Software development part

The Frontend

Introduction

In the context of the Real Estate Swipe project, the role of frontend development was critical for delivering an accessible, engaging, and user-friendly application interface. The project involved the creation of a React Native application, providing users with a platform to explore and interact with real estate listings in a novel and intuitive way.

The frontend development team, composed of three software engineering students, was responsible for designing and implementing the user interface, ensuring smooth navigation throughout the application, and handling any user interactions. Working collaboratively, the team strived to create an application that was visually appealing, intuitive to navigate, and responsive to user input, thereby providing an optimal user experience.

This report delves into the journey of the frontend development process of the Real Estate Swipe project, including the decision-making processes, the challenges encountered, and the strategies employed to overcome them. The document serves as a comprehensive account of the frontend development process, offering insights into the practical application of software engineering principles and methodologies in a real-world project context.

Technology and Tools

The front-end development of the Real Estate Swipe project was executed using JavaScript and React Native. JavaScript, a high-level programming language, was chosen for its versatility and wide usage in web development. React Native, a JavaScript framework, allowed the team to build a native mobile application that is compatible with both Android and iOS platforms.

The development environment for the project was set up using Visual Studio Code, a popular source code editor that offers features like syntax highlighting, intelligent code completion, and an integrated terminal, among others. It facilitated the writing and debugging of the code for the application.

NodeJS, an open-source, cross-platform, JavaScript runtime environment, was used to execute the JavaScript code outside of a web browser. This was particularly important for running the package manager that was used for handling dependencies in the project.

GitHub, a web-based hosting service for version control, was used for managing changes to the project and facilitating collaboration between the team members. This was particularly crucial due to the international composition of the team, as it allowed for effective coordination and collaboration.

The team used Expo with React Native, which provided a set of tools and services for building, deploying, and quickly iterating on iOS, Android, and web apps from the same JavaScript/TypeScript codebase. It expedited the development process and made it possible to write one project that runs natively on all mentioned platforms.

In terms of project management, the Kanban method was used with Asana. This visual system for managing work as it moves through a process allowed the team to visualize the workflow, limit work-in-progress, and efficiently manage tasks. It enabled the team to adapt quickly to changes, prioritize tasks effectively, and hence contribute to the overall productivity and efficiency of the project.

User Interface Design and Development

The team focused on simplicity and modernity when designing the user interface, drawing inspiration from the intuitive navigation structure of the popular application Tinder.

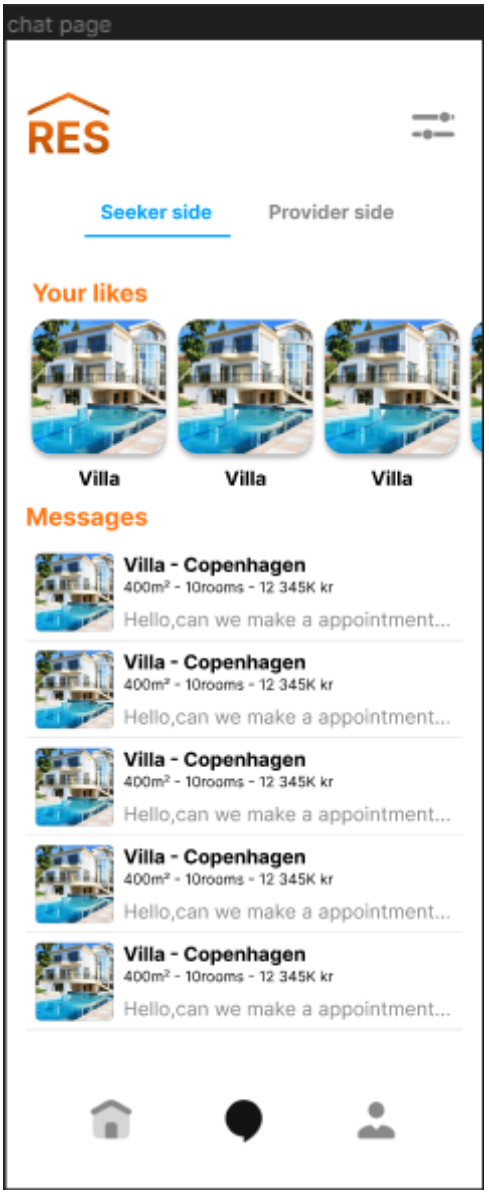
The design process was iterative, starting with a section of the interface, then simulating it on devices to understand its strengths and weaknesses. Feedback from other team members was collected and necessary corrections were made, before moving on to the next design phase. This cycle was repeated several times to ensure an optimal user experience.

The team used Figma, a powerful UI design application, to help with the design and prototyping process. The initial design played a pivotal role in defining the primary navigation structure of the application.

The main interface of the application consists of three pages, accessible via a bottom tab navigation:

- The 'Home' page, where users can browse through a stack of property cards, swiping to interact with them.

- The 'Conversation' page, which lists all the properties the user has 'liked', as well as all the open conversations they have.



- The 'Settings' page, where users can modify their profile, manage the properties they offer, and change various application settings.

The team ensured that the design was responsive and could adapt to different device sizes and orientations by testing the prototype on various devices using Figma. This allowed the team to validate their design decisions and adjust as needed, ensuring a smooth and intuitive user experience.

Developed React Components

During the development of this application, several React components were created to implement various features and interfaces. Here is a description of some of the key components that were developed:

1. **HomeScreen**

   The HomeScreen component is the main screen of the application, which is essentially a stack of property cards that users can swipe left or right to express interest or disinterest.
   At the beginning, it imports necessary modules from React, React Native, and local files. It also defines SCREEN_HEIGHT and SCREEN_WIDTH to store the dimensions of the device screen, which are later used for positioning and sizing the swipe cards.

```
1   import React, { useRef, useState } from "react";
2   import {
3     StyleSheet,
4     Text,
5     View,
6     Image,
7     Dimensions,
8     Animated,
9     PanResponder,
10  } from "react-native";
11  import SwipeCard from "../../components/SwipeCard";
12  import { testProperties } from "../../testProperties";
13
14  const SCREEN_HEIGHT = Dimensions.get("window").height;
15  const SCREEN_WIDTH = Dimensions.get("window").width;
```

   The component uses React's useState and useRef hooks to maintain state and references. pan is a reference to an Animated.ValueXY object which stores the current position of a card being swiped. properties holds the list of properties, liked and passed store the IDs of properties the user has liked or passed on, and rendered is a state variable that holds the properties currently being rendered.

```
17  export default function HomeScreen() {
18    const pan = useRef(new Animated.ValueXY()).current;
19
20    //This has to be a list of all properties ids we will stack on the deck
21    //and fetch datas from the database when we put the id into the rendered list state
22    const properties = useRef(JSON.parse(JSON.stringify(testProperties)));
23    const liked = useRef([]);
24    const passed = useRef([]);
25
26    const [rendered, setRendered] = useState(properties.current.slice(0, 2));
```

Swipe Handlers: Two functions, swipeLeftHandler and swipeRightHandler, are defined to handle swipe actions. They add the current property's ID to the passed or liked array, remove the property from properties, and update rendered with the next set of properties.

```
28  const swipeLeftHandler = () => {
29    passed.current.push(properties.current[0].id);
30    properties.current = properties.current.slice(1);
31    setRendered(properties.current.slice(0, 2));
32  };
33
34  const swipeRightHandler = () => {
35    liked.current.push(properties.current[0].id);
36    properties.current = properties.current.slice(1);
37    setRendered(properties.current.slice(0, 2));
38  };
```

The PanResponder object handles the user's touch events on the cards. It listens for movement and release events and updates the position of the card being swiped with Animated.timing or Animated.spring.

```
39  const panResponder = useRef(
40    PanResponder.create({
41      onStartShouldSetPanResponder: () => true,
42      onPanResponderMove: (event, gesture) => {
43        pan.setValue({ x: gesture.dx, y: gesture.dy });
44      },
```

```
45      onPanResponderRelease: (event, gesture) => {
46        if (gesture.dx > 120) {
47          // Swipe à droite, afficher la carte suivante
48          Animated.timing(pan, {
49            toValue: { x: SCREEN_WIDTH, y: gesture.dy }, // Déplacer la c
50            duration: 180,
51            useNativeDriver: false,
52          }).start(() => {
53            swipeRightHandler();
54            pan.setValue({ x: 0, y: 0 }); // Réinitialise la position de
55          });
56        } else if (gesture.dx < -120) {
57          // Swipe à gauche, afficher la carte suivante
58          Animated.timing(pan, {
59            toValue: { x: -SCREEN_WIDTH, y: gesture.dy }, // Déplacer la
60            duration: 180,
61            useNativeDriver: false,
62          }).start(() => {
63            swipeLeftHandler();
64            pan.setValue({ x: 0, y: 0 }); // Réinitialise la position de
65          });
```

```
      } else {
        // Réinitialiser la position de la carte si le mouvement n'est
        Animated.spring(pan, {
          toValue: { x: 0, y: 0 },
          friction: 4,
          useNativeDriver: false,
        }).start();
      }
    },
  })
).current;
```

In the render method, the component maps over the rendered array and creates SwipeCard components for each property. It also uses the Animated.View component to animate the position of the cards as they are swiped. The cards are styled using the StyleSheet object created at the end of the file.

```
return (
  <View style={styles.screen}>
    {/*<Text>
      {JSON.stringify(passed.current) + JSON.stringify(liked.current)}
  </Text>*/}
    <View style={styles.deck}>
      <Text>{"There is no longer properties to show you... \n Come later to see new ones!"}</Text>
      {rendered
        .map((property, index) => {
          if (index === 0) {
            return (
              <Animated.View
                key={property.id}
                style={[styles.card, styles.currentCard, animatedCardStyles]}
                {...panResponder.panHandlers}>
                <SwipeCard property={property} />
              </Animated.View>
            );
          }
          return (
            <Animated.View
              key={property.id}
              style={[styles.card]}
              {...panResponder.panHandlers}>
              <SwipeCard property={property} />
            </Animated.View>
          );
        }).reverse()}
    </View>
  </View>
);
}
```

In essence, the HomeScreen component is responsible for maintaining the state of the property stack, handling user interactions, and rendering the swipe cards. It uses React Native's Animated and PanResponder APIs to create a smooth and interactive user experience.

2. **ConversationScreen**

The ConversationScreen component is a key part of the application's interface that allows the user to switch between two views: 'Seeker Side' and 'Provider Side'.

- In the 'Seeker Side' view, the component displays a list of properties the user has liked and a list of ongoing chat conversations.

- In the 'Provider Side' view, the component displays a list of properties posted by the user.

The component allows for user interaction through touchable opacity elements that switch the active view when pressed.

- A menu container with two touchable opacity elements for switching the active view.

- A content container, which conditionally renders different elements based on the activeMenu state.

The LikedCard component is used to render each item in the 'liked' properties list.

The ConversationScreen component is a functional component and does not use lifecycle methods. It uses the useState hook to manage its state.

This component interacts with the LikedCard component. It passes each 'liked' property to the LikedCard component as a prop.

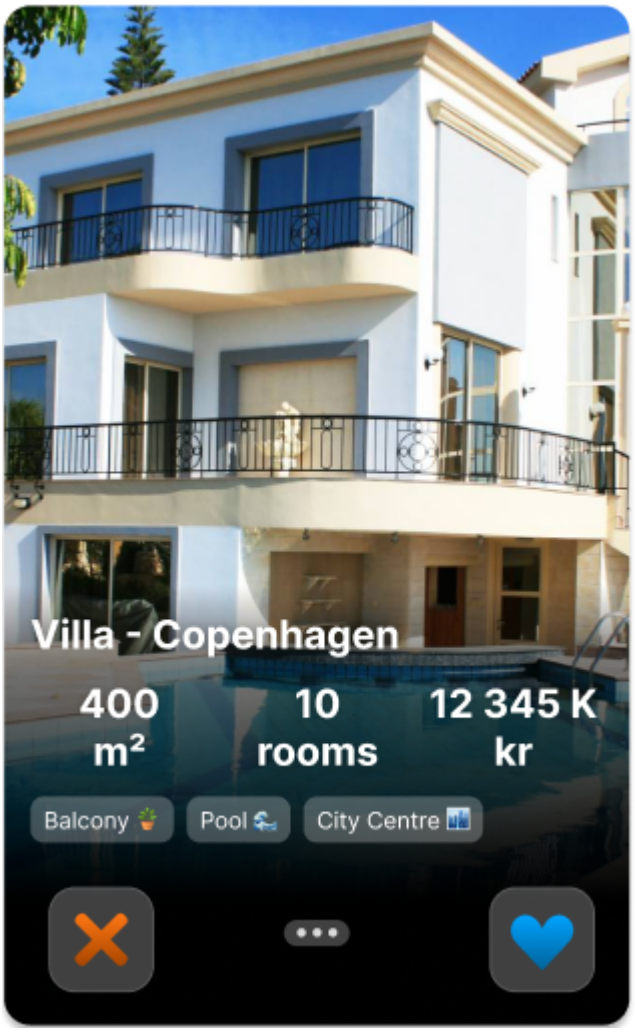
The main challenge in developing this component would likely have been managing the switch between the 'Seeker Side' and 'Provider Side' views. This was effectively handled using conditional rendering based on the activeMenu state.

Another challenge would be fetching the 'liked' properties from the database. For now, this is simulated with a placeholder dataset (testProperties), but in a real-world application, this data would be fetched from a backend database, potentially using an effect hook (useEffect).

3. **SettingsScreen**

   This component allows the user to modify their profile, the properties they offer, and various application settings. It employs stack navigation to navigate between different sub-menus.

4. **SwipeCard**



The SwipeCard component is used to display property information in a card format. The card includes a photo of the property, the property's label, surface area, number of rooms, and price. It provides a visual representation of each property to the end-user in a swipe card format.

This component receives a single prop, property, which is an object containing information about a property. This object includes fields such as photos, label, surface, nbrRoom, price, and isRental. The component does not manage any state variables itself.

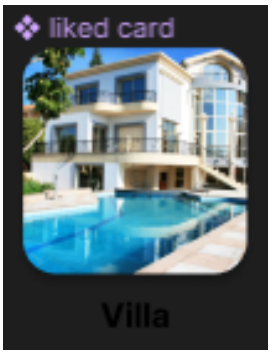
The component structure is straightforward. It has a parent View component, within which an Image component and another View component are nested. The nested View component contains a Text component for displaying the property label, and another View component that displays the property's details (surface, nbrRoom, price). The details are arranged in a row using three View components each containing two Text components.

As a functional component, SwipeCard does not have lifecycle methods. Furthermore, it doesn't use any hooks such as useState or useEffect, as it does not manage state or side effects.

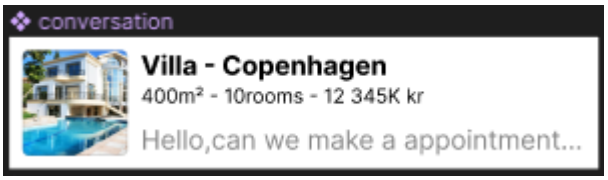
The SwipeCard component is likely used within a parent component that manages a list of properties and handles the swiping behavior. This parent component would pass the individual property object to SwipeCard as a prop.

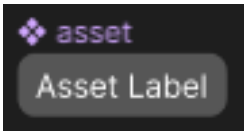
5. **LikedCard**

   A component like SwipeCard but used to display liked properties on the Conversation screen.



In addition to these, many other smaller components were developed to accomplish specific tasks, like buttons, text input, etc. Each component was designed to be reusable, so it could be used in multiple places in the application without code duplication.





Application Navigation

The navigation structure of the application is primarily based on bottom tab navigation, a common pattern in mobile applications that promotes an efficient and intuitive user experience. The application comprises three main pages: Home, Conversation, and Settings, all accessible via the bottom tab navigation.

The Home page is where users can browse through a stack of property cards that they can swipe through. The Conversation page is split into two sections: the Seeker Side, displaying a list of liked properties and ongoing conversations when searching for a property, and the Provider Side, showing conversations as a property provider. The Settings page hosts a standard stack navigation for the various menus, where users can modify their profile, the properties they are offering, and various application settings.

In the Home page, a button for setting search filters opens a bottom overlay, providing a smooth transition without leaving the page. The Conversation page presented a unique navigation challenge. It was resolved with a more traditional approach using conditional rendering in React, creating a custom top bar navigation with two tabs: Seeker Side and Provider Side. This solution, while unconventional, proved effective in achieving the desired navigation flow.

Transition effects between screens are kept to default settings, maintaining simplicity and consistency throughout the application. The absence of custom transitions reduces potential distractions, allowing users to focus on the core functionality of the application.

The team faced some challenges with navigation. Most notably, implementing the top bar navigation in the Conversation page was a significant hurdle. A more traditional approach using conditional rendering with React was adopted to overcome this. This involved changing the displayed content based on which tab (Seeker Side or Provider Side) was active, providing an efficient solution to the problem.

While the project didn't utilize any advanced features of React Navigation, it employed core components to create a seamless and intuitive user experience. The intuitive navigation of the application was ensured through the prototyping phase, where the navigation was tested extensively before the development. Feedback and observations from these tests were crucial in refining the navigation and enhancing its user-friendliness.

Challenges and Solutions

Throughout the development of the application, the team faced several challenges which required thoughtful solutions and sometimes necessitated changes in the approach. Here are some of the significant challenges encountered and the solutions that were employed to overcome them:

1. **Prototyping for a variety of devices:** It was crucial to ensure that the application's user interface is optimized for various devices with different screen sizes and resolutions. The challenge was to create a flexible design that looks good on all devices. This was achieved through extensive testing on various devices during the prototyping phase using Figma. Feedback from these tests was used to refine the design and make it more responsive.

2. **Top bar navigation on the Conversation page:** Implementing a custom top bar navigation with two tabs on the Conversation page was a significant challenge. The team initially struggled to create this using the standard components of React Navigation. The solution was to use conditional rendering in React. This involved dynamically changing the displayed content based on which tab (Seeker Side or Provider Side) was active, providing an efficient and elegant solution to the problem.

3. **Interfacing with React Native and Expo:** As the team was using React Native and Expo for the first time, there was a learning curve involved in understanding these technologies. The team overcame this by investing time in learning these technologies, exploring their documentation, and gaining hands-on experience through development.

4. **Collaboration and Project Management:** Managing a collaborative project with multiple team members working concurrently presented its own challenges. The team used GitHub for version control and adopted the Kanban methodology using Asana to manage tasks and progress effectively. This ensured smooth collaboration, efficient task distribution, and tracking of project progress.

5. **Error handling:** The team encountered an error in React Navigation where the 'component' for a screen could not be found. This was resolved by ensuring that the components were correctly imported, exported, and used throughout the application. If a component was exported as default, it was imported without curly braces, while named exports were imported with curly braces.

Facing and overcoming these challenges not only led to the successful development of the application but also provided the team with valuable learning experiences that improved their problem-solving and development skills.

Testing and Validation

1. **Prototyping and Initial Testing:** The team used Figma for initial prototyping and design testing. The design was simulated on various devices to ensure compatibility and responsiveness. Feedback was collected and incorporated into the design to improve usability.

2. **Integration Testing:** After initial testing, integration tests were carried out to ensure that all components of the application worked together correctly. This helped identify issues that could arise when different parts of the application interacted.

3. **Regression Testing:** After each major update or modification in the application, regression testing was performed. This ensured that the new changes did not break any existing functionality and that the application continued to perform as expected.

4. **Performance Testing:** The application was tested for its performance under various conditions, including different network speeds, device capabilities, and user loads. This helped ensure that the application was robust and could handle real-world conditions.

5. **Navigation Testing:** The team also specifically tested the navigation of the application to ensure it was intuitive and user-friendly. This was particularly crucial for the top bar navigation on the Conversation page, which was one of the unique challenges of the project.

Future Improvements in Testing

1. **Unit Testing:** For future projects, the team acknowledges the importance of conducting unit testing on individual components and functions. This would help catch bugs and issues early in the development process. The tests would be regularly updated as the code evolves.

2. **User Acceptance Testing (UAT):** Also, User Acceptance Testing is another area the team would like to focus on in future projects. This would involve getting the app in the hands of actual users and collecting feedback on its functionality, usability, and overall experience. This feedback would then be used to make further improvements to the application.

The current testing and validation process ensured the quality and reliability of the application and helped provide a seamless experience for the end-users. However, the team realizes that further improvements can be made to their testing processes for future projects.

Conclusion

Designing and developing the user interface and application navigation were key elements in this project. Drawing inspiration from the Tinder app led to a sleek and modern user interface, while using Figma for prototyping streamlined the design and testing process.

Implementing navigation presented challenges, most notably creating a top navigation bar for the conversation page. However, through judicious use of React's conditional rendering, these challenges were overcome to create smooth and intuitive navigation.

In retrospect, it's clear that prototyping and testing the user interface and navigation were critical to the project's success. They allowed issues to be identified and resolved early on, ensuring a high-quality user experience.

However, while functional and validation testing was conducted, it's acknowledged that more extensive testing, such as unit testing and user acceptance testing, could have further ensured the application's quality and reliability.

Overall, this project underscored the importance of careful user interface design and effective navigation in the successful development of mobile applications.

The Backend

The concept of backend development

Backend development refers to the process of building and maintaining the server-side of a web or software application. It focuses on the underlying logic, databases, and infrastructure that enable the user interface (what we usually call the frontend) to function properly.
In a typical web application, the frontend is responsible for presenting the user interface, collecting user input, and displaying information. However, when a user interacts with the frontend, there is a need for data processing, storage, and retrieval. This is where the backend comes into play.
Overall, backend development focuses on building the underlying infrastructure and functionality that supports the frontend of an application. It involves programming, databases, APIs, security, and performance optimization to ensure a robust and efficient system.

Regarding the tools used

In order for the backend of our application to work with its frontend, there were things that first needed to be done.
We first had to choose which programming language would be used to code the backend, and as the person working on this part has had experience with the Django language during his semester in PCL courses, it was decided that he would use it for the backend development.
Django is a high-level Python web framework that simplifies the process of building web applications. It provides a set of tools, libraries, and patterns that enable developers to create robust, scalable, and secured web applications quickly.
Some of the most important features of Django that we made use of are the following :

Object-Relational Mapping (ORM) : Django includes a powerful ORM that abstracts the database layer, allowing developers to interact with databases easily using Python objects and methods rather than writing complex SQL queries. The ORM supports various database backends and provides features like model definition (a model in Django is the equivalent of a database table) and querying.

URL Routing and View Handling : Django provides a URL routing system that maps URLs to corresponding views. Views are Python functions or classes responsible for handling HTTP requests and returning responses to the frontend. The framework offers flexible URL patterns, allowing developers to define complex routing configurations and handle dynamic URLs easily.

Admin Interface : Django offers an admin interface that automatically generates a functional administrative interface for managing application data. With minimal configuration, developers get a user-friendly interface for performing common CRUD (Create, Read, Update, Delete) operations on the application's database, which was used to test the relations and data integrity within the database.

Form Handling and Validation : Django provides a robust form handling system that simplifies form creation and validation. Developers can define forms using Python classes, and Django takes care of rendering, handling user input, and validating the submitted data. They were very useful for information communication with the frontend.

Django's versatility and comprehensive feature set make it suitable for building a variety of web applications. Its emphasis on convention over configuration allows developers to focus on application logic rather than just code, resulting in faster development and increased productivity. Hence their catchphrase : “The web framework for perfectionists with deadlines”.

The setup of The Django project in itself was very simple, as this had already been experienced in PCL courses, but the more tedious parts were the installation of all the dependencies with the fact that a project like this needs to be set up in a virtual environment, as to not cause any damage to potential other projects by updating or overriding already

existing softwares on the computer's system, rendering useless things that would only work on those older versions. After all these troubles were solved, the setup was complete, and is composed of a Django application inside a Django project, making the structure of the project dynamic in case of a restructuring or if the need appears for a new functionality to be added, as any number of applications can be created and linked inside of a project (an application could be anything from a functionality to a whole pan of the project).

Finally, and as the frontend people choose to work with the React Native framework, a need for a means of communication between this framework and Django was created, and what was chosen to fill that gap was the REST framework of Django.
The REST framework, or at least what was used of it in this project, allows us to send JSON responses containing relevant informations from the database to the chosen web pages through the use of components called serializers, the important point being the JSON format, as it is one of the most used data interchange formats, can be comprehended and used by the React Native frontend.

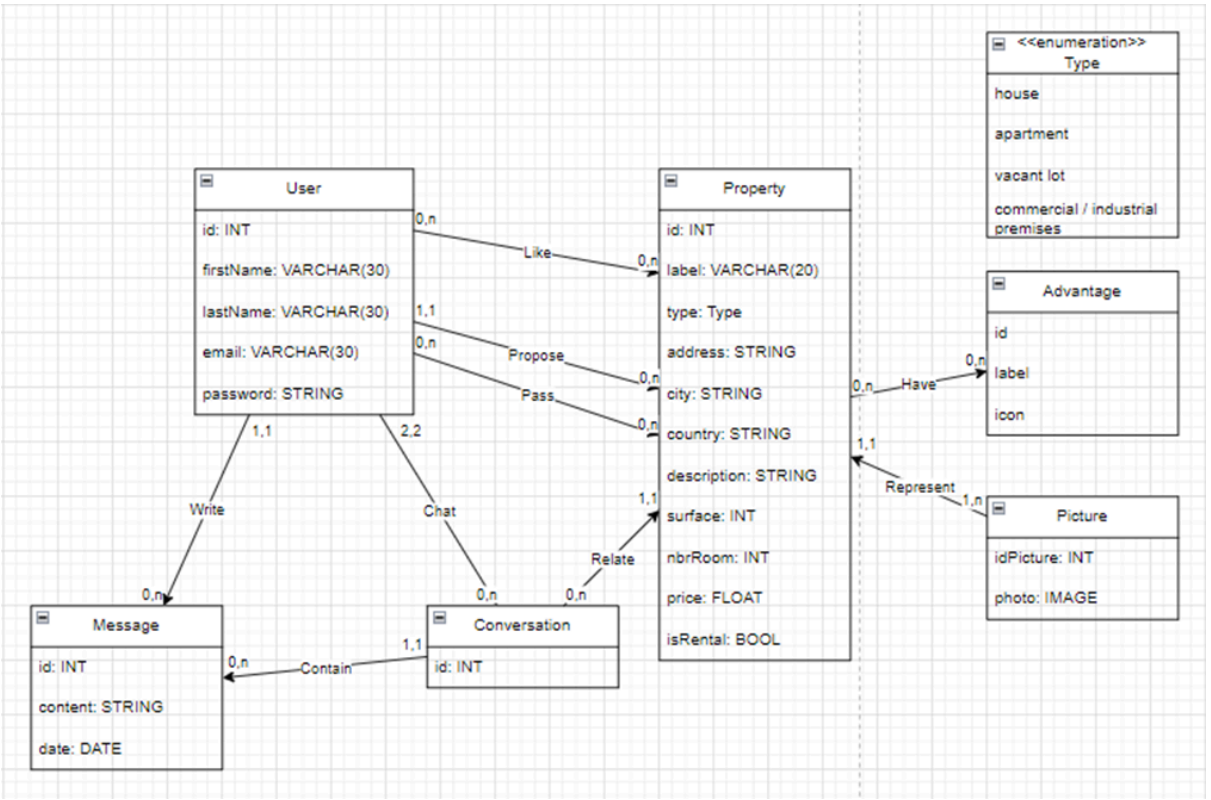
The conception of our database

Analyzing the Project with Merise

Merise, a methodical approach for data modeling and system design, was selected for the analysis phase of this project due to its emphasis on clear, structured system modeling and its wide acceptance in database design. The methodology provided a systematic framework to accurately represent and organize the information requirements of the real estate application, laying the foundation for a robust and efficient backend system.

Conceptual Data Model (MCD)

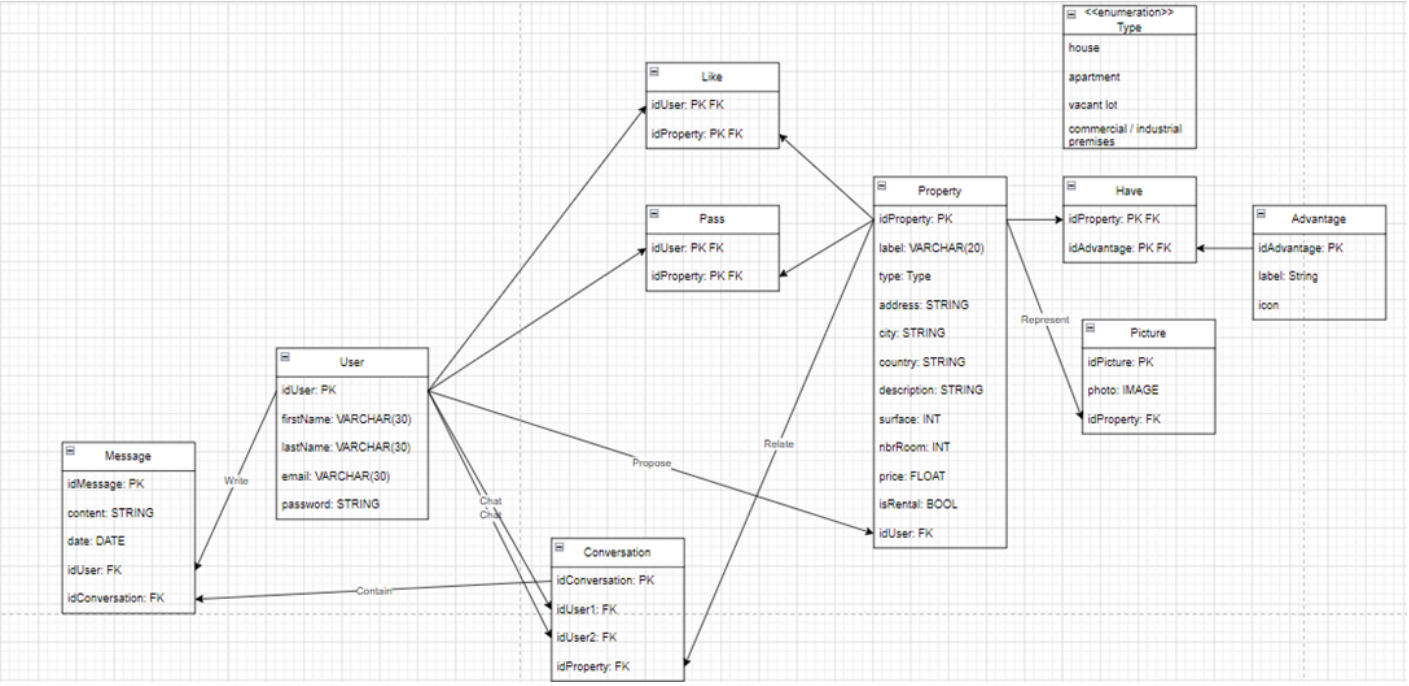
The creation of the Conceptual Data Model (MCD) was a critical step in understanding the data requirements of the application. Entities were identified representing key components of the real estate platform such as User, Property, Conversation, and Messages. Attributes for each entity were carefully considered to capture necessary details, while relationships between the entities were established to reflect the logical connections within the application.



The MCD served as a visual representation of the data structure and relationships, aiding in the comprehension of the application's data requirements, and facilitating communication among the team members. It was a critical tool in designing a database that would effectively support the application's functionality.

Physical Data Model (MPD)

Transitioning from the MCD to the Physical Data Model (MPD) involved refining the entity-relationship diagram to align with the specific constraints and capabilities of the chosen database system. Changes were made to conform to the physical database design principles, including normalization rules, data types, and index considerations.



The MPD served as the blueprint for the actual database implementation, providing a detailed and tangible plan to guide the database development process. The careful creation of the MPD was instrumental in ensuring a smooth transition from the design phase to the implementation phase of the database system.

The different parts of the Real Estate Swipe application

The models

In a Django project, models represent the data structure and define the logical structure of the application's data. Models define the entities or objects in the application and their relationships with each other. They serve as an intermediary between the application and the database, allowing developers to interact with the database using Python code.
Here are the main aspects of the Django models :

- In Django, a model class is a Python class that represents a database table. Each attribute of the class represents a field in the table. Django provides a rich set of field types such as IntegerField, CharField, DateTimeField, and ForeignKey, among others, to define the characteristics of the fields, like their data type, size, and constraints.
- Field types are used to handle different types of data, including integers, text, dates, booleans and more, as said earlier. These field types provide built-in validation and data conversion, ensuring that the data stored in the database complies with the defined rules and constraints.
- Models in Django can establish relationships with other models. The commonly used relationship types are one-to-one (OneToOneField) to define a one-to-one relationship between two models, one-to-many (ForeignKey) to establishes a one-to-many relationship between two models, and many-to-many (ManyToManyField), to represent a many-to-many relationship between two models.
- Model classes can define methods that perform operations specific to the model. These methods can be used to retrieve related objects, calculate derived values, perform data transformations, implement custom business logic, or can be overridden to change their purpose.
- Django includes a powerful migration system that allows developers to manage changes to the database schema over time. Migrations keep track of the modifications made to models and generate the necessary SQL commands to update the database schema. This ensures that the database structure stays in true to the model definitions.

A model from our application

```
# The model class definition
class Property(models.Model):

    # A string field
    label = models.CharField(max_length=20,null=False)

    # Definition of a choice field
    class TypeChoice(models.TextChoices):
        HOUSE = 'HO', ('House')
        APPARTMENT = 'AP', ('Appartment')
        VACANTLOT = 'VL', ('Vacant Lot')
        COMMINDUS = 'CI', ('Commercial / Industrial Premises')
    # Usage of said choice field
    type = models.CharField(max_length=2, choices=TypeChoice.choices,
default=TypeChoice.APPARTMENT)

    # String fields of different maximum lengths
    address = models.CharField(max_length=100,null=False)
    city = models.CharField(max_length=50,null=False)
    country = models.CharField(max_length=50,null=False)
    description = models.CharField(max_length=500,null=False)

    # Integer fields
    surface = models.IntegerField(null=False)
    nbrRoom = models.IntegerField(null=False)

    # Float field (can contain non-natural numbers)
    price = models.FloatField(null=False)

    # Boolean field (contains either True or False)
    isRental = models.BooleanField(default=True)

    # ManyToMany field to be able to stock many instances of the
"Avantage" model
    advantages = models.ManyToManyField(Advantage)

    # Overriding a model method to change its output
    def __str__(self):
        return self.name
```

As can be seen when looking at both this model and the schematic relational database presented earlier, this model is the implementation of the Property table, converted in Python

code. This same conversion has of course been done for all the other tables of the database.

Models play a crucial role in Django projects as they define the application's data structure and handle interactions with the database. By using models, developers can work with the application's data using Python code, without directly dealing with complex SQL queries or database operations.
Additionally, Django's models contribute to the framework's other components, such as the ORM, admin interface, forms, and serializers, by providing a consistent representation of data across the different parts of the application.

The views

In Django, views play a crucial role in handling HTTP requests and generating HTTP responses. They are responsible for processing user input, interacting with models and databases, and rendering appropriate templates or returning data in various formats. Views bridge the gap between the frontend and backend of a Django application.

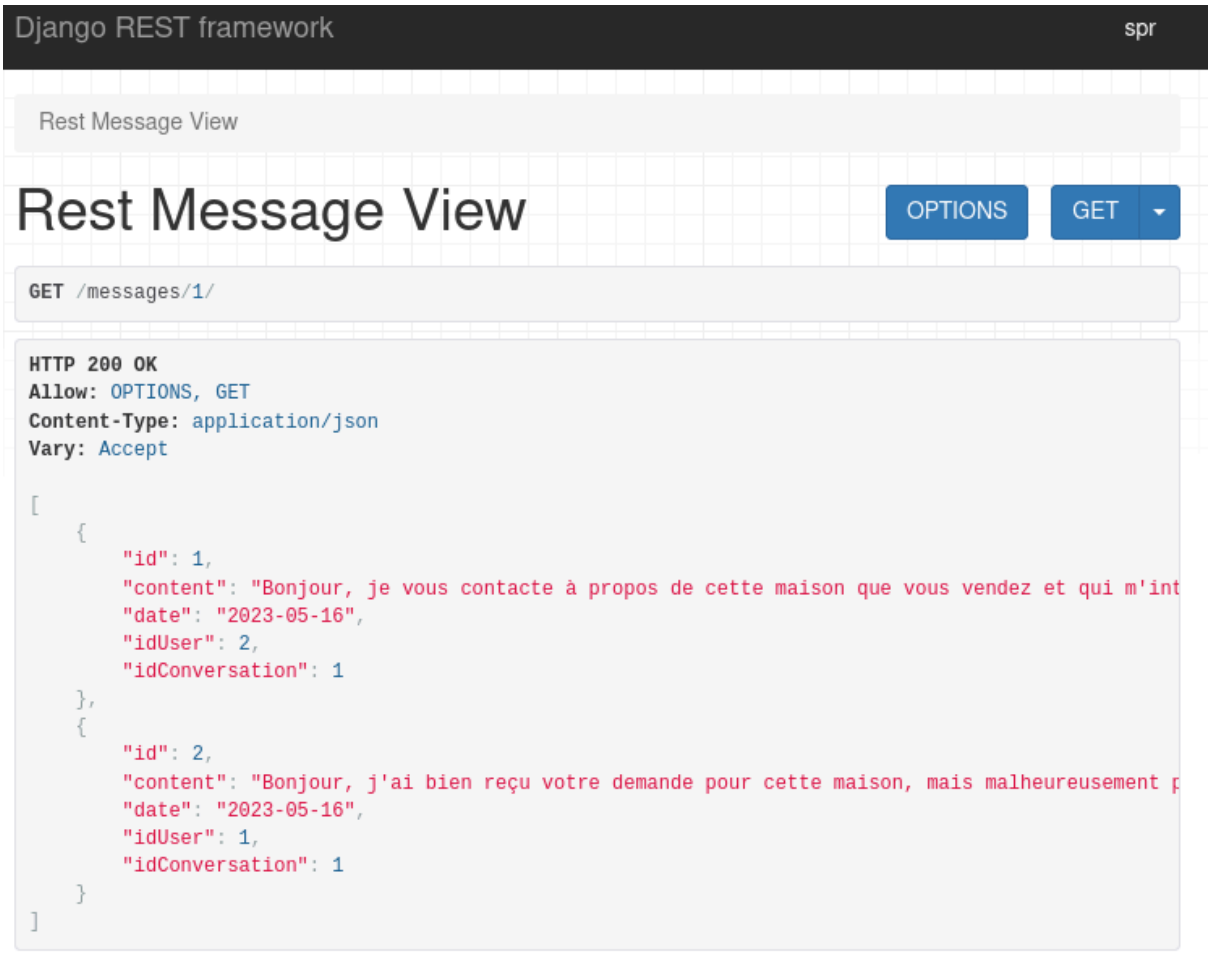
In our application, we have several views used to, as said previously, link the different parts of the backend and interact with the frontend. These interactions consist of receiving HTTP requests from the user interface, and sending back the appropriate HTTP responses.
In our case, these responses contain mainly the data asked for by the frontend after being transformed in the JSON format by the serializers.
The following code shows how one of the views from our application works in order to deliver the appropriate response to the frontend.

A view from our application, returning message informations

```
# Allows the REST framework to return a special view when retrieving
information from the models
@api_view(['GET', ])

# Definition of the view, idConv is the id of the Conversation model
def rest_message_view(request,idConv):

    # Trying to get messages that appear in a certain conversation
    try:
        message = Message.objects.all().filter(idConversation=idConv)
    # If there are no such messages
    except Message.DoesNotExist:
        # Then return a 404 error to the frontend
        return Response(status=status.HTTP_404_NOT_FOUND)

    # Send the messages found to a specific serializer to transform the
data into the JSON format
    serializer = MessageSerializer(message,many=True)

    # Send the transformed data from the serializer to the frontend
    return Response(serializer.data)
```

The type of response that can be sent by this view corresponds to the following screenshot.

A JSON response returned by the Message view



It can be seen in this screenshot of the server-side website, that there are two objects from the Message model that were returned when asking for the messages of the Conversation number one (it is also a model). These instructions were sent by the user interface by going into the /messages/1/ window of the application. This format of data is the JSON format that was needed for the frontend to work with.
However, note that this screenshot is not representative of our application, as it is only a built-in view of the REST framework made to easily look into the data of the HTTP response.

The routes

In Django, routes, also known as URL routing or URL configuration, define the mapping between URLs and view functions. Routes play a crucial role in determining which view should handle a particular HTTP request. This allows for easy navigation, proper handling of HTTP requests, and the ability to build dynamic and flexible web applications.

For our project, we have at our disposal two different urls.py files with the routes of our project : one in the Django project, and one in the Django application. The first one is created by default with the project, and is the one that Django automatically looks up to when he needs to load a view. To remedy that (because what we wanted was to stop the routes from the project to interfere with the views of the application), we created the second urls.py file inside the files of the application (in order for it to manage his own views), and referenced it in the original routes file. This has given us the following files :

Original urls.py file from the Django project

```
# The list which contains the routes of the Django project
urlpatterns = [

    # Route redirecting to the application
    path('', include('RealEstateSwipe_APP.urls')),

    # Route redirecting to the built-in Django Admin view
    path('admin/', admin.site.urls),
]
```

Example routes from urls.py file from the Django application

```
# The list which contains the routes of the Django application
urlpatterns = [
   # Path returning to the view with the JSON responses of the Message
model; idConv is the id of the conversation that we want to respond
    path('messages/<idConv>/', rest_message_view,
name="messages_rest"),

    # Path returning to the view with the JSON responses of the Picture
model; idProp is the id of the property from which we want the pictures
    path('pictures/<idProp>/', rest_picture_view, name="picture_rest"),
]
```

The serializers and forms

In Django, serializers and forms are components used to handle data input and output in web applications. They provide a convenient way to validate, transform, and render data for various purposes, such as processing user input or serializing data for APIs.

The serializers

Serializers in Django are used to convert complex data types, such as Django model instances, into a format that can be easily rendered into JSON (as needed for the link between Django and React Native), XML, or other content types.

Serializers provide different mechanisms to, among other things :

- Validate data based on the specified fields and their constraints
- Handle converting complex data types into simpler representations, allowing the serialization and deserialization of data exchanged between the client and the server when used through Django's views (this is the main reason as to why we used serializers in our application)

A serializer from our application

```
# Serializer definition
class MessageSerializer(serializers.ModelSerializer):
    class Meta:

        # Telling REST to fetch data from the "Message" model
        model = Message

        # Telling REST which fields of the model can be fetched
        fields = '__all__'
```

This serializer allows, when called, to return the simplified data we want from the Message model, as you have seen already in the views part.

The forms

Forms in Django provide a convenient way to handle user input and validate it before processing. They encapsulate fields and their associated validation rules, rendering them as forms for user interaction.

Their main purpose is to make it easier to create and process user-submitted data, and the key features allowing that are the following :

- They validate input data based on defined rules and constraints, ensuring the submitted data is valid.
- Forms handle populating fields with initial data and binding user-submitted data back to the form object.
- They handle error messages and provide convenient methods to display errors alongside form fields.

A form from our application, allowing user to register an account in the “User” model

```
# Form definition
class UserForm(forms.ModelForm):
    class Meta:

        # Telling Django in which model there will be one more instance
        model = User

        # Telling Django which fields will be inputted so that it can
verify if rules and constraints are respected
        fields = '__all__'
```

There are, however, no examples of data being sent from the user interface to the backend, as it is both impossible to visualize and still not working in our application by the time this is written.

Django forms are versatile and widely used for various purposes, including user registration, data input, search forms, and more. They provide an abstraction layer for handling user input and data validation, making it easier to maintain clean and secure data in web applications.

Sources

Meta Platforms, 2023. Setting up the development environment. *React Native documentation* [online] Available at: https://reactnative.dev/docs/environment-setup

Expo, 2023. *Expo documentation*, [online] Available at: https://docs.expo.dev/

React Navigation, 2023. Bottom Tabs Navigator. *React Navigation documentation*, [online] Available at: https://reactnavigation.org/docs/bottom-tab-navigator/

React Navigation, 2023. *React Navigation documentation*, [online] Available at: https://reactnavigation.org/docs/getting-started

Florian Marcu, 2021. How to Build React Native Swipe Cards Inspired by Tinder. *Instamobile*, [e-journal] Available through: https://instamobile.io/react-native-controls/react-native-swipe-cards-tinder/

Meta Platforms, 2023. Animations. *React Native documentation* [online] Available at: https://reactnative.dev/docs/animations/

Hassan, A., 2017. Creating React Native apps with Django rest-api. *Medium*, [e-journal] Available through : https://medium.com/@hassanabid/creating-react-native-apps-with-django-rest-api-59e8417865e9 [Accessed 21 April 2023]

Nakul, S., 2020. React Native and Django for Beginners. *Crowdbotics*, [e-journal] Available through : https://www.crowdbotics.com/blog/react-native-django-for-beginners [Accessed 21 April 2023]

Meta Open Source, 2023. *Integration with Existing Apps*. [online] Available at : https://reactnative.dev/docs/integration-with-existing-apps [Accessed 22 April 2023]

Django Software Foundation, 2023. *Django Documentation*. [online] Available at : https://docs.djangoproject.com/en/4.2/ [Accessed 18 April 2023]

Encode OSS Ltd., 2011. *Django REST framework*. [online] Available at : https://www.django-rest-framework.org [Accessed 7 May 2023]

Encode OSS Ltd., 2011. *Serializers*. [online] Available at : https://www.django-rest-framework.org/api-guide/serializers/#serializers [Accessed 7 May 2023]

Courage, Z., 2022. Django Imagefield: How to Upload Images in Django. *CodingGear*, [e-journal] Available through : https://codinggear.blog/how-to-upload-images-in-django/ [Accessed 13 May 2023]

Django Software Foundation, 2023. *Working with forms*. [online] Available at : https://docs.djangoproject.com/en/4.2/topics/forms/ [Accessed 18 May 2023]