**Real Estate Sharks**   
**Software Design Documentation**

Date: 2/19/2022

Version: 1.0.0



Source (<https://www.dreamstime.com/shark-house-vector-logo-design-home-icon-image158864365>)

Table of Contents

1. Introduction...................................................................................................

1.1 Purpose

1.2 Scope

1.3 Referenced

1. Requirements.........................................................................................

2.1 Section Details

2.2 Functional Requirements

2.2.1 Property Database

2.2.2 Maintenance

2.2.3 Explore

2.2.4 Shared Documents

2.2.5 Property Details

2.2.6 User Account

2.2.7 Help/Contact Info

2.3 Non-Functional Requirements

2.3.1 Reliability

2.3.2 Usability

2.3.3 Security

2.3.4 Accessibility

2.3.5 Scalability

1. Data Design................................................................................
2. Architecture...................................................................................................

4.1 System Architecture (Diagram 4.1, Diagram 4.1.1)

4.2 Database Architecture (Diagram 4.2)

1. Component Design........................................................................................

5.1 Component Descriptions (Diagram 5.1)

1. Interface Design..............................................................................................

6.1 User Interface Design

6.1.1 Web Page Tree (Diagram 6.1)

6.2 Use Cases

6.3 Web Application Architecture (Diagram 6.3)

6.3.1 HTTPS Certification

6.3.2 SSL session certificate

1. Patterns Used...........................................................................................

7.1 Architecture Design

7.2 Software Development Patterns

1. Design Concept Review..................................................................................

8.1 Design Concepts reflection

8.1.1 Abstraction

8.1.2 Architecture

8.1.3 Patterns

8.1.4 Separation of Concerns

8.1.5 Modularity

8.1.6 Information hiding

8.1.7 Functional Independence

8.1.8 Refinement

8.1.9 Aspects

8.1.10 Refactoring

8.1.11 Object-Oriented Design Concepts

8.1.12 Design Classes

8.1.13 Dependency Inversion

8.1.14 Design for Test

1. Design Concept Review..................................................................................

9.1 Economy

9.2 Visibility

9.3 Spacing

9.4 Symmetry

9.5 Emergence

1. Component Design Principles.........................................................................

10.1 Open-Closed Principle

10.2 Liskov Substitution Principle

10.3 Dependency Inversion Principle

10.4 Interface Segregation Principle

10.5 Release Re-Use Equivalency Principle

10.6 Common Closure Principle

10.7 Common Re-Use Principle

**Section 1: Introduction**

**1.1 Purpose of This Document**

This is the software design description for a property management web application for Westcott Properties (WP). WP owns, manages and rents several properties using an outdated system involving a notebook and excel spreadsheets. This document describes the scope, objectives and goals of the web application being built for WP and 3rd party users.

**1.2 Scope of Project**

The scope of this project is a web-based system that will support the management of all Westcott properties assets.   
  
Excel spreadshee3ts and a notebook will be replaced by this new system. In addition to the new system, changes to Westcott Properties workflow will be implemented via the automation of some tasks. Increased data analytics will contribute to positive changes in Westcott Properties workflow.

**1.3 References**

1. Title of reference
   1. Link
   2. Where used
   3. Notes
2. MVC vs MVT
   1. <https://medium.com/dsc-umit/mvc-vs-mvt-architectural-pattern-d306a56dce55>
   2. Section 4
3. Integrating django with reactjs using django rest framework
   1. <https://www.geeksforgeeks.org/integrating-django-with-reactjs-using-django-rest-framework/?ref=rp>
4. <https://sourcemaking.com/design_patterns/builder>
   1. Builder Design Pattern
5. <https://dev.to/sm0ke/django-tutorial-mvt-architecture-custom-commands-19nb>
   1. MVT Architecture
6. [managing-houses-and-properties-a-real-estate-agency-data-model](https://www.vertabelo.com/blog/managing-houses-and-properties-a-real-estate-agency-data-model/)
   1. Database inspiration

Section 1 Approval: {GMC,CDO}

**Section 2: Requirements**

**2.1 Section Details**

In this section, the application's functional and non-functional requirements will be described in detail. Terms user and tenant will be used interchangeably. The terms manager and the owner will be used interchangeably. In all requirements, the backend and frontend are considered two different applications that communicate and interact with each other via a rest API.

**2.2 Functional Requirements**

**2.2.1 Property Database**

The Django application backend must allow for post/fetch requests from the front end for the application architecture to function properly. Front end will interact with the Django rest API to manipulate, record, and store information provided by the user through JSON objects.

**2.2.2 Maintenance**

Users will be able to make maintenance requests through the web application. They must first be on the Maintenance tab of the website. They will be able to enter the title, body, and add attachments. The request will be sent directly to the assigned landlord. Requests made by users will be time-stamped, recorded, and tracked throughout the life cycle of the request. A request can be assigned back to the user/tenant by the landlord for response or closing of request and vice-versa.

**2.2.3 Explore**

The system must allow for searching of current property database. Once users log in, they must be able to search available rentals. A User must be on the “Property Database” tab on the website. They will be able to search the current rentals with the following possible search criteria; price, town, size (sqft), pet-friendly, house, apartment, and condo. The results will be displayed in a list format.

**2.2.4 Shared Documents**

Users will be able to view, upload, and download the following documents. Rental Contract, Utility bills, and insurance documents. These documents will be accessed by request on the shared documents tab of the website. First, a search will be performed for the given document. Then the resulting document will be downloadable or viewable. Both the user/tenant and the landlord/manager will have access to these documents.

**2.2.5 Property Details**

The user will be able to access detailed information on the apartment they are renting, requesting to rent, or “exploring” based on the “explore” requirement. The user will need to be on the property details tab of the website. This page will simply show detailed information on the apartment they are renting if at all.

**2.2.6 User Account**

All users will be able to edit the following information on the website account. Nickname, name, address, email, phone, and bio. This will be done by clicking on the “user account” tab on the header of the website.

**2.2.7 Help/Contact Info**

All users will be able to access the property owner's contact information as provided by the property owner. The website information will also be displayed on the website footer on all tabs. The property owner will have permissions to update this information.

**2.3 Non-Functional Requirements**

**2.3.1 Reliability**

The application will be fully functional if the is an internet connection. The application domain host will be run on a server with surge proctor and battery backup

**2.3.2 Usability**

The Tenant will be able to use the application after one hour of training. The dashboard will be intuitive and straightforward. There will be a Help page that provides information on where to find things and what to click for what.

**2.3.3 Security**

The Tenant will log in to the application with a password protected account. Each Tenant will be assigned a user authentication certificate. Security questions for password reset will be provided.

**2.3.4 Supportability**

The dashboard and database will be able to accommodate new data categories without major re-engineering. The application will be viewable from all major web browsers. The application will have a registered domain name and be certified by an accredited certificate authority.

**2.3.5 Availability**

The application will be available 24/7 365 days a year, accept when website maintenance is being performed. Website maintenance will be scheduled at least 1-week in advance and all users will be emailed.

**2.3.6 Scalability**

The application will be initially capped at 500 users as there will be limited space on the server. The number of properties in the database will be capped at 200. The application will be scalable in that parallel property owner/managers will be able to start their own property database and rental system.

Section 2 Approval: {GMC,CDO}

**Section 3: (Not Used) We are not using section 3 because the database is the foundation of the architecture with Django. Objects are made from the database through Django models.**

Section 3 Approval: {GMC,CDO}

**Section 4: Architecture**

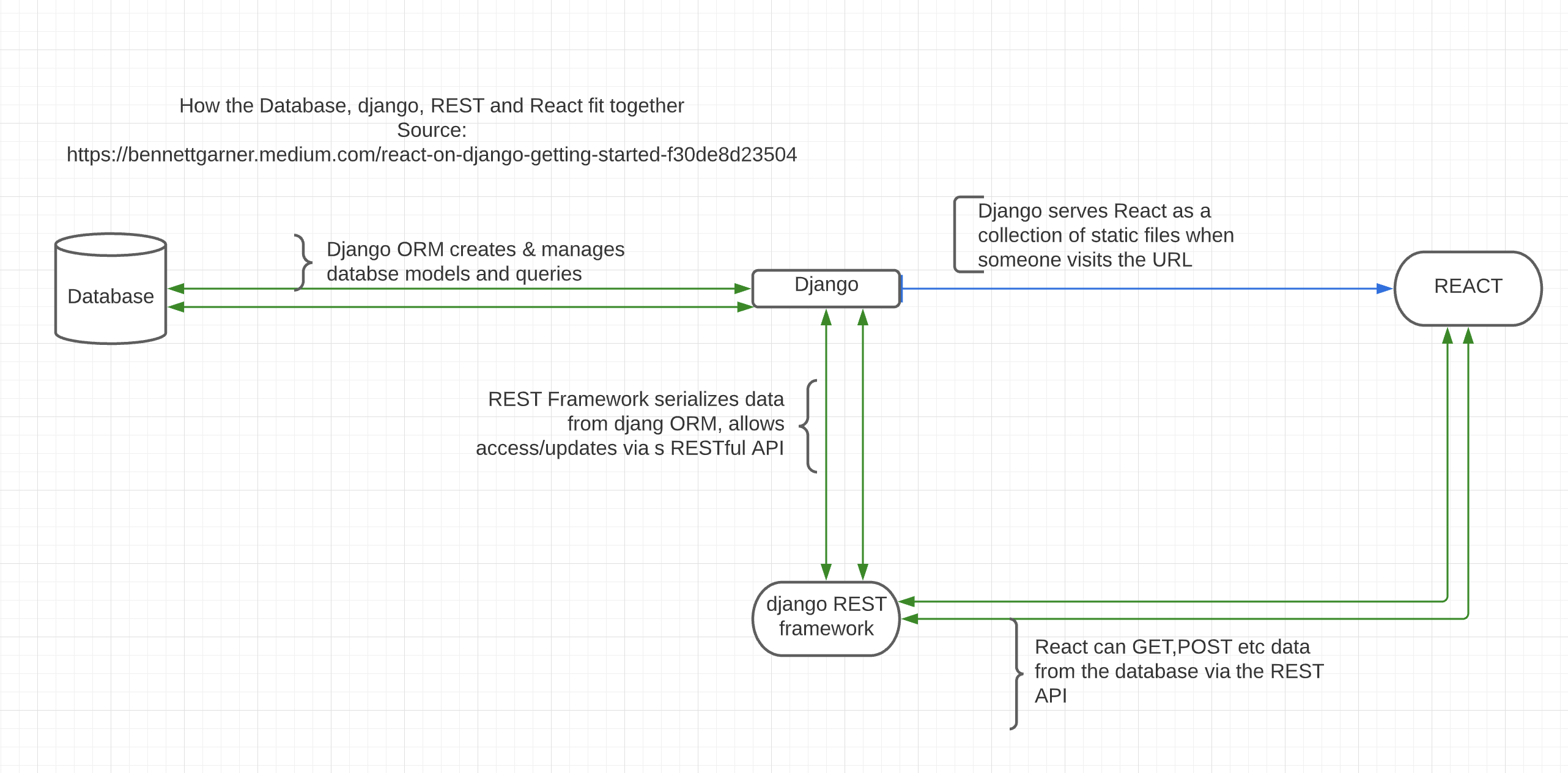
**4.1System Architecture**

The chosen architecture pattern for this system is the builder pattern. “The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.”(sourcemaking) This system will represent its data in a database. The Django ORM will query the database to obtain data. The Django REST framework will take the data and build objects that will be sent to the ReactJS frontend. (see Diagram 4.1.1)

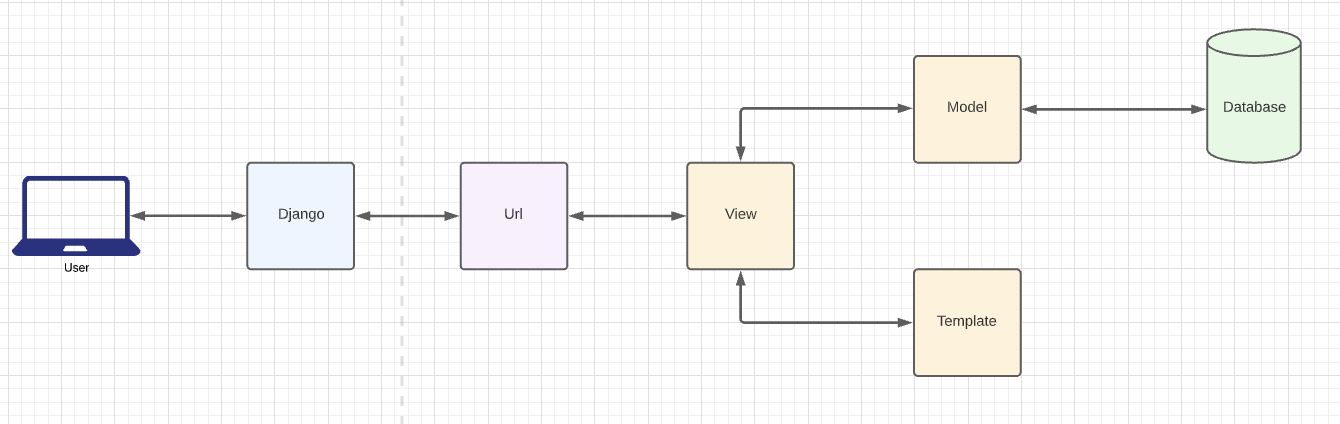
Django uses a Model View Template framework. When we build a web page we need the data, layout and the logic. The model is for handling the data in the database needed for a request. Each entity on the webpage will be linked to its own model logic. The front end of the system will be written in React JS. Templates on the backend will be used to dynamically format the data in a readable way for the front-end . The templates are written in Django Template Language. The views will take the data object from the model and look at the template associated with the request to create a JSON object for the front end. The views in the MVT architecture are different than traditional views in MVC architecture because they contain the “business logic” for the back end. The view acts more like a controller and the template acts like a view. The reason we do not have a dedicated controller is because the django framework is a controller because it is configuring everything.

To summarize, when the user makes a request, the Django framework will receive a URL and pick the appropriate view to send it to. The view will decide which template to use to format the data from the data object from the model for the front end. (see Diagram 4.1.2)

**Diagram 4.1.1**



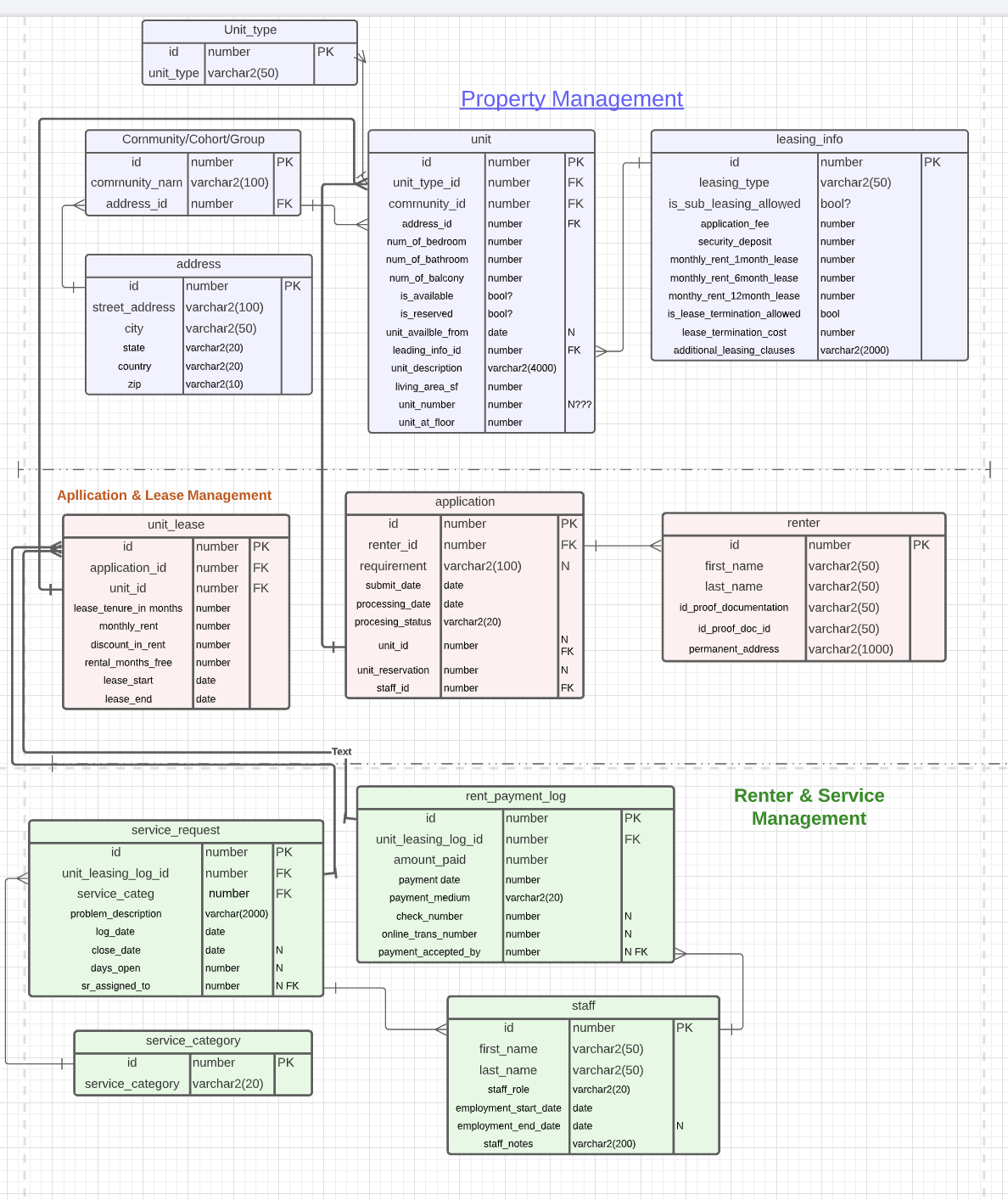
**Diagram 4.1.2**



**4.2 Data Base Architecture**

The foundation of this system will be its database. The database is broken down into three major categories; “property management”, “application and lease management”, “renter and service management”.(diagram 4.2) This database is a relational database because tables are connected through foreign keys. A relational database design was chosen due to its simplicity and modularity. Because tables are connected, querying will be easier and can be done through the Django ORM. Querying separate databases can be very difficult to do with the Django ORM. The database is modular because rows can be added or deleted without a major redesign.

**Diagram 4.2**



Section 4 Approval: {GMC,CDO}

**Section 5: Component design**

**5.1 Component Descriptions**

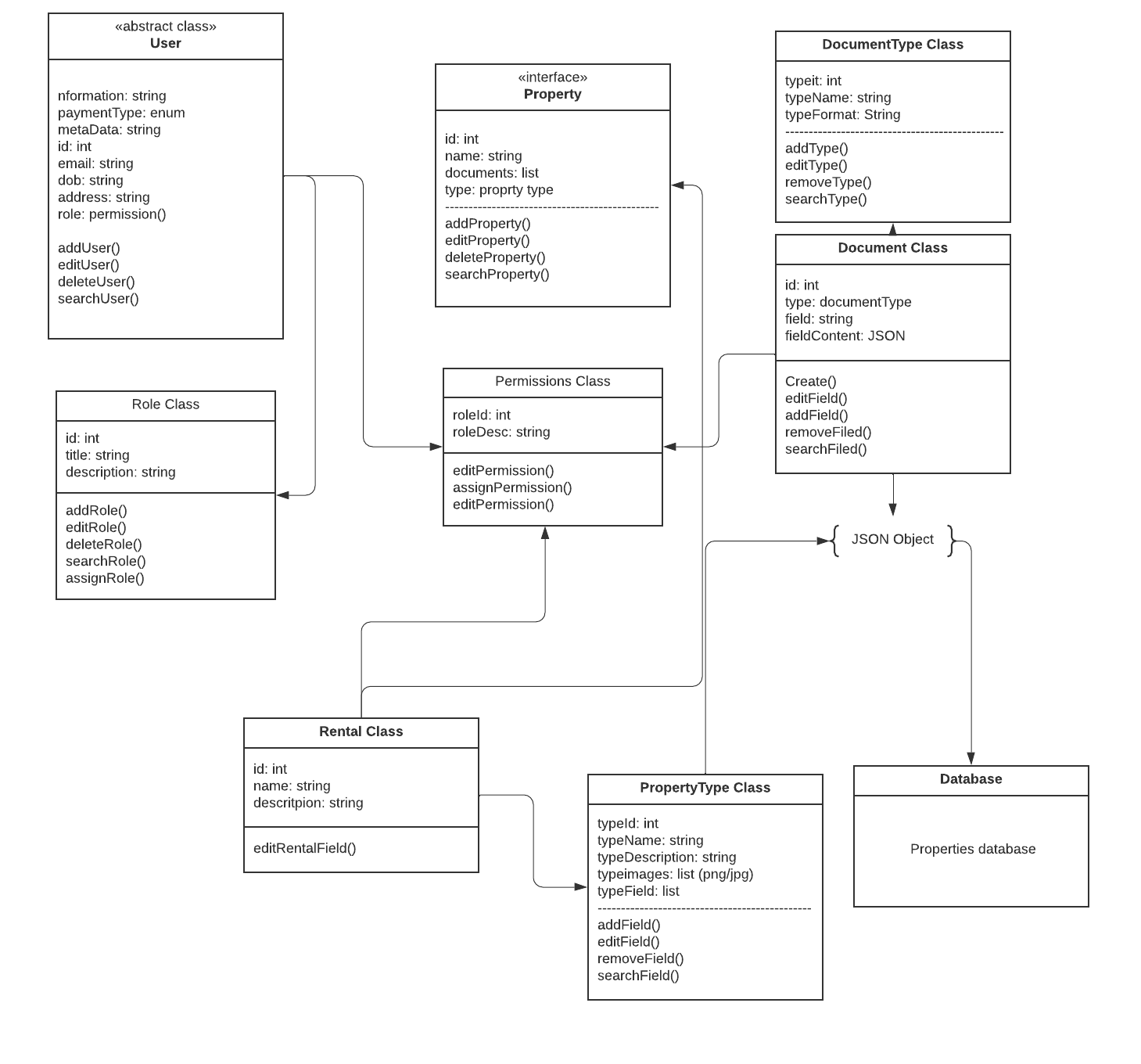
The components of the system are made up of the endpoints of the backend. There will be one of four endpoints: get, post, delete and put. A get request will get data and push it to the frontend to be rendered as seen in diagram 5.1. A post request will add data to the database. A put request will modify data in the database. A delete request will delete a row of data. The components will be separated into three unique Django apps that will interact with each other through the models. The apps are Property management, Lease management and Service management. Each system is represented by a color in diagram 4.2.

For the property management application, the unit model will be the main model. A post request will create the unit and all its dependent fields. A get request will get a listing of all the units. Another get request can get units within a certain category.

For the Application and service management app, the application will be the main model. All endpoints within this app will be changed via the application. A post request will create an application which will also create a renter.

For service management the staff model is the main model with all its associated services. All endpoints within this app will be changed via the staff connection. The post request will create staff and the second view will have separate endpoints that modifies its services.

**Diagram 5.1**



Section 5 Approval: {GMC,CDO}

**Section 6: Interface design**

**6.1 User Interface Design**

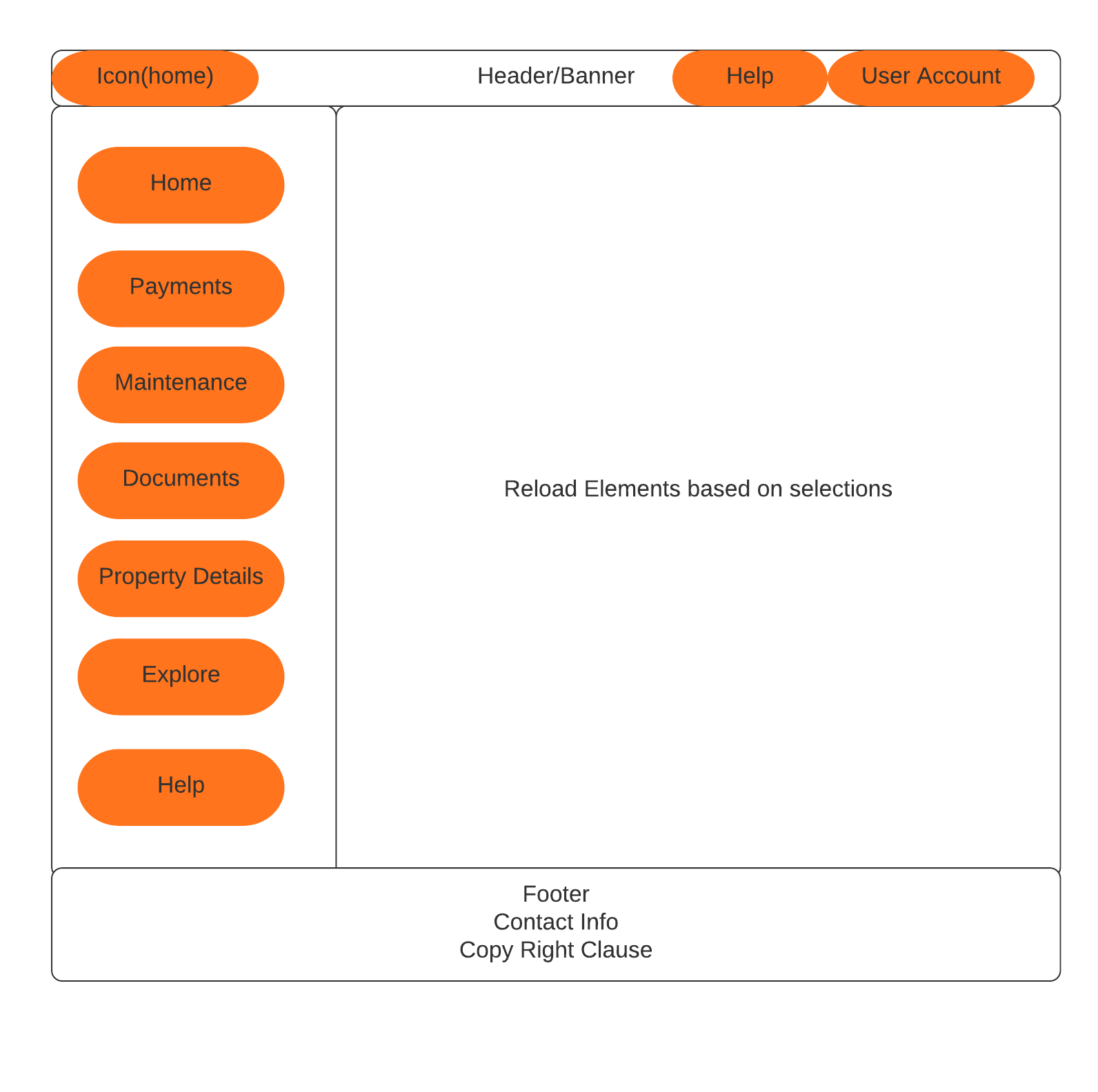
**6.1.1 Web Page Tree**

The User interface web page design is represented by a by a tree-web design diagram below. The purpose of this diagram is to show how the navigation through the web pages will be achieved. When navigating to the website, all user profiles default view will be the home page. From the home page we will be able to navigate to the maintenance, documents, explore, payments, property details, and help pages. From the maintenance, documents, explore, payments, property details, main, and help pages, we will be able to navigate using the navigation pane on the left-hand side of the screen to all other pages just as on the home page.

Property details will be able to navigate to each property set up in the database. It will reload page elements given the property selected. Links to third party website will be available if existent.

A ribbon will be present on all web pages that will allow navigation to the home via a symbol link. The ribbon will allow access the help and current users account information.

**Diagram 6.1**



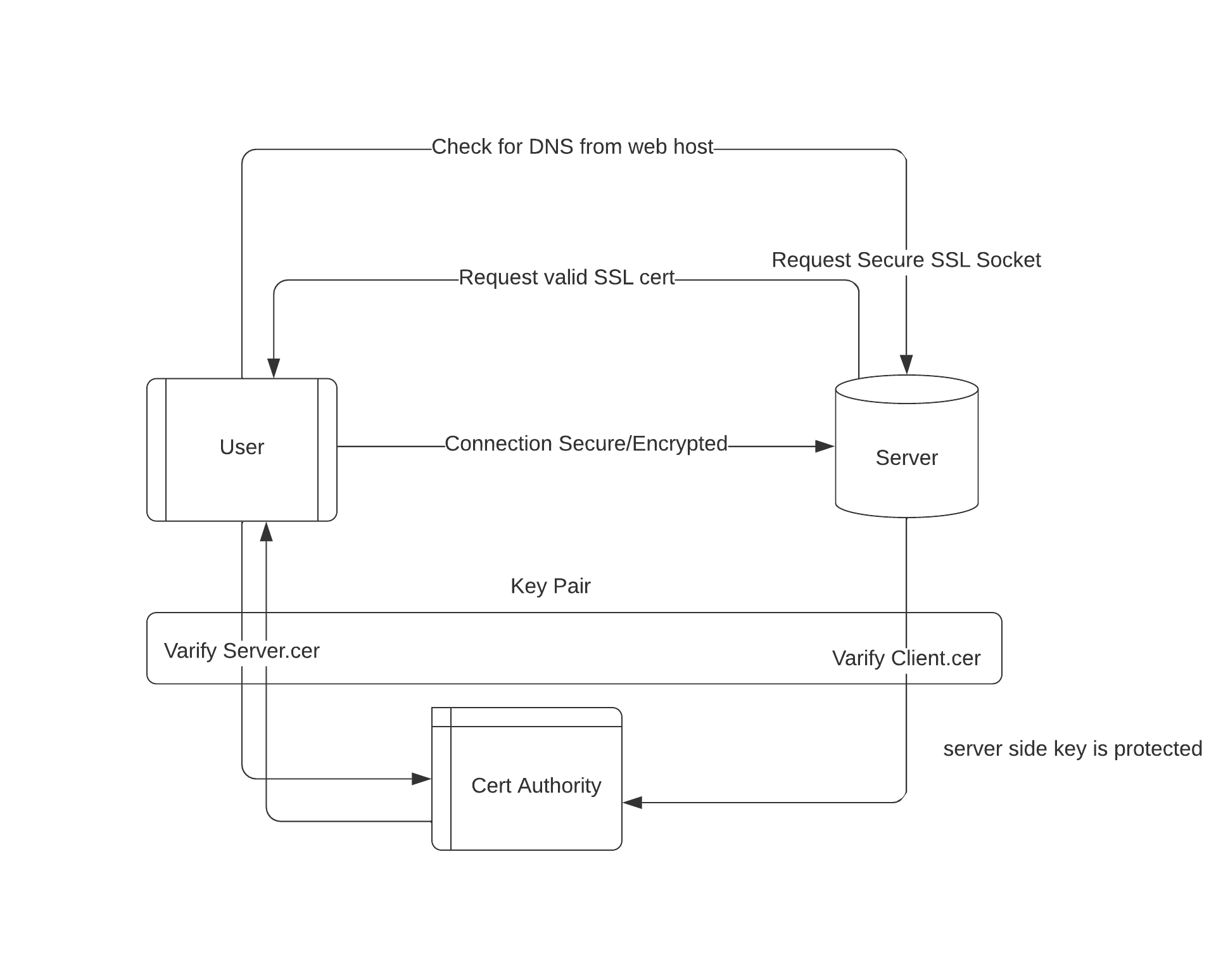
**6.2 Web Application Architecture**

The application will be utilizing the MVC design interface. The user interface will be the role of the view in the design. The view will be completely integrated too the controller which in-tern will filter the user input. Once filtered the controller will give the fetch/post request to the model. Once the fetch/post request of been processed the controller will initiate the view to reload page elements with the updated data.

**6.3 Web Application Security Design**

The Server-side web application will be utilizing the HTTPS communication protocol SSL to secure the client to server socket connections.

**Diagram 6.3**



**6.3.1 HTTPS certification**

The website will have a certificate maintained and authorized by a accredited certificate management website/organization.

**6.3.2 SSL session certificate**

To ensure users' connections are secure, each user session will be encrypted using public and private key pair. To achieve this the website will l utilize a CSR (Certificate Signing Request) on the server side. This creates a private and public key pair on the server-side. The CRS file is sent to the SSL certificate authority where the public key is processed. The server-side private key is then compared to the public key maintained by the authority without being compromised.

An intermediate certificate is installed that establishes the credibility of the SSL certificate by “chaining” it to the root certificate. Once the certificate is authenticated in what is known as a “handshake” the data transport layer is no longer sent in plain text but encrypted by an algorithm defined in the SSL standard.

Section 6 Approval: {GMC,CDO}

**Section 7: Procedural design**

**7.1 Architecture Design**

The overall architecture design was inspired by previous systems that utilize a Django backend with a ReactJS front end. Instead of reinventing the wheel we looked at how others have used Django and ReactJS and found that there is a singular concrete way to utilize them via the Django REST framework as outlined in Diagram 4.1.

The Django framework is configured to a certain format that follows the MVT pattern. The main choice of someone who is implementing Django is how important templates are to the system. Django templates can be used to create HTML or any other formation of string data. Our implementation of Django and ReactJS will result in the templates instructing the views to create JSON objects from the data inside of objects created by the model.

The Builder design pattern was not chosen, instead it is the default pattern for the design of the system architecture. This system is based on its database and manipulation of the data to create different representations of it. Django separates the process of parsing through the database and the process of building objects with the data. It is very possible that any system based on Django uses the builder pattern because of the way Django has been built.

**7.2 Software Development Patterns**

The incremental process model and spiral model will be implemented for the development of this system. Each model will be associated with a phase of the development. The first phase will implement the incremental process model. This phase will consist of two pre-planned increments. The first increment will focus on building a React-js based Front end. The second increment will focus on building the Back end which will interact with the front end created in the first increment.

Once the increments are completed, the incremental process model will be replaced with a spiral model. The second phase will be the final and perpetual phase of the project.

These models were chosen because they best suit a team of two where the development steps needed to achieve a product with the known essential functions. Achieving basic functionality is clearly broken up into distinct increments. These best suit the incremental process model because of the well-defined requirements of each increment. The spiral model was chosen for the second phase of the project because of its flexibility. During the second phase, the team members can work independently on features or together on a singular feature.

Section 7 Approval: {GMC,CDO}

**Section 8: Design Concept Review**

**8.1 Abstraction**

For the design of this application, we will be abstracting out many of the class functionalities. For example, we will abstract out the Property class so that other classes can inherit and all work properly together, providing a similar but different function. Procedural Abstraction was used in determining how a user requirement will be handled from the front end down to the database.

**8.2 Architecture**

For this application we used both the Structural and Functional models to construct and well-organized and functional hierarchy of how architectural and class components.

**8.3 Patterns**

We are using a builder design pattern. We think this is appropriate because the similarities of properties and their functional requirements are such that putting in the extra work to allow for one class to instantiate new objects with multiple similarities using a single constructor fit well.

**8.4 Separation of Concerns**

Separation conjunction complexity is a huge concept to consider when trying to mingle with too many objects. To help alleviate this complexity we believe using the builder pattern will help reduce this issue.

**8.5 Modularity**

To help reduce complexity we will be modularizing each function to its own component where possible. We will also be abstracting out all similar functionalities among dissimilar objects of the same nature or type.

**8.6 Information hiding**

To increase information hiding we will be using JSON objects of the same pattern to pass information between the front and back ends. This enables the passing of information in the same format even when the module has changed because it is required by the interface to construct its JSON in a specific format. This format is then described in the API documents so that any permissible requester protocol can translate the data. In addition, all object properties will be private except where absolutely needed.

**8.7 Functional Independence**

Django is inherently de-coupled given how objects are created

**8.8 Refinement**

Throughout the programming and design process we will be re-visiting the model/class inheritance to re-evaluate what needs to be abstracted to a higher level.

**8.9 Aspects**

To help prevent crosscuts given system aspects we will be de-coupling all system aspects from multiple functional requirements. Meaning no two requirements should block each other. Instead, they should include the functionality necessary to check such crossover.

**8.10 Refactoring**

To help reduce the ability to refactor code, each model will have as specific a function as possible. Our goal throughout the project is to be as precise and non-redundant as possible.

**8.11 Object-Oriented Design Concepts**

With the use of python, and it being duck typed, it has made it easier to plan out class hierarchies since simply passing a class or function to another during the initial declaration gives it such functionality.

**8.12 Design Classes**

We will be making use of the following design classes. User Interface which has been layed out in React, Persistent data store using Django, System designed to accommodate these designs, and a process to implements each step.

**8.13 Dependency Inversion**

Python makes it easier to follow dependency inversion. Objects will be open for interaction but closed to change by simply adding the using and extending the functionality of another class by creating a new one. Based on our Component UML diagram (Diagram 6.2), we can see the relationship between our classes and how functionality is split into multiple classes for the same functional system.

**8.14 Design for Test**

For the front end, we will simply be testing by navigating and instigating each functional requirement. The back end, we will be testing by unit tests utilizing postman which will make api requests.

Section 8 Approval: {GMC,CDO}

**Section 9: Architectural design considerations**

**9.1 Economy**

The architecture of this system is very straightforward . The builder pattern uses a common input (the database) to create many different outputs. There will be no unnecessary features driving this process.

**9.2 Visibility**

The overall design model will be obvious to engineers who view this model at a later time because it is not a novel design. Systems that combine Django and React have used a very similar design. The design will be documented and the code will have appropriate comments to guide people new to the system.

**9.3 Spacing**

The architecture is well spaced because objects only get created when a request is made for it. There are no hidden dependencies in the design. The flow of the system is dependent only on fetch/put requests from the front end that are triggered by user interaction.

**9.4 Symmetry**

From a systemic perspective, the design is very consistent and balanced. The overall architecture is very simple to explain and understand. It follows patterns that have been established by others who have implemented React and Django.

**9.5 Emergence**

Objects are created when needed in the backend when requested by the front end. While the backend will have some static files on hand for the front end, everything else in this system will be event driven. The database is extremely scalable and flexible to accommodate growth and change because it is relational. Any non-primary key or foreign key can be deleted or renamed. Additionally, rows can be added to any table that already exists.

Section 9 Approval: {GMC,CDO}

**Section 10: Component design principles**

**10.1 Open-Closed Principle**

To ensure the open-closed principle is adhered to, we are using a relational database architecture where objects are created using the builder pattern. Given the use of the builder pattern, objects will dynamically be created without, closed to modification but open for extension given the creation of new objects.

**10.2 Liskov Substitution Principle**

To adhere to the Liskov substitution principle a builder design pattern will be used where objects are built from Django relational templates. Sub-classes will be given the dependency and functionality of superclass's.

**10.3 Dependency Inversion Principle**

Top down object construction through the builder pattern will ensure high level modules will not depend on lower-level modules.

**10.4 Interface Segregation Principle**

To ensure Interface segregation, information hiding will be used to hide elements not needed by the user. Such segregation will be achieved by fully separating the react.js front-end view with the Django backend. First, the front end will clean the user input and scrape for such things as SQL injection, then the REST API determines what object to be constructed, sent, and rendered by the front end.

**10.5 Release Re-Use Equivalency Principle**

With use of the Builder pattern any component dynamically constructed (there will only be static files used as templates to construct the objects), will be both extensible and discardable to re-use in a different way.

**10.6 Common Closure Principle**

The Common Closure Principle is inherent in the Django programming architecture and program libraries.

**10.7 Common Re-Use Principle**

The Common Re-Use Principle is inherent in the Django programming architecture and program libraries. By using model templates to determine how to dynamically create objects, Django adheres to the Re-Use principle.

Section 10 Approval: {GMC,CDO}