.

Student Housing Solution (SHS) Report 2 (Group 8)

-Ansh Gambhir, Rishi Shah, Kyle Tran, Srinivasniranjan Nukala, Michael Giannella, John Yager, Rishab Ravikumar, Sneh Shah, Ketu Patel, Sahil Patel

https://mgiannella.github.io/StudentHousingSolution-Landing/

Submitted 3/22/20

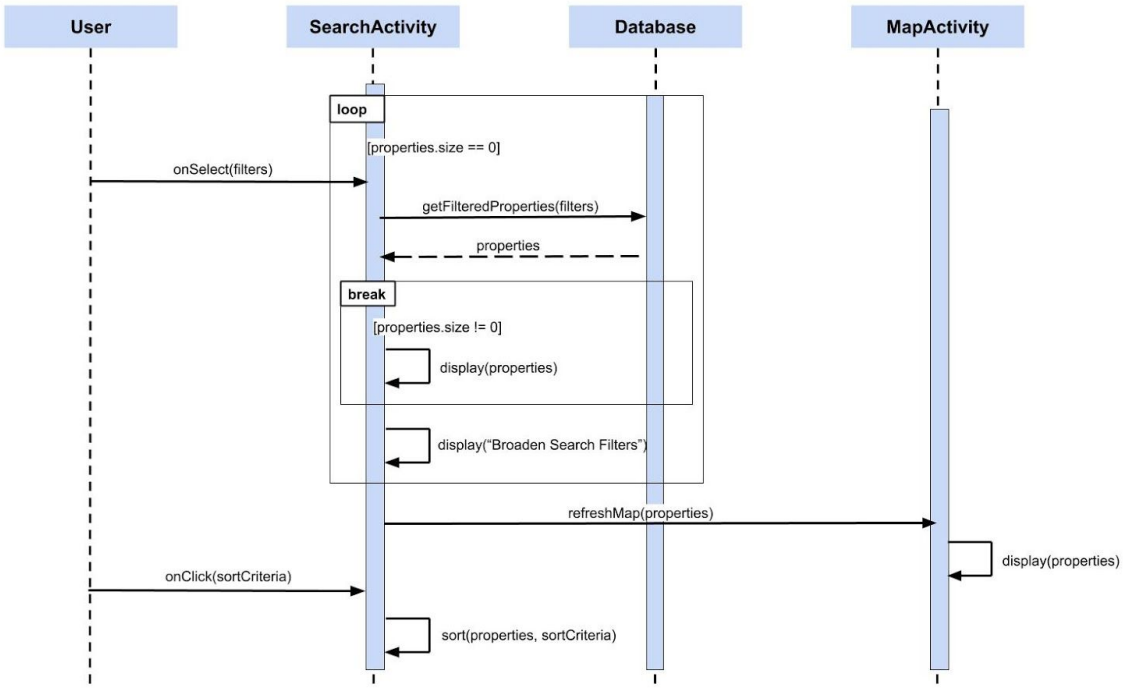
Individual Contribution Breakdown

Parts	Rishi	Kyle	Sneh	Ketu	Sahil	Michael	Rishab	John	Srinivas	Ansh
Interaction Diagrams	12.5	12.5	12.5	0	0	12.5	12.5	12.5	12.5	12.5
Class Diagrams and Interface Specification	10	10	10	10	10	10	10	10	10	10
System Architecture and System Design	10	10	10	10	10	10	10	10	10	10
Algorithms and Data Structures	10	10	10	10	10	10	10	10	10	10
User Interface Design and Implementation	10	10	10	10	10	10	10	10	10	10
Design of Tests	10	10	10	10	10	10	10	10	10	10
Project Management and Plan of Work	10	10	10	10	10	10	10	10	10	10
References	10	10	10	10	10	10	10	10	10	10

Table of Contents

Interaction Diagrams

UC-1 : SearchProperties

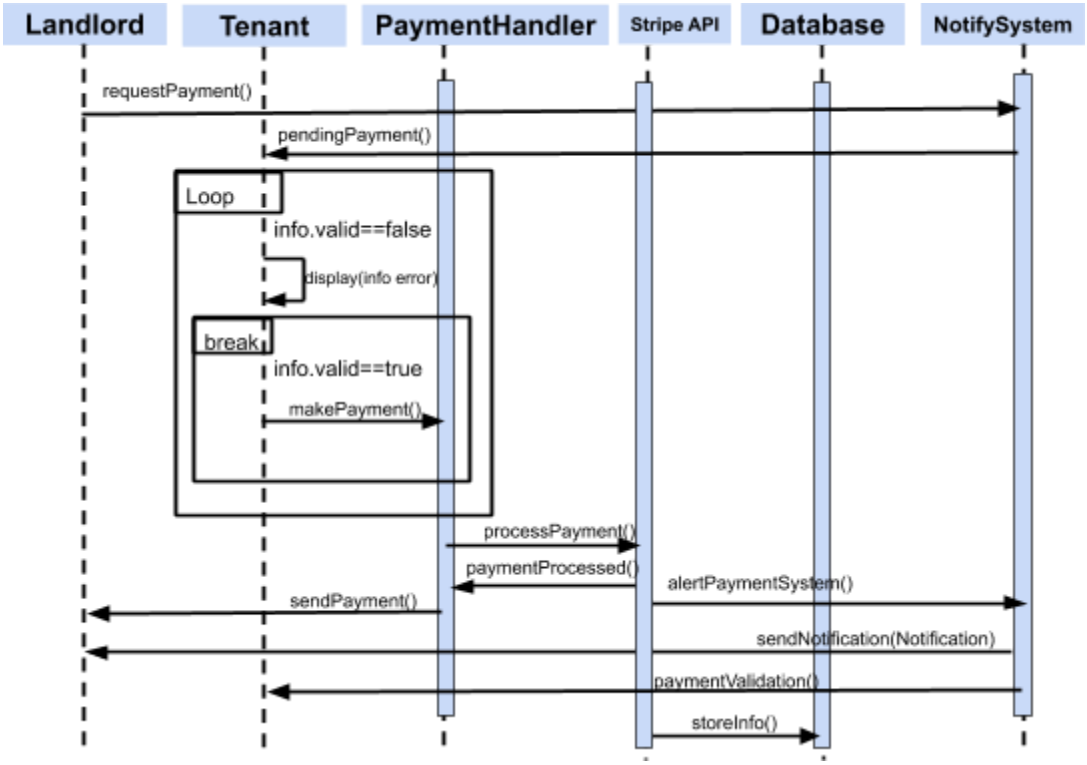


In this use case, we employ the expert doer, high cohesion, and low coupling design principles. Since the Database is the ultimate knowing module, and the SearchActivity and MapActivity modules derive their information directly from it, the expert doer in this case would be the database.

Additionally, we employ high cohesion, as no one module does everything. The SearchActivity and MapActivity receive requests from the user interface, and communicate with the database. There is no one module that does too many computations, as the database handles basic filtering, and the SearchActivity and MapActivity modules help display that to the user.

Moreover, the system uses the low coupling design principle, as there are no microservices, or really small entities. The main entities all have a significant share of work, however it is really even between the three which is how we employ the high cohesion principle.

UC-6 : Payment

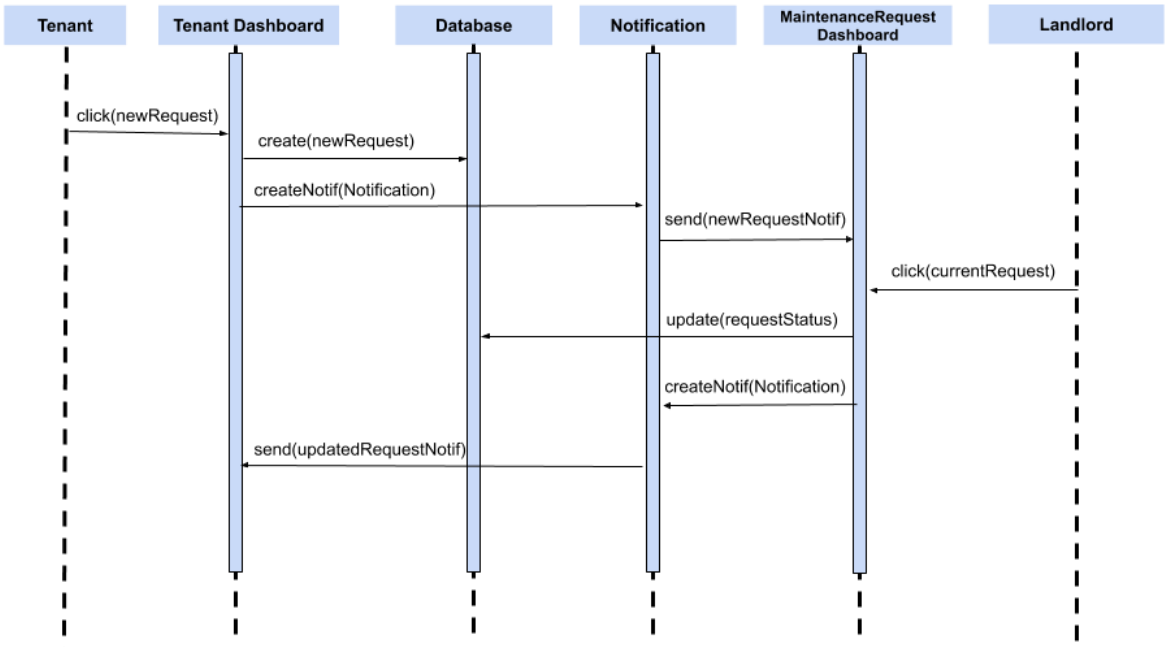


This use case uses the expert doer principle and a low coupling principle. The reason being is because there is mainly one type of information being cycled, and that is information relating to the payment from one user to the other. When a request is made from the landlord to the tenant, it is thrown to the NotifySystem which sends a pending payment notification to the tenant. The tenant then makes a payment based on the notification to the PaymentHandler. However, the payment information must be checked before it passes to the handler. Such as if the card that was entered is a valid one, and is there any missing information that is required to be filled in, or else an error will be thrown in the form of a message detailing the required inputs. Information such as that must be completed in order to go to the next step, which is the information being sent to the Stripe API. The API validates the payments and checks to see if it is correct, and sends an appropriate notification to the tenant. With the validation, the payment is then processed to the landlord.

The notification system takes in the information regarding the user's payment and its validation. This information is important since users must know if the transaction was made successful or not. If the transaction was not made successful, then the system alerts the tenant user about the error. However, if it is successful, then the user who made the payment will receive the

notification with information regarding a success statement, payment date, payment amount, and all other relevant information.

UC-7: MaintenanceRequest



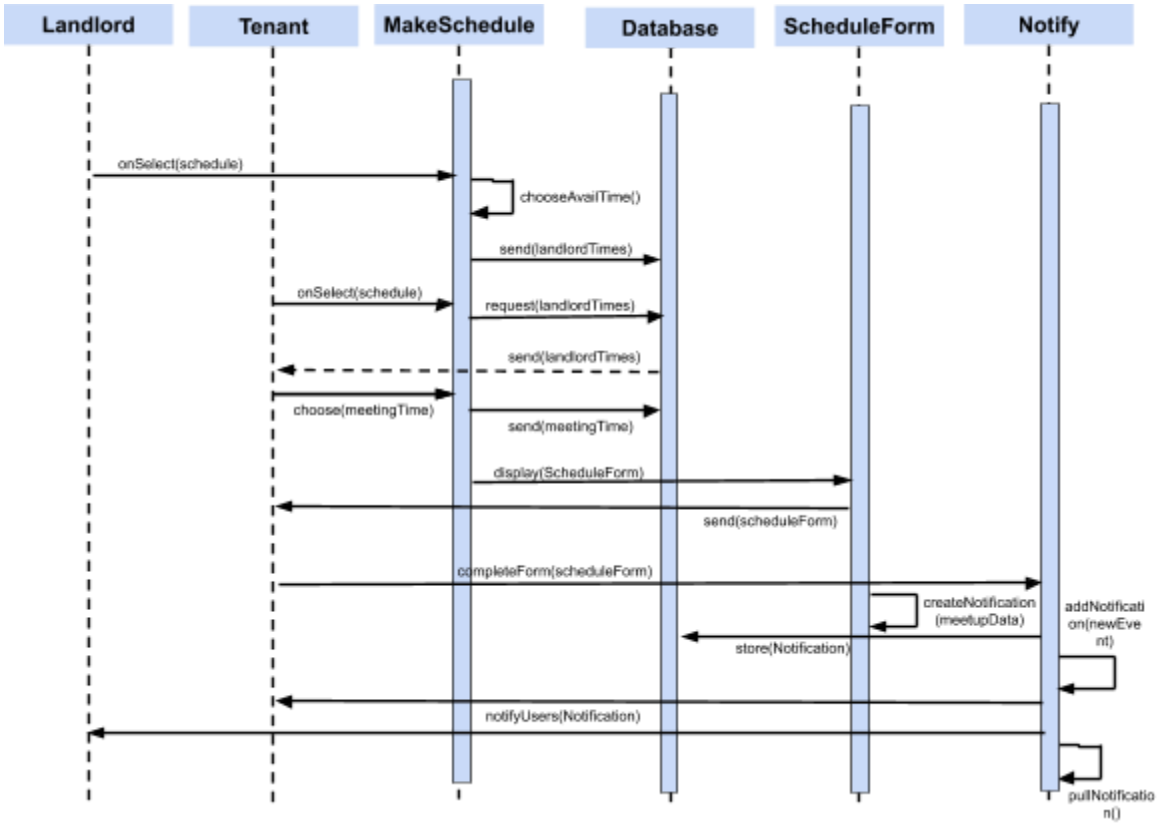
The initiating event for this use case would be a tenant clicking to create a maintenance request. Once the tenant submits all the information required to create the request, the request is then saved into the database and the landlord gets a notification about the request filed. The landlord can log in to see the notifications and click on the request for more details. The landlord can update the status of the maintenance request to alert the tenant if the issue has been dealt with yet. Once the landlord updates the status, this update is saved into the database and the tenant is notified of the change. If the tenant is not satisfied with the maintenance, the tenant can create another maintenance request.

With this use case, we use expert doer principle as landlords and tenants both interact with the database. The tenants can create a maintenance request in the database which is sent over to the landlord. Once the landlord receives the maintenance request, the landlord then can update the

status of the maintenance request and store that in the database. Finally, the tenant is notified of the change with the maintenance request. Therefore, all the updates with a maintenance request are stored in the database.

We also use high cohesion with this use case because each object has many responsibilities instead of one object doing most of the work. We use different dashboards with the tenant and landlords that can either create or update requests. Additionally, we have a separate class for notifications so that landlords and tenants can be notified of the status of their maintenance requests.

UC-10: ScheduleView



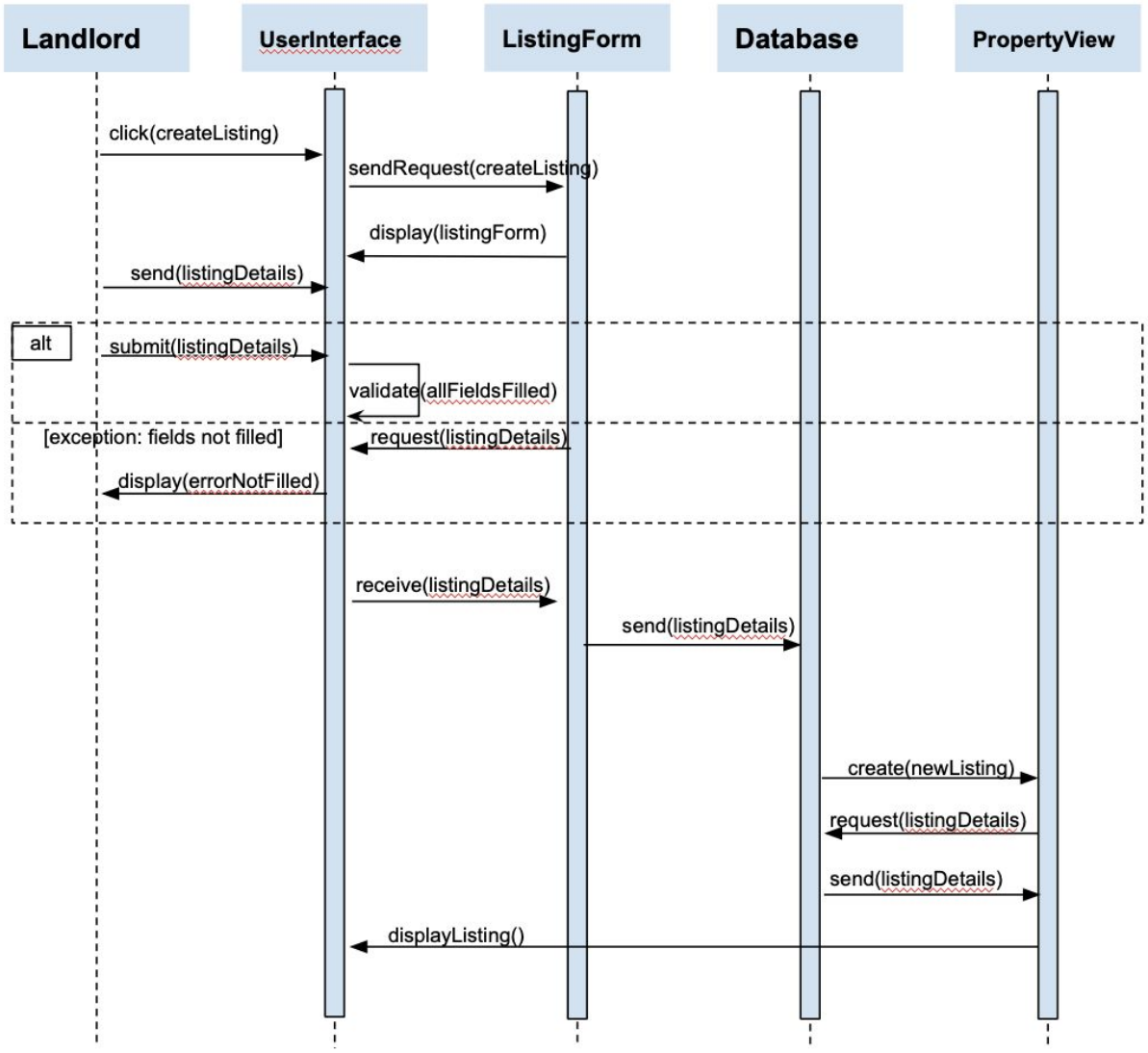
The initiating event for ScheduleView is when the landlord chooses to make his schedule from his respective dashboard. When the landlord opens his schedule page, he will choose times where he is available to meet. These times will get stored in the database so that when the tenant goes to make his schedule, he can choose the final meeting time for the times the landlord chose. After the tenant decides on a time, he will be prompted to fill out a form with even title, start and end time of the meeting, and a description of the meeting. When this is submitted, a notification

will be created and added onto the heap. Both the landlord and tenant can view a list of these notifications on their dashboard. This notification will only be pulled (taken out of the heap) once the event is completed.

The expert doer principle is applied with the database because it is the main tool that communicates to all parts of the system. The landlord and tenant both receive notifications from the system when a meeting is coming up. Likewise, makeSchedule and scheduleForm communicate with the database when they need information or need to create something.

With this use case, the high cohesion principle is used because we have minimized the number of responsibilities for each object. The landlord and tenant each have different roles in terms of the scheduling. Similarly, the makeSchedule and scheduleForm also have different responsibilities in the system.

UC-11: CreateListing



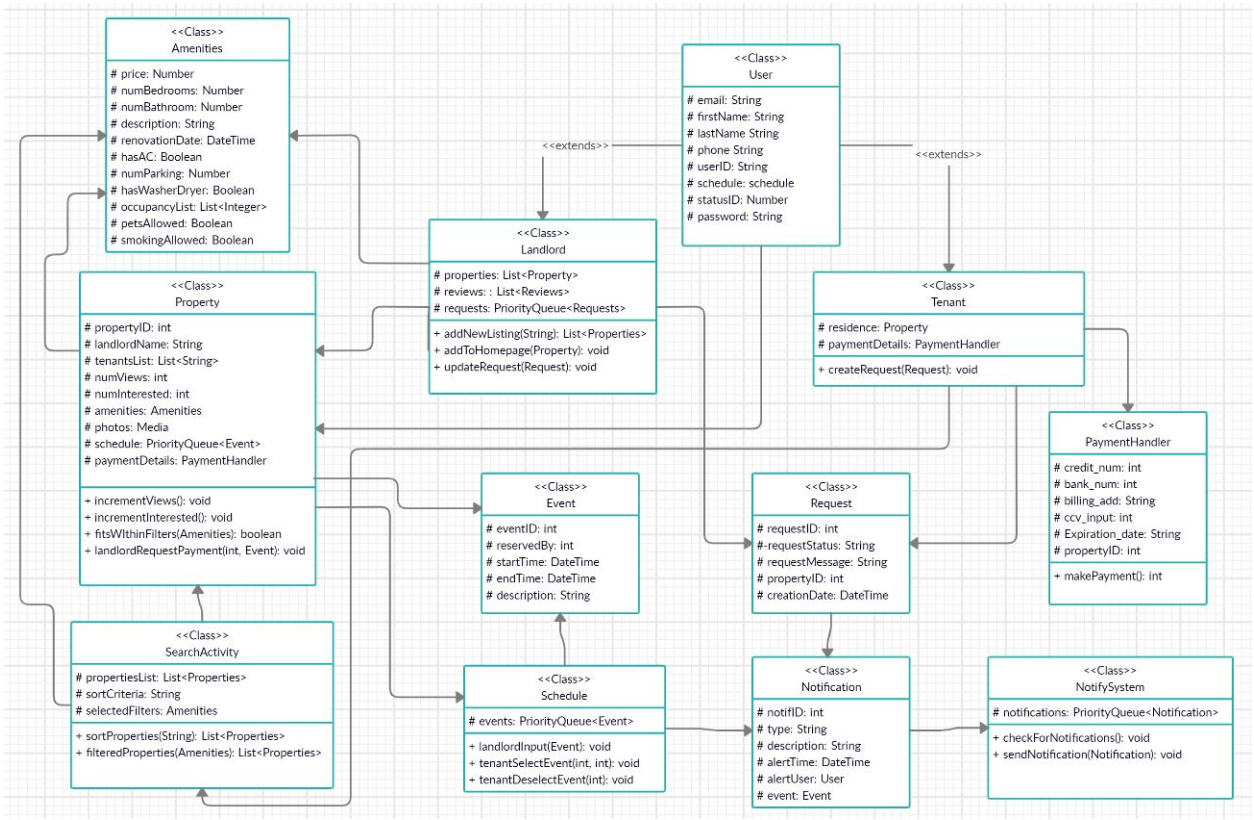
The initiating event is when the landlord clicks the create listing button. The listing form is displayed and the landlord fills in all the details of the form. When the form is submitted, the fields are validated and are only sent to the database if there are no errors. After this, when a landlord selects this property from the search page, the listing details and the corresponding property page are displayed.

For the CreateListing use case, the high cohesion principle is applied. No one object takes on too many computation responsibilities. Each object has a basic function that it performs and then communicates with the other objects to carry out a task. Therefore, the low coupling principle is also applied since not one object carries out many communication responsibilities. This creates

more dependency, but less stress on each object to perform work. By balancing the workload across objects, it creates an efficient chain of tasks to perform with minimal effort done by each object.

Class Diagram and Interface Specification

Class Diagram



Datatypes and Operation Signatures

Class: SearchActivity

Attributes:

- propertiesList: List<Properties>; Current list of properties being displayed to the user
- sortCriteria: String; User selected sorting preference of property listings
- selectedFilters: Amenities; User selected filters represented by an Amenities object

Methods:

- sortProperties(String): List<Properties>; Sorts propertiesList based on user's sortCriteria. Returns an updated list of sorted properties.
- filteredProperties(Amenities): List<Properties>; Makes database request to retrieve properties within the user's filter parameters (represented by Amenities object). Returns this filtered list of properties.

Class: Amenities

Attributes:

- price: int; Cost of rental property per month
- numBedrooms: int; Number of bedrooms in property
- numBathrooms: int; Number of bathrooms in property
- description: String; Description & additional information regarding the property
- renovationDate: DateTime; Last renovation date of the property
- hasAC: Boolean; Availability of A/C in the property
- numParking: int; Number of parking spaces provided by the property
- hasWasherDryer: Boolean; Availability of Washer & Dryer
- occupancyList: List<Integer>; List storing the number of possible tenants
- petsAllowed: Boolean; Info on if pets are allowed on the property
- smokingAllowed: Boolean; Info on if smoking is allowed on the property

Methods: (NONE)

Class: Property

Attributes:

- propertyID: int; Unique DB identifier for property listing
- landlordName: String; Landlord that owns the property
- tenantsList: List<String>; List containing tenant names if property is rented
- numViews: int; Number of prospective tenants who have viewed the listing
- numInterested: int; Number of prospective tenants who have expressed interest in the property

- amenities: Amenities; Important specifics/details associated with the property (ex. # of bedrooms/bathrooms etc.)
- photos: Media; Any pictures that will be included with the property's listing
- Schedule: schedule view for any upcoming events for the property itself

Methods:

- incrementViews(): void; Increments numViews every time a prospective tenant opens the property listing
- incrementInterested(): void; Increments numInterested when prospective tenant selects that they are interested in the property.
- fitsWithinFilters(Amenities): boolean; Checks to see if property (self) fits within a given set of filters. Returns True if within filters, False if not within filters.

Class: User

Attributes:

- statusID: int; Identifies if User is a landlord or tenant
- password: String; Password of the user
- email: String: Email of the user
- firstName: String: First name of the user
- lastName: String: Last name of the user
- phone: String: Phone # of the user
- userID: String: Unique DB identifier for the user
- Schedule: Schedule; scheduling feature for user

Methods: (NONE)

Class: Tenant

Attributes:

- residence: Property; Property being rented by the tenant. NULL if tenant is looking for housing.
- paymentDetails: PaymentHandler; contains an object with all the tenant's payment information

Methods:

- create(Request): void; Tenant can file a maintenance request

Class: Landlord

Attributes:

- properties: List<Property>; List of properties owned by the landlord (self)
- reviews: List<Reviews>; List containing reviews of the landlord (self)
- requests: PriorityQueue<Request>; List containing all the maintenance requests the landlord has, in chronological order

Methods

- addNewListing(String): List<Properties>; Method that the landlord is able to invoke that begins the process of creating a new listing
- update(Request): void; Landlord can update the status of a maintenance request
- addToHomepage(Property): Void: Landlord is able to create a listing and add it to list of properties and submits the listing for public view

Class: Schedule

Attributes

- events: PriorityQueue <Event>; a chronological queue that is used to view a user's or listing's schedule

Methods

- landlordInput(Event): void; When landlord arrives at the schedule page of one of their properties, they are able to create and input an Event
- tenantSelectEvent(int, int): void; When tenant arrives at the schedule page of a property listing, they are able to select an open event (i.e. Open House), and schedule a viewing. TenantID and EventID are passed to the method.
- tenantDeselectEvent(int); void; If a tenant can no longer make a scheduled viewing, they can deselect the event (EventID is passed to the method to cancel it).

Class: Event

Attributes

- eventID: int; unique identifier for each event that is created
- reservedBy: int; id of who created the event
- startTime: DateTime; date and time when the event begins
- endTime: DateTime; date and time when event ends
- description: String; any other necessary information about the event (location, phone number, etc.)

Class: Notification

Attributes

- notifID: int; unique identifier for each notification that is created
- type: String; what the notification is for (tour, maintenance request, etc.)

- description: String; information about the notification
- alertTime: DateTime; when the user will be notified
- alertUser: User; which user will be alerted
- event: Event; information about the upcoming meeting (see Event Class)

Class: NotifySystem

Attributes

- notifications: PriorityQueue<Notification>; chronological order of any upcoming notifications (of the events) for the user

Methods

- checkForNotifications(): void; Internal loop running at all times on a separate thread, which constantly checks to see if any notifications need to be sent
- sendNotification(Notification): void; When the Notification.alertTime has been reached for a Notification, this message sends the notification to the appropriate User

Class: PaymentHandler

Attributes:

- credit_num: String; Credit card number of the tenant
- billing_add: String; Billing address of the tenant
- ccv_input: Number; Card Verification Value of the credit card
- expiration_Date: String; Expiration date of the credit card
- propertyID: int; Unique identifier for property that tenant is paying for

Methods:

- makePayment(): int; Payment details are sent to a 3rd party service to complete the payment, and an exit code containing the status of the payment (i.e. success, failure, etc.) is returned.

Class: Request

Attributes:

- requestID: int; Unique identifier assigned to the maintenance request
- requestStatus: String; Current status of the maintenance request
- requestMessage: String; Short description of what kind of maintenance is required
- residence: int; contains propertyID of the property that maintenance is required for

Traceability Matrix

	Domain Concepts																		
Software Classes	**SearchPage**	**FilterSelector**	**SortSelector**	**SearchController**	**PropertyContainer**	**MakePayment**	**ProcessPayment**	**PaymentReview**	**PaymentPage**	**NotifyTenantPending**	**TenantPaymentPage**	**andLordPaymentPage**	**LandlordPage**	**ScheduleMaker**	**ScheduleForm**	**Notify**	**Listing Form**	**View Property**	**UpdateRequest**	**InputRequest**
User														X		X				
Landlord									X	X		X	X				X		X	
Tenant								X	X		X							X		X
SearchActivity	X	X	X	X																
Property		X			X								X	X		X	X	X		
Amenities		X			X												X	X		
PaymentHandler						X	X	X	X		X									
NotifySystem						X				X		X	X	X		X				
Schedule													X	X						
Event						X				X				X	X	X				
Notification						X				X	X	X	X	X		X			X	X
Request																			X	X

- ScheduleMaker
    - User: Every user of the website has the ability to create a schedule for important events and meetings
    - Property: Every property will also have a schedule associated with it for things like housing tours or maintenance requests.
    - NotifySystem: Everytime an event is created, a notification is added to this chronological list of upcoming events.
    - Event: Event represents the important information about the meeting.
    - Notification: When an event is created, a notification containing an event is also created and added to the list of user's upcoming events.
    - Schedule: Schedule is the user or property's tool to create and view any events
- ScheduleForm
    - Event: Event represents all the pertinent information that the user fills out about the meeting
- Notify
    - User: Every user will have a list of notifications for upcoming events they have.
    - Property: Every property will also have a schedule where upcoming meetings and events can be seen.
    - Event: Event is the description of the meeting that the users are being notified about.

    - Notification: Each individual alert and the event description that is associated with it to remind the user that they have an event coming up
    - NotifySystem: Chronological list of all upcoming event notifications that the tenant or landlord should be aware of
- ListingForm
    - Landlord: When the landlord wants to create a new listing on the website, they have to use a form to input all the important information.
    - Property: Once the landlord fills everything out, a property page will be created containing all of that information.
    - Amenities: The landlord has to fill out all the information about the features of the house (the number of bathroom, bedrooms, rent amount, etc.)
- ViewProperty
    - Tenant: every tenant should be able to view any listings that are created and posted by the landlords
    - Property: the actual page that contains pictures, descriptions, and any other important information about the house
    - Amenities: features of the house (number of bathrooms, bedrooms, price, etc.)
- LandlordPage
    - Landlord: every user that is a landlord has a dashboard for all their important tools
    - Property: every landlord's dashboard has a list of their properties
    - NotifySystem: chronological list of all upcoming events for the landlord
    - Schedule: every landlord has a scheduling feature on their dashboard for choosing times that they are available to meet
    - Notification: individual alerts and event description for upcoming events for the landlord
- SearchPage
    - SearchActivity: The SearchActivity populates the visual interface once it has retrieved/sorted the properties
- FilterSelector
    - SearchActivity: Selected filters are stored by the SearchActivity and filtered properties are returned & displayed
    - Property: Filters are passed to each Property to check if they fit within the specifications
    - Amenities: Selected filters are stored as an Amenities object
- SortSelector
    - SearchActivity: User specified sort criteria are stored by the SearchActivity and sorted properties are returned & displayed
- SearchController

    - SearchActivity: Makes database requests once filters and sort criteria are selected, and displays the results
- PropertyContainer
    - Property: DB identifiers and at-a-glance details are stored in Property objects
    - Amenities: Specifications of each property are stored separately in Amenities objects
- InputRequest
    - Request: Tenant is allowed to create a Request to send to his/her landlord
    - Notification: Tenant will be notified when landlord has dealt with the issue
- UpdateRequest
    - Request: Landlord can update the status of a current Request
    - Notification: Landlord is notified when the tenant has a maintenance request
- MakePayment
    - PaymentHandler: Tenant enters information needed to make a payment
    - NotifySystem: Tells system that payment is being made
    - Event: Event represents all information about the current payment being made
    - Notification: Sends notification to tenant/landlord that payment is being made
- ProcessPayment
    - PaymentHandler: Processes the payment with the given information
- PaymentReview
    - PaymentHandler: Will allow user to review information once payment is processed
- PaymentPage
    - PaymentHandler: Displays prompt for user to enter payment information
- NotifyTenantPending
    - NotifySystem: System sends notification to tenant that there is a pending payment
    - Event: Event is the pending payment sent out by the landlord to the tenant
    - Notification: Tenant will receive notification that there is a pending payment
- TenantPaymentPage
    - PaymentHandler: Displays information for accessing payment handler to enter payment information
    - Notification: Displays any notifications regarding payments on the tenant page
- LandlordPaymentPage
    - NotifySystem: Displays option to send a notification of pending payments to system
    - Notification: Displays notifications for payments made by the landlord’s tenants

System Architecture and System Design

Architectural Styles

The architecture of our system can be described by several common styles of System Design. Primarily, the system will be designed as a 3-tiered client-server system. The client-server delineation is necessary, because the client, users that are landlords and tenants, are separated and abstracted from the server side. Additionally, the client makes requests that trigger reactions from servers, a hallmark of client-server systems. Here, we denote the server as containing a controller and database, thus giving the architecture 3 tiers.

Another important philosophy behind the design of the system is making the architecture component based. This can be seen in the way our project has been decomposed into 4 main components throughout this and the previous report. The 4 components are: tenants searching for housing, landlords seeking tenants, tenants, and landlords. By clearly defining the four types of users, the experience can be tailored into four discrete types of interactions, so users get more relevant features and information. Along with this, there is easy reuse for many different tenant or landlord users based on a framework that will dictate how they interact with the site. Making the system component based also helps encapsulate the system so that users only interact with methods relevant to themselves, not other classes of users.

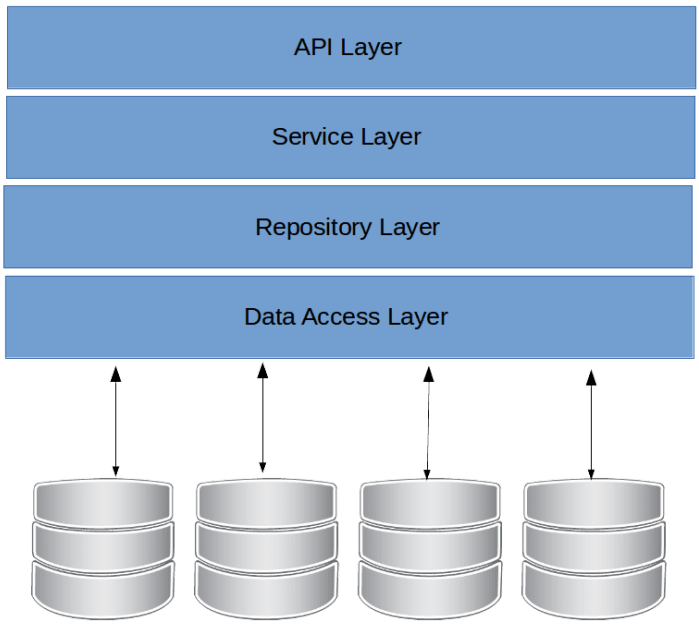
Another important architectural feature that comes as a result of our system design, is that we are incorporating many tenants of a data-centric system. Our project involves heavy use of a standard relational database (SQL) to store necessary data about properties, users, and more. Additionally, this storage takes place on the server side rather than client side, a necessary component for ensuring proper architecture in a data-centric system. Two decisions related to this design philosophy were to use table-driven logic, because it is more flexible, and to use some stored procedures that run on database servers in order to prevent injection attacks. On a larger scale, the overall architecture of the project has shared data models. For example, property objects have the same construction/data for landlords and tenants; this commonality between aspects of the system is due to the data-centric approach.

For the front-end aspects that users interact with we have chosen to follow Representational State Transfer (REST) for the web API. This is useful because it allows for interoperability between computer systems on the internet. For this project, that will be necessary, because many different clients will be interacting with the site from many different computers and types of systems. Because of the diversity of systems interacting with the site, we will implement stateless protocol

and standard operations so that the website is malleable. Additionally, using REST will lead to a reliable and uniform web representation.

Identifying Subsystems

The three subsystems in the system are the client, controller, and database. The below diagram models the structure of our back end, with the layers being part of the controller, and the controller having access to the database. The API Layer directly interfaces with the front-end, which is what the user sees, this service layer applies any logic, the repository layer provides encapsulation of the way the data is actually stored, and the data access layer handles storing and retrieving data.

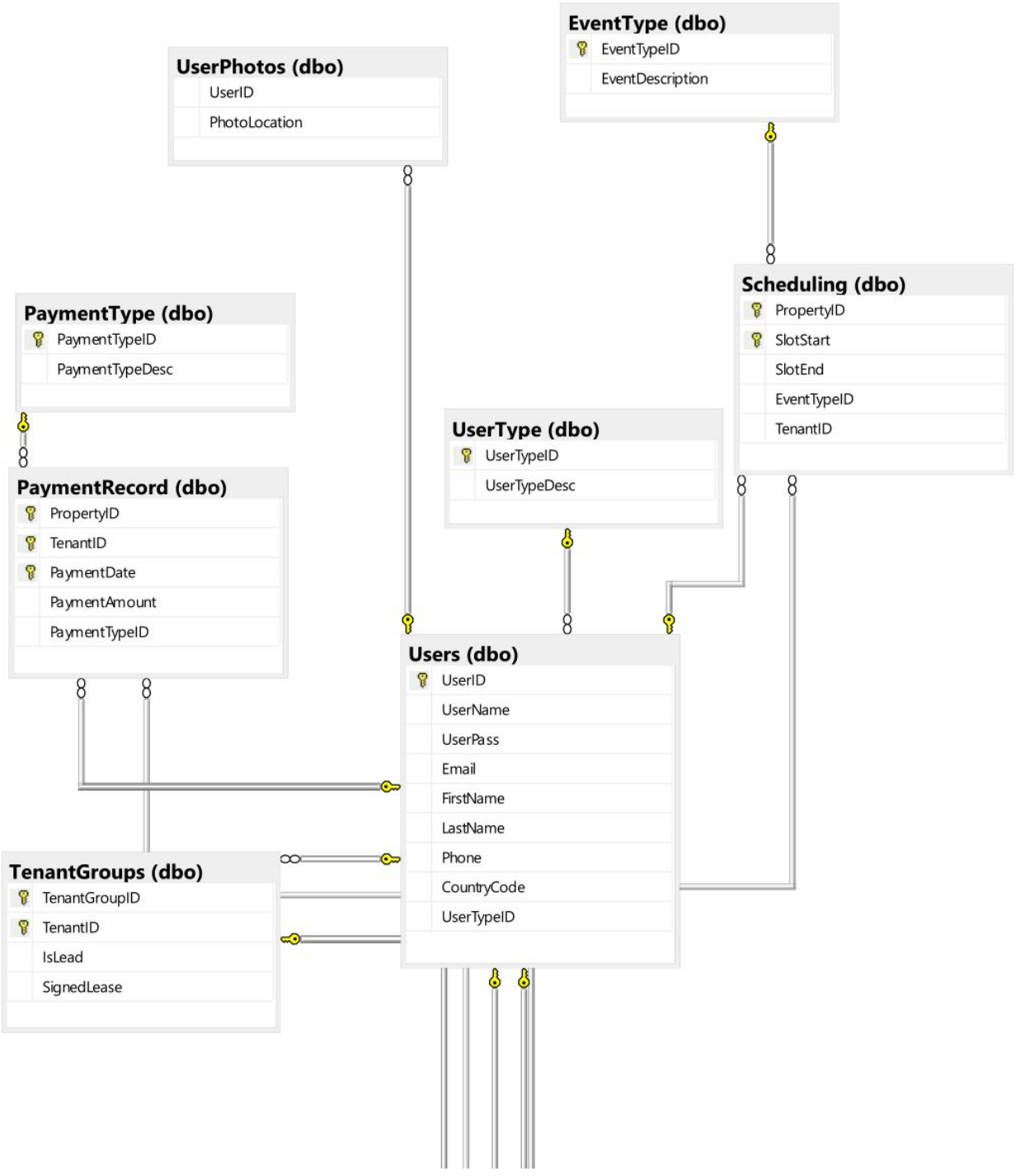


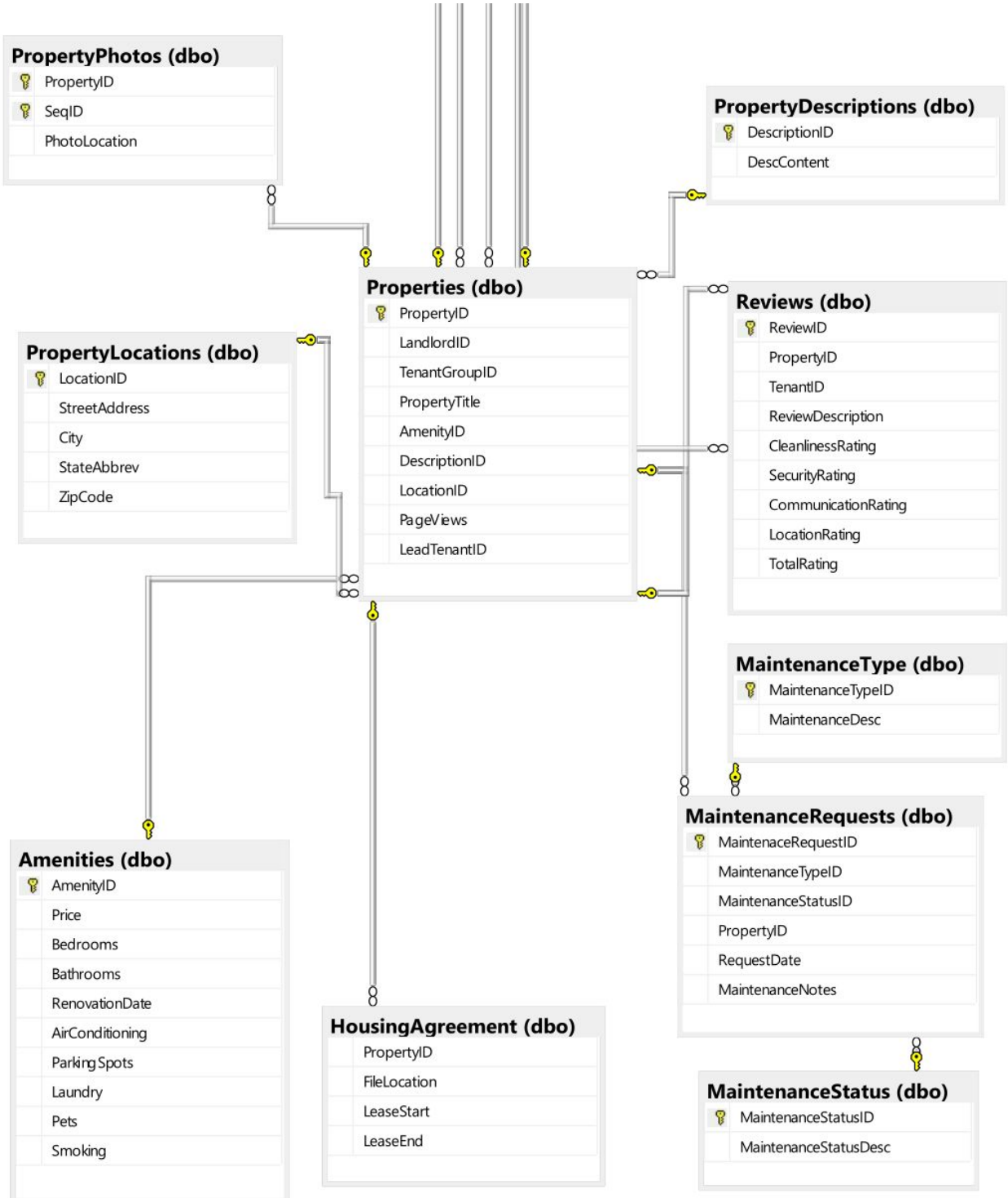
Mapping Subsystems to Hardware

The system will need to run on multiple computers. The server, including the controller and database, will be hosted using Microsoft Azure, and running independently of the client subsystem. The client subsystem will run on a web browser of a different device for each user.

Persistent Data Storage

The system architecture pairs with the use cases and design of the system to require that it save data that needs to outlive a single execution of the system. Permanent objects will be stored using a relational database: a SQL server hosted by Microsoft Azure. The database will hold information about each user account, both tenant and landlord, as well as property information. The database diagram can be seen below:


EventType (dbo)
EventTypeID
EventDescription
UserPhotos (dbo)
UserID
PhotoLocation
Scheduling (dbo)
PropertyID
SlotStart
SlotEnd
EventTypeID
TenantID
PaymentType (dbo)
PaymentTypeID
PaymentTypeDesc
UserType (dbo)
UserTypeID
UserTypeDesc
PaymentRecord (dbo)
PropertyID
TenantID
PaymentDate
PaymentAmount
PaymentTypeID
Users (dbo)
UserID
UserName
UserPass
Email
FirstName
LastName
Phone
CountryCode
UserTypeID
TenantGroups (dbo)
TenantGroupID
TenantID
IsLead
SignedLease


PropertyPhotos (dbo)
PropertyID
SeqID
PhotoLocation
PropertyDescriptions (dbo)
DescriptionID
DescContent
Properties (dbo)
PropertyID
LandlordID
TenantGroupID
PropertyTitle
AmenityID
DescriptionID
LocationID
PageViews
LeadTenantID
Reviews (dbo)
ReviewID
PropertyID
TenantID
ReviewDescription
CleanlinessRating
SecurityRating
CommunicationRating
LocationRating
TotalRating
PropertyLocations (dbo)
LocationID
StreetAddress
City
StateAbbrev
ZipCode
MaintenanceType (dbo)
MaintenanceTypeID
MaintenanceDesc
MaintenanceRequests (dbo)
MaintenaceRequestID
MaintenanceTypeID
MaintenanceStatusID
PropertyID
RequestDate
MaintenanceNotes
Amenities (dbo)
AmenityID
Price
Bedrooms
Bathrooms
RenovationDate
AirConditioning
Parking Spots
Laundry
Pets
Smoking
HousingAgreement (dbo)
PropertyID
FileLocation
LeaseStart
LeaseEnd
MaintenanceStatus (dbo)
MaintenanceStatusID
MaintenanceStatusDesc

Network Protocol

As our project can run on potentially up to three machines, we do use multiple network protocols for connecting the machines. At a minimum, however, it runs on two machines, as the database is a managed database ran on the Azure platform, and the client and middleware can be run on the same server in separate containers. Either way, to connect the client and middleware, we use the HTTP(S) protocol, as it allows us to send GET/POST/PUT/DELETE/UPDATE requests through the REST architectural style, and it is a widely adopted standard, with plenty of documentation and implementation examples. To connect the middleware with the database we use JPA, which is built on top of JDBC. It uses JDBC to connect the database, but once connected, you can interact with the database using object models, and very few lines of extra code to persist or access data. Although this doesn't offer a whole lot of flexibility, it works well with our model, and is widely accepted and used with Java Spring, so there are many examples and documentation sources available.

Global Control Flow

- *Execution Orderness:* This system is an event-driven system, as the user controls the flow of events that occur, and different users can see different items depending on if they are a landlord or tenant, and the system waits for input from the user before generating the next action, which varies based on what the input is, such as a user selecting different filters to search for different listings.
- *Time dependency:* Since our system is dependent on user input, our system is an event-response system. One time-dependent component of our project is our weekly email digests. After prospective tenants filter their results based upon what types of properties they are looking for, they will receive an email every week notifying them of any new properties that are posted that fit their criteria.
- *Concurrency:* Every time a user connects to the website, a new thread is created that corresponds to that user. Apache Tomcat web server will handle all of the concurrent requests that are made to the website through a thread connection pool. Each incoming request is assigned to a new thread, and once that thread is finished, it can be reused for another task. Multiple threads running at the same time allows for multiple users to be logged into the site at the same time. Additionally, a thread is needed for every request that is made to handle the interaction between the system and the database.

Hardware Requirements

Our application has separate requirements for the client and server side. On the client end, the user needs a device that has the ability to run a web browser. The application is setup responsively so that it should work on most, if not all screen resolutions, although for the full user experience, larger resolution is always better (1280x720 or greater). Hard disk space is not necessarily needed, although to persist cookies it is recommended to have at least 32MB of space available. A network connection is also required, we recommend at least 500 Kbps for an ample user experience. Additionally, more modern browsers such as Chrome, Edge, Opera, and Firefox are recommended for all the functions of the site to operate properly.

On the server side we require at least two servers. One being the managed SQL database, which is hosted by Microsoft Azure, and the other being a hosted VPS on Azure, running Linux, to handle both the client and middle-tier portions of the application. The database resources are not calculated in the traditional sense, Azure uses DTUs or Database Transaction Units, of which we have 10 DTUs. The amount of storage we have is 250 GB, which should be more than necessary. The VPS should have at least 20GB of hard drive space, 4 GB of RAM, and at least a 4-Core processor. A gigabit network connection is also important to ensure little to no network latency on the server side.

Algorithms and Data Structures

Algorithms

No algorithms that implement mathematical models are used in this project.

Data Structures

In this project we are making use of 3 different data structures. Arrays, Queues, and Hashmaps.

Arrays & ArrayLists
An ArrayList is a special implementation of the Array data structure. It dynamically allocates more space (more Arrays) if more elements need to be added.

When searching for properties, a user selects filters it wants applied, in order to narrow down properties that are relevant to their interests. On the backend, once filters are applied in the database to find relevant properties, the results are returned as an ArrayList of Property objects. ArrayLists are mutable, so their content and length can be changed as needed. With the necessary index of an item in an array, lookup takes O(1) time, making further access to the individual properties very fast, which is essential if a user is to click on each property for further information.

On our homepage, we have an autocompleting search bar which allows the user to select which university they want to view property listings for. Every time a user types in a character, the front end needs to sift through the entire university array and return universities that contain the input characters. We decided that an Array was the best data structure for this task because it had a spatial complexity equal to other data structures (O(n)) and had the best access time complexity (O(1)), which was important to us considering the function needed to access every element each time it was called.

Queue & PriorityQueue
A PriorityQueue is a special implementation of the Queue data structure. It automatically sorts the contents of the Queue by a specified Comparator.

In our project, we will be using PriorityQueues to store notification data and our specified Comparator will be the DateTime a user needs to be notified of an event. This allows us to store notifications in a FIFO (first in, first out) chronological order. We decided that a PriorityQueue

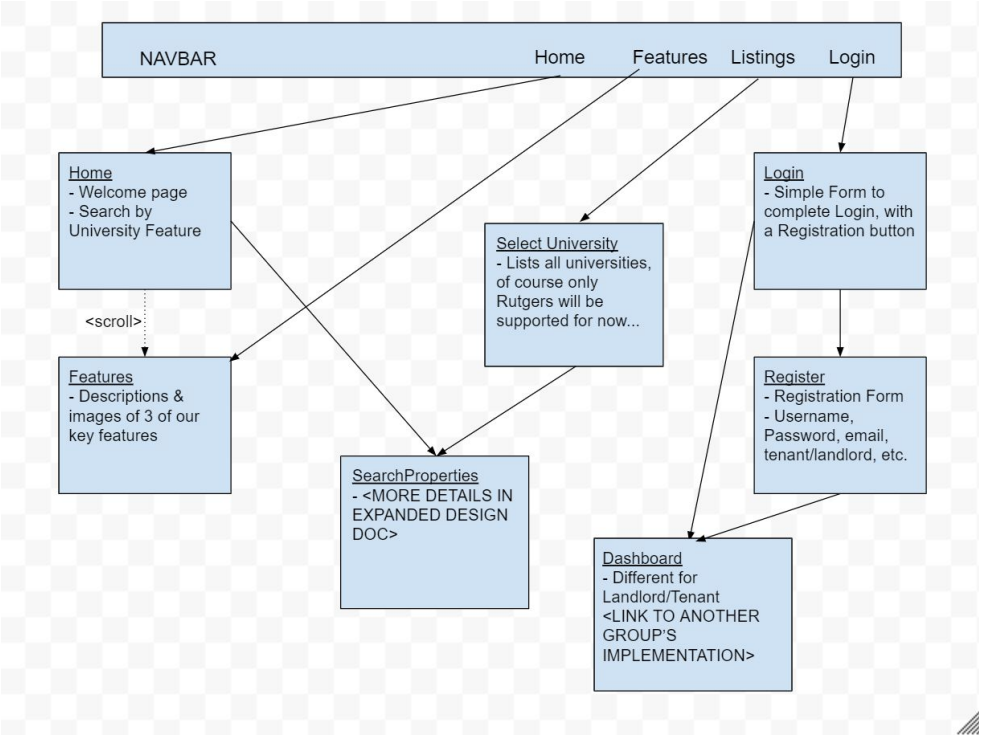
was the best data structure for this task because it had the same spatial complexity as other data structures (O(n)) and for accessing the most urgent notifications the access complexity would be O(1), as the most urgent notifications will be at the top of the PriorityQueue. For our function, this is the fastest option.

Another important use of PriorityQueue in our project is for the email digest feature. With this feature, each user is sent a weekly tailored email featuring all new properties that fit their desired specifications. To accomplish this, users who sign up for the digest will be placed in a priority queue, with their "priority" being the desired time of sending for that user. A timer object will be running, and the time will constantly be compared against the send time for the top priority user. This will be accomplished using the "peek()" functionality of a priority queue. When the times match, the email is sent, and the user is popped from the queue, given a new priority, and then pushed back onto the queue. Priority queues are implemented similarly to heaps, thus, each insertion and deletion (push and pop) have an average runtime of O(logn), which is overall best case for the desired features of individual email send times.

Hashmap

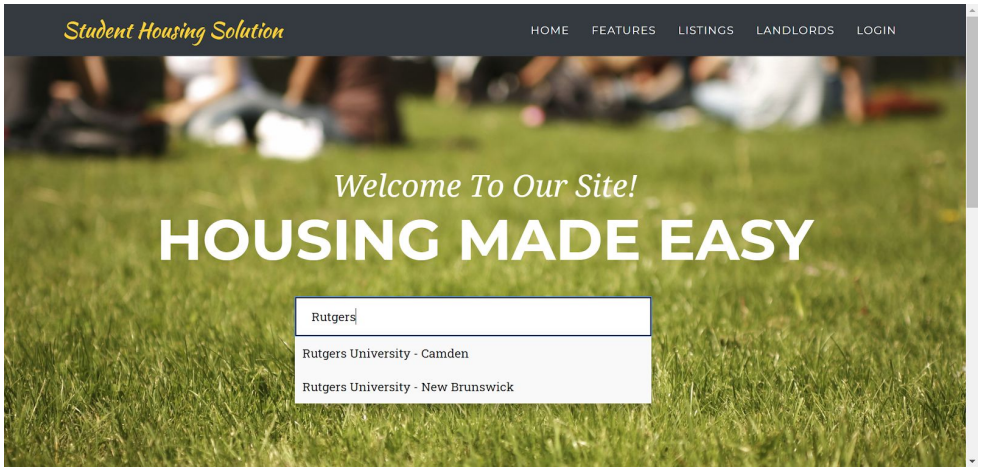
Through Java, the Stripe API was imported, and it allowed the implementation of certain data structures to be easily handled. These data structures were Hashmaps and Arrays, and they were necessary with how data was being stored and traced. Hashmaps are used frequently within the payment system to correctly input data to a certain key. When inputting information within the hashmap, the big-O operation is O(1), while searching throughout the map the average and worst case for big-O operations is O(logn). Within the payment system, an outer map is created to represent customer information, and inner maps are made within the outer map to represent other relevant data such as credit card information, bank account, and Stripe account details. For example, a customer is given an id which is used to identify the outer map that represents that specific information of the customer. The id is able to be referenced from the inner maps so that any data that is stored in the inner maps can be traced back to the outer map, which in this case is the customer. Additionally, when a Stripe account is created, a customer is given certain capabilities that allow them to perform actions within the system. A key named requestedCapabilities can be called to output an Array of strings that contain the capabilities of a user.

User Interface Design and Implementation

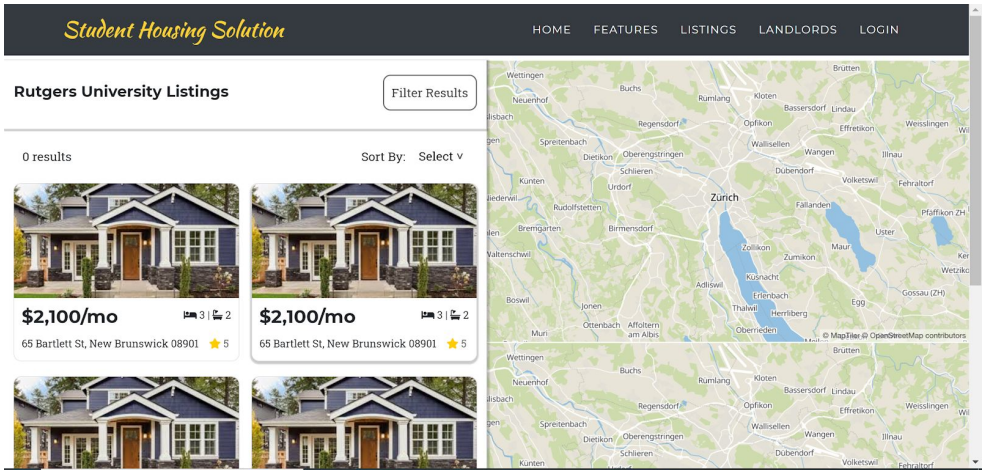


The above navigation path diagram shows how to access the SearchProperties page, and the Login/Registration pages (which are elaborated on below).

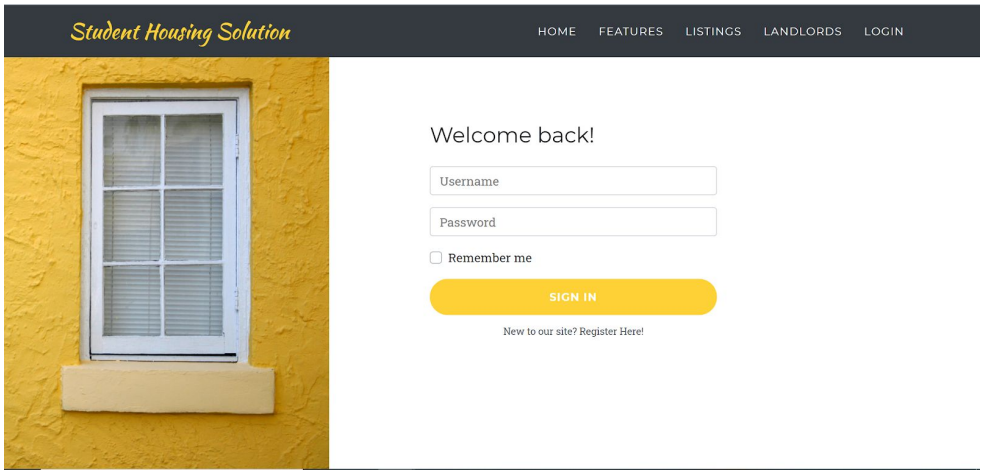
- To access the Login/Registration pages, minimal user effort is required (1 mouseclick for Login, 2 mouseclicks for Registration).
- For the Search Properties page, if we are taking the Rutgers University - New Brunswick listings page as an example, it requires a minimum of:
    - 2 mouseclicks (click on Listings tab, then click on “Rutgers University - New Brunswick”)
    - 2 keystrokes, 1 mouseclick (type in “Ru” in the search bar on the homepage, which will autocomplete and you can click on “Rutgers University - New Brunswick”)

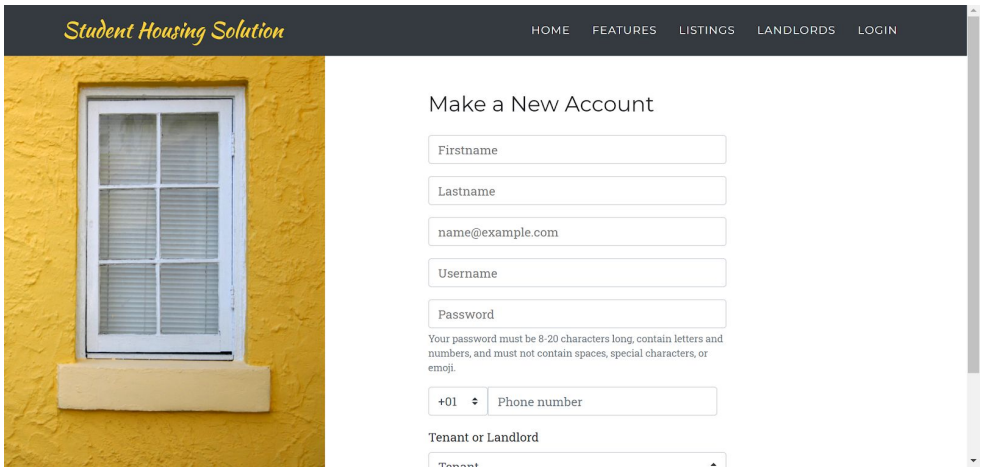


Our homepage implementation was designed with the user in mind: it is simple, clean, and easy to navigate. The design followed along with our rough mockups from Report 1, as there is a navigation bar on the top with links to several important areas of the site. Additionally, the homepage's main feature is an auto-fill search bar that populates with different universities to find housing for. All of our front end was implemented using ReactJS, which allowed us to keep this page very similar to the screen mock-ups and easily implement dynamic features like the autocomplete bar.

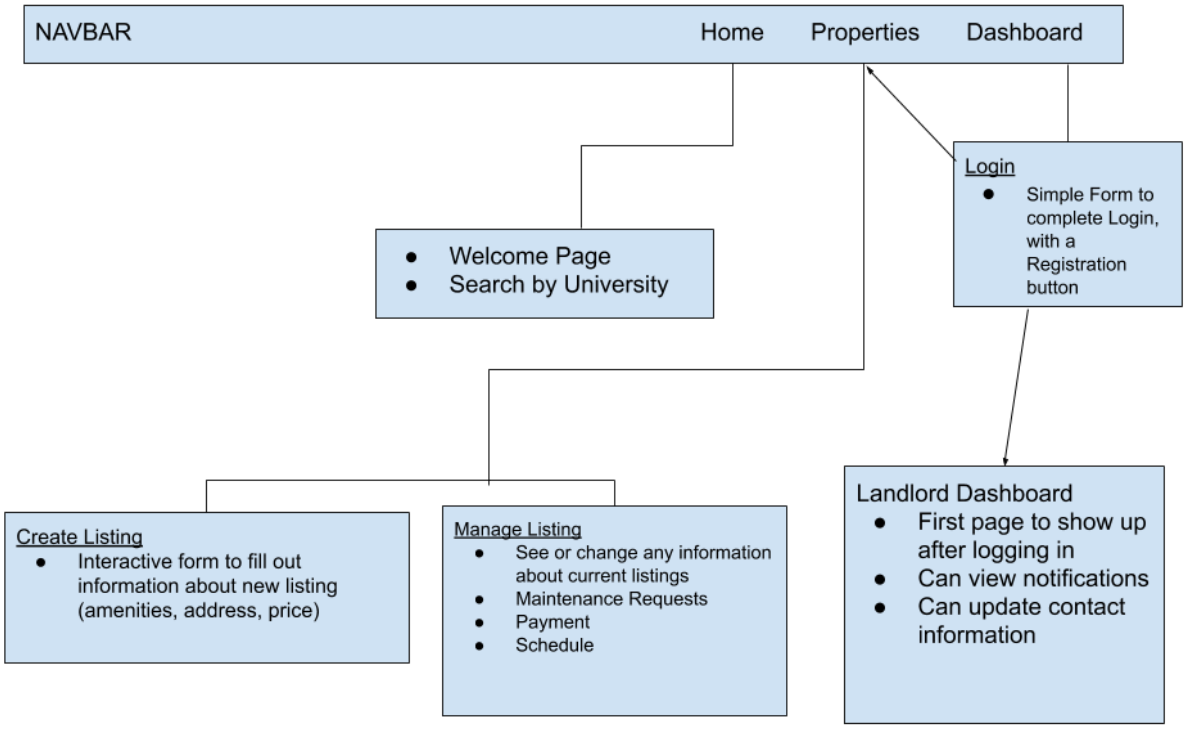


Our current implementation of the “View Listings” feature largely follows the design from report one, but has been simplified and streamlined to be more user friendly. The most obvious difference is the placing of the map on the right side of the screen instead of the left. This was made with the user in mind, as it showed to be more intuitive in this manner; implementing this change was very simple. Currently, the property view on this screen does not show page views or number of people interested in the property. While this still could be changed, this decision was made to cut down on the information on the screen to keep it clean and prevent it from becoming overwhelming.



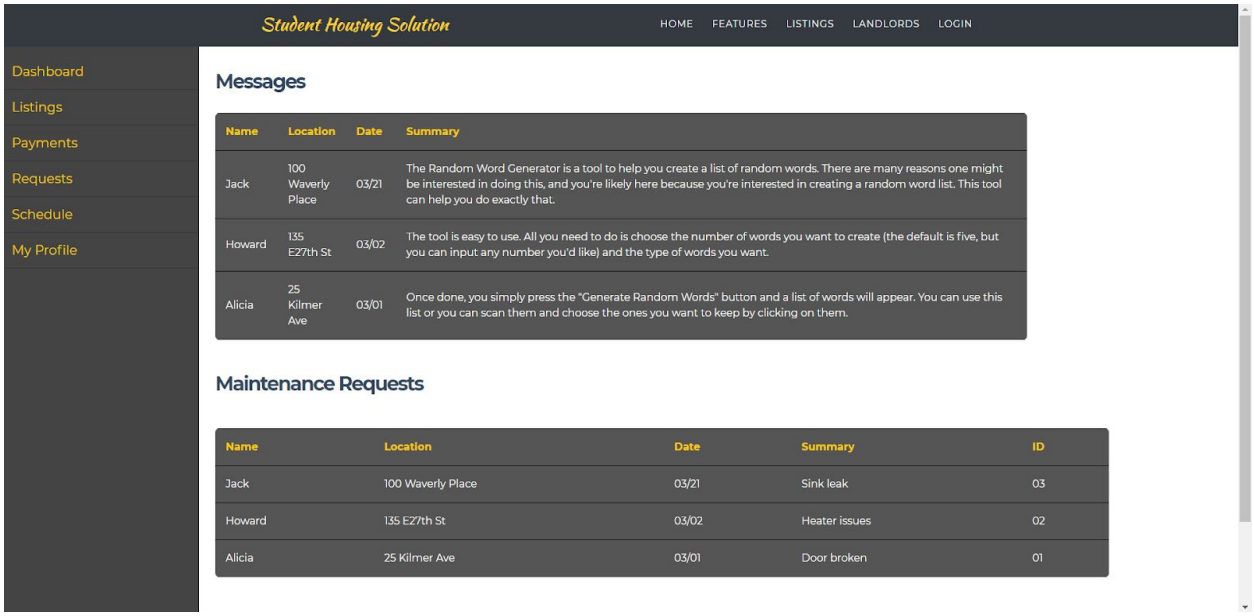


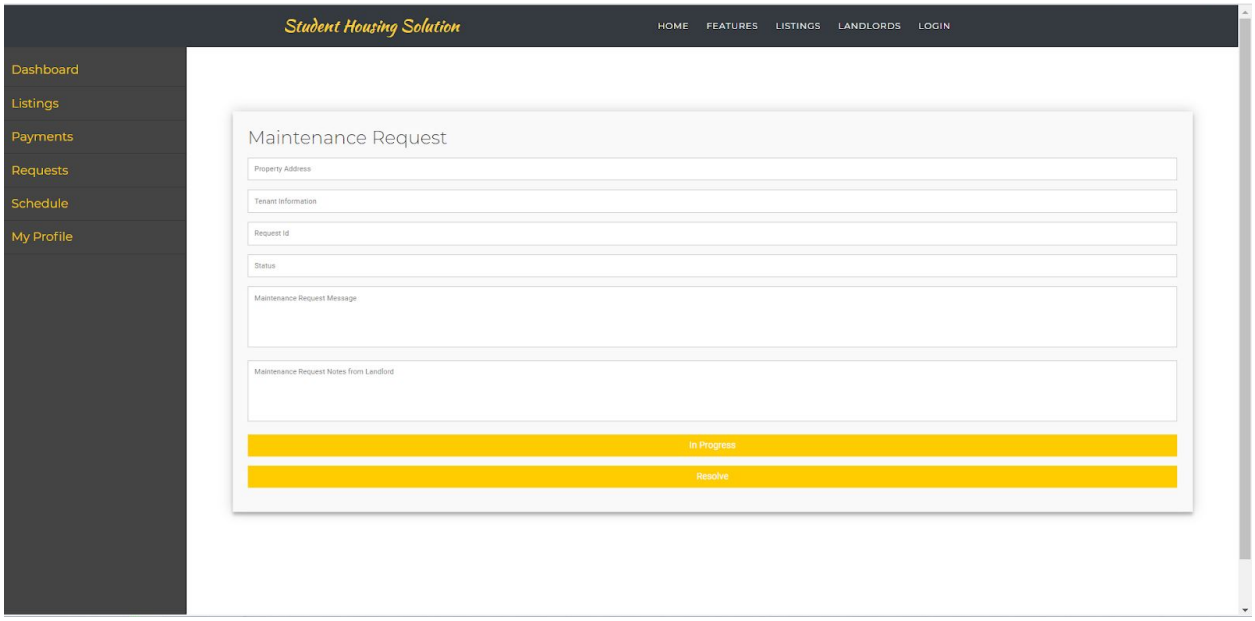
The login and registration pages were not included in the UI designs for the first report, but since they are fully functional and largely finished, they were included to demonstrate the continuity of UI theme throughout the site. These pages are minimalist and cleanly designed. In particular for the registration page, there was thought put into how to best make this form so that the user can easily create an account while still being visually appealing. The page automatically provides messages to make sure that all inputs are correctly validated, i.e. the requirements on the screen for the password. Input validation is done client side, so that the user doesn't have to attempt to make an account, wait for the site to make a post request, and then find out their password is incorrectly formatted.



The above diagram shows how to access the Dashboard and Maintenance Request Page:

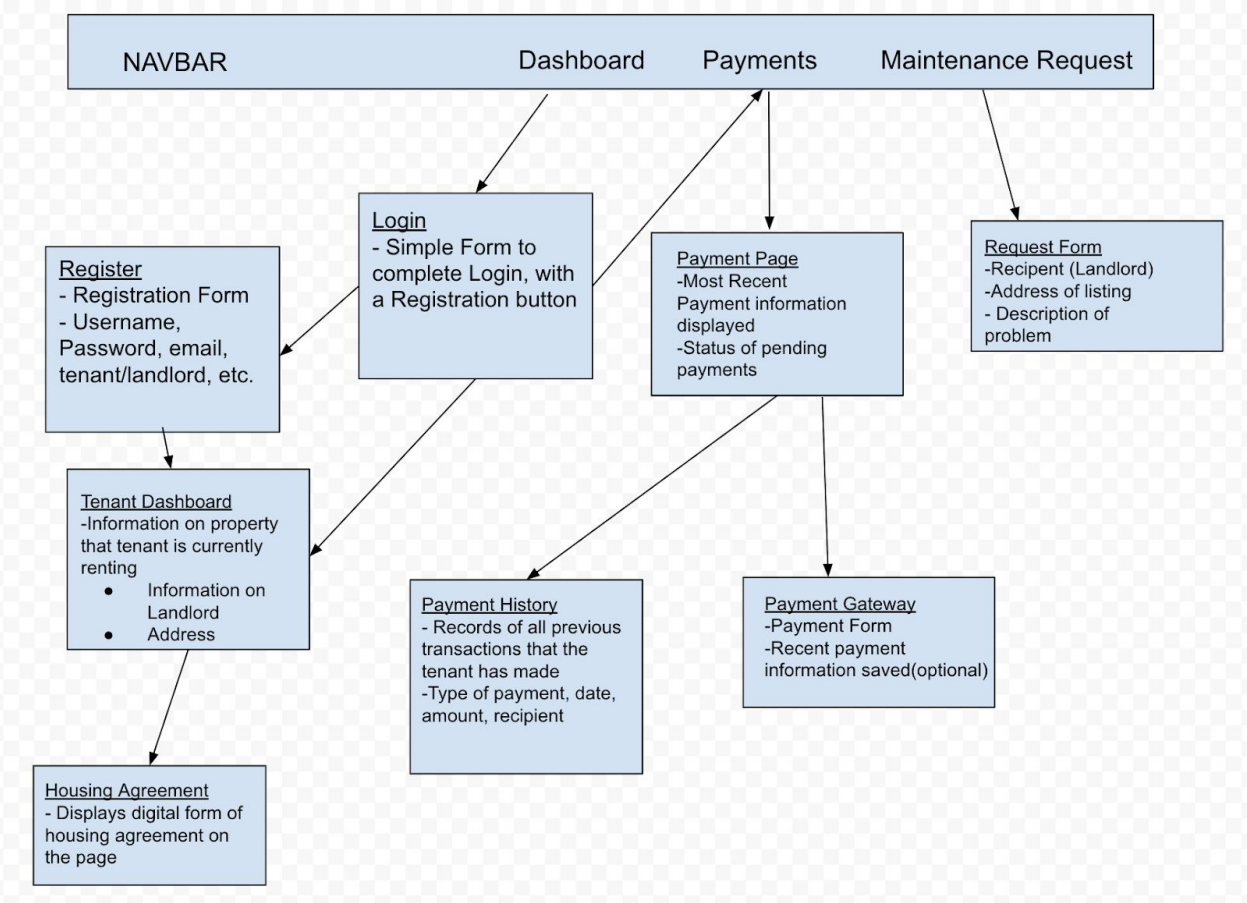
- To access the Dashboard Page, the minimal user effort required is 2 user inputs for log in, 1 mouse click to log in and 1 mouse click on Dashboard/Login
    - The number of keystrokes would depend on the landlord's username and the password. For example, if the username was "landlord123" and the password was "password123," it would take a total of 22 keystrokes to reach the Landlord Dashboard.
- To access the Maintenance Request Page, 2 user inputs and 3 mouse clicks are required. The 2 user inputs would be to log in, 1 mouse click to log in, 1 mouse click to click on Dashboard, and then 1 more mouse click to get to the Maintenance Request page. The number of keystrokes would again depend on the username and password as mentioned above.





The dashboard and maintenance are cleanly designed to go with the UI theme throughout the site. The dashboard was particularly designed to be visually appealing and offers easy navigation throughout the website. It provides easy access to inquiries, maintenance requests, payments, listings and messages for the tenants. This page acts as a hub for the user as it allows easy navigation all around. Maintenance request page offers a simple way for tenants to submit a request. In order to minimize user effort for Landlords and Tenants, it only asks for necessary information for each request. Landlords could easily find the property and tenants through the submitted request. It also offers textboxes for tenants and owners to keep track and update the

ongoing request. These pages were created to access all possible details and ease mode of communication between both parties.

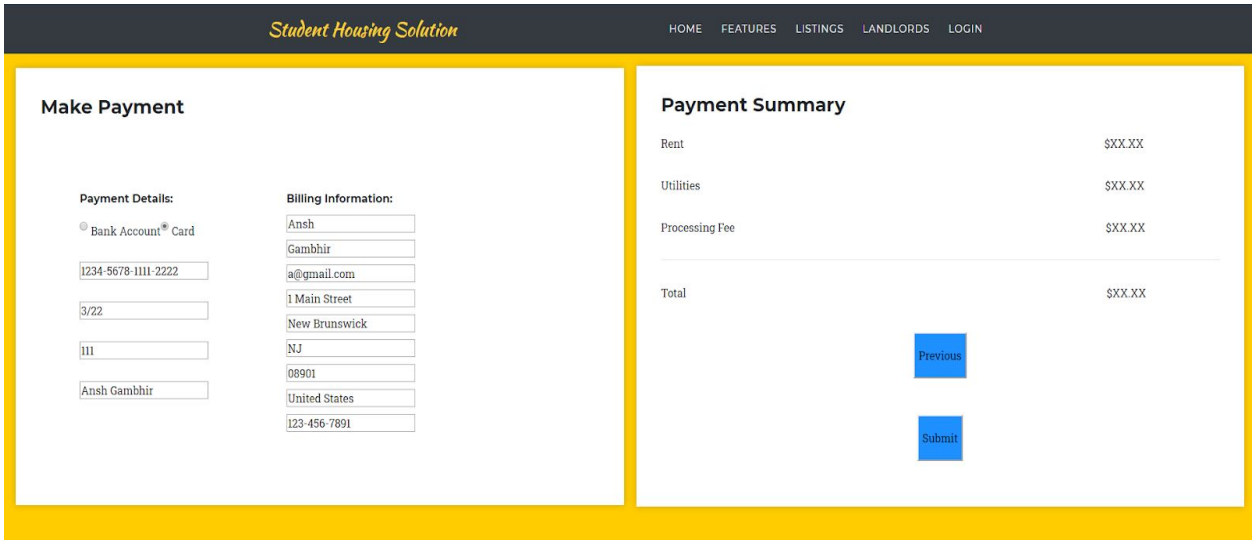


The above mapping diagram displays the navigation path to get to the payment gateway, which contains the payment checkout form.

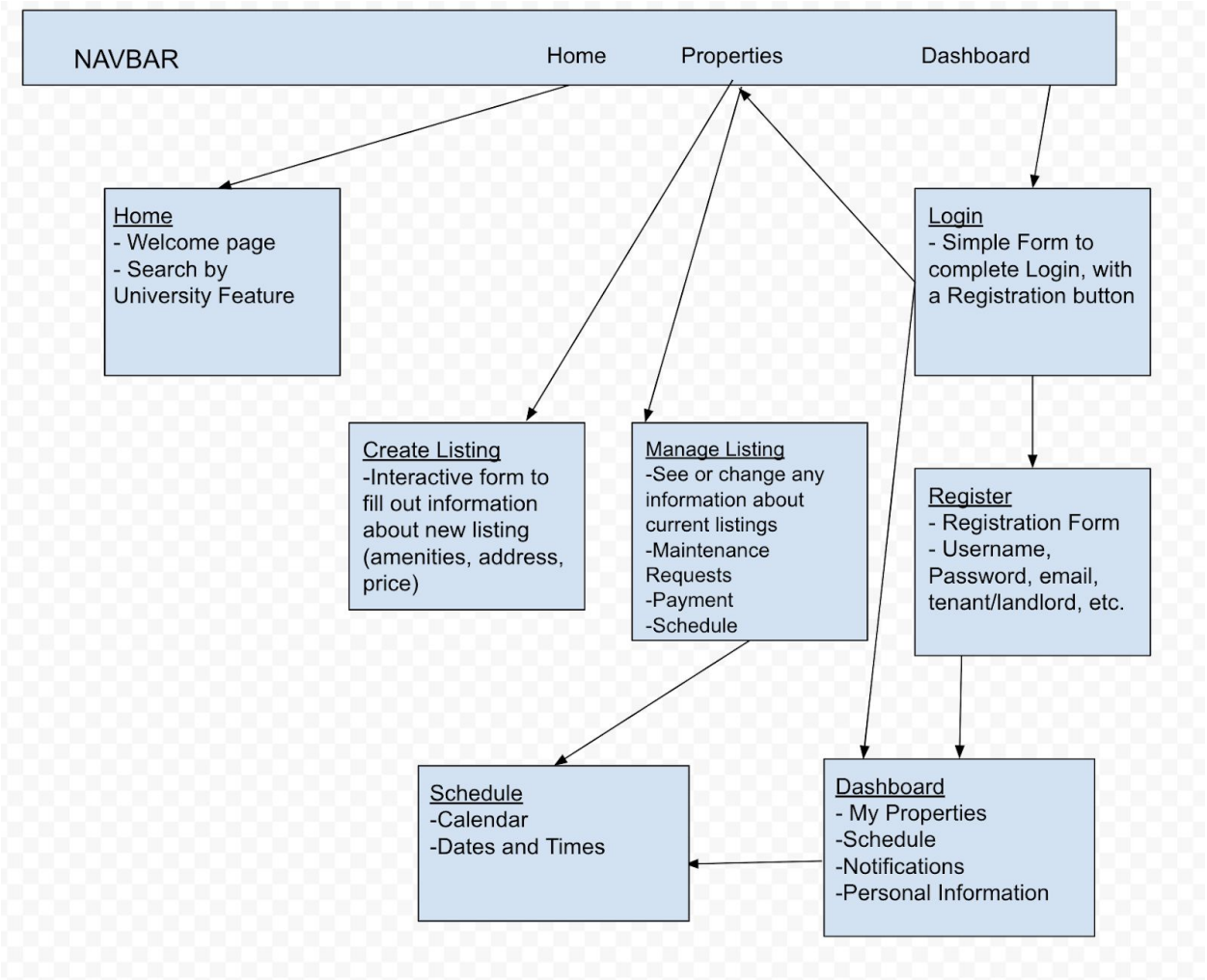
-The minimum user effort to get into the payment form is 3 clicks and 2 user inputs: 2 user inputs to enter username & password, 1 click to login to the website, 1 click to get to the payment page, and one click to get to the payment checkout form.

- The number of keystrokes it takes to get to this page is equivalent to the length of a username and password. For example, if the username was “srinivas” and the password was “nukala”, it would take 14 keystrokes to reach the create listing page.

-On the payment checkout form, to successfully make a payment, 2 clicks and 13 different keystrokes are needed: 1 click to select payment type, either bank account or card, 13 keystrokes to enter all of the necessary information, counting one keystroke per field, and then one more click to submit the information.

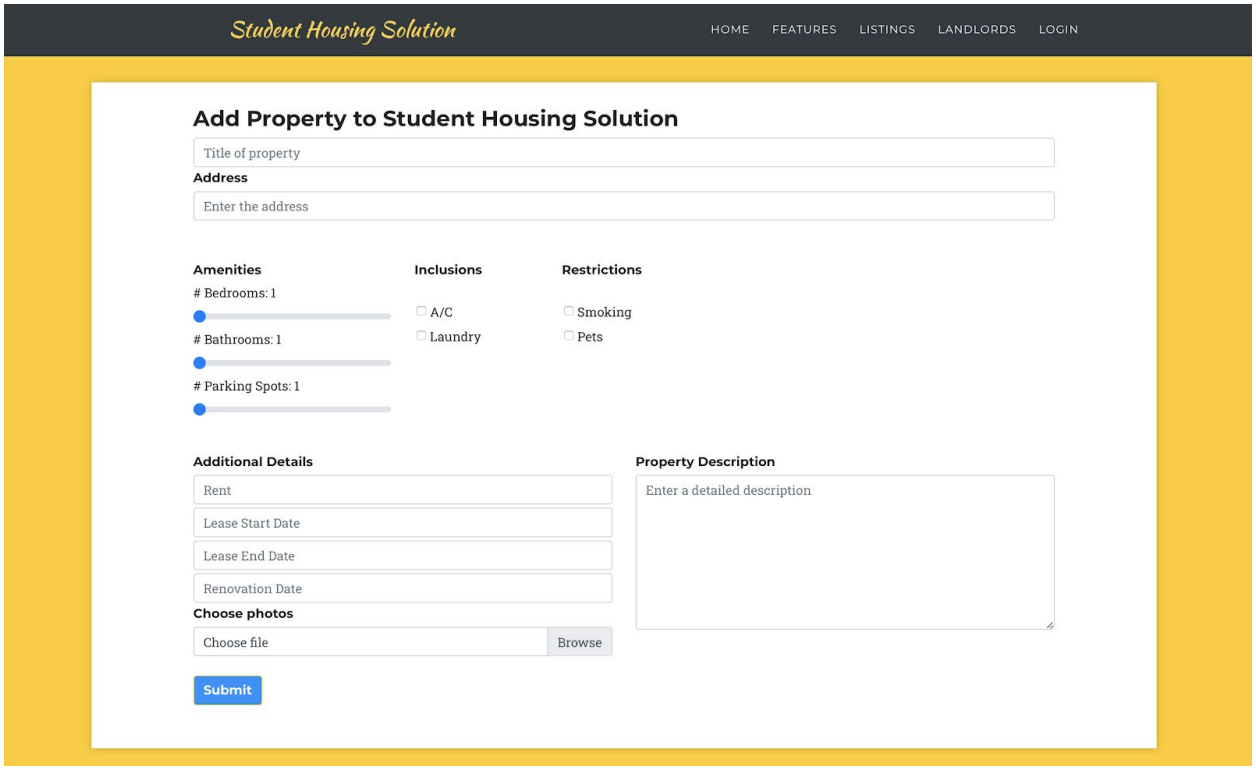


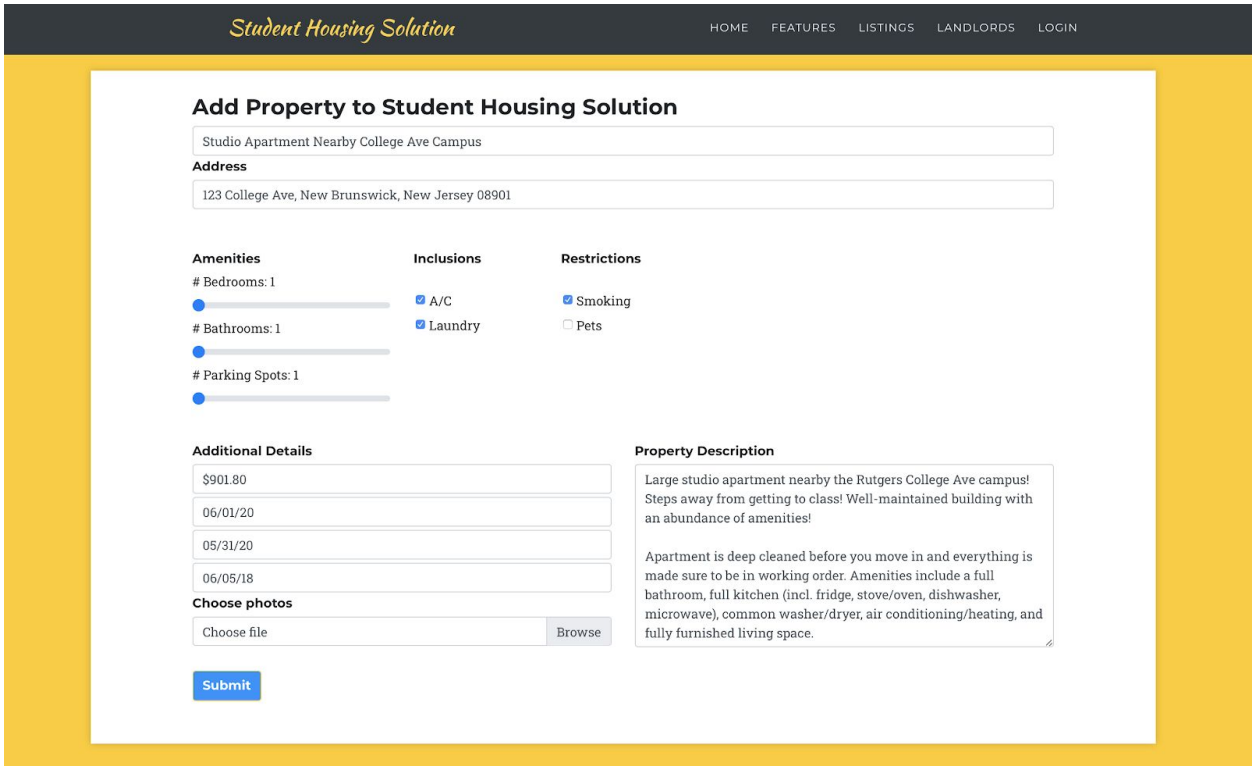
The user interface for the payment checkout form was not included in the first report, but the page was included to showcase in the second report since the whole payment process is going to be demonstrated in the first demo. In order to design this page, we considered all the possible information that we needed in order for a tenant to make a successful payment in our system. We also wanted to include a payment summary so the user can review the amount that they are paying before finalizing the payment, and if they may choose to either go back to the home page or submit, which will initiate the transaction. In order to minimize user effort, the user only has to enter information that is necessary, and although there is a large amount of items to fill out, it is necessary in order for our system to securely make the payment. Also, the user only needs to navigate from their dashboard to a payment page which shows all their upcoming payments, and then from there they can go to the checkout form in order to make the payment.



The picture above shows the navigation path to get to Create Listing and Schedule.

- To get to Create Listing, the minimal user effort required is 3 mouse clicks. (1 for Log in, 1 to go to the properties page on the NavBar, and 1 to hit the create listing button)
    - The number of keystrokes it takes to get to this page is equivalent to the length of a username and password. For example, if the username was “rishi” and the password was “shah”, it would take 9 keystrokes to reach the create listing page.
- To get to the Schedule page, it requires a minimum of 2 mouse clicks (1 for login which takes you to the dashboard, and then 1 more to click the schedule button)
    - The number of keystrokes it takes to get to this page is equivalent to the length of a username and password. For example, if the username was “kyle” and the password was “tran”, it would take 8 keystrokes to reach the schedule page.



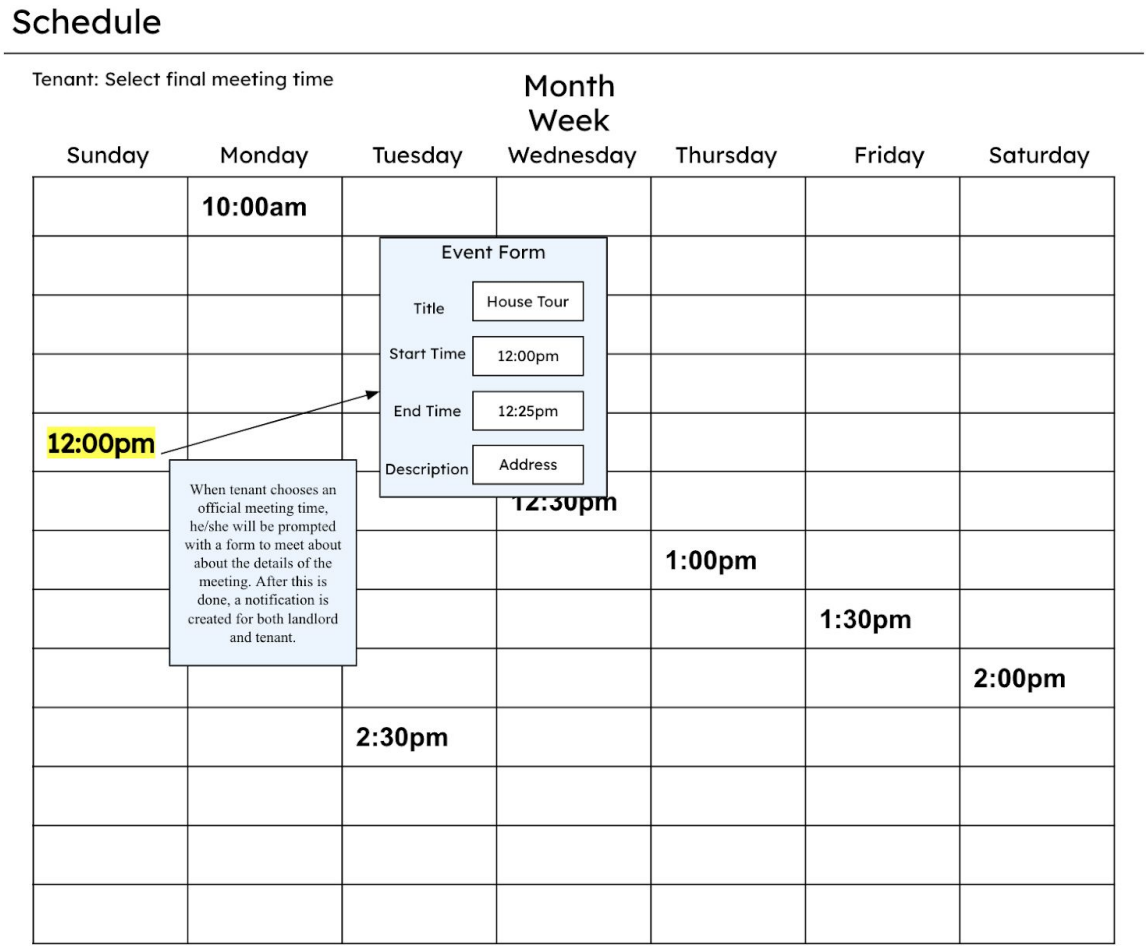


Create Listing UI (Front End)

An example input sequence is shown above. The number of keystrokes in this example is equal to the total number of characters that the landlord typed in.

The UI for the create listing form was not included in the first report. We designed this listing form by considering the different information that had to go into the form and making an input type that it was best suited for. For example, for the types of amenities such as whether or not the house has air conditioning or allows pets, we used check boxes because they will be boolean variables. The address will be autocomplete so as you type in your address, different locations will pop up similar to how the Search Properties part of the website works when you are searching for a college. Under additional details, the rent input box will be set to a monetary input that only accepts numbers and will format it as a price. The lease start, lease end, and renovation date will be formatted as “mm/dd/yy” accordingly. Property description will allow the landlord some flexibility with formatting and will allow the landlord to include anything that he/she believes is important to include about the property. This can include amenities not included above such as complimentary included services such as, internet, trash pickup, and general utilities. When clicking the submit button, the required fields must be checked to make sure they are all filled in, then the property listing can be created and stored in the database.

Landlord View of Schedule



Tenant View of Schedule

The landlord's schedule will have a list of times as displayed in the first picture above. He will select the times where he is available to meet (bolded). This is what the tenant's schedule view will look like. They will receive the times that the landlord selected and choose a final meeting time (highlighted) and fill out the form shown above. An example input sequence is shown above in this form. The number of keystrokes is equal to the total number of characters that the tenant typed in to complete this form.

We did not implement the official UI for ScheduleView yet. We will do that for the second demo. However, we redesigned our schedule UI from the previous report and made a better mockup of it. We added a schedule form that would appear once a final meeting time was decided for the landlord and tenant. The tenant would then fill out information about the meeting such as the title, start and end time, and a description of the meeting. After submission, the time

that was selected should be grayed out and unavailable to be selected again and the landlord should receive a notification alerting them that a new appointment was requested.

Design of Tests

Test Cases

Test Case Identifier: TC-1

Use Case Tested: UC-1 (SearchProperties)

Pass/Fail Criteria:
The test passes if:
- Correct amount of properties are shown when filters are applied
- Properties are shown in correct order when sort criterion is selected
- If no properties are found (too many filters applied), system asks user to expand filters

Input Data: Filter Criteria (# of Bedrooms, # of Bathrooms, Renovation Date, Availability of: Air Conditioning, Parking Spots, Laundry facilities, Allowance of: Pets, Smoking) & Sort Criteria (Price: Low to High, High to Low; Quality Ratings, Most Recent)

Test Procedure	**Expected Results**
Step 1. Enter the desired filter criteria from a list of filters and click submit.	The frontend will receive a list of property listings that match the filter criterion and display them in a list.
Step 2. Click on the desired sorting order from a dropdown menu.	The frontend will rearrange the list of property listings according to the sorting order.
Step 3. Select too many filters/enter very specific criteria from the list of filters and click submit.	The user will be prompted to loosen their filter criteria because no properties could be found under their (too specific) conditions.

Test Case Identifier: TC-2

Use Case Tested: UC-2 ViewInformation and UC-5 CompareListings

Pass/Fail Criteria: The test passes if the user is able to successfully view the property's

information, and compare it to another property. **Input Data:** PropertyID, comparePropertyID (can be null if not comparing listings)	
Test Procedure	**Expected Results**
Step 1. Select a property from search results. Step 2. Select a property to compare to the currently selected property. Step 3. Select another property to compare to the currently selected property.	The frontend will request the full details from the server, which will return those details, which will be displayed to the user. The frontend will request the full details from the server for the second property, and will only display the amenities and their differences, without including descriptions, so the user can see the properties side by side. The frontend will request details about another property, and the property that is currently being compared against will be replaced in the side by side comparison.

Test Case Identifier: TC-3

Use Case Tested: UC-3 MapView

Pass/Fail Criteria:
The test passes if:
- Location markers are pointing to the correct locations
- Updated property markers are shown depending on User Pan/Zoom of the MapView
- Updated property markers are shown depending on User selected filters
- Correct links to detailed property listings are accessible from property markers

Input Data: Latitudes/Longitudes of Properties, Filter/Sort Criteria, User Pan/Zoom Information

Test Procedure	**Expected Results**
Step 1. Pan/Zoom along the MapView in various directions. Step 2. Enter the desired filter criteria from a	The frontend will receive an updated list of property listings that are within the geographical bounds of the MapView and display their location markers. The frontend will receive a list of property

list of filters and click submit.	listings that match the filter criterion and display their location markers on the MapView.
Step 3. Click on any property marker from the MapView.	The user will be taken to the corresponding property listing page where they can view more details on the property.

Test Case Identifier: TC-4

Use Case Tested: UC-4 EmailDigest

Pass/Fail Criteria: The test passes if the user (tenant) is able to successfully sign up for and receive an email digest with updates according to user-set filters, as well as be able to unsubscribe from the digest.

Input Data: User Selected Filters: (# of Bedrooms, # of Bathrooms, Renovation Date, Availability of: Air Conditioning, Parking Spots, Laundry facilities, Allowance of: Pets, Smoking)

Test Procedure	**Expected Results**
Step 1. Request an email digest and apply filters then press submit.	The user will receive an initial email, which has the option to unsubscribe, as well as contains properties which fit their filters.
Step 2. Click the unsubscribe button in the email.	The user should now be unsubscribed, and will not receive anymore emails.
Step 3. Request an email digest without applying filters.	The frontend will alert the user telling them to apply filters in order to receive emails.
Step 4. There are no new properties that match the filters.	There will be no email sent, unless there are properties to share.
Step 5. The user applies for an email digest, but the email address attached to their account is not valid.	The user will not receive any emails, however they will still technically be signed up for said emails.
Step 6. The user changes the email on their account after applying for a digest.	The user will now receive their email digest at this new email address.

Test Case Identifier: TC-5

Use Case Tested: UC-6 Payment

Pass/Fail Criteria:. The test passes if the tenant is able to make a debit card/bank payment using a test card and if the payment is successfully sent to a landlord's test bank account. The test fails if the payment is not received by the landlord account at the end of the process.

Input Data:
Bank Details: Account Number, Routing Number, Type of Account, Name of account holder
Card Details: Card Number, Expiration Date, CCV, Name on Card.
Billing Information: First name, Last name, Address, City, State/Providence, Zip/Postal, Country, Phone Number

Test Procedure	**Expected Results**
Step 1. The user inputs information for a test credit card/ bank account within the checkout page.	The payment system checks to see if the input information is correct and valid depending on the type of payment.
Step 2. The user inputs information for a test credit card/bank account, however the user forgets to input name and address.	Payment is not sent, payment checkout page is reloaded with an error stating missing information at the specific regions.
Step 3. The user sends the payment amount to the payment handler.	Payment information is received and details are visible within the payment dashboard, under the section called "customer".
Step 4. The payment handler sends information to the Stripe API, to validate the payment details.	The API validates the payment, and sends it back to the payment handler for further processing.
Step 5. The approved payment is processed and is sent to the landlord's bank account.	Within the dashboard, under the section called "accounts", the payment transaction details are visible within the respective account, thus showing the approved payment.

Test Case Identifier: TC-6

Use Case Tested: UC-7 MaintenanceRequest

Pass/Fail Criteria: The test passes if the tenant is able to create a maintenance request and if the landlord is able to update the maintenance request and both these changes are saved in the database.

Input Data: Request message, statusDesc, date

Test Procedure	**Expected Results**
Step 1. The tenant creates a maintenance request and fills out the request message with what is wrong with the property.	The front end will send this maintenance request to the database. The landlord gets the notification that a tenant has a maintenance request and is able to update the status of the request.
Step 2. The tenant creates a maintenance request but does not fill out the request message.	The system notifies the tenant that the request has not been filled out properly.
Step 3. The landlord updates the status of the request that the tenant has from “In Progress” to “Resolved”.	The maintenance request’s status ID is updated in the database and the tenant is notified of the change.

Test Case Identifier: TC-7

Use Case Tested: UC-8 Rating

Pass/Fail Criteria:. The test passes if the current tenant’s rating of their respective property is able to be displayed to another user viewing the said property. The test fails if the review is not accessible by other users to view the ratings of a specific property.

Input Data: Cleanliness rating out of 5, Security rating out of 5, Communication rating out of 5, Location rating out of 5, Total rating out of 5, Review description.

Test Procedure	**Expected Results**

Step 1. Current tenants select an option to create a rating of the property they are staying at their tenant homepage.	Review pop up page will appear for the user to input a review.
Step 2. User selects numbers for each category of rating.	When selected, user will be prompt to explain the reason of the rating (the review description)
Step 3. User does not write a description for the review.	The page will ask the user to confirm if they wish to proceed without making a review, or give the option to make one.
Step 4. The user clicks the option to submit the review	Review is submitted within the system, and is stored within the database, along with their propertyID, TenantID. Creates a ReviewID as the review is submitted.
Step 5. Review is published under the property listing.	The review is accessible by other users when looking at the property that the current tenant is staying.

Test Case Identifier: TC-8

Use Case Tested: UC-9 ViewAgreement

Pass/Fail Criteria: The test passes if the landlord is able to upload the lease agreement and if the tenant(s) are able to view the agreement.

Input Data: A file of the lease

Test Procedure	**Expected Results**
Step 1. The landlord uploads the lease file.	The landlord is able to see the file he/she uploaded onto the website.
Step 2. The landlord has not uploaded the lease file.	The system prompts/notifies the landlord that the lease needs to be uploaded.
Step 3. The tenant logs on and goes to his/her	The tenant is able to click on the lease and

dashboard.	view the file.

Test Case Identifier: TC-9

Use Case Tested: UC-10 ScheduleView

Pass/Fail Criteria: The test passes if the landlord is able to choose times where he is available to meet and those times are then sent to the tenant who is able to choose a final meeting time. This selection creates an event and notification that both the landlord and tenant are able to view.

Input Data: Event Title, Start Time, End Time, Description

Test Procedure	**Expected Results**
Step 1: Landlord selects times where he is available to meet.	These times are sent to the database, waiting for the tenant to request them when he wants to choose the final meeting time.
Step 2: Tenant goes to his schedule and receives the list of times that the landlord chose. He selects a final meeting time from those options.	The final meeting time is sent to the database and a form is displayed for the tenant to fill out regarding the meeting.
Step 3. Tenant inputs the event title, start time, end time, and a description of the event.	Upon submission, form data is sent to the database and notification is created and added to the landlord and tenant's list of upcoming events that they are both able to view.

Test Case Identifier: TC-10

Use Case Tested: UC-11 CreateListing

Pass/Fail Criteria: The test passes if the correct information is sent to the database once the form is submitted. The property should be added to the list of the landlord's current properties. A prospective tenant should be able to search for the house.

Input Data: Property Title, Address, Photos, Amenities, Price, and Description	
Test Procedure	**Expected Results**
Step 1. The landlord creates a new listing by filling out the listing form with all the necessary information.	The data is then sent to the database.
Step 2. Upon submission, the landlord checks to see if the property has been added to his current list of properties.	When the landlord views his list of properties, the recently submitted property should be added on to the list.
Step 3. Prospective tenant should be able to search for the house.	The tenant is able to view the property and all the information that the landlord had inputted.

Test Case Identifier: TC-11

Use Case Tested: UC-12 Login

Pass/Fail Criteria: The test passes if the user (tenant or landlord) is able to successfully login with their credentials, and if the user (tenant or landlord) does not enter valid credentials and receives an alert.

Input Data: username, password

Test Procedure	**Expected Results**
Step 1. Type in a valid username and password and press submit.	The front end will receive a login token from the server which is persisted in SessionStorage, and the user will be redirected to the home page.
Step 2. Type in an invalid username and password and press submit.	The front end will display an alert to the user telling them their details are invalid, after not receiving a token from the server.
Step 3. Type in a SQL injection attack type query into the username and password fields to attempt to bypass the authentication and press submit.	The user will receive an alert telling them their details are invalid, as the server will not provide a token to the front end.

Test Case Identifier: TC-12

Use Case Tested: UC-13 Create Account

Pass/Fail Criteria: The test passes if the user (tenant or landlord) is able to successfully create an account by entering valid information, and if the user (tenant or landlord) does not enter valid information, and is prompted to fix their errors.

Input Data: username, password, email, first name, last name, phone number, phone country code, user type (tenant or landlord)

Test Procedure	**Expected Results**
Step 1. Type in valid details (password is correct length and has the correct requirements) and select a user type and click submit	The frontend will send this data to the server, where the user's account will be created and persisted in the database, and the user will be redirected to the homepage with their login token persisted in SessionStorage.
Step 2. Type in invalid details (e.g. password is too short or doesn't have a numeric character, etc.) and click submit.	The frontend will validate this information, and the user will receive an alert telling them which field is invalid, and what is invalid about it.
Step 3. Do not fill out all the fields and click submit.	The frontend will detect this, and the user will receive an alert telling them that they are missing fields, and which fields they are missing.

Test Coverage

All of our test cases are designed to cover all of the use cases we had fledged out in Report One. As we continue development, we will adjust and add more test cases as we see necessary. We are taking the equivalence testing approach, as we are dividing our possible input into groups where the behavior of the application is equivalent, and running tests on each of these groups instead of every possible case in those groups. In addition, we are making sure to have edge coverage in our test cases to make sure that they behave appropriately, but we are not solely focused on the edge cases. This is important so that every part of the control branch is tested, as without it, we are unable to know if that section is fully bug-free. We will make sure that before our code is merged to the master, or production, branch, it will be fully tested and functional. The specificity of our test cases is to ensure coverage for all possible scenarios. Our goal is to ensure a bug-free experience for the user, and to do so, we must be comprehensive in the design of our tests.

Integration Testing

We are tackling the development process by separating into different functional units, decided by the different types of users our site will have. Due to the methodology we used to divide and conquer, it made the most sense for us to use the bottom-up horizontal integration testing technique. We start by combining the units at the bottom of the metaphorical “tree”, or units that do not have any units relying on them. These units are tested first. From there we test units that have reliance from other units at an increasing rate until we reach the units with the most reliance from other units. At the top of our “tree” we have units like the UserController and AuthController, as well as the JwtService class. All three of these have essential pieces that each piece of the application needs in order to authorize requests, as well as view information on a user and decide if they should have permission to access something. At the bottom of the tree would be units such as the MaintenanceRequests classes, as they are only essential to that function, and other functions can operate without them.

More specifically, our testing begins without integration, as explained above, each unit must be thoroughly tested before being merged into the main development branch, and then tested there until it can be merged into production. We start with the small units that are not as mission-critical or reliant on by other units, and make our way to the top. But in the process, as we get closer to the top, we are also indirectly testing its functionality, as the units at the bottom start to become more and more reliant on the top as you go up the “tree”.

Project Management and Plan of Work

Merging the Contributions from Individual Team Members

In order to merge contributions from individual team members we made use of different platforms for different parts of the project. In the case of the report, we use Google Docs, as it allows us to seamlessly edit and work together. We also choose to divide each category of the report up evenly amongst our “sub-teams” (each assigned to a particular function of the application, and the use cases that go along with that). We ensure consistency, as well as uniform formatting and appearance by making use of built-in features for headers/text etc in Google Docs, as well as by keeping in contact through bi-daily stand-up meetings (will be discussed further in next section). We also make sure to allot time in order to proofread and revise the report before submission to address these issues.

In the case of the actual code, we use GitHub for collaboration. This allows us to create branches for each of our sub-issues and once they are resolved, we can merge back into the main. We use a configuration manager setup, where the master branch is locked, and can only be merged to through approved pull requests. This makes sure that our master branch is always fully tested and working, and that we don’t have any hiccups in this process. We maintain uniform formatting in the front-end by discussing standards, and using shared components such as the navigation bar in order to keep a unified look. This is also discussed in our bi-daily stand-up meetings.

Project Coordination and Progress Report

Due to our breakdown of subgroups by use case rather than specialty, our progress has resulted in several fully or partially developed features that can operate independently of other features still being developed.

Completed Use Cases:

UC-12: Login

The entire login use case is now operational: users click on the login button on the navigation bar, enter their username and password, and are redirected to a landing page for their profile. On the backend, we are able to successfully authenticate users in the database.

UC-13: CreateAccount

Users are able to register their account as either a landlord or a tenant, and their registration info, including name, username, password, email, and phone number is stored in the database.

Partially Completed Use Cases (to be done by first demo):

UC- 1: SearchProperties

The SearchProperties use case has largely been handled. On the front end, users are able to select filters and scroll through a list of properties on the main search page. Additionally, database modelling is done. To be completed soon is creating a functional controller and service layer that can interact between the frontend and database.

UC- 7: Maintenance Requests

Several iterations of a front-end interface for Maintenance Requests have been completed, but we are still finalizing what the user will see. We are also continuing to work on the backend of making and viewing these requests from the tenant and landlord sides.

UC- 11: CreateListing

Work has been done to create a user-friendly form for landlords to upload information about their property, then have it persist in the database.

UC- 6: Payment

The backend portion of the payment use case is largely done: using Stripe API, users can enter their credit card info and securely send payment to the controller, which then routes it to the landlord's bank account. Yet to be completed is a functional user interface so that users can send and receive money through our website.

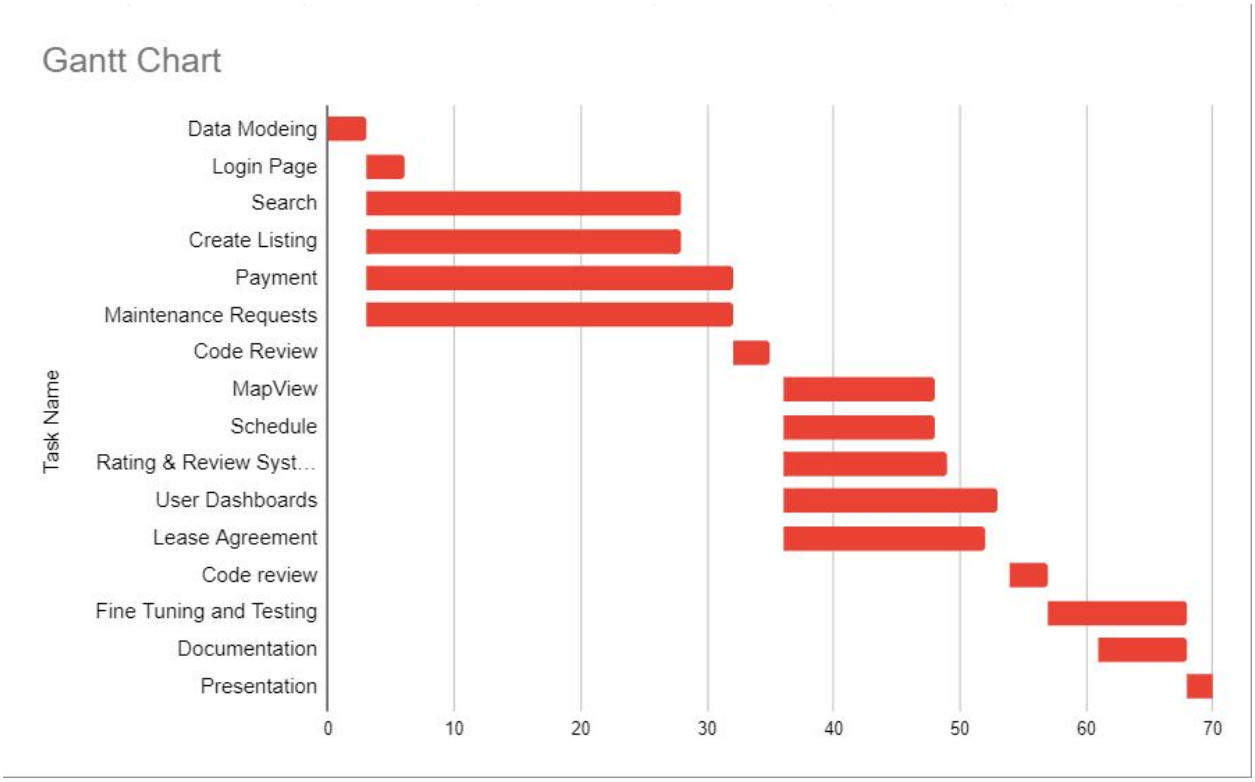
Use Cases Not Yet Implemented:

UC-2: ViewProperty
UC-3: MapView
UC-4: EmailDigest
UC-5: CompareListings
UC-9: ViewAgreement
UC-10: ScheduleView
UC-8: Rating

In addition to implementing these use cases, a significant amount of work has been done in order to set up the project and site in an organized, scalable, and sustainable manner. Michael has taken on the role of configuration manager. With this responsibility, he has created master and development branches in the github repository, and dictated a workflow. When working on new features, team members should check out from the development branch and create a new branch with the feature's name, i.e. Login branch. Once completed and rigorously tested, those smaller updates are merged with development. Michael also undertook learning and implementing the authentication process on the backend. This is critical to ensure that all passwords and kept secure: only single-use tokens are returned from the database on login, not passwords. Rishab has contributed in ways beyond the report by serving as a leader for the front end. With four subgroups each working in unique components, it is important to maintain a unified theme and coding manner. Rishab set up the development environment in ReactJS and chose colors/themes to unify the site.

In order to keep this project well-coordinated we are making use of a Trello board. It allows us to work in a SCRUM-like manner, and we can create cards and move them from column to column. Each card represents a task, and it is assigned to a sub-group, and particularly a person or two. Then that card is placed in the working on column, once someone picks it up and is ready to work on it. When it is completed it is moved to the needs to be tested column, and once tested, it is merged into master branch and moved to the done column.

Plan of Work



	Task	Start	End	Start on Day	End on Day	Duration	Team
Up to First Demo							
	Data Modeling	2/23/2020	2/26/2020	0	3	3	PT
	Login Page	2/26/2020	2/29/2020	3	6	3	PT
	Search	2/26/2020	3/22/2020	3	28	25	PT
	Create Listing	2/26/2020	3/22/2020	3	28	25	LNT
	Payment	2/26/2020	3/26/2020	3	32	29	CT
	Maintenance Requests	2/26/2020	3/26/2020	3	32	29	LWT
	Code Review	3/26/2020	3/29/2020	32	35	3	Everyone
Up to Second Demo							
	MapView	3/30/2020	4/11/2020	36	48	12	PT
	Schedule	3/30/2020	4/11/2020	36	48	12	LNT
	Rating & Review System	3/30/2020	4/12/2020	36	49	13	PT
	User Dashboards	3/30/2020	4/16/2020	36	53	17	Everyone
	Lease Agreement	3/30/2020	4/15/2020	36	52	16	LWT
	Code review	4/17/2020	4/20/2020	54	57	3	Everyone
Up to Final Submission							
	Fine Tuning and Testing	4/20/2020	5/1/2020	57	68	11	Everyone
	Documentation	4/24/2020	5/1/2020	61	68	7	Everyone
	Presentation	5/1/2020	5/7/2020	68	74	6	Everyone

The above is the Gantt chart we created for our project. Our deadlines have slightly shifted since report one as there were complications due to the University closing because of the COVID-19 outbreak. Allocating time for people to get situated, we will still be able to hit all our goals for the first demo, just not along the same timeframes. This did shrink our deadlines for the second demo deliverables, however, we feel that we will still have enough time, we will just have to be more efficient with that time.

Breakdown of Responsibilities

Team Name	Team Description	Team Members
Prospective Tenants (PT)	Tenants that have not yet sourced housing.	Rishab Ravikumar John Yager Michael Giannella
Current Tenants (CT)	Tenants that have already sourced housing through the platform.	Srinivasniranjan Nukala Ansh Gambhir
Landlords without Tenants (LNT)	Landlords that are looking to acquire tenants through the platform.	Rishi Shah Kyle Tran
Landlords with Tenants (LWT)	Landlords that have sourced tenants through the platform.	Ketu Patel Sahil Patel Sneh Shah

Prospective Tenants:
This team is tasked with working on the use cases related to tenants that are looking for housing. These particular use cases include SearchProperties (UC-1), ViewInformation (UC-2), MapView (UC-3), EmailDigest (UC-4), CompareListings (UC-5), Login (UC-12), and CreateAccount (UC-13).

Individually, Michael is working on designing the models, repositories, controller, and service layer for this portion of the application. In addition, he set up the system-wide authentication using JWTs and Swagger for documenting the API of the application. He is also responsible for the back-end setup and integration, as well as designing git workflows for the entirety of the application.

Rishab is designing & building the front end user interface for this portion of the application using ReactJS, and is responsible for the merging of all groups' UI components in ReactJS. In addition, he will be responsible for integrating the MapView into the application using the Google Maps API.

John implemented parts of the frontend using ReactJS. In particular, his work has focused on interacting with the controller and backend via post and get requests. He also provided a working

model for the login and registration pages that successfully authorize users from and adds users to the server.

Current Tenants:
This team is tasked with working on the use cases related to tenants that already have housing through the platform. These use cases include Payment (UC-6) and Rating (UC-8).

Srinivas is responsible for the entire backend operation for the payment handling between tenants and their landlords, by using Java Spring. He is responsible for creating user accounts, handling payment information, and payment transactions between users, by using Stripe API.

Ansh is responsible for creating and designing the payment checkout page using ReactJS and also for testing Stripe account creation and transactions between users. He also is responsible for the front end and back end of the rating system.

Landlords without Tenants:
This team is tasked with working on the use cases related to landlords who haven't found tenants, and are searching for them using our platform. The use cases assigned to this group are ScheduleView (UC-10) and CreateListing (UC-11).

Rishi is responsible for the backend for CreateListing. This includes making the controller for this use case. He has to request data from the landlords such as property address and amenities and make sure they are sent to the database correctly. He will be responsible for the front end of ScheduleView. This will be done with ReactJS.

Kyle is responsible for the front end user interface for the listing form. This involves using CSS and ReactJS to design what the actual form will look like on the website. He will be responsible for the backend of the ScheduleView, which will be done using Java Spring.

Landlords with Tenants:
This team is tasked with working on the use cases related to landlords who have sourced tenants through our platform. The use cases assigned to this group are MaintenanceRequest (UC-7) and ViewAgreement (UC-9).

Sneh is responsible for the backend of MaintenanceRequests using Java Spring. Depending on the status of the user, the user should either be able to create a maintenance request and save it to the database or update the status of the maintenance request and save the changes to the database. She will also be working on the frontend of ViewAgreement for both landlords and tenants using React.

Sahil was tasked with developing the front end for MaintenanceRequests using ReactJS. This includes developing the frontend for MaintenanceDashboard and the landing page for landlords. He will be responsible for the backend of ViewAgreement for both landlord and tenants.

Ketu is responsible for the frontend of MaintenanceRequests for Landlords with Tenants. This included creating a base template using ReactJS for the MaintenanceRequests and MaintenanceDashboard pages. Ketu wIll also be helping out with the backend of ViewAgreement for landlords and tenants.

Integration and Testing

Rishab and Michael are coordinating the integration for the frontend (ReactJS) and backend (Java Spring & SQL Database) portion of the application, respectively. They have both created the initial projects, and imported any libraries necessary for the entire application, as well as established a standard for how the code should be organized. Michael has taken the role of configuration manager, and has set up the GitHub repositories for both parts of the application.

`

Testing is being handled by each group for their respective use-cases, until it is merged to the main development branch, where it will be tested as a whole by all of the sub-groups. The point is to make sure that the features are usable by members of the group who may not have worked on it, and that it is easy to use and understand. At this point, suggestions can be made and applied to the respective sections, and finalized. Once finalized, the changes are merged to the master branch, which takes the role of our production-ready branch in our version control model.

References

Website Title: Introduction - Bootstrap

URL: https://getbootstrap.com/docs/4.4/getting-started/introduction/

Website Title: Creating Spring Boot and React Java Full Stack Application with Maven

URL: https://www.springboottutorial.com/spring-boot-react-full-stack-crud-maven-application

Website Title: Use React and Spring Boot to Build a Simple CRUD App

URL: https://developer.okta.com/blog/2018/07/19/simple-crud-react-and-spring-boot

Website Title: Software Engineering – Domain Modeling

URL:

https://computersciencesource.wordpress.com/2009/11/26/year-2-software-engineering-domain-modelling-2/

Website Title: What is Sequence Diagram?

URL:

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/

Website Title: Complete React (& Redux) Tutorial

URL:

https://www.youtube.com/watch?v=OxIDLw0M-m0&list=PL4cUxeGkcC9ij8CfkAY2RAGb-tmkNwQHG&index=1https://www.youtube.com/watch?v=OxIDLw0M-m0&list=PL4cUxeGkcC9ij8CfkAY2RAGb-tmkNwQHG&index=1

Website Title: Step by Step: Making a Simple CRUD Application Using Java Servlet/JSP

URL:

https://www.mitrais.com/news-updates/step-by-step-making-a-simple-crud-application-using-java-servlet-jsp/

Website Title: How to pass data from child component to its parent in ReactJS?
URL:
https://stackoverflow.com/questions/38394015/how-to-pass-data-from-child-component-to-its-parent-in-reactjs

Website Title: Stripe API Reference
URL:https://stripe.com/docs/api

Website Title: Spring Boot API Reference
URL: https://docs.spring.io/spring-boot/docs/current/api/

Website Title: Create APIs with JWT Authorization using Spring Boot
URL: https://dev.to/cuongld2/create-apis-with-jwt-authorization-using-spring-boot-24f9

Website Title: JPA Many to One mapping
URL: http://www.thejavageek.com/2014/01/15/jpa-many-one-association/

Website Title: Creating Spring Boot and React CRUD Full Stack Application with Maven
URL: https://dzone.com/articles/creating-spring-boot-and-react-crud-full-stack-app