience</h2>
      </div>
      <div className="box">
        <h1>200</h1>
        <h2>Award Gained</h2>
      </div>
      <div className="box">
        <h1>2000+</h1>
        <h2>Property Ready</h2>
      </div>
    </div>
    </div>
    </div>
    <div className="imgContainer">
      
    </div>
  </div>
);
}

```

export default HomePage;

**Rest of the code has been omitted to save space and for quick overview of the high level view of the code.**

### **Prisma Schema ORM:**

```

generator client {
  provider = "prisma-client-js"
}
datasource db {
  provider = "mongodb"
  url      = env("DATABASE_URL")
}
model Post {
  id      String    @id @default(auto()) @map("_id") @db.ObjectId
  title   String
  price   Int
  images  String[]
  address String
  city    String
  bedroom Int
  bathroom Int
  latitude String
  longitude String
  type    Type
  property Property
  createdAt DateTime @default(now())
  user    User       @relation(fields: [userId], references: [id])
  userId  String      @db.ObjectId
  postDetail PostDetail?
}

```



```

    savedPosts SavedPost[]
}
enum Type {
    buy
    rent
}
enum Property {
    apartment
    house
    condo
    land
}
model PostDetail {
    id      String  @id @default(auto()) @map("_id") @db.ObjectId
    desc    String
    utilities String?
    Pet     String?
    Income  String?
    Size    Int?
    school  Int?
    bus     Int?
    restaurant Int?
    post    Post    @relation(fields: [postId], references: [id])
    postId  String  @unique @db.ObjectId
}
model SavedPost {
    id      String  @id @default(auto()) @map("_id") @db.ObjectId
    user    User    @relation(fields: [userId], references: [id])
    post    Post    @relation(fields: [postId], references: [id])
    userId  String  @unique @db.ObjectId
    postId  String  @unique @db.ObjectId
    createdAt DateTime @default(now())
    @unique([userId, postId])
}
model User {
    id      String  @id @default(auto()) @map("_id") @db.ObjectId
    email   String  @unique
    username String  @unique
    password String
    avatar  String?
    createdAt DateTime @default(now())
    posts   Post[]
    savedPosts SavedPost[]
    chats   Chat[]  @relation(fields: [chatIDs], references: [id])
    chatIDs String[] @db.ObjectId
}
model Chat {
    id      String  @id @default(auto()) @map("_id") @db.ObjectId
    users   User[]  @relation(fields: [userIDs], references: [id])

```

```

    userIDs String[] @db.ObjectId
    createdAt DateTime @default(now())
    seenBy String[] @db.ObjectId
    messages Message[]
    lastMessage String?
  }
  model Message {
    id String @id @default(auto()) @map("_id") @db.ObjectId
    text String
    userId String
    chat Chat @relation(fields: [chatId], references: [id])
    chatId String @db.ObjectId
    createdAt DateTime @default(now())
  }

```

### **Back End Code to authorize Users and allow them to Post Listings:**

```

import prisma from "../lib/prisma.js";
import bcrypt from "bcrypt";

export const getUsers = async (req, res) => {
  try {
    const users = await prisma.user.findMany();
    res.status(200).json(users);
  } catch (err) {
    console.log(err);
    res.status(500).json({ message: "Failed to get users!" });
  }
};

export const getUser = async (req, res) => {
  const id = req.params.id;
  try {
    const user = await prisma.user.findUnique({
      where: { id },
    });
    res.status(200).json(user);
  } catch (err) {
    console.log(err);
    res.status(500).json({ message: "Failed to get user!" });
  }
};

export const updateUser = async (req, res) => {
  const id = req.params.id;
  const tokenUserId = req.userId;
  const { password, avatar, ...inputs } = req.body;

  if (id !== tokenUserId) {
    return res.status(403).json({ message: "Not Authorized!" });
  }

```

```

let updatedPassword = null;
try {
  if (password) {
    updatedPassword = await bcrypt.hash(password, 10);
  }

  const updatedUser = await prisma.user.update({
    where: { id },
    data: {
      ...inputs,
      ...(updatedPassword && { password: updatedPassword }),
      ...(avatar && { avatar }),
    },
  });

  const { password: userPassword, ...rest } = updatedUser;

  res.status(200).json(rest);
} catch (err) {
  console.log(err);
  res.status(500).json({ message: "Failed to update users!" });
}

export const deleteUser = async (req, res) => {
  const id = req.params.id;
  const tokenUserId = req.userId;

  if (id !== tokenUserId) {
    return res.status(403).json({ message: "Not Authorized!" });
  }

  try {
    await prisma.user.delete({
      where: { id },
    });
    res.status(200).json({ message: "User deleted" });
  } catch (err) {
    console.log(err);
    res.status(500).json({ message: "Failed to delete users!" });
  }
};

export const savePost = async (req, res) => {
  const postId = req.body.postId;
  const tokenUserId = req.userId;

  try {

```

```

const savedPost = await prisma.savedPost.findUnique({
  where: {
    userId_postId: {
      userId: tokenUserId,
      postId,
    },
  },
});

if (savedPost) {
  await prisma.savedPost.delete({
    where: {
      id: savedPost.id,
    },
  });
  res.status(200).json({ message: "Post removed from saved list" });
} else {
  await prisma.savedPost.create({
    data: {
      userId: tokenUserId,
      postId,
    },
  });
  res.status(200).json({ message: "Post saved" });
}
} catch (err) {
  console.log(err);
  res.status(500).json({ message: "Failed to delete users!" });
}
};

export const profilePosts = async (req, res) => {
  const tokenUserId = req.userId;
  try {
    const userPosts = await prisma.post.findMany({
      where: { userId: tokenUserId },
    });
    const saved = await prisma.savedPost.findMany({
      where: { userId: tokenUserId },
      include: {
        post: true,
      },
    });

    const savedPosts = saved.map((item) => item.post);
    res.status(200).json({ userPosts, savedPosts });
  } catch (err) {
    console.log(err);
    res.status(500).json({ message: "Failed to get profile posts!" });
  }
};

```

```

    }
  };

export const getNotificationNumber = async (req, res) => {
  const tokenUserId = req.userId;
  try {
    const number = await prisma.chat.count({
      where: {
        userIDs: {
          hasSome: [tokenUserId],
        },
        NOT: {
          seenBy: {
            hasSome: [tokenUserId],
          },
        },
      },
    });
    res.status(200).json(number);
  } catch (err) {
    console.log(err);
    res.status(500).json({ message: "Failed to get profile posts!" });
  }
};

```

#### **Authorization Code for Users:**

```

import bcrypt from "bcrypt"; //Hashing users passwords
import jwt from "jsonwebtoken";
import prisma from "../lib/prisma.js";

```

```

export const register = async (req, res) => {
  const { username, email, password } = req.body;

  try {
    // HASH THE PASSWORD

    const hashedPassword = await bcrypt.hash(password, 10);

    console.log(hashedPassword);

    // CREATE A NEW USER AND SAVE TO DB
    const newUser = await prisma.user.create({
      data: {
        username,
        email,
        password: hashedPassword,
      },
    });

    console.log(newUser);
  }
};

```

```

    res.status(201).json({ message: "User created successfully" });
  } catch (err) {
    console.log(err);
    res.status(500).json({ message: "Failed to create user!" });
  }
};

export const login = async (req, res) => {
  const { username, password } = req.body;

  try {
    // CHECK IF THE USER EXISTS

    const user = await prisma.user.findUnique({
      where: { username },
    });

    if (!user) return res.status(400).json({ message: "Invalid Credentials!" });

    // CHECK IF THE PASSWORD IS CORRECT

    const isPasswordValid = await bcrypt.compare(password, user.password);

    if (!isPasswordValid)
      return res.status(400).json({ message: "Invalid Credentials!" });

    // GENERATE COOKIE TOKEN AND SEND TO THE USER

    // res.setHeader("Set-Cookie", "test=" + "myValue").json("success")
    const age = 1000 * 60 * 60 * 24 * 7;

    const token = jwt.sign(
      {
        id: user.id,
        isAdmin: false,
      },
      process.env.JWT_SECRET_KEY,
      { expiresIn: age }
    );

    const { password: userPassword, ...userInfo } = user;

    res
      .cookie("token", token, {
        httpOnly: true,
        // secure: true,
        maxAge: age,
      })

```

```

        .status(200)
        .json(userInfo);
    } catch (err) {
        console.log(err);
        res.status(500).json({ message: "Failed to login!" });
    }
};

export const logout = (req, res) => {
    res.clearCookie("token").status(200).json({ message: "Logout Successful" });
};

```

### **App.js Back End Server Code:**

```

import express from "express";
import cors from "cors";
import cookieParser from "cookie-parser";
import authRoute from "./routes/auth.route.js";
import postRoute from "./routes/post.route.js";
import testRoute from "./routes/test.route.js";
import userRoute from "./routes/user.route.js";
import chatRoute from "./routes/chat.route.js";
import messageRoute from "./routes/message.route.js";
const app = express();
app.use(cors({ origin: process.env.CLIENT_URL, credentials: true }));
app.use(express.json());
app.use(cookieParser());
app.use("/api/auth", authRoute);
app.use("/api/users", userRoute);
app.use("/api/posts", postRoute);
app.use("/api/test", testRoute);
app.use("/api/chats", chatRoute);
app.use("/api/messages", messageRoute);
app.listen(8800, () => {
    console.log("Server is running!");
});

```

## **Chapter 6: Testing**

### **6.1 System Testing:**

In these stages the test group of the development team, using the cases and the test data already prepared will test the programs. Only after all the functions are tested singularly, an integrated testing will be performed to see that inter-function dependability is satisfied. Separate test cases and test data will be worked out for the integrated testing. Following are the activities we planned to test the system.

1. This system is indeed an evolutionary system, so every unit of the system is continuously under testing phase.
2. One test activity “Basis Path Testing” that will try to cover all paths in the system. This activity identifies all paths that provide different functionality of the system, and other paths to reach that functionality.
3. Another testing activity is “Control Structure Testing”, which will test each condition with positive and negative data combination.
4. This testing activity will also perform “Data Flow Testing” in which it will be tested how the data re following the system. And will also check whether the data entered from one procedure, is reflected whenever it requires or not.
5. All conditions will be tested with “Boundary Value Analysis” where different input will be given to test whether the system is functioning with boundary values or not.
6. Along with the boundary value analysis, the system is also tested with “Range Value Tested” where editable values will be tested with ranges of values.
7. The system is being tested in “Unit Testing” manner where at the completion of one unit that is tested thoroughly with above mentioned testing activities.
8. The integration testing will also be performed to ensure that the integrated unit is working properly with other units or not.

### **5.2 Test Levels**

#### **5.2.1 CONTENT TESTING:**

Errors in Project content can be as trivial as minor typographical errors such as incorrect information, improper organization, or validation of intellectual property laws. Content Testing attempts to uncover this and many other problems before the user encounters them.

**Content Testing Objectives:**

There are three types of objectives.



- ☐ To uncover syntactic errors in text-based documents, graphical representation, and other media.
- ☐ To uncover semantic errors in any content object represented as navigation error.
- ☐ To find errors in organization or structure of content that is presented to the end user.

### **5.2.2 INTERFACE TESTING:**

Interface design model is reviewed to ensure that generic quality criteria established for all user interfaces have been achieved and that application specific interface design issues have been properly addressed.

Interface testing strategy:

The overall strategy for interface testing is to (1) Uncover error related to specific Interface mechanisms (2) uncover errors in the way the interface implements the semantics of navigation, Web Application functionality, or content display. To accomplish this strategy, several objectives must be achieved:

- Interface futures are tested to ensure that design rules, aesthetics and related visual content are available for the user without error.
- Individual interface mechanisms are tested in a manner that is analogous to unit testing.

Testing Interface Mechanisms: -

When a user interacts with a system, the interaction occurs through one or more interface mechanisms.

Client-side scripting: -

Black box tests are conducted to uncover any error in processing as

The script is executed. These tests are coupled with forms testing because script input is often derived from data provided as part of forms processing.

Application specific interface mechanisms: -

Test conforms to a checklist of functionality and features that are defined by the interface mechanism.

- ☐ Boundary test minimum and maximum number of items that can be placed in to shopping chart.
- ☐ Test to determine persistence of image capture contents.
- ☐ Test to determine whether the system can record co-ordinate content at some future date.

### **5.2.3 USABILITY TESTING: -**

Usability test may be designed by Project engineering team.

- ☐ Define a set of usability testing categories and identify goal for each.
- ☐ Design test that will enable each goal to be evaluated.
- ☐ Select participants who will conduct the test.
- ☐ Instrument participant's interaction with system while testing is conducted.
- ☐ Develop a mechanism for assessing the usability of the system.

The following test categories and objective illustrate establish testing:

- Interactivity- Are interaction mechanisms easy to understand and use?

Layout- Are navigation mechanisms, content and function placed in a manner that allows the user to find them quickly?

Readability- Is the text well written and clear?

Aesthetics- Do layout color, typeface, and related characteristics lead to ease of use?

Display Characteristics- Does the system make optimal use of screen size and resolution?

Time Sensitivity- Can important features, functions and content be used in a timely manner?

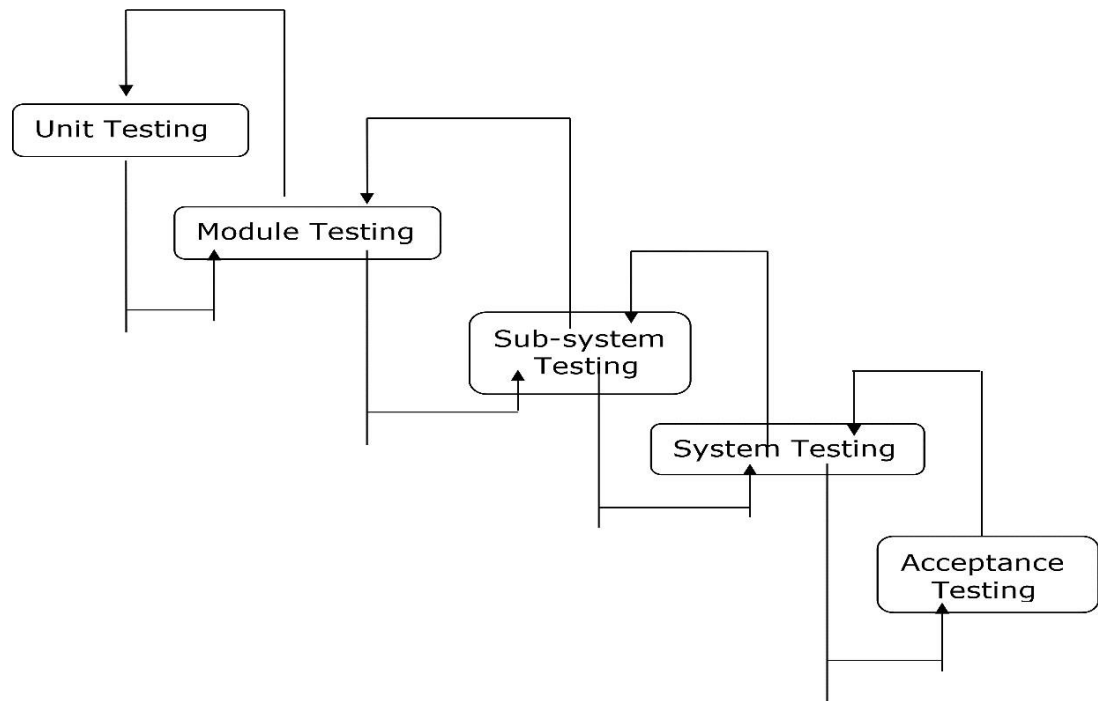
Accessibility- Is the system accessible to people who have Disabilities?

### **5.2.4 COMPATIBILITY TESTING:-**

Project must operate within environment that differs from one another. Different computers, display device, OS, browser and network connection speed can have significant on system operation. The Project team derives a series of compatibility, validation tests, derived from existing interface tests, navigation tests, performance tests and security tests.

### **5.3 Testing Methods:**

Analyze and check system representation such as the requirement document, design diagrams and the program source code. They may be applied at all stages of the process.



There are different Models of testing. Based on testing methods there are two types of testing:

1. White-box testing.
2. Black-box testing

#### 1. WHITE-BOX TESTING

White-box testing, sometimes called glass-box testing, is a test case design method that uses the control structure of the procedural design to drive the test case.

- Logical errors and incorrect assumptions are inversely proportional to the probability that a program will be executed. Errors tend to creep into our work we design and implement function, condition or control that is out of the mainstream tends to be well understood.
- We often believe that a logical path is not likely to be executed when in fact it may be executed on a regular basis. The logical flow of a program times counter

intuitive.

## 2. BLACK-BOX TESTING:

For our project we have periodically tested our software using black-box testing.

Thinking as a client we have evaluated the software for its ease and convenience.

### ☐ Unit Testing:

During the programming stages each form, modules and class treated unit has been put into the test data. Every module is tested independently. The steps are follows:

☐ Manual code is tested like spelling checks, logic and errors.

☐ Once the manual checking is over the complication has been done. Syntactical errors, if any must be corrected.

☐ After the clean complication of the program, some dummy data, as specification, has been used for testing of the module to see if it works as specified.

### ☐ Integration Testing:

After our individual modules were tested out, we go the integrated to create a complete system. This integration process involves building the system and testing the resultant system for problems that arise from component interaction.

### ☐ Performance Testing:

Performance testing is designed to test the runtime performance of the system within the context of the system. These tests were performed as module level as well as system level. Individual modules were tested for required performance.

### ☐ Condition Testing:

Performance testing is a test case design method that exercises the logical conditions.

### ☐ Interface Testing:

Interface sting is an integral part of integration. We examined the code to be tested and explicitly listed each call to an external component. In the system standards tests for GUIs have been performed, which are as follows:

☐ The position and related labels for all controls were checked.

☐ Validations for all inputs were done.

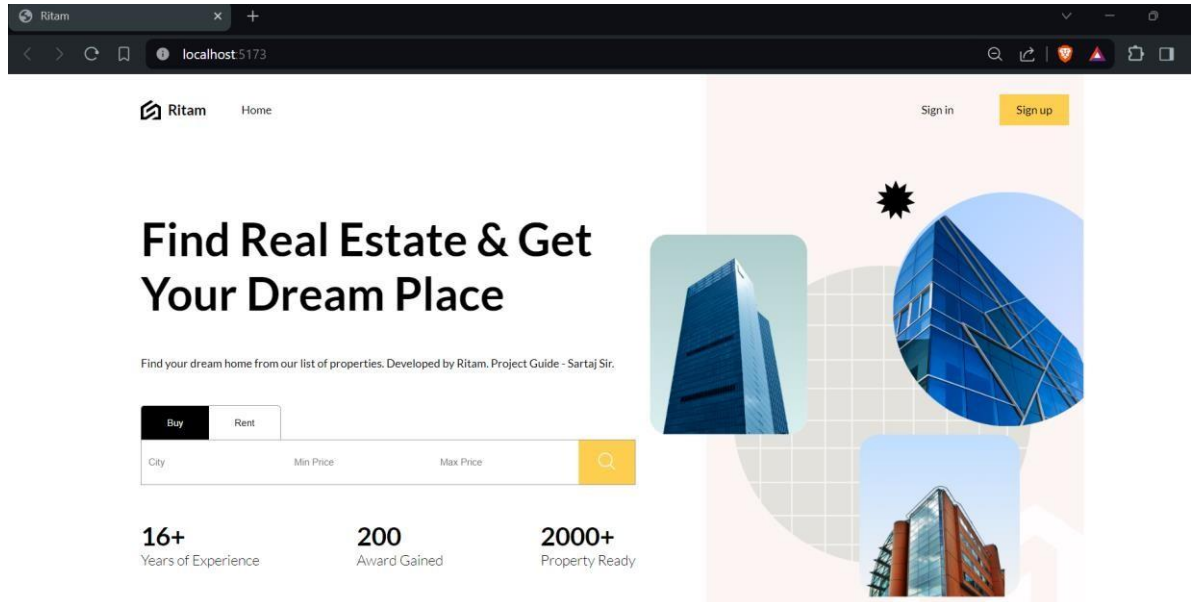
☐ Pull down controls were verified for proper functionality.

☐ Whether the non-editable text controls disabling, and it was also verified that it doesn't exceed the maximum allowed length.

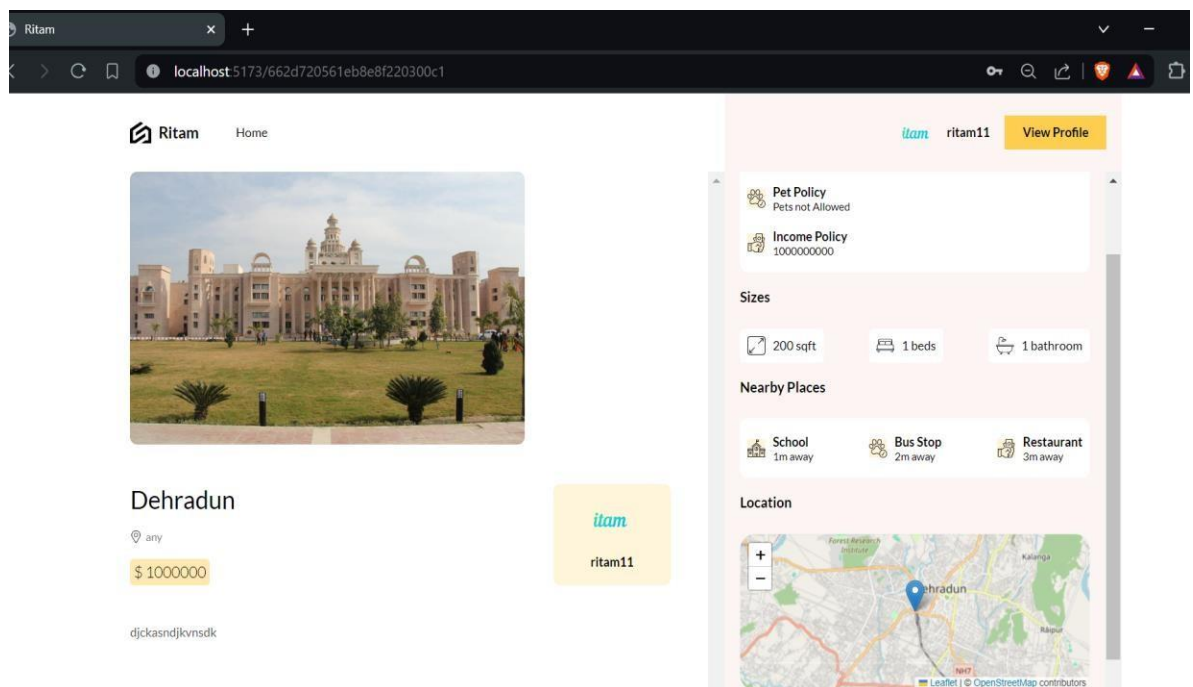
## Chapter 7: OUTPUT SAMPLES and CONCLUSION

### 7.1 OUTPUT of the Final Application

#### HOME PAGE:



#### PROPERTY PAGE:



## CREATING NEW PROPERTY LISTING:

RitamHome

Add New Post

Title

Price

Address

Description

Normal B I U

City

Bedroom Number

Bathroom Number

Latitude

Longitude

Type

ritamritam11View Profile

Upload

## Registering as a New User:

localhost5173/register

RitamHome

Sign inSign up

Create an Account

Username


Email

Password

Register

[Do you have an account?](#)

## Property Listing:

 Ritam Home

SEARCH RESULTS FOR

Location

City Location

Type

any

Property

any

Min Price

0


Max Price

0

Bedroom

any

Search




Dehradun

any

\$ 1000000

1 bedroom 1 bathroom

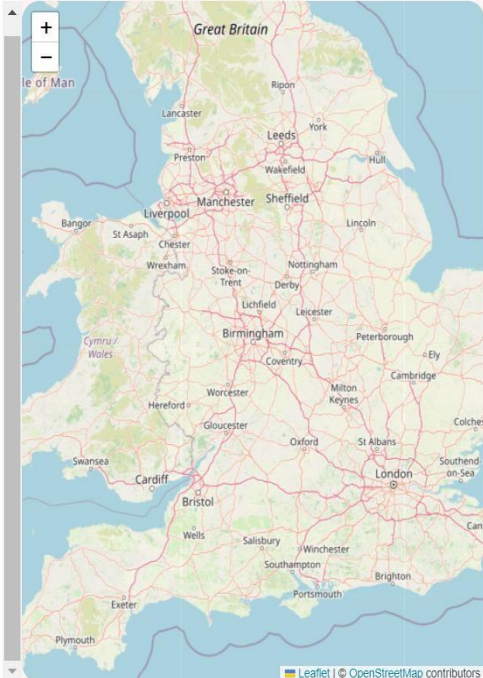


new

skjfnjskdfn

\$ 1000000000

2 bedroom 1 bathroom



## MongoDB Data Cluster: (Data Being stored in MongoDB)

Data | Cloud: MongoDB Cloud

https://cloud.mongodb.com/v2/6624111821fd4c6482b8beca#/metrics/replicaSet/662411d97d16d657bf7f9fb/explorer/es/

Recruiting tips | Deloitte... Learn web development [020] 1. Overview | Pit... Week 1 Interview Que... Tips to Land a Job | Re... How to Host HTML, C... Smashing Magazine ...

Atlas Ritam's Org ... Access Manager Billing All Clusters Get Help Ritam

Rental Website ... Data Services App Services Charts

Overview

DEPLOYMENT

Database

Data Lake

SERVICES

Device & Edge Sync

Triggers

Data API

Data Federation

Atlas Search

Stream Processing

Migration

estate

Chat

Message

Post

PostDetail

SavedPost

User

Find

Indexes

Schema Anti-Patterns

Aggregation

Search Indexes

Generate queries from natural language in Compass

INSERT DOCUMENT

Filter

Type a query: { field: 'value' }

Reset Apply Options

```
_id: ObjectId('662d6d3f48c75d910f9a8f70')
email: "ritam1@gmail.com"
username: "ritam11"
password: "$2b$10$Vqv4Qj8B/9gep3s7BpzoM0seVZ.ouAzPKsgv0LSFIA3GIhMyHavJ."
createdAt: 2024-04-27T21:25:19.767+00:00
avatar: "https://res.cloudinary.com/lamadev/image/upload/v1714254431/avatars/wh_"
```

System Status: All Good

©2024 MongoDB, Inc. Status Terms Privacy Atlas Blog Contact Sales

## 7.2 Conclusion

The development of a real estate web application using the MERN stack offers a robust, scalable, and efficient solution for property management and listing. Through the integration of MongoDB, Express.js, React.js, and Node.js, we've created a dynamic platform that seamlessly handles data storage, server-side logic, and client-side rendering.

By leveraging React.js for the front-end, users benefit from a responsive and intuitive interface, enabling smooth navigation and enhanced user experience. Meanwhile, Node.js provides a fast and scalable backend environment, ensuring efficient data processing and seamless communication with the client-side application.

Moreover, MongoDB's flexible and scalable database architecture empowers the application to manage large volumes of property data efficiently, offering robust search and filtering capabilities for users. The Express.js framework facilitates the development of RESTful APIs, enabling seamless communication between the frontend and backend components.

Overall, the MERN stack's combination of powerful technologies has enabled us to create a feature-rich real estate web application that meets the diverse needs of property buyers, sellers, and agents. With its scalability, flexibility, and performance, our application is poised to revolutionize the real estate industry, providing a modern and streamlined platform for property transactions and management.

## 7.3 Future Scope of this Application:

- This project is developed as a master's project and still gives lot of scope for its extension which could be made to the project if it is going to be developed as commercial product.
- A Real time Chat Feature using Socket.io will be implemented in the future so that the seller and buyer is not required to disclose their personal information.
- The feature of providing Google Maps within this application can add up to the functionality of the website. With the advancement of technology, dynamic maps can be generated which helps the buyer locate a particular area.
- Buyer testimonials can also be added in future if required.
- Inclusion of all these features would make the application feature rich. The advantages of putting these functionalities in the project are described in detail in the following sections.



## Bibliography

1. Documentation Referred from ReactJS Website. (<https://react.dev/learn>)
2. Connecting Back End to Front End. ([https://axios-http.com/docs/api\\_intro](https://axios-http.com/docs/api_intro))
3. Express JS, a NodeJS framework (<https://expressjs.com/en/guide/routing.html>)
4. Styling SCSS (<https://sass-lang.com/documentation/syntax/>)
5. Various Node Package Managers (<https://docs.npmjs.com/>)
6. YouTube Channel like (Llama Dev, Web Dev Simplified)