



UNIVERSIDAD DE CHILE  
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS  
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

## **DESARROLLO DE UN SISTEMA DE GESTIÓN DE INMUEBLES**

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

IVÁN AARON LARRAÍN MUÑOZ

PROFESOR GUÍA:  
SERGIO OCHOA DE LORENZI

PROFESOR CO-GUÍA:  
IGNACIO VALLEJOS QUINSACARA

MIEMBROS DE LA COMISIÓN:  
ÉRIC TANTER  
HUGO MORA RIQUELME

SANTIAGO DE CHILE  
2024

## Resumen

Este trabajo de memoria se realizó en el marco de un proyecto para la empresa Efirent, la cual está dedicada a la administración de inmuebles para sus clientes, encargándose de la interacción con los arrendatarios y el manejo de los cobros y pagos de arriendo mes a mes.

Uno de los procesos críticos de Efirent involucra cuadrar manualmente los pagos recibidos. Esta actividad presenta actualmente una gran limitación, puesto que luego de pagar, los arrendatarios deben enviar el comprobante de pago por algún canal de comunicación (por ej., por email). Luego, un ejecutivo de Efirent debe revisar los comprobantes y cuadrarlos en las hojas de cálculo manualmente, lo que ocupa una gran parte del tiempo disponible para trabajo, especialmente en periodos críticos, como lo son el fin y principio de cada mes.

Para abordar el problema antes planteado, en esta memoria se desarrolló un sistema que permite el ingreso de los datos de las propiedades administradas, y sus contratos de arriendo, generando automáticamente los cobros correspondientes mes a mes. Además, el sistema permite a los arrendatarios realizar pagos mediante *Webpay* o de manera manual, pero asociando automáticamente el comprobante correcto al contrato correspondiente.

Este sistema fue evaluado por usuarios expertos (empleados de Efirent) quienes analizaron el proyecto en base a su utilidad percibida para resolver el problema de cuadrar pagos. Estos usuarios consideraron el desafío de usar el proceso actual, y también dificultad para adaptarse a un nuevo sistema. Para esta comparación se consideraron los tiempos dedicados al proceso de cuadro de pagos en el sistema actual, y su equivalente con el nuevo software. También fue evaluado por un usuario externo que representó al perfil de usuario de un arrendatario ajeno al funcionamiento del negocio.

Los resultados de las evaluaciones, aunque son preliminares, fueron satisfactorios, por lo que se puede decir que los objetivos del proyecto a priori se cumplieron. Sin embargo, aún queda bastante espacio para mejorar, sobre todo en la usabilidad de las interfaces y las funcionalidades ofrecidas para la recopilación de datos.

Como parte del trabajo a futuro también podemos mencionar que se requiere mejorar el acceso a la información cruzada en la plataforma. Además, se puede mejorar la claridad y redacción de los textos e instrucciones presentados, tanto a usuarios de administración como a los arrendatarios, permitiendo así disminuir la barrera de entrada para el uso de la aplicación.

*A mi familia,  
y a mis amigos.*

## Tabla de contenido

1. Introducción	1
1.1. Justificación	2
1.2. Objetivos	3
1.2.1. Objetivo general	3
1.2.2. Objetivos específicos	4
1.3. Solución propuesta	4
1.4. Estrategia de evaluación de la solución	7
2. Marco teórico	8
2.1. Aplicaciones de gestión de inmuebles	8
2.1.1. ComunidadFeliz	8
2.1.2. Inmogesco	8
2.1.3. Breal	8
2.1.4. Habitatsoft	9
2.1.5. Skualo	9
2.2. Análisis de Espacio de Datos Abordado	9
2.3. Justificación del desarrollo de una nueva plataforma	11
3. Concepción de la Solución	13
3.1. Requisitos de la plataforma	13
3.1.1. Gestión de unidades arrendadas	13
3.1.2. Gestión de contratos	13
3.1.3. Gestión de pagos	14
3.2. Perfiles de usuario soportados	15
3.3. Arquitectura de la Solución	15
3.4. Modelo de datos	17
3.5. Procedimiento de cálculo de los cobros	19
3.6. Diseño de interfaces de usuario	20
3.7. Tecnologías escogidas para la implementación	20
3.7.1. Backend	20
3.7.2. Frontend	21
3.7.3. Entorno de desarrollo	22
4. Implementación de la Plataforma	23
4.1. Portal de Administración	23
4.1.1. Estructura del proyecto	23
4.1.2. Layouts de la aplicación	24
4.1.3. Acceso a los CRUDs	26
4.1.4. Contratos	36
4.1.4.1. Lista de contratos	36
4.1.4.2. Formulario de contratos	38
4.1.4.3. Administración de contratos vigentes	44
4.2. Portal de Arrendatarios	46

4.2.1. Búsqueda por RUT	47
4.3. Backend de la plataforma	51
4.3.1. Estructura de cada app	51
4.3.2. Aplicación base y utils	53
4.3.3. Aplicación de propiedades	53
4.3.2. Aplicación de contratos	56
4.3.3. Aplicación de monedas	61
4.3.4. Manejo de usuarios	61
4.4. Descripción de la API	62
4.4.1. Aplicación de propiedades	62
4.4.2. Aplicación de contratos	64
4.4.3. Usuarios	65
4.4.4. Aplicación de monedas	65
5. Evaluación de la solución	67
5.1. Usuarios expertos	67
5.1.1. Ejercicio realizado	67
5.1.2. Evaluación de usabilidad	67
5.1.3. Evaluación de la utilidad percibida	70
5.1.4. Comparación de tiempos	72
5.1.5. Opiniones de los usuarios	73
5.2. Usuario externo	74
5.2.1. Ejercicio realizado	74
5.2.2. Evaluación de usabilidad	74
5.2.3. Opinión del usuario	75
6. Conclusiones y trabajo a futuro	77
Bibliografía	79
Anexos	80
Anexo A: Mockups de Interfaces de Usuario	80
A.1. Diseño de interfaces de usuario	80
Anexo B: Schema de datos	84

# 1. Introducción

Esta propuesta de memoria se enmarca en la operatoria de la empresa *Efirent*<sup>1</sup>, la cual opera en el ecosistema de empresas inmobiliarias chilenas. Ésta se desempeña como una administradora de propiedades para arriendo, ofreciendo el servicio tanto de corretaje (encontrar un arrendatario para una vivienda) como de administración de inmuebles. El cliente principal de este negocio es entonces la persona dueña de la vivienda, conocida como “cliente” o “inversionista”.

El servicio de corretaje es un proceso que comienza cuando un cliente registra su propiedad con Efirent. En ese momento la empresa pasa a publicar la propiedad en diversos portales inmobiliarios y otras plataformas, para así conseguir arrendatarios para el inmueble. Durante ese proceso se deben programar visitas con potenciales inquilinos, y luego de que se llegue a un acuerdo, se debe manejar la generación de contratos de arriendo y confirmación de la firma de estos.

Una vez firmado el contrato, el cliente puede decidir si terminar el proceso, encargándose personalmente de la administración del inmueble durante el periodo de arriendo, u optar por delegar esta tarea a Efirent.

El servicio de administración de la propiedad incluye la notificación a arrendatarios de los pagos mensuales, tanto de arriendo como de otros gastos (gastos comunes, utilidades, garantías en cuotas, etc.), la recepción y confirmación de estos (si es que son realizados), la recepción de solicitudes a post-venta del inmueble o de mantenimiento, las inspecciones periódicas que correspondan a la propiedad, y la generación y entrega de información a los clientes/inversionistas, permitiéndoles conocer el estado de su propiedad y de los pagos correspondientes. Todas estas actividades se llevan a cabo de manera continua, mientras dure el arriendo de la propiedad.

Por otra parte, cuando se decide terminar un arriendo, la empresa se encarga del término del contrato, inspecciones posteriores y, si el cliente lo desea, volver a publicar el inmueble para realizar un nuevo arriendo.

Actualmente, la “plataforma” Efirent no existe como una aplicación propia, sino que corresponde a un sistema manejado manualmente mediante herramientas tradicionales (*Excel* principalmente). Este escenario genera dolores importantes en el proceso de administración de inmuebles, pues éste es un proceso continuo que involucra muchos intercambios de información, tanto entre la plataforma y fuentes externas, como entre

---

<sup>1</sup> <https://efirent.cl>

secciones internas de ésta. Para ello se requiere un alto nivel de atención y dedicación para que estos procesos funcionen de manera razonable.

Para ilustrar el punto, por ejemplo, el seguimiento y cobranza de arriendos y/o utilidades actualmente se realiza todo de manera manual, y mediante transferencias/depósitos. Frecuentemente ocurre que no se cuenta con toda la información necesaria para confirmar que un pago ha sido realizado, o quién es su autor, lo que dificulta el proceso de relacionar un pago a una propiedad, y a sus pagos pendientes. Por ende, se deben cuadrar los depósitos con los arrendatarios correspondientes, lo que resulta en un proceso largo y en donde es posible equivocarse, lo que usualmente tiene importantes consecuencias económicas negativas para el negocio.

Previo al pago de un cliente, se deben enviar las notificaciones de cobro en las fechas correspondientes, y notificaciones de morosidad si es que no se realizan los pagos, lo que también se realiza manualmente. Además, se maneja el pago de la garantía en el caso de que ésta sea pagada en cuotas.

Esto se repite para el resto de funciones que pertenecen al proceso de administración del inmueble, lo que conlleva una alta complejidad, pues los datos deben manejarse a mano e incluso en hojas de cálculo distintas. Esto significa que cualquier tarea que implique una propagación de cambios a otras hojas, resulta costosa y riesgosa, pues el error humano puede ser fácilmente obviado cuando se deben actualizar varios archivos para dejarlos consistentes.

En su estado actual, esta limitación es abordada asignando un mayor número de horas dedicadas a ese proceso. Esto limita de forma severa la capacidad de la empresa de expandir su negocio (“escalar”) y volverse más competitivo.

## 1.1. Justificación

Ante el escenario descrito, la primera pregunta que surge es ¿por qué desarrollar una nueva solución y no usar o adaptar una aplicación existente? Para dar respuesta a esta pregunta, es importante reconocer que existen varias plataformas de software, como *Valorízate*<sup>2</sup>, en donde se ayuda a los clientes (“inversionistas”) a encontrar buenas propiedades para comprar. Cuando un inversionista concreta una compra mediante esta plataforma, se le incentiva a utilizar el servicio de Efirent para encontrar un arrendatario y administrar este arriendo (pagos, liquidaciones, mantenimiento, etc). Esto es importante de destacar, pues implica que a futuro la plataforma de software de Efirent recibirá datos de otras plataformas, por lo tanto, debe ser capaz de interactuar con ellas.

---

<sup>2</sup> <https://valorizatechile.cl>

El uso de una solución existente impone límites respecto a qué tanto se puede integrar dicho software, con los datos que manejan actualmente las plataformas que generan la información. Esto implica que podría haber cuellos de botella al transformar los datos entre plataformas, o simplemente, carencia de datos para alimentar el software de Efirent.

Otra razón por la que se descarta utilizar una solución ya existente, es que el proceso que lleva adelante Efirent incluye la captación clientes (por ej., arrendatarios para una propiedad), y la administración de los contratos. Estos procesos tienen diversos sub-procesos ad hoc a la empresa, como lo son las visitas a casas o departamentos, los pagos (tanto de arriendos, como de utilidades y/o gastos comunes), la generación de contratos, el envío de liquidaciones, la realización de inspecciones, etc.

Si bien hay herramientas para apoyar parte de estos procesos (por ejemplo, la firma electrónica de certificados), ninguna de las aplicaciones inspeccionadas cubre suficientes partes del proceso, o son poco flexibles para abordar las necesidades propias del negocio de Efirent. Por lo tanto, si se decidiera contratar diversos servicios para cada parte del negocio, de todos modos, tendría que existir un proceso para compatibilizar los datos que fluyen de un extremo al otro. En consecuencia, la ganancia que entregaría al usar estos productos, podría ser menor que el costo de contratarlos y manejar la interoperabilidad de datos entre estos.

Existen otras empresas dedicadas a un mercado similar (la más notoria es Houm<sup>3</sup>), pero éstas son “competencia” y no corresponden a un servicio que se podría usar para cumplir las necesidades del negocio de Efirent.

## 1.2. Objetivos

### 1.2.1 Objetivo general

El objetivo general de este trabajo de memoria es diseñar e implementar una plataforma de software para Efirent, que permita realizar al menos la administración de las propiedades de sus clientes. Esto implica la implementación de diversos flujos de operación, los cuales consisten en: 1) el ingreso de propiedades a la plataforma (incluyendo posibles modificaciones posteriores), 2) el ingreso de usuarios tanto de administración como arrendatarios, 3) la asociación de arrendatarios y propiedades mediante un contrato de arriendo generado por el sistema, 4) el cálculo mes a mes de los pagos a realizar por los arrendatarios y 5) la interacción con un sistema de pagos externo para permitir la realización de estos de manera simple.

---

<sup>3</sup> <https://houm.com/cl/propietario/arriendo>



La solución desarrollada debe ser fácil de utilizar para arrendatarios, y eficiente para trabajadores de Efirent, otorgando acceso a la información de manera rápida y relevante para las tareas a realizar.

### 1.2.2. Objetivos específicos

Los objetivos específicos que se desprenden del objetivo general son los siguientes:

- Desarrollo del módulo de gestión de propiedades (inmuebles), incluyendo la gestión de entidades relacionadas a estas y necesarias para su creación.
- Desarrollo del módulo de usuarios con sus roles y permisos asociados.
- Desarrollo del módulo de creación de contratos de arriendo, incluyendo la generación de los documentos correspondientes.
- Desarrollo del módulo de administración de contratos en curso, el cual calcula los montos a pagar por el arrendatario según los datos existentes del contrato generado (y potencialmente otros factores).
- Desarrollo de la integración del sistema con un mecanismo de pagos online existente (por ejemplo, Webpay, Khipu, o Boacompra, entre otros). Este servicio debe generar un link de pago con los montos calculados para el mes en curso, y asociar automáticamente un pago exitoso con su inmueble correspondiente.
- Desarrollo del módulo de envío de correos que notifica a arrendatarios y propietarios de los pagos realizados.
- Integración de los módulos para dar origen al servicio de administración de la nueva plataforma de Efirent.

### 1.3. Solución propuesta

La Figura 1 muestra el ambiente operacional de la solución, los principales actores y los macro-componentes de la plataforma Efirent. Esta plataforma soporta tres perfiles de usuario: *arrendatarios*, *ejecutivos comerciales* y *gerentes*.

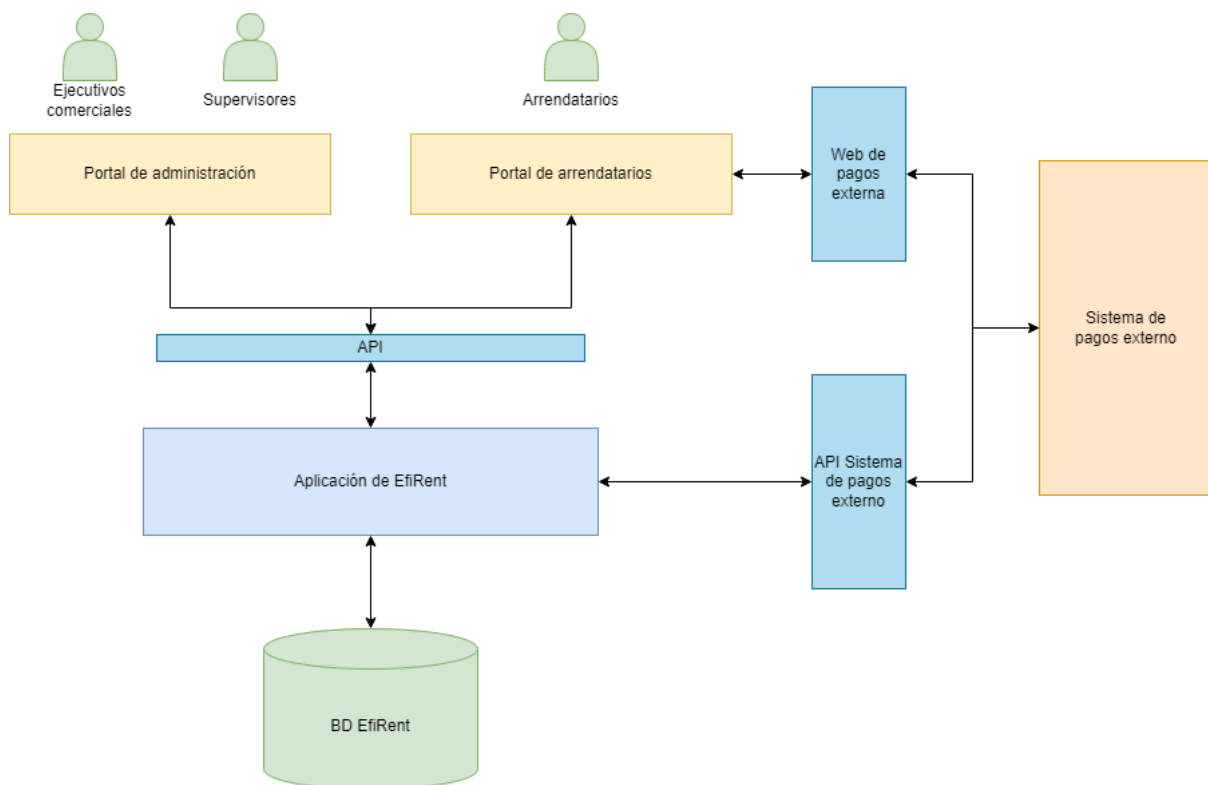


Figura 1. Estructura general de la solución

La plataforma EfiRent es una aplicación Web que sigue la estructura del patrón MVC (Modelo-Vista-Controlador) [6]. Esta aplicación cuenta con un *back-end* (el controlador) alimentado por una base de datos relacional (el modelo), la cual almacena todos los datos relevantes para el funcionamiento de la plataforma (propiedades, registro/historial de pagos, clientes, arrendatarios, etc.).

El *back-end* está desarrollado en *Python*, usando el *framework Django*, y tiene como responsabilidad llevar a cabo las acciones que requiere la lógica de negocio. En este caso, la lógica maneja la creación, edición, eliminación y obtención de los datos desde el *modelo*; es decir, los *CRUDs* de las distintas entidades, principalmente de las propiedades, pero también de unidades asociadas a éstas, como lo son las bodegas y/o estacionamientos.

Con estos datos, se debe permitir a los usuarios ingresar los datos y documentos de los contratos de arriendo para estas propiedades, estableciendo la relación entre el arrendatario y la propiedad. Esto incluye ingresar el valor del canon de arriendo que el arrendatario deberá pagar mes a mes, entre otros costos (garantías, por ejemplo).

Para un contrato de arriendo, el sistema debe calcular los montos a pagar mensualmente, los cuales dependen de los valores establecidos en el contrato de

arriendo, y generar pagos pendientes de realización (registros de pago), así como poder enviar correos automáticos para notificar estas acciones.

El componente *vista* (del modelo MVC), como se mencionó antes, corresponde a una aplicación web desarrollada tentativamente en *Typescript* (*Javacript* con tipado estático) con el *framework* *Vue3*<sup>4</sup>. Ésta deberá permitir a los empleados de Efirent ingresar nuevas propiedades a la plataforma, visualizar las unidades existentes, y ver sus datos detallados, revisar sus historiales de pago (registros de pago) y el estado de estos, y también ingresar registros de pago de manera manual en casos excepcionales.

Además, la vista de la plataforma debe contener una interfaz para que los arrendatarios puedan acceder a la información de sus pagos realizados y/o pendientes. Para realizar los pagos se integrará la solución con una plataforma de pagos externa (véase *Webpay*, *Boacompra*, *Khipu*, *Flow*, etc.).

Esta integración se realiza en parte por *back-end*, mediante la conexión a la API de la plataforma elegida y el envío de *requests* para generar links de pago por los montos calculados en el registro de pago. La integración se finaliza por *front-end* mediante la redirección al link ya mencionado para que el usuario realice el pago mediante la plataforma y mostrando las vistas correspondientes a los resultados del flujo de pago. Si existen inconvenientes para que un arrendatario realice pagos mediante la plataforma externa, se puede hacer uso del ingreso manual de un registro de pago para la propiedad correspondiente.

La comunicación entre la vista (*front-end*) y el *back-end* se realizará mediante una API REST, la cual está implementada con *Django Rest Framework*<sup>5</sup>. Para ambos existe manejo de permisos, ocultando o bloqueando acciones en la vista para los usuarios correspondientes y realizando un chequeo de permisos a la hora de hacer llamadas a la API. Esto se usará como un respaldo de seguridad para impedir el acceso a usuarios maliciosos que sobrepasen los controles realizados por el *front-end*.

Los perfiles de usuarios mencionados anteriormente poseen acceso a vistas del portal. La vista de administración es aquella en donde se puede ver la información de las entidades del sistema, y manipular datos en ésta. En la versión actual de la solución, un ejecutivo puede realizar casi todas las acciones del portal de administración (ingreso de propiedades, ingreso de contratos), pero existen algunas acciones las cuales solo pueden ser realizadas por gerentes; éstas corresponden a acciones relacionadas a manejo monetario (por ejemplo, realizar modificaciones a los registros de pago).

---

<sup>4</sup> <https://vuejs.org>

<sup>5</sup> <https://www.django-rest-framework.org>

Por su parte, el usuario con perfil de arrendatario (o vista de arrendatario) sólo podrá ver el monto del arriendo que debe, y también realizar el pago con el método escogido.

#### 1.4. Estrategia de evaluación de la solución

La solución fue evaluada en términos de su usabilidad y utilidad por un usuario experto (empleado de Efirent) en el proceso actual de la empresa. Para medir el primer atributo, una posibilidad es utilizar la encuesta SUS (*System Usability Scale*) [1] y para medir el segundo, emplear una versión reducida de la encuesta de TAM (*Technology Acceptance Model*) [2].

Por otra parte, se obtuvo información del tiempo actual que toma realizar los procesos de cobro y cuadro de pago de arriendos, para luego compararlo contra el mismo proceso, pero usando la nueva solución. Se elige esta métrica en específico (tiempo de demora) porque el tiempo y recursos humanos dedicados actualmente a esta labor es una de las principales dolencias del negocio.

Se recopiló también la opinión del usuario experto con respecto al resto de procesos de la plataforma, especialmente aquellos de ingreso manual de datos. Los comentarios obtenidos indican que los usuarios perciben como más usable a la nueva solución, en comparación a sus procesos legados (con hojas de cálculo). Esto permite identificar varias ventajas otorgadas por el sistema.

## 2. Marco teórico

Actualmente en el mercado existen diversos sistemas de administración inmobiliaria ajustados a distintos requisitos y procesos o partes de un mismo proceso. Estas presentan posibles soluciones al problema enfrentado, por lo que se deben analizar para justificar el no utilizarlas.

### 2.1. Aplicaciones de gestión de inmuebles

A continuación se presentan diversas aplicaciones de gestión inmobiliaria que se recopilaron como posibles soluciones al problema.

#### 2.1.1. ComunidadFeliz<sup>6</sup>

Esta plataforma ofrece la capacidad de administrar un edificio y sus inmuebles, otorgando acceso a pagos automáticos mediante *Webpay* para los gastos comunes, así como canales de comunicación entre la administración y arrendatarios, además de diversos módulos de reportería. Esta plataforma está enfocada en el manejo de un edificio o condominio en específico y su operación mensual, por lo que no tiene soporte para contratos ni pagos de arriendo, lo que la descarta como una opción para solucionar el problema actual.

#### 2.1.2. Inmogesco<sup>7</sup>

Inmogesco ofrece un software de administración robusto para llevar un inventario de las propiedades manejadas y sus estados, soportando el proceso de captación de arrendatarios. Permite incluso administrar las publicaciones en portales inmobiliarios. Debido a su enfoque en el proceso de captación, no posee soporte para el manejo de los arriendos y por ende los pagos automáticos de mensualidad, y por ende tampoco responde a la necesidad de Efirent.

#### 2.1.3. Breal<sup>8</sup>

Breal es una plataforma de administración de propiedades y arriendos que cumple con una buena parte de los requisitos de la empresa, permitiendo el manejo de propiedades, arriendos, propietarios e inquilinos, pero no ofrece un sistema de pagos automáticos para los arrendatarios. Por lo tanto, no cumple con uno de los requisitos principales de Efirent, que es el permitir el pago mediante alguna pasarela de pagos. Intentar integrar un sistema para permitir esto puede resultar perjudicial para la eficiencia de la operación, en vez de ayudar a su eficiencia.

---

<sup>6</sup> <https://www.comunidadfeliz.cl>

<sup>7</sup> <https://inmogesco.com>

<sup>8</sup> <https://www.breal.cl>

#### 2.1.4. Habitatsoft<sup>9</sup>

Habitatsoft ofrece una plataforma de administración para inmobiliarias pero enfocada en el proceso de captación/venta de éstas, y no se entrega soporte para el manejo de una propiedad una vez arrendada/vendida. Por lo tanto, y al igual que *Inmogesco*, no cumple con los requisitos para abordar los problemas de la empresa.

#### 2.1.5. Skualo<sup>10</sup>

Skualo provee una plataforma de administración para empresas completa, con manejo de pagos, clientes, recursos humanos e inventarios. Sería posible utilizar estas funcionalidades para manejar propiedades, adaptando lo que ofrece ahora mismo para el uso requerido (los pagos de arriendo pueden considerarse como ventas, así como las propiedades pueden ser inventario). Sin embargo, esto significa adaptarse a las limitaciones que daría un software no pensado para el caso específico, lo que puede terminar no entregando la eficiencia requerida para la operación.

### 2.2. Análisis de Espacio de Datos Abordado

Las hojas de cálculo actualmente utilizadas por Efirent manejan diversos datos de las propiedades, los usuarios y los contratos, a partir de los cuales se han seleccionado los relevantes para esta memoria. Luego de recolectar, depurar y unificar las entidades de datos y sus atributos desde las planillas, se muestran a continuación los atributos de cada entidad de datos que maneja el sistema.

#### *Propiedad:*

- *Código:* identificador de la propiedad.
- *Inmobiliaria:* la inmobiliaria a la que pertenece la propiedad.
- *Datos de dirección:* dirección, número, comuna, edificio (si aplica), torre (si aplica), piso (si aplica), edificio (si aplica).
- *Tipo de vivienda:* si es casa o departamento.
- *Otros datos de vivienda:* orientación, cantidad de habitaciones y baños, superficie (interior, terraza y total), si es amoblada, si es nueva, si acepta mascotas, amenidades, fecha de publicación.
- *Estado:* si se encuentra arrendada, publicada u otro estado.
- *Estacionamientos y bodegas:* número de estacionamiento/bodega, cantidad de estacionamientos y bodegas.
- *Capacidad:* número máximo de ocupantes.
- *Tipo de plan:* qué plan contrató con Efirent el propietario de esta propiedad.

---

<sup>9</sup> <https://www.habitatsoft.com>

<sup>10</sup> <https://www.skualo.cl>

- *Datos de arriendo:* fecha de inicio/término de arriendo, número de contrato, arrendatarios, arrendatario principal, codeudores, fecha de entrega a arrendatario, fechas de inspección (última y próxima), monto estimado de gastos comunes.
- *Otros datos:* datos de contacto de administración del edificio (si aplica) y post-venta de la inmobiliaria (si aplica).

#### *Contrato:*

- *Partes del contrato:* arrendatario, propietario.
- *Fechas de vigencia:* fecha de inicio de vigencia, fecha de término o reajuste del contrato.
- *Fechas de pago:* fecha de pago de arriendo, fecha de pago efectivo, fecha de pago a propietario.
- *Arrendatario y deudor:* arrendatario principal, arrendatario(s) secundarios, codeudor (si aplica).
- *Canon de arriendo:* monto a pagar por arriendo mes a mes, moneda, día de cobro, valores de estacionamiento/bodegas.
- *Renovación:* variación de IPC, periodicidad del reajuste, nuevo monto de arriendo post-reajuste.
- *Garantía:* número y valor de garantías, monto y número de cuotas (si aplica).
- *Otros:* comisiones por administración (dependen del plan que tenga el propietario).
- *Anexos:* rebajas/descuentos/ajustes hechos al canon de arriendo (pueden ser permanentes o temporales).

#### *Propietario:*

- *Datos personales:* nombre, RUT, datos de contacto.
- *Plan contratado:* qué plan contrata con Efirent (junto con sus valores correspondientes).
- *Datos bancarios:* datos de cuenta bancaria para recibir los pagos que le correspondan. Pueden variar por propiedad si es que se le administran varias propiedades.

#### *Arrendatario:*

- *Datos personales:* nombre, RUT, datos de contacto.

- *Fechas*: fecha de ingreso a la propiedad, fecha de salida.

#### *Pagos de arriendo:*

- *Monto*: monto del pago.
- *Fecha del pago*: fecha en que se realizó el pago.
- *Medio de pago*: con qué medio se realizó el pago (actualmente depósito/transferencia).
- *Comprobante*: enviado manualmente por el arrendatario.
- *Propiedad/contrato*: a que contrato/propiedad corresponde el pago

#### *Pagos a propietario:*

- *Monto*: monto del pago.
- *Fecha del pago*: fecha en que se realizó el pago.
- *Detalle*: comisiones cobradas (administración/corretaje).

Existen otros datos referentes principalmente al mandato (contrato entre Efirent y el propietario), el cual contiene información de fechas de pago al propietario, comisiones, etc. Estos datos no se han detallado en esta sección debido a que no son relevantes para el problema abordado, pues no influyen en el proceso de pago de arriendos.

## 2.3. Justificación del desarrollo de una nueva plataforma

Dada la recopilación de información realizada, se decide por desarrollar una nueva plataforma debido a varios factores. En primer lugar, Efirent existe dentro de un ecosistema de empresas dedicadas al rubro inmobiliario en distintas etapas, por lo que una plataforma propia permite desarrollar con la integración de estas en mente.

Dentro de Efirent en específico también se maneja el proceso de captación, lo que involucra una lógica distinta al manejo de propiedades y contratos, y por ende requiere planificación dedicada a optimizar el proceso de la empresa. Las plataformas existentes no comprenden todo el abanico de tareas que debe realizar Efirent durante su operación, ni permiten el nivel de integración entre procesos y empresas que es necesario para obtener una operación escalable y mantenible a largo plazo, tanto en términos monetarios como en tiempo utilizado.

Es importante considerar que el utilizar plataformas existentes conlleva un posible costo de integración con lo existente, el cual se mantiene en el tiempo. Esto difiere del costo de integración con un sistema desarrollado a medida, que si bien en un comienzo puede ser alto, la ventaja que posee es que con el tiempo va disminuyendo a medida



que el proceso implementado se aproxima cada vez más a cubrir la operación real de la empresa. Además, a futuro permite implementar nuevas funcionalidades, integradas a lo existente a pedido del negocio, en vez de depender de las prioridades de negocio de las plataformas externas. Por estos motivos, se decide desarrollar una nueva plataforma.

## 3. Concepción de la Solución

En este capítulo se presenta el trabajo que se llevó a cabo en pos del logro de los objetivos definidos. A continuación se detalla el espacio de datos a abordar con el sistema y los principales requisitos. Luego se describen los usuarios del sistema y se presenta el modelo de datos del sistema. Finalmente, se describe el procedimiento de cálculo de cobros y los mockups de las principales interfaces del sistema.

### 3.1. Requisitos de la plataforma

Como ya se mencionó en el capítulo 1, la nueva plataforma de administración de inmuebles posee una serie de requisitos mínimos para poder considerarse un producto viable, y que además aporte valor a las operaciones de Efirent. Estos han sido divididos para abordarse en 3 incrementos, o conjuntos de requisitos a alcanzar en el proyecto. A continuación se describe brevemente la funcionalidad asociada a cada uno de los tres incrementos.

#### 3.1.1. Gestión de unidades arrendadas

El primer grupo de requisitos corresponde a los relacionados a la gestión de datos de unidades arrendadas (ingreso, edición, eliminación y visualización), y otras entidades que entran dentro del manejo de propiedades de la empresa. Esto implica que debe existir un sistema de gestión de datos de propiedades, propietarios, bodegas y/o estacionamientos. Además, debe ser posible mantener otros datos como edificios/proyectos inmobiliarios, amenidades, y otros elementos asociados a las propiedades. Esto permite la organización de las entidades de datos administradas dentro de la plataforma, y también prevenir variaciones innecesarias entre datos que deben hacer referencia a las mismas entidades físicas. Particularmente, se busca evitar es que hayan edificios o entidades en general con el mismo nombre, pero ligeramente distintos. Por ejemplo, que una propiedad esté en el “Edificio Juan Gómez” y otra esté en el “Edificio Juan Gomes”.

Aparte de lo ya mencionado, debe permitirse agregar datos de arrendatarios para poder asociar a las propiedades en arriendo, según se define en las agrupaciones posteriores. Todas las entidades mencionadas en esta agrupación deben además ser modificables y/o eliminables bajo ciertas condiciones, y según permisos asignados a los distintos perfiles de usuario del sistema. Estos permisos son definidos en una sección posterior del presente informe.

#### 3.1.2. Gestión de contratos

El segundo hito corresponde al ingreso y gestión de datos de contratos de arriendo, para generar las relaciones correspondientes entre arrendatario y propiedad. Se

requiere poder ingresar manualmente los datos, o mediante algún archivo externo (preferiblemente en formato de Excel), para así poblar la base de datos de la plataforma con los contratos de arriendo. Se debe poder subir, además de los datos mismos, el documento del contrato en un formato acordado (preferiblemente PDF).

El proceso de generar un contrato, a partir de los datos de la plataforma, se relega a un carácter opcional (nice-to-have), para así priorizar otros requisitos que se alinean más con un sistema factible de desarrollar en el tiempo disponible en el marco de esta memoria.

Una vez ingresado un contrato al sistema, se debe poder subir otros documentos como anexos a este contrato. Además, se debe poder editar los datos del contrato bajo ciertas condiciones y permisos según el tipo de usuario del sistema.

### 3.1.3. Gestión de pagos

El grupo de requisitos más importante para el sistema corresponde al proceso de cobrar los arriendos de las propiedades administradas. El sistema necesita poder, según las fechas definidas en los contratos de arriendo, generar cobros que sigan una serie de reglas de cálculo, y entreguen los montos correspondientes a pagar en el mes en curso. Estas reglas serán expuestas en una sección posterior de este informe.

Para cada cobro generado debe poder agregarse ajustes y/o recargos dados ciertos motivos. Los cobros deben poder modificarse de manera manual en situaciones excepcionales, y bajo expresa autorización de un usuario administrador del sistema, manteniendo ante este caso un historial de cambios.

Un arrendatario deberá poder ingresar a un portal simple en donde, mediante su RUT, podrá revisar si tiene algún cobro pendiente, y realizar el pago cuando sea conveniente. Para ello, se le dará la opción de utilizar alguna plataforma de pagos externa, o bien realizar un pago manual con la obligación de subir un comprobante de pago.

Un pago realizado de la segunda forma debe además pasar a un estado de revisión, para que sea confirmado como correcto. Si el pago no es correcto, o faltase dinero para completar el monto requerido, se podrán asociar más pagos al cobro. Además, debe proveerse una forma de ver los cobros y/o pagos pendientes, los que están en revisión y los ya revisados.

Para cada cobro generado y pago realizado deben enviarse mensajes de notificación, tanto a arrendatarios como a propietarios, notificando las acciones realizadas. Estos deberán ser por correo o a través de alguna aplicación de mensajería instantánea (por ejemplo, Whatsapp).

## 3.2. Perfiles de usuario soportados

Tal como se mencionó antes, dentro de la plataforma existen diversos tipos de usuarios, los cuales tendrán distintas restricciones según su posición en la jerarquía de permisos de la plataforma. Estos perfiles han sido extraídos del funcionamiento actual de la empresa, particularmente de su área de administración de propiedades.

El primer tipo de usuario corresponde al *arrendatario* que ingresa a la plataforma para pagar su arriendo. Este usuario debe poder ver lo que adeuda de arriendo y realizar el pago correspondiente, así como recibir las notificaciones pertinentes.

El segundo tipo de usuario corresponde a un *ejecutivo* del sistema. Éste es quien hace mayor uso de la plataforma, y debe tener acceso al grueso de funcionalidades definidas en los requisitos. Sin embargo, se ve limitado al momento de querer modificar datos identificados como sensibles, o importantes dentro de las entidades de la plataforma, por ejemplo, modificar el canon de arriendo de una propiedad ya ingresada.

El tercer tipo de usuario corresponde al *supervisor de ejecutivos*. Su interacción con el sistema permite realizar todas las acciones posibles del ejecutivo, pero además debe ser capaz de autorizar la modificación (o realizarla por su propia cuenta) de información que haya sido identificada como requirente de autorización.

## 3.3. Arquitectura de la Solución

La figura 2 muestra la arquitectura del sistema, la cual consta de cuatro macro-componentes: aplicación de backend, el portal de administración, portal de arrendatarios y el sistema de pagos externos. A continuación se explican cada uno de estos componentes.

La *aplicación de backend* es la encargada del manejo de los datos de propiedades, contratos, cobros y pagos (recuadro azul con título “Aplicación de EfiRent”). Esta aplicación guarda su información en una base de datos relacional (cilindro verde) y expone una API para que otros componentes de la solución se comuniquen con ésta. Entre sus responsabilidades está el manejo y procesamiento de la información que los usuarios ingresan a la plataforma, el cálculo de los montos de pago, el encolamiento y envío de mensajes de recordatorio, el procesamiento de las acciones que se lleven a cabo sobre los datos ya ingresados, y la comunicación con plataformas externas a la solución.

El *portal de administración* (recuadro amarillo con título “Portal de administración”) puede ser accedido mediante una cuenta de administración y entregará una interfaz gráfica en forma de aplicación web para que los usuarios puedan interactuar con la aplicación descrita anteriormente. Esta tiene la responsabilidad de recibir los datos ingresados por usuarios y enviarlos al backend, así como recibir datos almacenados y

procesados por éste para mostrarlos en un formato legible y ordenado a los usuarios, permitiendo una comprensión rápida de la información y poniendo a disposición acciones que gatillen una comunicación con el backend de la aplicación para realizarlas.

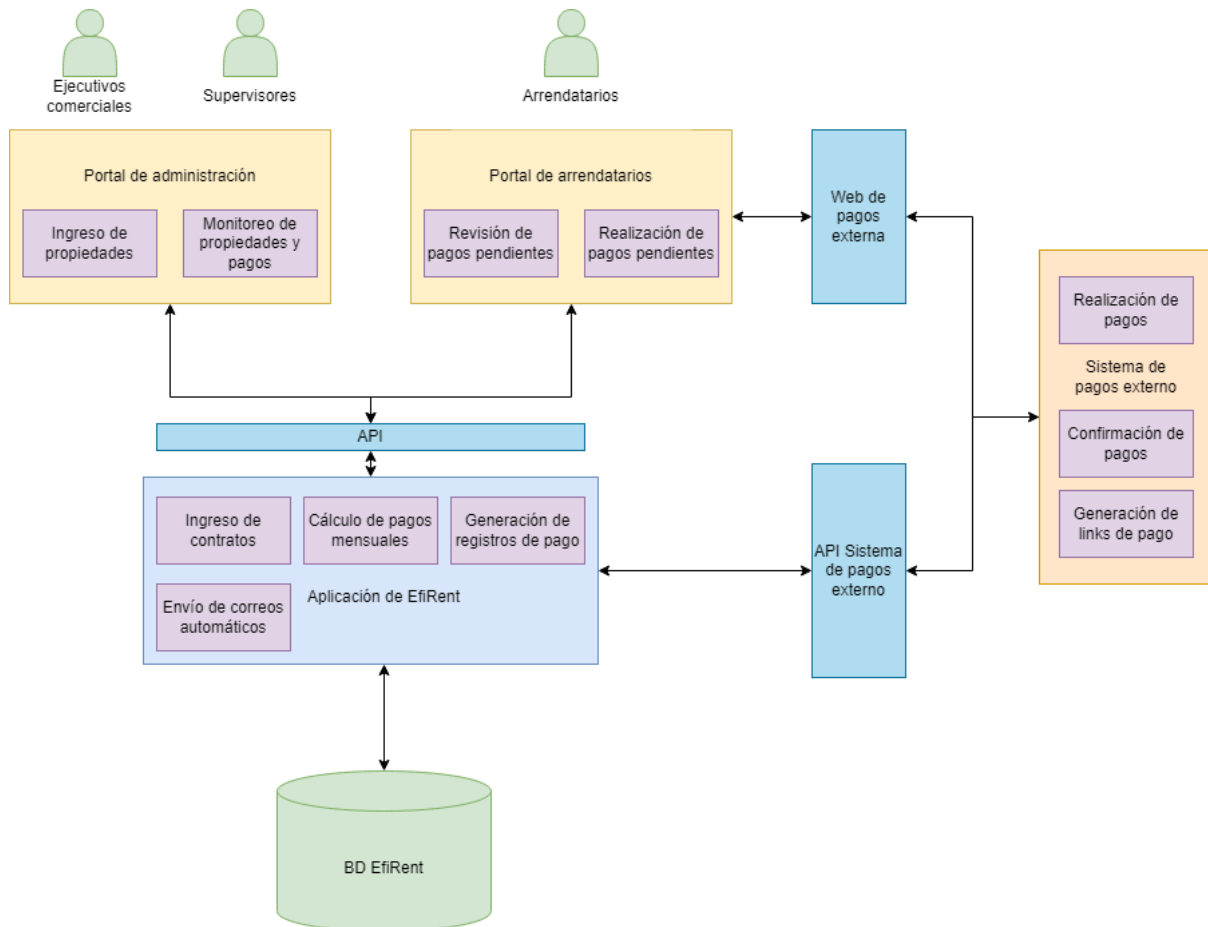


Figura 2. Arquitectura de la solución

El *portal de arrendatarios* (recuadro amarillo con título “Portal de arrendatarios”) corresponde a un portal de libre acceso en donde un arrendatario podrá, usando su RUT, buscar sus pagos pendientes y realizar el pago de estos mediante dos modalidades: un pago manual o un pago mediante plataforma externa de pagos. En el caso del pago manual, éste se realiza completamente mediante la API expuesta por el backend, mientras que para el pago externo se realiza con una combinación de llamadas a la API y redirecciones a una web de pagos externa.

El *sistema de pagos externo*, el cual expone mediante una API las acciones para generar links de pago y confirmar el estado de las transacciones. La responsabilidad de manejar los casos de éxito y fracaso del pago se delegan a la solución desarrollada.

La comunicación con la API de la plataforma externa se realiza exclusivamente mediante el backend, debido a que se deben manejar credenciales que no deben ser manejadas a nivel de frontend (debido que pueden ser inspeccionadas fácilmente). La única interacción entre la plataforma externa y el frontend se realiza mediante los links de pago, los cuales se usan para redireccionar hacia y desde esta plataforma de pagos.

### 3.4. Modelo de datos

Basado en el análisis del espacio de datos presentado en la sección 2.2, se diseñó el modelo de datos de la plataforma, el cual involucra 21 tablas (Fig. 3). Dichas tablas se describen brevemente a continuación.

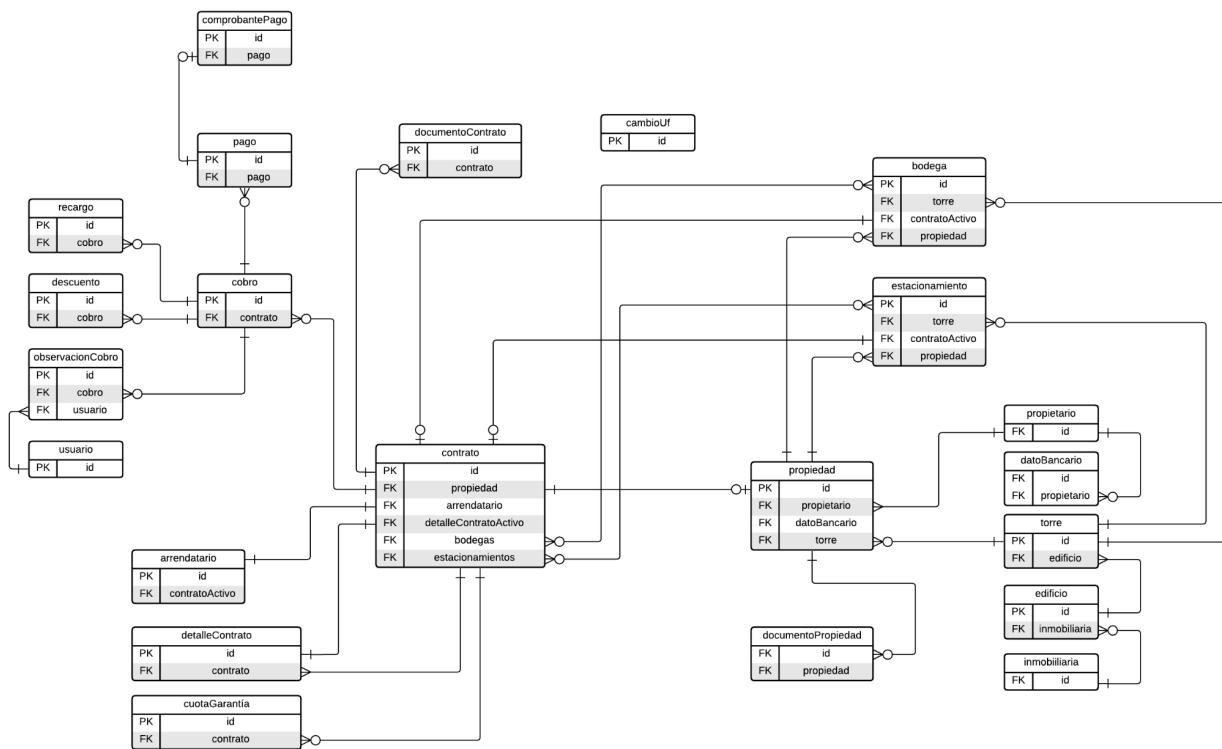


Figura 3. Modelo de datos del sistema

- **Propiedad:** Esta tabla almacena la información de una propiedad, y tiene llaves foráneas a las tablas de *Propietario* (*many-to-one*), *Torre* (*many-to-one*) y *Contrato* (*one-to-one*, corresponde al contrato actualmente activo para esta propiedad).
- **DocumentoPropiedad:** corresponde a un documento subido para una propiedad. Posee una llave foránea a la tabla de *Propiedad* (*many-to-one*).
- **Propietario:** Ésta corresponde a un propietario de una propiedad. Contiene sus datos personales.
- **DatoBancario:** Ésta almacena los números de cuenta usados para recibir los pagos por una propiedad. Contiene una llave foránea a *Propietario* (*many-to-one*)

- *Inmobiliaria*: Esta entidad corresponde a los datos de la inmobiliaria que administra una propiedad.
- *Torre*: Ésta tiene los datos de una torre, que corresponde a una subdivisión de un edificio o complejo de edificios. Por ejemplo, un proyecto inmobiliario puede venderse como un edificio, pero consistir de dos o más torres distintas. En el caso en que el edificio sea único, se mantiene una torre para tener consistencia. Esta tabla posee una llave foránea a la tabla de *Edificio* (*many-to-one*).
- *Edificio*: Esta entidad corresponde a un edificio (no necesariamente el edificio físico, puede referirse a un proyecto inmobiliario). Puede tener una o más torres, y tiene una llave foránea a la tabla de *Inmobiliaria* (*many-to-one*).
- *Bodega*: Esta entidad corresponde a otro tipo de unidad arrendable. Puede estar asociado a una *Propiedad* mediante una llave foránea (*many-to-one*), a una *Torre* (*many-to-one*) y a un *Contrato* (*one-to-one*, representa el contrato actualmente activo).
- *Estacionamiento*: Es similar a bodega, o sea, también es un espacio arrendable. Tiene las mismas relaciones que la bodega.
- *Contrato*: Representa el contrato de arriendo de una propiedad. Tiene llaves foráneas a *Propiedad* (*one-to-one*), *Estacionamiento* (*many-to-many*), *Bodega* (*many-to-many*) y *DetalleContrato* (*one-to-one*, representa el detalle de contrato actualmente activo). Un contrato puede tener varios estacionamientos y bodegas, y éstos pueden pertenecer a varios contratos, pero solo pueden tener un contrato activo, el cual es asignado en el campo de contrato activo en sus modelos correspondientes.
- *DetalleContrato*: Esta entidad contiene datos del contrato para cálculos y otros fines. Posee llave foránea a *Contrato* (*many-to-one*). Un contrato puede tener muchos detalles, pero solo uno de ellos es el detalle activo. Todos los valores del contrato que puedan ser modificados una vez el contrato esté en vigencia se almacenan en esta tabla. El propósito de que el contrato señale su detalle activo es para que el resto de detalles puedan seguir asociados al contrato, otorgando entonces un posible historial de detalles anteriores.
- *Arrendatario*: Ésta corresponde a un arrendatario de un contrato. Tiene una llave foránea a *Contrato* (*one-to-one*, corresponde al contrato actualmente activo para este arrendatario).
- *DocumentoContrato*: Esta entidad corresponde a un documento de un contrato (puede ser el contrato mismo o un anexo). Posee una llave foránea a *Contrato* (*many-to-one*).
- *CuotaGarantía*: Corresponde a una cuota de garantía, usada para luego crear los cobros con los valores correspondientes. Posee una llave foránea a *Contrato* (*many-to-one*).

- *Cobro*: Ésta representa un cobro generado para un contrato según las reglas de cálculo que serán definidas más adelante. Posee una llave foránea a *Contrato* (*many-to-one*).
- *Recargo*: Corresponde a un recargo (aumento del monto) de un cobro. Se relaciona con *Cobro* (*many-to-one*) mediante una llave foránea.
- *Descuento*: Corresponde a un descuento de un cobro. Se relaciona de la misma manera de recargo con *Cobro*.
- *Pago*: Esta entidad corresponde a un pago realizado, asociado mediante una llave foránea a *Cobro* (*many-to-one*).
- *ComprobantePago*: Ésta corresponde a un respaldo (comprobante) de pago para los casos en que se realice un pago mediante depósito/transferencia/otro. Tiene una llave foránea a *Pago* (*one-to-one*).
- *ObservacionCobro*: Corresponde a una observación realizada en un cobro como resultado de una acción tomada sobre éste. Posee una llave foránea que apunta a *Cobro* (*many-to-one*) y a *Usuario* (*many-to-one*).
- *Usuario*: Éste corresponde a un usuario de la plataforma.
- *CambioUf*: Corresponde al valor de cambio de UF a pesos chilenos para un día en específico.

### 3.5. Procedimiento de cálculo de los cobros

Los cobros generados automáticamente se calculan tal como se indica a continuación. Para el mes del cálculo, se suman el valor del canon de arriendo definido en el contrato, la cuota de garantía del mes (si aplica) y/u otros recargos (si aplican). Luego se le resta el valor total de los descuentos aplicados al cobro (si aplican), y finalmente se le suma el interés por atraso de pago. Este último se calcula como un interés simple del 1% sobre el canon de arriendo, multiplicado por cada día de atraso. Visto en una fórmula se representa de la siguiente manera:

$$T = C_a + G + \sum_{i=0}^n R_i + \sum_{i=0}^n D_i + 0.01 * C_a * A_d$$

donde:

$C_a$  = canon de arriendo

$G$  = garantía

$R$  = recargo

$D$  = descuento

$A_d$  = días de atraso

$T$  = total del cobro



## 3.6. Diseño de interfaces de usuario

Usando el software *Balsamiq*<sup>11</sup>, se crearon varios *mockups* para representar cómo se vería la interfaz del sistema a grandes rasgos, y de esa manera poder realizar una validación inicial con los usuarios de la plataforma. El detalle de los mockup desarrollados se reporta en el Anexo A.

## 3.7. Tecnologías escogidas para la implementación

A continuación se presentan las tecnologías utilizadas en la implementación de la plataforma, y se expone la justificación para el uso de cada una de éstas.

### 3.7.1. Backend

El backend de la plataforma se desarrolló en *Python*, usando el framework *Django*. Se elige usar estas tecnologías por dos principales razones: 1) tal como se mencionó en la sección 1.1, existe una plataforma para la empresa *Valorízate* la cual fue desarrollada por un equipo de desarrollo externo anterior, y 2) ésta fue hecha usando la misma combinación de tecnología (*Python* y *Django*).

Adyacente a esto, existió en su momento un proyecto para crear otra plataforma específica para *Efient* dedicada a su proceso de captación de clientes, también desarrollada en el mismo conjunto de tecnologías. Esta plataforma fue construida parcialmente pero nunca puesta en marcha, pero sí existen planes de retomar su desarrollo e iniciar su operación a futuro.

Dadas estas plataformas existentes dentro del proceso de la empresa, se decide utilizar el mismo *stack* para el backend, para así maximizar la compatibilidad entre estos y reducir el esfuerzo de aprendizaje para introducir nuevos desarrolladores a estos proyectos. De esa manera, solo es necesario aprender un conjunto de tecnologías para poder desarrollar en todas las plataformas.

La segunda razón por la que se elige *Python* y *Django*, es la versatilidad que otorga este framework para el desarrollo de plataformas complejas. *Django* cuenta con un ecosistema amplio de librerías y extensiones que permiten implementar una gran diversidad de funcionalidades, embebidas directamente en el framework y sin malgastar recursos en solucionar problemas de compatibilidad.

El proyecto utiliza fuertemente la extensión *Django Rest Framework*, la cual permite crear una API REST utilizando una estructura y sintaxis que sigue los estándares existentes de *Django*. Esta extensión se utiliza debido a que el proyecto está dividido en dos aplicaciones distintas, y era necesario poder comunicar ambas sin tener que

---

<sup>11</sup> <https://balsamiq.com>

realizar una integración complicada de las arquitecturas de dichas aplicaciones. Una API REST permite una comunicación entre ambas aplicaciones siguiendo convenciones de uso común y a la vez permite mantener la separación de responsabilidades entre el frontend y el backend de la aplicación.

Además de *Django Rest Framework*, se utiliza la cola de tareas *Celery*, la que permite encolar tareas de manera asíncrona, permitiendo realizar llamados periódicos a acciones de la plataforma que se ejecutarán sin la intervención de un usuario y con la posibilidad de reintentar tareas fallidas. Para usar *Celery*<sup>12</sup>, también es necesario usar un servicio de *message broker* para permitirle a esta cola enviar y recibir mensajes para la ejecución de las tareas. Para este rol se usa *RabbitMQ*<sup>13</sup>, el cual provee este servicio y es recomendado por la documentación de *Celery*<sup>14</sup>.

### 3.7.2. Frontend

Para el frontend de la plataforma se crea una aplicación de *Typescript*. Se elige este lenguaje por sobre *Javascript* simplemente para poder tener un tipado estático, lo que permite generar archivos con tipos que representan las respuestas que entrega la API, y así poder tener certeza de qué tipo de datos se tienen en el frontend luego de recibirlos desde el backend. Esto entrega una estructura más cohesiva a los datos de la plataforma y permite prevenir errores de tipos que ocurren en *Javascript*, que solo pueden encontrarse durante el *runtime*.

Por sobre *Typescript* se utiliza el framework *Vue*, el cual permite construir aplicaciones modulares, separando el comportamiento de la aplicación en componentes con su propia lógica, los cuales pueden conversar entre sí mediante eventos y pasarse la información necesaria para su funcionamiento. Una de las ventajas de *Vue* es que es un framework que introduce *reactividad* al funcionamiento. Esto implica que, al ocurrir algún cambio a los datos que maneja una vista de la aplicación, ésta puede actualizar lo que se muestra en la interfaz de usuario sin necesidad de recargar la página.

Al momento de administrar datos de las propiedades, existen acciones que deben poder realizarse sin perder el estado actual de la página pero que son demasiado complejas como para poder implementarse con *Javascript* por sí solo. Este requisito deja como opciones varios frameworks, como lo son *React*, *Angular*, *Svelte* y *Vue*, entre otras.

Se elige utilizar *Vue* debido a la existencia de *Quasar*<sup>15</sup>, un “framework sobre *Vue*” el cual pone a disposición una completa librería de componentes y plugins que agilizan el

---

<sup>12</sup> <https://docs.celeryq.dev/en/stable/index.html>

<sup>13</sup> <https://www.rabbitmq.com>

<sup>14</sup> <https://docs.celeryq.dev/en/stable/getting-started/backends-and-brokers/rabbitmq.html>

<sup>15</sup> <https://quasar.dev>

desarrollo de interfaces complejas. Entre los componentes que entrega se encuentra una tabla de datos con múltiples configuraciones y funcionalidades útiles para un software de administración, por ejemplo, el soporte *built-in* para filtrado y ordenamiento de datos.

Debido a esto se decide utilizar *Vue* junto a *Quasar* para desarrollar el frontend de la aplicación, pudiendo enfocar el trabajo en el resto de comportamientos que requieren un desarrollo más en profundidad.

### 3.7.3. Entorno de desarrollo

Para desarrollar la aplicación de backend se utiliza un entorno virtual de *Python* 3.10, manejado mediante el paquete *pipenv*<sup>16</sup>. Esto permite empaquetar las dependencias en un archivo *pipfile*, y exportarlas en un archivo *requirements.txt* para poder realizar el desarrollo en cualquier máquina que posea un ambiente con una versión compatible de *Python*.

Para el frontend se utiliza un entorno con *NodeJS*<sup>17</sup> como base (requisito para usar *Vue*), y el administrador de paquetes *yarn*<sup>18</sup>. Todo esto es desarrollado en una máquina Windows 10 con Ubuntu instalado en el *Windows Subsystem for Linux*<sup>19</sup>.

---

<sup>16</sup> <https://pipenv.pypa.io/en/latest/>

<sup>17</sup> <https://nodejs.org/en>

<sup>18</sup> <https://yarnpkg.com>

<sup>19</sup> <https://ubuntu.com/desktop/wsl>

## 4. Implementación de la Plataforma

Considerando los componentes de la arquitectura descritos en la sección 3.3, a continuación se describe cada uno de ellos.

### 4.1. Portal de Administración

El portal de administración al ser una aplicación de *Quasar* sobre *Vue* sigue la estructura de archivos establecida por *Quasar*, lo que se traduce en una serie de archivos y carpetas encargados de distintas funciones de la *app*.

#### 4.1.1. Estructura del proyecto

Cada interfaz del proyecto corresponde a un archivo en formato *.vue*, el cual integra HTML, *Typescript* y CSS para poder escribir la presentación y lógica de la aplicación en un mismo lugar. Estos archivos se dividen en 2 carpetas distintas: *pages*, en donde se guardan las páginas mismas de la *app* (cada una corresponde a una url en el navegador), y *components*, los cuales son archivos que encapsulan interfaz y lógica que puede ser reutilizada en *pages* o incluso en otros *components*, permitiendo eliminar repetición de código y modularizar el funcionamiento. Por ejemplo, existe un componente encargado de ser la base de los *popups* de la aplicación (llamado *PopupContainer*), e incluye la lógica para manejar su apariencia base y su comportamiento en algunos casos necesarios y de uso reiterado.

Este componente se utiliza como base para todo *popup* mostrado, y también para otros componentes que definen *popups* más específicos. Cada *page* de la *app* está entonces compuesta de código propio y múltiples componentes que entregan lógica lista para usar en los casos necesarios.

Existe además una carpeta *composables*, que corresponde a una metodología introducida en *Vue 3* para encapsular solamente lógica (sin encargarse de la presentación, a diferencia de un componente) reutilizable. Esta carpeta puede ser utilizada en componentes y *pages*.

Por otra parte, en en este proyecto se utilizan *composables* para encapsular la lógica del manejo y presentación de errores al usuario, así como para centralizar la recolección de datos de cobros para mostrar al usuario, lo cual debe hacerse en más de una ubicación y por ende se convierte en un *composable*.

La base de la apariencia de la aplicación se define en la carpeta de *layouts*. En esta se incluyen los archivos que definen los componentes que se muestran en todas las *pages* de la *app*, como lo son la barra lateral, el *header* (con el logo de *EfiRent* y la info de

usuario/botón de cierre de sesión), y los colores del fondo sobre el cual se colocan todas las *pages*.

La carpeta *stores* incluye los datos que deben mantenerse a lo largo de toda la aplicación (o sea, de manera global), independiente de si se cambia de página dentro de esta. Un *store* define los datos que recibe y las acciones que pueden realizarse sobre estos. Para esta *app* se utiliza un *store* para los datos de usuario y autenticación. Dentro de esta se mantiene el usuario actualmente logueado, su *token* de sesión (este es explicado más a detalle en la sección de *backend*) y se definen las acciones de inicio y cierre de sesión. Sin esta *store*, un usuario podría necesitar estar constantemente ingresando sus credenciales entre cada cambio de vista.

El directorio *router* contiene todas las urls de la aplicación (por parte del *frontend* exclusivamente) y su asociación a cada *page* definida con anterioridad. Dentro de éste se definen los nombres, rutas y atributos permitidos (ej: “/propiedades/:id” envía a la vista de la propiedad con el id especificado, y esa *page* tendrá disponible en su código el id recibido).

Finalmente, existen las carpetas *utils*, que contiene métodos auxiliares para ayudar al funcionamiento de la app (formateo de unidades, validaciones de datos, etc); *models*, la cual almacena los tipos utilizados en el resto de la aplicación; *endpoints*, que maneja los llamados a la API del *backend* para poder así centralizar su comportamiento a lo largo del proyecto, y *i18n*, que contiene los textos de la *app* que son insertados en esta mediante el uso de *i18n*, que corresponde a un *framework* de internacionalización, lo que permite cambiar el idioma del texto mostrado según el idioma establecido por el usuario. Por ahora esta carpeta tiene solo los textos en español, pero el uso de este *framework* permite agregar traducciones nuevas sin tener que modificar el código del resto de la *app*.

#### 4.1.2. *Layouts* de la aplicación

Dentro de la *app* existen 2 *layouts* distintos: el de *login* y el *principal*. El *layout* de *login* (Figura 4) es una vista simple que muestra una caja con *inputs* de correo y contraseña, y aparece cada vez que se ingresa a alguna de las vistas del portal de administración, sin tener una sesión iniciada o se cierra una sesión previamente abierta.

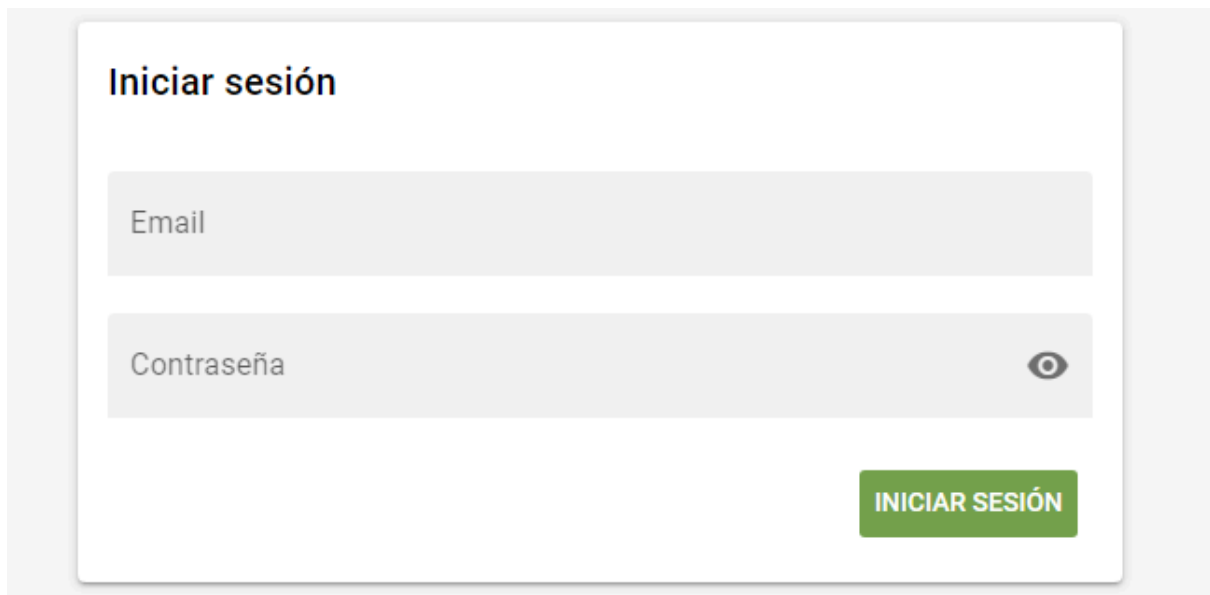


Figura 4. Vista de login.

En el *layout* principal (Figura 5) es donde ocurre la mayoría de la interacción con la aplicación, y éste consta de 3 elementos principales:

- La *barra superior o header*, que contiene el logo de la empresa y un pequeño panel desplegable con el correo del usuario actualmente en sesión. Además, tiene un botón para cerrar esta sesión y uno para abrir la barra lateral.
- La *barra lateral o drawer*. Ésta aparece al mover el cursor sobre la sección izquierda de la pantalla, o al apretar en el botón dedicado a esto en el *header*. Esta barra lateral cumple la función de navegación en la plataforma, mostrando al usuario las páginas de “alto nivel” a las que puede acceder. Que sean de “alto nivel” quiere decir que son páginas desde las cuales se accede al resto de funcionalidades de la plataforma. Por ejemplo, uno de los enlaces es “propiedades”, y hacer click sobre éste lleva a la vista principal de propiedades, con una tabla y botones para dirigirse al resto de páginas o realizar las acciones que sean pertinentes a esta entidad.
- El *contenedor de las páginas o router view*. El contenedor corresponde a toda la zona demarcada por el *header* por arriba y los bordes de la pantalla, y es en donde se muestran las páginas definidas en la carpeta *pages*.

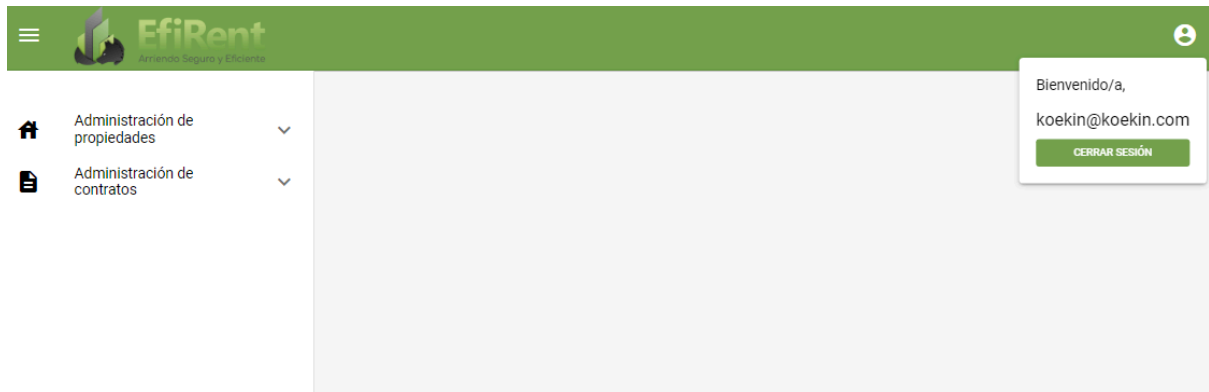


Figura 5. *Layout* principal de la aplicación.

#### 4.1.3. Acceso a los CRUDs

Desde la barra lateral se puede acceder a las páginas de Inmobiliarias, Edificios, Propiedades, Estacionamientos, Bodegas, Propietarios y Arrendatarios. Cada una de éstas lleva a una vista de tabla con las entidades existentes y las acciones correspondientes.

Para cada entidad, se presentan las acciones básicas de editar, ver detalle y eliminar (Figura 6). Sobre cada tabla se encuentra el botón “agregar” que lleva a la vista para crear una nueva entidad. El botón de “editar” lleva a la vista para editar la entidad. El botón “eliminar” muestra un *popup* confirmando la eliminación, y el botón “ver detalle” lleva a la vista para ver los datos de la entidad (no siempre se puede ver toda la información en una tabla por motivos de legibilidad y espacio).

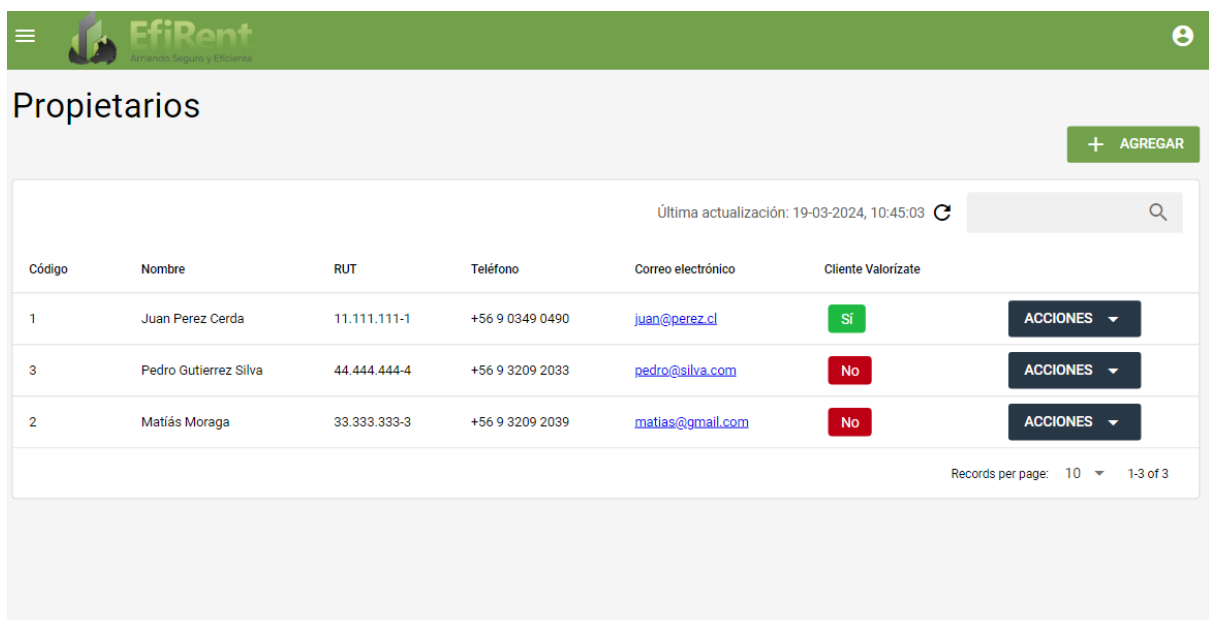


Figura 6. Listado de propietarios.

Dentro de cada tabla, existen algunas columnas especiales; si la columna hace referencia a otra entidad, el usuario puede hacer click en esta para dirigirse al detalle de esta otra entidad. Por ejemplo, la tabla de propiedades tiene una columna edificio, la cual al ser presionada lleva a ver el detalle del edificio al que corresponde la propiedad. Si la columna hace referencia a algún estado, ésta despliega un color representativo del estado. Si la columna corresponde a algún correo electrónico, se abre la aplicación de correos del sistema del usuario.

Todo el comportamiento de las tablas está definido en un componente, llamado *ListComponent*, el cual es reutilizado a lo largo de la aplicación. Éste es una extensión de un componente de tabla entregado por *Quasar*, y le agrega funcionalidades comúnmente usadas, como por ejemplo:

- Una barra de búsqueda, que realiza un filtrado de texto simple, y puede ocultarse si la vista no la requiere.
- Una acción para recargar los datos con información de la fecha y hora de la última recarga.
- La apariencia de las columnas especiales, para lo cual se extienden las configuraciones de las columnas de la tabla original para poder utilizar estos casos especiales fácilmente.
- Acciones, tanto fijas como variables, según una condición de cada fila. Para ello se crea un método que recibe la fila actual, y un método de filtrado de acciones. Luego, por cada fila se aplica este método y se decide qué acciones mostrar.
- El comportamiento de las acciones, las cuales pueden recibir un método *handler* que define qué hacer al seleccionar la acción, o una ruta de la aplicación para que al hacer click sobre ésta, se redirija al usuario, además de otras configuraciones como el nombre, color e ícono.

Todo esto permite tener esta funcionalidad extendida a lo largo de la aplicación. Para el caso especial del servicio para listar propiedades, se agrega además un filtro por inmobiliaria, mediante un selector posicionado encima de la lista. Si se elige alguna opción, se recargan los datos para obtener solo las propiedades de esa inmobiliaria, o bien todas si es que no se selecciona una opción (Figura 7).



Última actualización: 19-03-2024, 10:48:07

Código	Número	Superficie total	Tipología	Edificio	Torre/Block	Inmobiliaria	Propietario	Estado	Acciones
2	1B	35 m2	2D 1B	<a href="#">Edificio 1</a>	Torre 1	<a href="#">Inmobiliaria Uno</a>	<a href="#">Juan Perez Cerda (11.111.111-1)</a>	Arrendado	ACCIONES
1	1A	35 m2	2D 1B	<a href="#">Edificio 1</a>	Torre 1	<a href="#">Inmobiliaria Uno</a>	<a href="#">Juan Perez Cerda (11.111.111-1)</a>	Disponible	ACCIONES
3	3A	35 m2	2D 1B	<a href="#">Edificio 1</a>	Torre 1	<a href="#">Inmobiliaria Uno</a>	<a href="#">Juan Perez Cerda (11.111.111-1)</a>	Disponible	ACCIONES
4	3B	35 m2	2D 1B	<a href="#">Edificio 2</a>	Torre 1	<a href="#">Inmobiliaria Dos</a>	<a href="#">Juan Perez Cerda (11.111.111-1)</a>	Disponible	ACCIONES

Records per page: 10 1-4 of 4

Figura 7. Listado de propiedades con el filtro por inmobiliaria.

Al presionar el botón “crear”, el sistema lleva al usuario a la página correspondiente al formulario de creación del objeto. Ésta contiene una serie de *inputs* que corresponden a las columnas del modelo, aunque no todas las columnas tienen un *input*, pues algunos de los datos se configuran automáticamente. La página para editar y revisar el detalle tiene una estructura similar. Sin embargo, en ambos casos los *inputs* vienen precargados con los datos del objeto seleccionado para edición. Para la acción de revisar el detalle, esta información es de solo lectura.

Para componer cada una de estas páginas de acciones, se realizan los pasos que se indican a continuación. Primero se crea un componente base de formulario para un objeto, que contiene solamente los *inputs* necesarios para obtener los datos y crear la entidad. Este componente incluye campos de texto, campos de selección, además de una configuración que permite poner el formulario en formato de solo lectura.

Para complementar a los componentes de formularios se crean componentes para *inputs* especiales, que se utilizan como cualquier otro *input*, pero con atributos pre-configurados. En los formularios de CRUD de esta sección se usan dos (Figura 8): 1) un *input* para RUT que incluye *masking* (formatear lo que el usuario ingresa para que se vea como un RUT) y revisión de la validez del rut mediante una fórmula para los dígitos de éste<sup>20</sup>, y 2) un *input* para números de teléfono, que también utiliza *masking* para formatear el número a un estilo reconocible (+56 X XXXX XXXX) y validando que

<sup>20</sup> <https://validarutchile.cl/calcular-digito-verificador.php>

corresponda a un número con código de país correcto. Existen otros *inputs* que se utilizan en formularios más complejos y que son explicados más adelante.




Figura 8. *Inputs* de RUT y número de teléfono, con un error de validación en el RUT.

También existen reglas de validación simples para otros *inputs* de texto, como lo son revisar el formato de un correo, o asegurarse de que el usuario haya ingresado algún dato para los campos que son obligatorios. Estas validaciones se realizan para agregar una capa extra de seguridad sobre aquella validación hecha por el backend, y también para poder entregar feedback más específico al usuario sobre los campos con error.

Adicionalmente, en algunos de los formularios de cada objeto se incluye lógica para pedir un listado de otras entidades a la API, en el caso de que sean necesarias para su creación (por ejemplo, para crear un edificio hace falta elegir una inmobiliaria).

Para conveniencia de los usuarios, se agrega un botón al lado de algunos *inputs* de selección para abrir un *popup* que utiliza el componente de creación para el objeto correspondiente (Figuras 9 y 10). Éste permite realizar la creación en el mismo lugar, usando la lógica ya implementada que llama a la API y crea el objeto. Luego, se toma este objeto creado y se agrega a la lista de objetos para seleccionar, permitiendo su uso inmediatamente sin tener que salir de la página, crear el objeto y volver a introducir los datos de lo que se estaba creando.

Debido a que estos *inputs* de selección, representan objetos del modelo de datos, se utiliza un formato específico para almacenar sus opciones. Cada una es (como base) un objeto *JSON* con campos *value* y *label*, siendo el primero el ID del objeto en base de datos, y el segundo siendo el texto que se muestra en la opción. Éste puede ser tan simple como el nombre del objeto, así como ser un texto procesado y generado a partir de otros atributos de la opción. Las opciones pueden tener otros campos para este fin, pero siempre deben presentar los atributos *value* y *label*.



Figura 9. *Input* para edificio con botón para añadir uno nuevo.

Figura 10. *Input* para edificio con el *popup* para crear un nuevo edificio abierto.

Si el formulario es puesto en modo de sólo lectura, estos componentes de selección no cargan la lista de opciones, pues no son necesarias. Al usar cualquier *input* que seleccione algún objeto de otro modelo en la plataforma, sus datos se envían al backend en forma de IDs para realizar su asociación, en vez de tener que enviar sus datos completos.

El formulario de edificio tiene un comportamiento algo distinto a esto; es decir, las torres del edificio se crean junto a éste, mediante una interfaz al final del formulario de edificio, permitiendo al usuario escribir el nombre de la nueva torre y agregarla a la lista, así como eliminarlas antes de guardar el formulario (Figura 11).

El *payload* enviado incluye tanto los datos de edificio como los de las torres, y se envían al backend siendo éste el encargado de crear y asociar estas torres al edificio siendo creado. Estos también pueden verse en un formato de solo lectura, bloqueando cualquier acción sobre estas torres en el formulario.

Figura 11. Interfaz de creación de torres para un edificio.

De manera similar a edificio, el formulario de propietario contiene una sección para agregar datos bancarios, los cuales son manejados completamente dentro del componente de formulario (pudiendo crearlos, editarlos y/o eliminarlos) pero cuyos datos no se envían al backend para su creación y asociación hasta que se confirme la creación/edición de propietario (Figura 12).

A diferencia de edificio y torre, los datos bancarios se crean en un *popup* con los campos necesarios. Al ser creados, estos se almacenan en una lista con la opción de editar el dato o eliminarlo, todo esto de manera local y sin operaciones en la base de datos. Al igual que en el edificio, existe una versión de solo lectura de estos datos, evitando que el usuario pueda tomar acciones sobre estos.

Banco	Tipo de cuenta	Número de cuenta	RUT
Banco BICE	Corriente	0000191231213	11.111.111-1

Figura 12. Interfaz de datos bancarios dentro del formulario de propietario.

Luego, para cada acción se crea un componente que incluye, dentro de su contenido, el formulario del objeto. Por ejemplo, para la creación de propiedades se crea un

*PropertyCreate.vue* que utiliza el componente *PropertyForm.vue*. Esto se realiza con el fin de poder utilizar un solo componente para el formulario, y para mantener consistentes los *inputs* a lo largo de todas las acciones. Sin embargo, se divide cada acción en un componente distinto, para poder implementar lógica única a cada una en caso de ser necesario. En este componente de acción se implementa para el caso de la creación y la edición la llamada a la API para realizar la acción en sí, utilizando alguno de los métodos definidos en la carpeta *endpoints*.

Para llamar a estos métodos, primero se valida el formulario, mostrando todos los errores de validación encontrados. Luego, una vez validados todos los campos, se utiliza un método comúnmente denominado *buildPayload()*, que arroja de vuelta un objeto *JSON* con los datos ingresados por el usuario, después de aplicarles cualquier formato u lógica adicional que deba aplicarse previo a su envío al backend. Por ejemplo, para el caso de las opciones de selección que correspondan a objetos de la base de datos (los mencionados con atributos *value* y *label*), el método toma el campo *value* como el dato a utilizar en el *payload*.

Para el caso de la creación/edición de torres dentro de un edificio, el método toma los datos de las torres ingresadas y las recopila en una lista de objetos *JSON* con los datos de cada torre. De esa forma, el backend puede recorrer esta lista y en cada iteración tener los datos necesarios para crear. Este objeto es entregado al método correspondiente a su acción, y en caso de ser una acción de edición se agrega además el ID del objeto siendo modificado.

En la acción de crear una propiedad se incluye además la lógica adicional para manejar la subida y descarga de documentos asociados a ésta, mostrándose al final del formulario la interfaz para realizar estas acciones. En el caso de la creación, se muestra un simple *input* que permite seleccionar múltiples archivos desde el dispositivo del usuario, así como también eliminarlos de la selección si es que se comete un error (Figura 13).

Al guardar el contrato, primero se suben los archivos al backend, con un *endpoint* aparte que retorna los IDs de los documentos subidos. Dichos documentos son enviados, junto con el *payload* de creación de propiedad, para su asociación.



Figura 13. Interfaz para subir documentos al crear una propiedad.

Para el caso de las acciones de detalle y edición, el componente incluye además la lógica para que, al ser cargado, haga la llamada a la API con el ID del objeto en sí y traiga sus datos. Estos datos pueden, según cómo lo formatee el backend, venir con cierta cantidad de información explícita. Esto es de particular interés al momento de formatear las opciones para un *input* de selección que corresponda a un objeto del modelo de datos, pues en estos casos se recibe un objeto con los campos necesarios para recrear una opción del selector. Por ejemplo, si se tiene un selector de edificios, se recibe desde la API un objeto *JSON* con el nombre y el ID del edificio, así como cualquier otro campo que se utilice para armar el *label* de la opción.

Con estos datos se crea un objeto que sigue la convención de *value* y *label* descrita con anterioridad, y se inserta en el formulario como la opción seleccionada actualmente. Si este formateo es hecho correctamente, un usuario que abra el selector verá correctamente situada la opción correspondiente al valor que tenía el objeto en base de datos. Esto último no es relevante para la acción de detalle; si bien se realiza el formateo para mostrar la opción de manera consistente con cómo se ve al crear/editar, el formulario no tiene las opciones cargadas para seleccionar (y el *input* de selección no se puede abrir para verlas), tal como se describe en la sección donde se presentan estos *inputs*. Por lo tanto, no es necesario que haya un *value* válido en el objeto; es decir, con el *label* basta.

La interfaz de documentos en el caso de la acción de editar una propiedad presenta comportamiento adicional. En primer lugar, se muestran en una lista los documentos subidos con anterioridad (Figura 14). Allí cada elemento posee dos acciones: 1) descargar, la cual abre la interfaz del navegador para guardar el archivo, y 2) borrar, la cual marca el documento para su eliminación al guardarse la propiedad. O sea que, mientras no se haya guardado la edición de ésta, el documento seguirá disponible. Es más, al marcarse para su eliminación, la acción de borrar es reemplazada por una acción para deshacer el borrado, evitando que al guardar la propiedad el documento se elimine del sistema.

Debajo de la lista se encuentra el mismo *input* de archivos visto previamente, y éste puede ser usado para subir más archivos a la propiedad. Al guardar el formulario nuevamente se suben primero los nuevos documentos, y luego se envía al backend la lista de los IDs de estos documentos, así como otra lista con los IDs de aquellos marcados para eliminación.

**Documentos**

Cualquier cambio realizado no se verá reflejado en el contrato hasta que se guarde.

Nombre	Acciones
comprobante_pago_10 (1).jpg	Este archivo será eliminado al guardar
The_Princes_in_the_Tower_by_John_Everett_Millais_(1878).png	

Subir más archivos:

Elegir archivos

Liquidacion-Febrero2024.pdf

**GUARDAR**

Figura 14. Interfaz de documentos para editar propiedad.

Esta interfaz se repite para la acción de “ver detalle”, pero al ser una acción de solo lectura, se ocultan las acciones de borrar/deshacer y el *input* inferior, mostrando solamente la lista de documentos subidos con el botón de descarga. Adicionalmente, la acción ver detalle de la propiedad muestra una interfaz similar al resto, pero con componentes adicionales (Figura 15).

**Detalles de la propiedad**

**Información de la propiedad**

**Arrendado** [IR A CONTRATO](#)

Propietario\*  
Juan Perez Cerda (11.111.111-1)

Edificio\*  
Edificio 1

Torre/Block  
Torre 1

**Dirección:** Las Rosas 2933, Las Condes, Región Metropolitana de Santiago

Número\*  
18

Piso\*  
2

Orientación\*  
N

Habitaciones\*  
2

Baños\*  
1

**Tipología:** 2D 1B

Superficie interior\*  
30.00 m<sup>2</sup>

Superficie terraza\*  
5.00 m<sup>2</sup>

**Superficie total:** 35 m<sup>2</sup>

**Datos de propietario**

Nombre\*  
Juan

Primer apellido\*  
Perez

Segundo apellido  
Cerde

RUT\*  
11.111.111-1

Teléfono\*  
+56 9 0349 0490

Correo electrónico\*  
juan@perez.cl

¿Es cliente de Valorizate? **SI**

**Datos bancarios**

Banco	Tipo de cuenta	Número de cuenta	RUT
BCI / Mach	Corriente	1213131231	33.333.333-3

Figura 15. Detalle de propiedad con información del propietario.

Además de presentar la información de propiedad, se muestra para conveniencia del usuario la información extendida de su propietario, los documentos del contrato activo actual para la propiedad (o nada, si no se encuentra arrendada) y la lista de estacionamientos y/o bodegas asociados a esta propiedad (Figura 16). Esto permite centralizar la información y evitar desplazamientos innecesarios por la plataforma.

The screenshot displays a web interface for property details. At the top left, there are two input fields: 'Superficie interior\*' with the value '30.00' and unit 'm2', and 'Superficie terraza\*' with the value '5.00' and unit 'm2'. Below these, a summary states 'Superficie total: 35 m2'. A dropdown menu for 'Tipo\*' is set to 'Nuevo', and 'Ocupación máxima\*' is '3'. A 'Documentos' section indicates 'No se han subido archivos'. Below this, a 'Documentos de contrato activo' section shows a file named 'diagramaefirent.drawio.png'. On the right side, a table displays bank information: 'Banco BCI / Mach', 'Tipo de cuenta Corriente', 'Número de cuenta 1213131231', and 'RUT 33.333.333-3'. At the bottom right, there are two sections: 'Estacionamientos' and 'Bodegas', each with a search bar and a table with columns 'Código', 'Número', 'Estado', and 'Contrato'. Both tables currently show 'No data available'.

Figura 16. Detalle de propiedad con documentos de contrato, bodegas y estacionamientos.

Finalmente, se inserta el componente de cada acción dentro de una *page* dedicada a esta, con el fin de definir las urls de la aplicación que llevan a cada una. Se agrega cada acción en una *page* separada para poder así dividir las urls de manera lógica (ej: *properties/create*, *properties/:id/edit*, etc.) ya que cada *page* va asociada a una sola url.

El uso de estas *pages* es sencillo: la *page* con la acción crear se abre sin datos previos y muestra el formulario vacío, y las para editar y detalle reciben en su url un ID correspondiente al objeto a editar y lo piden a la API; en caso de no existir, se muestra un error (puede ocurrir si el usuario ingresa escribiendo una url directamente en su navegador).

Finalmente, para la acción de eliminar (y otras acciones que requieran confirmación) se utiliza el componente *ConfirmationPopup*, que tiene como base el *PopupContainer* mencionado en la sección 4.1.1 y le agrega botones de aceptar o rechazar la acción y una forma rápida de mostrar una barra de carga en el *popup*, la que se muestra mientras se realiza la acción y se espera la respuesta de la API (Figura 17).



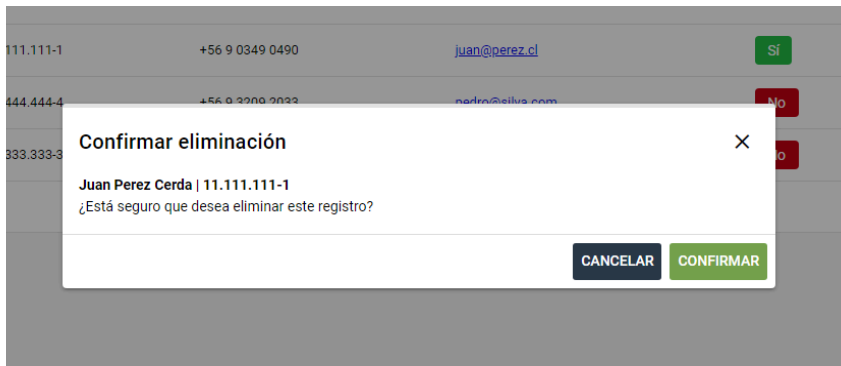


Figura 17. *Popup* de confirmación para la acción de eliminar propietario.

Teniendo todo esto en consideración, se implementan entonces las páginas de listar, crear, editar y ver detalle, así como las acciones para eliminar para las entidades mencionadas al principio de la sección, junto con toda su lógica específica en caso de requerirlo.

Los usuarios utilizan estas vistas ya explicadas para llevar un registro de sus propiedades manejadas de una manera estructurada: se pueden seguir claramente las relaciones entre las entidades, manteniendo una consistencia de los datos y pudiendo acceder a colecciones de datos de manera rápida: es posible ver el propietario, contratos y otros adjuntos de una propiedad o los edificios de una inmobiliaria sin tener que manipular información de manera manual, pues el modelo de datos en sí otorga la capacidad de acceder a la información. Sin embargo, el propósito principal de poder ingresar estos datos es utilizarlos en el manejo de contratos.

#### 4.1.4. Contratos

La siguiente gran sección del portal de administración corresponde a todo lo que involucra contratos: su creación y administración, además de sus entidades relacionadas. En el menú lateral se puede ir a la tabla de contratos, la que funciona de manera idéntica a las tablas descritas en la sección anterior, pero con algunos comportamientos introducidos para esta vista.

##### 4.1.4.1. Lista de contratos

En primer lugar, existe una columna que muestra el valor de arriendo establecido por contrato, en pesos chilenos. Este dato se recibe desde el backend como un valor numérico calculado siguiendo reglas que serán explicadas en su sección correspondiente. Para mostrar este número de una manera fácilmente reconocible como un monto monetario, se crea en la carpeta de *utils* un archivo *currency.js* con métodos auxiliares para manejar el formateo de los números a monedas, utilizando la *ECMAScript Internationalization API*<sup>21</sup>, la cual recibe un código de moneda (al peso

<sup>21</sup> [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Intl](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl)

chileno le corresponde “CLP”) y transforma un número simple a un *string* formateado (ejemplo: convierte 12035 en \$12.035) (Figura 18).

Código	Arrendatario	Propiedad	Estacionamientos	Bodegas	Moneda	Valor de arriendo	Estado
12	Juan Perez (11.111.111-1)	N/A	0	1	CLP	\$1	Pendiente
14	Juan Perez (11.111.111-1)	N/A	0	1	CLP	\$1	Pendiente
15	Juan Perez (11.111.111-1)	N/A	0	1	CLP	\$1	Pendiente
28	Juan Perez (11.111.111-1)	<a href="#">3 (Edificio 1 Torre 1.3A)- Juan Perez Cerda (11.111.111-1)</a>	0	0	CLP	\$1	Pendiente
29	Juan Perez (11.111.111-1)	<a href="#">3 (Edificio 1 Torre 1.3A)- Juan Perez Cerda (11.111.111-1)</a>	0	0	CLP	\$1	Pendiente
22	Pedrito Perez (22.222.222-2)	<a href="#">2 (Edificio 1 Torre 1.1B)- Juan Perez Cerda (11.111.111-1)</a>	0	1	CLP	\$400.000	Vigente
1	Iván Larrain Muñoz (20.192.352-2)	<a href="#">1 (Edificio 1 Torre 1.1A)- Juan Perez Cerda (11.111.111-1)</a>	1	0	CLP	\$550.000	Cerrado
31	Juan Perez (11.111.111-1)	<a href="#">3 (Edificio 1 Torre 1.3A)- Juan Perez Cerda (11.111.111-1)</a>	0	0	UF	UF 10,30	Cerrado

Figura 18. Lista de contratos.

Adicionalmente, las acciones de esta vista tienen diferencias con las acciones en las otras tablas presentadas con anterioridad: se mantiene la acción de agregar, pero el resto de acciones de cada fila se muestra dependiendo del estado actual del contrato. Para un contrato recién creado y que aún no entra en vigencia se permiten todas las acciones tradicionales: editar, eliminar y ver detalles. En conjunto con la acción de agregar, éstas se componen de la misma forma que las acciones presentadas anteriormente, es decir, un componente de formulario base insertado en componentes específicos para cada acción, con el formulario estando encargado de llamar a la API para obtener los datos que necesita para mostrar sus *inputs* de manera correcta, y los componentes de acción formateando el *payload* y haciendo las *requests* a la API para llevar a cabo las acciones.

Aparte de estas acciones se agrega una para pasar a vigente un contrato, la cual muestra un *popup* de confirmación (usando el componente descrito antes en las acciones de eliminar) para realizar la acción. Una vez estando en vigencia el contrato, las acciones disponibles cambian: el contrato no se puede pasar a vigente (por obvias

razones) ni eliminar, y se agrega la acción de administrar y cerrar contrato. Para un contrato cerrado solo se permite la acción de ver detalles.

#### 4.1.4.2. Formulario de contratos

El formulario de contrato contiene en primer lugar un selector de propiedades, bodegas y estacionamientos disponibles. Un contrato puede contener un número de cualquiera de estas tres entidades. Es posible crear uno con solo una propiedad, solo un estacionamiento, solo una bodega o una combinación de estos.

El input de selección de propiedad permite seleccionar solo una, mientras que los inputs de bodega y estacionamiento permiten elegir varias opciones (Figura 19). La apariencia de las que son seleccionadas se modifica para dar más claridad sobre sus datos, pues el comportamiento por defecto de una selección múltiple es listar los nombres separados por comas.

The screenshot shows a web form titled "Información del contrato". At the top, there is a dropdown menu for "Propiedad" with the selected value "4 (Edificio 2, Torre 1, 3B) - Juan Perez Cerda (11.111.111-1)". Below this is a section titled "Información de la propiedad" containing a table of details:

Edificio	Edificio 2	Dirección	Las Flores 125, Ñuñoa, Región Metropolitana de Santiago
Torre/Block	Torre 1	Tipología	2D 1B
Número	3B	Superficie total	35 m2
Propietario	Juan Perez Cerda		
Estacionamientos disponibles	2		
Bodegas disponibles	2		

Below the table are two checkboxes: "Incluir estacionamientos" (unchecked) and "Incluir bodegas" (checked). At the bottom, there are two multi-select dropdowns. The "Estacionamientos" dropdown shows one selected item: "2 - Propiedad 4 - Edificio Edificio 1 - Torre Torre 1". The "Bodegas" dropdown shows two selected items: "2 - Propiedad 4 - Edificio Edificio 1 - Torre Torre 1" and "3 - Propiedad 4 - Edificio Edificio 1 - Torre Torre 1".

Figura 19. Selector e información de propiedades, estacionamientos y bodegas.

Al seleccionar una propiedad se muestra un recuadro con información de esta, de manera de que el usuario pueda ver más datos sin sobrecargar el *label* del selector de propiedades. Dentro de este recuadro se muestran dos *checkboxes* que permiten

agregar al contrato los estacionamientos y/o bodegas que la propiedad tenga asociada y que estén disponibles para arriendo. Al seleccionar alguna de éstas, se agregan al selector correspondiente como si el usuario las hubiese seleccionado por su cuenta, y si se deseleccionan entonces se remueven del selector.

Cabe notar que, si el usuario elige los estacionamientos o bodegas que correspondan a la propiedad sin haber usado el *checkbox*, existe de todos modos lógica implementada para marcar el *checkbox* sin intervención del usuario y así mantener consistencia entre estos y el selector de estacionamientos y bodegas.

Estos *checkboxes* aparecen solo de manera condicional: si la propiedad seleccionada no está asociada a ningún estacionamiento o bodega, o estos no están disponibles para arriendo, el *checkbox* correspondiente no se mostrará.

Para lograr este comportamiento se configura el backend para que envíe junto con la propiedad una lista de estacionamientos y bodegas asociados, y se realiza un formateo de sus datos para cuadrarlos con la lista de estacionamientos y bodegas disponibles que se tiene en el formulario. Si por alguna razón se recibe algún objeto que no esté disponible en la lista, no se considera. Se usan estos datos para decidir si mostrar o no los *checkboxes*.

A continuación se encuentran los *inputs* para establecer las fechas de inicio y fin/renovación del contrato. Para esto se disponen dos cajas de texto en donde el usuario puede escribir una fecha a mano o usar un selector de fechas. En ambas cajas hay validaciones del formato de la fecha, para que ésta sea mostrada/ingresada siguiendo la estructura DD-MM-YYYY. Este formato luego se convierte al orden ISO 8601<sup>22</sup> (YYYY-MM-DD) al momento de enviarlo al backend.

La fecha de inicio es de libre elección, pero se muestra un mensaje de aviso en caso de que se elija una fecha en el mes actual o uno anterior a este, pues esto implica que el primer cobro del contrato una vez sea pasado a vigencia corresponderá al mes siguiente al presente (por definición del cliente) (Figura 20).

La fecha de término funciona de manera idéntica a la de inicio, pero cuenta además con validaciones adicionales para que esta no pueda ser menor a la fecha de inicio. Esto toma la forma de una validación simple mediante comparación de los *strings* de las fechas, pero también se aplica un bloqueo a las opciones en el selector de fechas. Esto sirve como una doble capa de seguridad: si por alguna razón se consigue desbloquear las opciones, al hacer *submit* del formulario de todos modos se activará la otra validación (y de todos modos existe un chequeo similar en backend).

---

<sup>22</sup> <https://www.iso.org/iso-8601-date-and-time-format.html>



Figura 20. Selector de fechas abierto.

La siguiente sección del formulario de contrato corresponde a la de información de pago, lo que comprende el valor de arriendo, fechas de cobro, montos y cuotas de garantía. Se dispone en primer lugar de un selector de moneda para elegir entre pesos chilenos y UF, siendo estas las dos monedas permitidas en la plataforma (por el momento). Seguido de este selector se encuentra el *input* para ingresar el monto mensual de arriendo, el cual se formatea según la moneda elegida. Si es pesos chilenos, se muestra de manera similar a cómo se hace en la columna de monto de arriendo en la página de listar contratos (Figura 21), mientras que si es UF, se muestra a un costado del *input* el valor de la UF correspondiente al día actual, y debajo de éste se dispone el valor convertido a pesos chilenos (Figura 22). Ambos casos utilizan la *Internationalization API* para convertir el número al formato correspondiente, permitiendo hasta dos decimales en caso de la UF.

Seguido de estos *inputs* se encuentra el selector del día de cobro, que corresponde a una grilla con 30 posibles opciones (se excluye el día 31 pues no todos los meses tienen 31 días). Se marca con un borde negro el día actual para más claridad, y se marca en verde el día seleccionado por el usuario. Se muestra además la información de que si se cambia el día de cobro de un contrato, este cambio no tiene efecto en el cobro del mes en curso, sino que comienza a ser válido desde el siguiente.

Información de pago

Moneda\*  
CLP

Valor de arriendo\*  
\$400.000

! Si se cambia el día de cobro, este tomará efecto en el siguiente mes.

Fecha de cobro de contrato: 16 de cada mes.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30

Figura 21. Interfaz de monto de arriendo en pesos chilenos y día de cobro.

Información de pago

Moneda\*  
UF

Valor de arriendo\*  
CLF 12,50

Valor UF al 18-03-2024: \$37.001

! Valor de arriendo en CLP: \$462.507

! Si se cambia el día de cobro, este tomará efecto en el siguiente mes.

Figura 22. Interfaz de monto de arriendo en UF

Después de elegir el monto de arriendo y día de cobro se permite ingresar el monto de garantía a pagar en pesos chilenos (no se trabaja con garantías en UF en Efirent), además del monto de esta ya pagado a contra contrato. Se muestra debajo de ambos *inputs* el monto restante a pagar (monto de garantía - monto pagado contra contrato), y según este monto se calculan las cuotas de garantía para el contrato. Las cuotas se despliegan como una lista con inicialmente una sola cuota por el monto restante y un selector para el número de cuotas deseadas, con un máximo de 10 (este número se elige arbitrariamente, en la práctica no suelen exceder las 3). Según la selección se recalculan las cuotas, siendo todas por el mismo monto.

En cada elemento de la lista de cuotas se encuentra un *checkbox* que permite marcar la cuota como pagada, lo que significa que esta cuota no será cobrada. Esto se incluye como funcionalidad principalmente para manejar los casos en que un contrato ya existente y que aún tenga cuotas de garantía por pagar se ingrese a la plataforma: se permite mantener la información de las cuotas que se pagaron, pero se evita tener que cobrarlas todas nuevamente y eliminar las que correspondan. En la cabecera de la lista se encuentra un *checkbox* que actúa como un “seleccionar todas” (Figura 23).

Valor de garantía\*  
\$500.000

Garantía pagada contra contrato  
\$100.000

Garantía restante  
\$400.000

! Las cuotas de garantía se calculan sobre la garantía restante.

Número de cuotas\*  
3

! Las cuotas no marcadas como pagadas se agregarán automáticamente a los primeros cobros de arriendo.

Cuotas		<input type="checkbox"/> Marcar como pagada
<input checked="" type="checkbox"/>	Cuota 1 \$133.333	<input checked="" type="checkbox"/> Marcar como pagada
<input type="checkbox"/>	Cuota 2 \$133.333	<input type="checkbox"/> Marcar como pagada
<input type="checkbox"/>	Cuota 3 \$133.333	<input type="checkbox"/> Marcar como pagada

Figura 23. Interfaz de cuotas.

A continuación se elige desde un selector al arrendatario del contrato. Se muestra el nombre completo de éste junto con su RUT, así como su información de contacto. A un costado se encuentra un botón para desplegar el formulario de creación de arrendatario y poder crearlo en el momento, al igual que en formularios anteriores.

Finalmente, se muestra la interfaz para subir documentos al contrato, la cual funciona de manera idéntica a la interfaz mencionada en la sección 4.1.3 para subir documentos de propiedad, mostrando un *input* de múltiples archivos. Al guardar el contrato, primero se suben los archivos al backend con un *endpoint* aparte que retorna los IDs de los documentos subidos, los cuales son enviados junto con el *payload* de creación de contrato para su asociación (Figura 24).

Información del arrendatario

Arrendatario\*  
Luis Perez (66.666.666-6) +

Datos de contacto

Correo  
luis@perez.com

Teléfono  
56990239029

Documentos de contrato

Elegir archivos  
The\_Princes\_in\_the\_Tower\_by\_John\_Everett\_Millais\_(1878).png x Liquidacion-Febrero2024.pdf x

GUARDAR

Figura 24. Sección final con selector de arrendatario y documentos de contrato.

Una vez creado el contrato, aparece en la lista de contratos con estado “Pendiente”. Durante este estado, es posible editar el contrato en su totalidad, lo que funciona de manera equivalente a crear, pero con sus datos pre-cargados. La mayor diferencia al editar reside en la interfaz para manejar los documentos del contrato, la cual nuevamente se comporta de la misma manera que la interfaz de documentos de propiedad, mostrando en una lista los documentos previamente subidos con acciones para eliminar/deshacer y descargarlos, además de agregar nuevos archivos al contrato (Figura 25).

Documentos de contrato

! Cualquier cambio realizado no se verá reflejado en el contrato hasta que se guarde.

Nombre	Acciones
comprobante_pago_10 (1).jpg	↓ 🗑️
Liquidacion-Febrero2024.pdf	Este archivo será eliminado al guardar ↓ ↶

Subir más archivos:

Elegir archivos  
The\_Princes\_in\_the\_Tower\_by\_John\_Everett\_Millais\_(1878).png x

GUARDAR

Figura 25. Interfaz de documentos en editar contrato.

Esta interfaz se repite para la acción de ver detalle, pero en una versión de solo lectura con la única acción disponible siendo el descargar. La acción de pasar a vigente cambia el funcionamiento de la edición del contrato: un contrato vigente solo permite editar el monto de arriendo, la fecha de pago y sus documentos; todo lo demás se muestra como de solo lectura.



### 4.1.4.3. Administración de contratos vigentes

Si se desea hacer cambios a la propiedad y/o estacionamientos/bodegas debe hacerse un contrato nuevo. Para poder tener disponibles estas entidades para la creación, éstas no deben estar en un contrato activo, y si lo están éste contrato debe cerrarse. Esta acción muestra un *popup* de confirmación (usando nuevamente *ConfirmationPopup*) y termina el contrato, anula sus pagos pendientes y libera las entidades relacionadas a éste.

Un contrato vigente otorga acceso a la acción de administrar, la cual abre una vista que contiene diversos componentes: una sección de detalle del contrato, la cual tiene la misma funcionalidad que la acción ver detalle ya existente, con la diferencia de que la interfaz de documentos no se muestra en el mismo lugar, si no que en una sección aparte en donde se permiten los controles de eliminar y subir nuevos archivos (Figura 26).

Administrar contrato

Información del contrato

Vigente

Propiedad  
2 (Edificio 1, Torre 1, 1B) - Juan Perez Cerda (11.111.111-1)

Información de la propiedad

Edificio	Dirección
Edificio 1	Las Rosas 2933, Las Condes, Región Metropolitana de Santiago
Torre/Block	Tipología
Torre 1	2D 1B
Número	Superficie total
1B	35 m2

Propietario:  
Juan Perez Cerda

Estacionamientos disponibles  
0

Bodegas disponibles  
0

Estacionamientos

Bodegas

1 - Propiedad 1 - Edificio Edificio 1 - Torre Torre 1

Fecha de inicio de contrato\*

Fecha de fin de contrato\*

Cobros

PENDIENTES EN REVISIÓN CONFLICTIVOS PAGADOS

Mes	Año	Fecha de cobro	Valor	Valor en CLP	ACCIONES
Mayo	2024	20-04-2024	\$450.000		ACCIONES
Junio	2024	15-06-2024	\$450.000		ACCIONES
Agosto	2024	15-08-2024	\$450.000		ACCIONES

Records per page: 10 1-3 of 3

Documentos

Nombre  
diagramaefrent.drawio.png

Subir más archivos:

Elegir archivos

GUARDAR

Figura 26. Vista de administrar contrato.

Se agrega además una sección para administrar los cobros del contrato. Esto se muestra como una lista de éstos, con cuatro pestañas para separarlos por estado: pendientes, en revisión, conflictivos y pagados. Todos tienen acceso a una acción, la cual cambia según su estado. Para los cobros en revisión se muestra la acción “administrar”, la cual muestra un *popup* con la información del cobro (Figura 27), lo que incluye un desglose del valor, una interfaz para agregar/editar/eliminar un recargo (lo que suma valor al cobro) o un descuento (resta valor). Para crear/editar alguno de estos se abre otro *popup* en donde se puede seleccionar el motivo de una lista de opciones, así como una opción “Otro” que permite ingresar un motivo manualmente.

Los recargos pueden tener como origen un cobro de intereses por sobre un cobro atrasado, lo que se refleja en el desglose como un recargo con motivo “Intereses” y el número de días de atraso. Si se elimina el recargo por motivo de intereses, el cobro queda marcado para que deje de generar intereses hasta que no vuelva a ser agregado este recargo.

Tanto recargos como descuentos se crean solamente en pesos chilenos, independiente de la moneda del contrato. En caso de que el contrato sea en UF, el desglose se muestra tanto en UF como en pesos chilenos.

Debajo del desglose del cobro se muestra una lista con los pagos realizados para este cobro y el estado de estos, además de la fecha de pago y si es que éste se realizó con atraso. Al ser un cobro en estado pendiente, todos sus pagos deben de estar rechazados, si no, éste estaría en un estado distinto. Para los pagos manuales se permite también descargar el comprobante que el arrendatario haya subido.

Se muestra además una acción para marcar el cobro como pagado, lo que genera un registro de esta acción y se pasa este cobro a revisión. Se requiere para esto subir un archivo que respalde el pago y el motivo por el cual se marca como pagado desde la plataforma de administración.

Esta acción queda registrada y se muestra al usuario en el costado del *popup* como una línea de tiempo que muestra la acción, quién la realiza y el comentario ingresado, el cual puede ser editado posteriormente a su creación para considerar casos en que el texto ingresado es erróneo (esto fue requerido por cliente). Cada una de las acciones realizadas en el *popup* de administrar el cobro es registrada de la misma forma.

Una vez estando en revisión un cobro, las acciones que se muestran son: aprobar, lo cual marca el cobro como pagado y cierra su flujo de acciones; observar, que no modifica el estado de éste y simplemente registra una acción de observación con un comentario; marcar como conflictivo, que junto al motivo ingresado cambia el estado del cobro a conflictivo, indicando que existe algún problema con éste y no debe ser aprobado aún, y devolver a pendiente, lo que rechaza el cobro y lo devuelve al estado pendiente para que pueda volver a intentarse un pago. Una vez el cobro se encuentra en estado pagado no se permiten más acciones sobre éste. Toda la lógica descrita para manejar los cobros se encapsula dentro de un componente *CobroManager*.

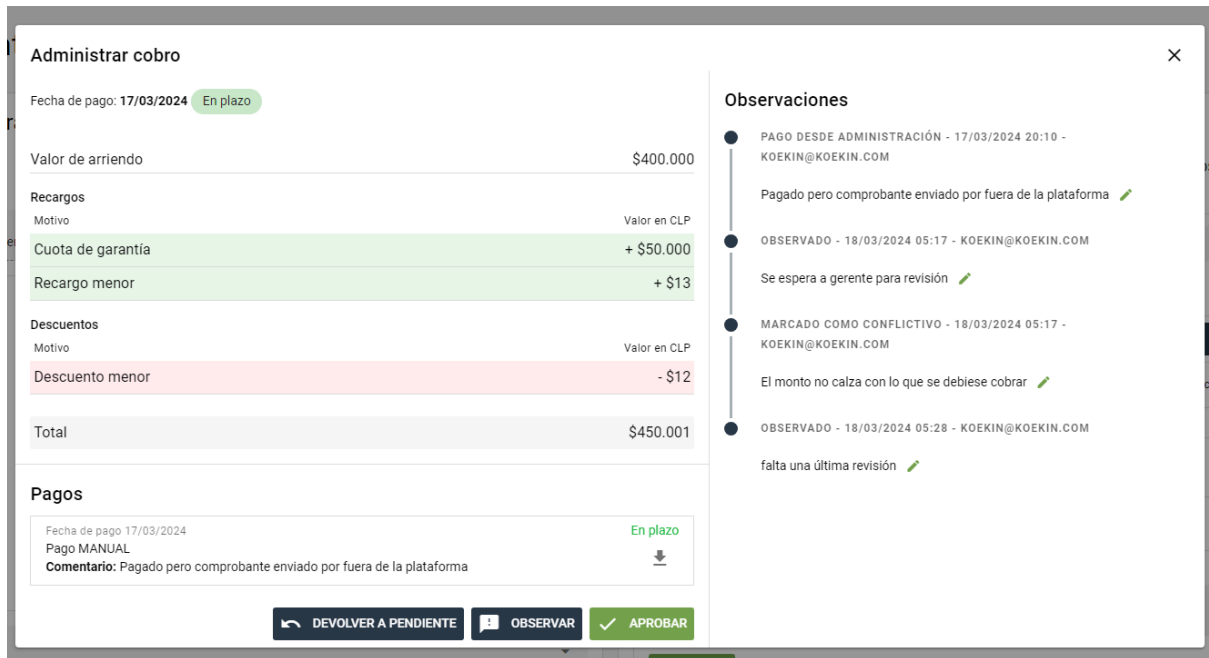


Figura 27. Interfaz de administración de cobro.

Se crea además una página de cobros separada de la de administrar contrato (Figura 28). Ésta corresponde a una lista de cobros por estado pero para todos los contratos de la plataforma, lo que permite ver el estado global de los cobros sin tener que buscarlos dentro de su contrato correspondiente. Esta página funciona de manera idéntica a la administración del cobro dentro de la página de administrar contrato, incluyendo las mismas acciones y el mismo flujo de estas. Para implementar esta página se reutiliza el componente de administración de cobros descrito arriba, pero con datos de cobros sin discriminar por contrato.



Figura 28. Página de cobros

## 4.2. Portal de Arrendatarios

El portal de arrendatarios se crea dentro de la misma aplicación que se usa para crear el portal de administración, pero éste corresponde a un *layout* distinto, el cual muestra solamente una pantalla con un botón para iniciar sesión (lo que lleva al portal de

administración) y el contenido de la página en el centro. A diferencia del portal de administración, no existe un menú lateral, y toda la interacción del usuario con este portal se realiza en una sola página.

#### 4.2.1. Búsqueda por RUT

El portal comienza con una barra de búsqueda por RUT con las validaciones necesarias para asegurar el formato y validez de éste. Al hacer la búsqueda, aparecen debajo de ésta los cobros pendientes encontrados para este RUT. Si no se encuentra ninguno, se muestra un mensaje informando al usuario. De encontrarse, se ordenan por el más cercano a su fecha de vencimiento (Figuras 29 y 30).

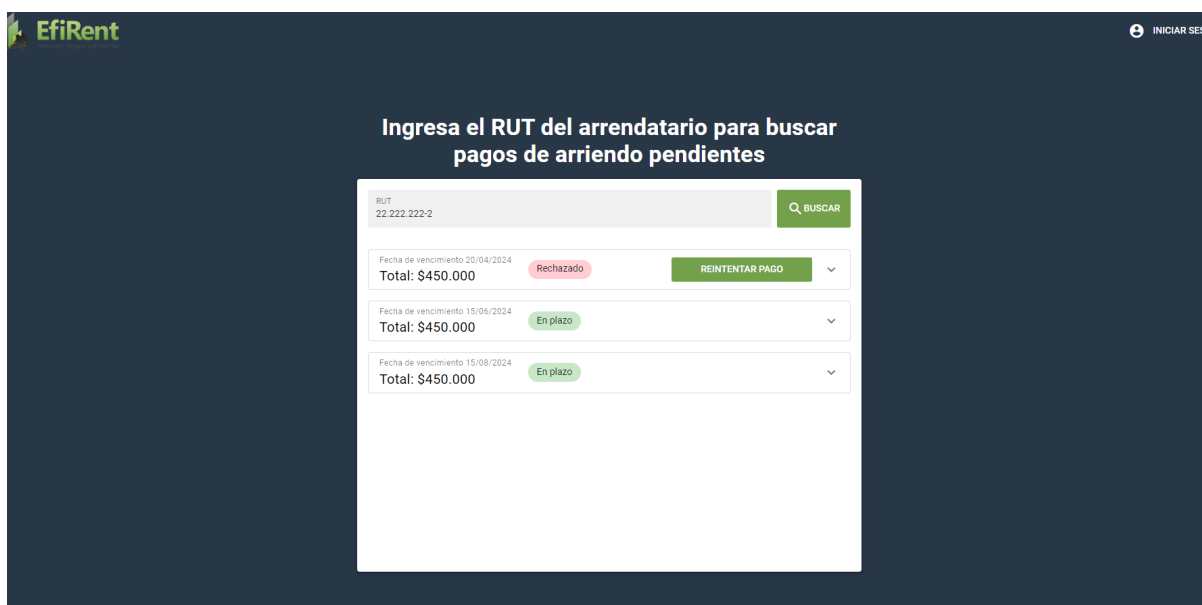


Figura 29. Portal de arrendatarios con resultados por RUT

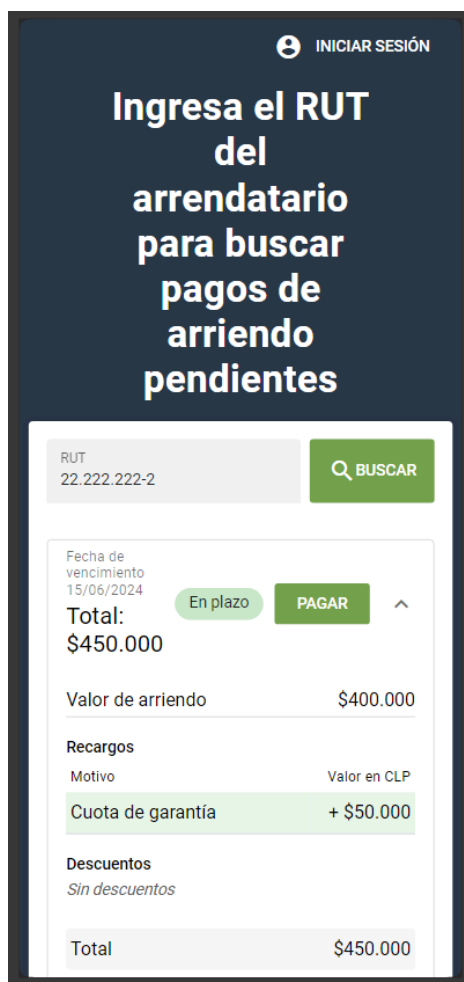


Figura 30. Portal de arrendatarios con resultados por RUT en tamaño *mobile*.

Los cobros pendientes se muestran como una lista que muestra la información de la fecha de vencimiento, el total a pagar, el estado del cobro (en plazo/atrasado/rechazado) y la acción de pagar (reintentar pago si es que ha sido rechazado anteriormente), que se muestra solamente para el primer cobro de la lista (para pagar un cobro deben haberse pagado los anteriores a este).

Al hacer click en cualquiera de los elementos de la lista, éste se expande para mostrar más información (Figura 31): el desglose del cobro, de manera idéntica a como se ve en la administración de cobro pero sin ninguna de las acciones posibles, así como el *timeline* de acciones tomadas sobre el cobro, con la diferencia de que el arrendatario sólo puede ver aquellas que hayan devuelto el cobro al estado pendiente, lo que se muestra a éste como la acción “rechazado”. Esto para dar a entender que la administración rechazó el pago manualmente, pero sin mostrar las acciones internas que se puedan haber tomado para llegar a esta decisión. Ambas secciones son implementadas reutilizando los componentes creados para la administración del contrato, los cuales admiten una opción para mostrarse en modo solo lectura.

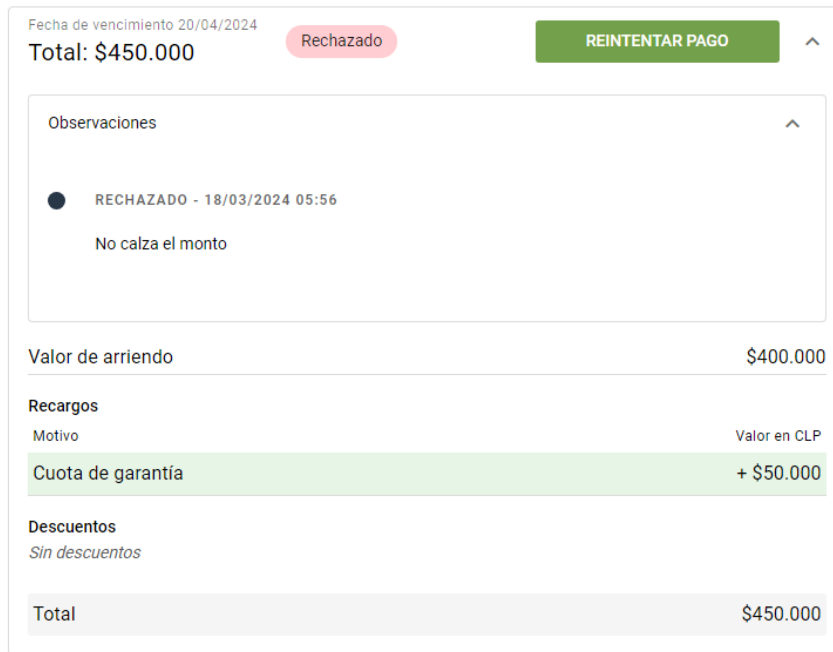




Figura 31. Información completa del cobro.

Al presionar el botón de pago, se abre un *popup* con las opciones disponibles para pagar. Por el momento estas corresponden a un pago manual y a *Webpay*. Para el pago manual, se muestra una interfaz que muestra el monto, los datos de transferencia (para efectos de este informe estos datos son ficticios), un *input* para subir un archivo, y otro para incluir un comentario opcional con el pago (Figura 32). Debajo de estos se muestran mensajes informativos al usuario indicando que el comprobante debe ser legible y por el monto correcto, y que el cobro pasará a estar en revisión una vez sea enviado el formulario.

La opción de *Webpay* muestra al usuario un botón para iniciar el pago, lo cual pide al backend un link de pago con el monto correspondiente y redirige a la página de *Webpay*, dentro de la cual el flujo es manejado completamente por la plataforma externa. Una vez terminado el proceso, independiente del resultado, el usuario es redireccionado a la página de confirmación de la transacción, la cual pide al backend que entregue el estado de ésta para mostrar al usuario el mensaje correspondiente. Tanto en el éxito como en el fracaso, se muestra un botón para volver a la página inicial de búsqueda (Figura 33).


**Pagar** ✕

**Pago manual**  Transferencias, depósitos, etc.

**Webpay** 


**Datos de pago**  
**Banco:** Santander  
**Tipo de cuenta:** Corriente  
**Número de cuenta:** 0000 1234 5678  
**Titular:** Efirent SpA.  
**RUT:** 11.111.111-1


**Monto: \$450.000**

 Comprobante\*

Comentario (opcional)


0 / 255

 Asegúrese de que el comprobante subido sea legible y sea por el monto exacto del cobro.


 Una vez enviado el comprobante el pago pasará a revisión.

**ENVIAR**

Figura 32. Interfaz de pago manual.

 INICIAR SESIÓN

**Resultado de la transacción**



**¡Pago exitoso!**

**Orden de compra:** 84259951  
**Fecha:** 19/03/2024 10:53  
**Medio de pago:** Tarjeta terminada en 6623  
**Monto:** \$450.000

**VOLVER AL INICIO**

Figura 33. Resultado del pago por Webpay.

## 4.3. Backend de la plataforma

La aplicación de *Django* que cumple el rol de backend de la plataforma está organizada siguiendo la estructura convencional de un proyecto de *Django*, utilizando como motor de base de datos *PostgreSQL*<sup>23</sup>. Es decir, consiste de varias *apps* que tienen un propósito y una responsabilidad definida dentro del funcionamiento de la solución. Además, existe una *app* base que se encarga de la configuración general de la aplicación. Cada *app* corresponde a una carpeta dentro de la carpeta principal del proyecto, y todas excepto la *app* base ya mencionada poseen una estructura en común.

### 4.3.1. Estructura de cada *app*

Cada *app* tiene un archivo `_init.py_` que permite incluir código que será ejecutado al momento de inicializar la aplicación. Existe un archivo `admin.py` en donde se registran las configuraciones para el *admin* de *Django*, el cual provee un portal de administración sencillo en donde se pueden realizar acciones sobre la base de datos. El archivo `apps.py` permite configurar el comportamiento de la *app* en ámbitos, como por ejemplo, su nombre (el que es usado por el resto de *apps* para acceder a esta), sus campos por default, entre otros.

El archivo `models.py` es el primer archivo importante de cada *app*, y en éste se definen todos los modelos de datos que correspondan al funcionamiento de la *app*. Cada uno de estos modelos corresponde a una tabla en la base de datos y está representado en el archivo como una *clase* de *Python* con sus atributos correspondientes, los cuales pueden ser campos sencillos (de texto, booleanos, etc) o identificar relaciones con otras tablas. También se pueden configurar otros atributos de cada modelo, como lo son su representación de *string* (de manera que al imprimir un modelo en la terminal o cualquier otra interfaz se vea escrito de manera legible) o sus restricciones (unicidad o cualquier otra restricción que se desee implementar). Además, se pueden crear métodos que apliquen cierta lógica al momento de guardar cambios a la base de datos o campos calculados cuyo valor dependa de una lógica especificada (no se guardan en base de datos, si no que son calculados en *runtime*).

Los modelos creados siguen el modelo de datos planificado en la sección 3.4, aunque su implementación en la práctica presenta variaciones ligeras. Notablemente, las relaciones *many-to-many* son representadas por *Django* con tablas intermedias entre las entidades. El diagrama generado a partir del *schema* de la aplicación se encuentra adjunto en el Anexo B.

El archivo `views.py` de cada *app* corresponde a los controladores de la aplicación, los cuales reciben las *requests HTTP* enviadas a la API y aplican la lógica escrita para

---

<sup>23</sup> <https://www.postgresql.org>



luego retornar, en este caso, un objeto *JSON*<sup>24</sup> con los datos requeridos. Cada *view* corresponde a una clase de *Python*, debido a que *Django Rest Framework* está pensado para usarse con *class-based views*<sup>25</sup>. Cada una de estas clases hereda de las clases entregadas por *DRF*, las que proporcionan un comportamiento común para recibir y responder a las *requests*.

Las *views* se encargan de procesar y tomar cualquier acción necesaria ante una *request*, lo cual puede significar realizar cálculos, crear objetos en base de datos, modificarlos y/o procesarlos según sea programado. Dentro de cada *view* se definen métodos *get*, *post*, *delete*, *put* y *patch*, los cuales al incluirse permiten que un controlador maneje acciones de este tipo. Por ejemplo: si un controlador solo tiene un método *get*, cualquier *request* que llegue a éste y que no sea de tipo *get* fallará.

El archivo *serializers.py* corresponde a un conjunto de clases que determinan cómo procesar los datos de los modelos para convertirlos a *JSON* o cómo leer un objeto *JSON* y convertirlo a un objeto que corresponda a un modelo de la aplicación, así como aplicar validaciones de datos. Estos permiten, por ejemplo, establecer un número de campos requeridos en una *request*.

Por su parte, una *view* define qué *serializers* ocupar en su operación y así poder rechazar *requests* que no tengan los campos definidos o entregar una respuesta formateada de una manera predecible y reutilizable en varias *views* (por ejemplo, si existen múltiples *views* que responden con un mismo tipo de objeto). Al usar las clases entregadas por *DRF*, a veces resulta innecesario definir explícitamente la creación/edición de un objeto, pues si se usa la clase correcta para la acción necesaria, ésta usará la información procesada por el *serializer* para crear/editar el objeto correspondiente. Por ejemplo, si se usa como base para una *view* la clase *CreateAPIView*, y se le entrega un *serializer* con una serie de campos, no es necesario escribir explícitamente las instrucciones para que se cree un objeto, si no que basta con darle qué modelo es el que se espera se cree (del archivo *models*), y esta *view* usará el *serializer* para procesar los datos, realizar validaciones y crear el objeto.

El archivo *urls.py* posee las declaraciones de las urls de los *endpoints* de la aplicación, cada uno correspondiendo a una *view*. Aquí se establece el string de la url y qué parámetros acepta. Debido a que una *view* puede tener más de un tipo de acción, una url puede recibir varios métodos HTTP.

Para algunas *apps* se agrega un archivo *utils.py* que incluye métodos auxiliares relacionados a la lógica implementada en el resto de la *app*. Estos pueden ser métodos para encapsular acciones repetitivas o para remover lógica innecesaria de otros

---

<sup>24</sup> <https://www.json.org/json-en.html>

<sup>25</sup> <https://docs.djangoproject.com/en/4.2/topics/class-based-views/>

archivos. También existe en algunas *apps* el archivo *tasks.py*, el cual define tareas que *Celery* (la librería de encolamiento de tareas utilizada) puede ejecutar. Éstas corresponden a lógica que en vez de ejecutarse de manera inmediata se pone en una cola de ejecución para que se haga cuando estén disponibles los *workers* o recursos necesarios.

Estas tareas suelen corresponder a cosas no urgentes como envío de correos/mensajería, y si bien en un ambiente local esto suele ser instantáneo, para un ambiente de producción es importante no ralentizar el funcionamiento de la aplicación esperando que se realice de manera síncrona una acción que puede ponerse en una cola para después.

Finalmente, las *apps* que tengan modelos definidos en *models.py* tienen una carpeta *migrations*, en donde se guarda el historial de cambios realizados a la base de datos, lo que corresponde a creación/modificación/eliminación de modelos y adiciones/cambios a sus atributos. Estas acciones crean archivos de migración automáticamente y permiten conocer las modificaciones que se han realizado y también cómo deshacerlas.

#### 4.3.2. Aplicación base y utils

La aplicación base del proyecto no sigue la misma estructura que el resto de *apps*. Esta posee archivos para definir los *handlers* de errores para el proyecto en general, un archivo *settings.py* en donde se configura el proyecto (aquí se aplican varias configuraciones para *DRF* y también para el funcionamiento de la autenticación en la aplicación). Cuenta también con un archivo *urls.py* pero en este caso sirve para indicar la base de la url para el resto de *apps* (por ej, se le asigna el *path* “app1/...” a la *app1*, por lo que todas las urls de esa *app* partirán con “app1”.

Existe además una aplicación *utils* en donde se define un modelo base para el resto de *apps*. En el archivo *models* de esta *app* se incluye una clase que crea un modelo de datos con campos de fecha de creación/edición actualizados automáticamente. Todos los otros modelos de datos descritos en el proyecto heredan de este modelo base, permitiendo que todos tengan columnas en la base de datos que señalen cuándo fueron creados y cuándo fue su última edición.

#### 4.3.3. Aplicación de propiedades

La aplicación de propiedades sigue la estructura descrita con anterioridad. En su archivo de *models* declara los modelos de datos correspondientes a las propiedades y otras entidades como lo son inmobiliarias, edificios, entre otros. En detalle, los modelos declarados en esta *app* son los siguientes:

- *Inmobiliaria*. Se agrega una validación que evita que se elimine una inmobiliaria si es que existe alguna propiedad o edificio que pertenezca a esta.

- *Edificio*: Se define su relación con inmobiliaria y se agrega una condición que evita su eliminación si existen propiedades o torres asociadas a este edificio. También se declara que no puede haber más de un edificio con el mismo nombre y la misma inmobiliaria.
- *Torre*: Se define su relación con edificio, se agrega una condición para que no se pueda eliminar una torre que tenga propiedades, estacionamientos y/o bodegas asociadas. También se prohíbe que exista más de una torre con el mismo nombre en el mismo edificio.
- *Propietario*: Se definen sus atributos, entre los cuales se crea un campo calculado para mostrar su nombre completo en las respuestas que entrega la API. También se agrega una condición que prohíbe eliminar un propietario que tenga propiedades asociadas.
- *DatoBancario*: Se define su relación con el modelo “propietario”. No presenta protecciones ante eliminación, y si un propietario se elimina, también se eliminan sus datos bancarios.
- *DocumentoPropiedad*: se define su relación con propiedad. No presenta protecciones ante eliminación, siendo borrado si es que su propiedad lo es.
- *Propiedad*: Se define su relación con propietario, dato bancario y torre. Se define también su relación con su contrato activo, que se define en la *app* de contratos. Se crean diversas variables calculadas para poder obtener su estado de disponibilidad (se define como disponible si está marcada como arrendable y si no tiene ningún contrato activo). Se agregan validaciones para evitar que se cambie el contrato activo de una propiedad sin haber cerrado su contrato activo actual con anterioridad, y además se evita que se pueda marcar como activo un contrato que no haya sido hecho para esta propiedad. Se establece también que no se puede eliminar una propiedad que esté en un contrato y/o que tenga bodegas y estacionamientos asociados a ésta. Además, se agrega una restricción para que no pueda haber otra propiedad asociada al mismo contrato activo que otra.
- *Estacionamiento y bodega*: Corresponde a dos modelos muy similares en su implementación. Se define su relación con propiedad y torre, además de su pertenencia a un contrato activo. También se agregan variables calculadas para saber si se encuentra disponible y se implementan restricciones similares a propiedad: no se permite cambiar el contrato activo si no se cierra el anterior, y no se puede poner como contrato activo un contrato en que no se haya incluido este/a estacionamiento/bodega.

El archivo de *views* se compone de diversos *viewsets*, que corresponden a clases entregadas por *Django Rest Framework* que agrupan los métodos HTTP comunes y permiten definir la lógica del CRUD de un objeto en una sola clase, en vez de tener que

definir múltiples clases. Se implementan dentro de este archivo los controladores para el CRUD de todos los objetos descritos en el archivo *models*. Para cada *view* se definen los *serializers* correspondientes en el archivo *serializers*.

- *Para inmobiliarias, propietario, propiedad, bodega y estacionamiento* se crea un CRUD sencillo que permite crear, editar y eliminar. Los *serializers* para estas *views* se encargan de exigir todos los campos necesarios, incluyendo los de las relaciones (un campo con el ID del objeto relacionado, por ejemplo, enviar el ID de propietario al crear una propiedad). En el caso de estacionamiento y bodega, se agrega también la validación de que debe enviarse al menos una torre o una propiedad al momento de crear/editar, la cual es utilizada para asociar el estacionamiento/bodega a estos.
- *Para edificios* se agrega a las acciones de crear y editar una validación de que se envíe al menos una torre en el *payload* que recibe el controller y que utiliza para la creación o modificación del objeto. También se encarga de crear/editar/eliminar las torres enviadas aparte de solo el edificio, y asociarlos como corresponden.
- *Para propiedades* se crean controladores para listar las propiedades de la plataforma, permitiendo filtrarlas por inmobiliaria (esto es usado por el *frontend* en la interfaz de la figura 7) y también listar aquellas que cumplan las condiciones para ser consideradas *arrendables*, o sea, que no tengan un contrato activo actualmente y que estén marcadas como disponibles para arriendo. Dentro de este controlador se consiguen además los datos de las bodegas y estacionamientos disponibles y que estén asociados a esta propiedad, y se empaquetan en un formato anidado (estacionamientos y bodegas se ponen como un atributo de propiedad, y corresponden a la lista de cada uno). En la creación/edición de propiedades se agrega la lógica para asociar a ésta los documentos enviados, así como eliminar aquellos que se marquen para eliminación.
- *Para estacionamientos y bodegas* se crean controladores que devuelven la lista de aquellos que sean *arrendables*, lo que al igual que propiedad corresponde a no tener un contrato activo y estar marcada como disponible.
- *Para documentos de propiedad* se crean controladores para subir el archivo así como para descargarlo, además de otros para listar estos documentos (solamente su nombre/id, se dispone el controlador de descarga para obtener el archivo en sí) tanto libremente como según propiedad.
- *Para propietarios* se agrega en el crear/editar la lógica para que cree/edite los datos bancarios enviados en el *payload* y los asocie a sí mismo, así como elimine los que no estén presentes en éste, por lo que al editar los datos de *propietario* se debe enviar la información de sus datos bancarios para evitar que estos sean eliminados.

### 4.3.2. Aplicación de contratos

La aplicación de contratos sigue la misma estructura descrita con anterioridad y se encarga de manejar la información pertinente a los contratos de arriendo, lo que significa que administra a los arrendatarios, los cobros/pagos, y todas las otras entidades que se utilizan para el funcionamiento del contrato. Específicamente, los modelos que declara esta aplicación son los siguientes:

- *Contrato*: se define su relación con la propiedad, estacionamiento y/o bodegas que se encuentran en el contrato, así como la relación al detalle del contrato activo actualmente. Se declaran además una serie de restricciones: los montos del contrato deben ser igual o mayores que 0, no puede existir más de un contrato activo por propiedad, arrendatario, bodega o estacionamiento; la fecha de término del contrato debe ser mayor a la fecha de inicio, y el monto de garantía pagado contra contrato no puede exceder al monto declarado de garantía. Estas restricciones se validan al momento de intentar crear/guardar algún cambio al contrato.
- *DetalleContrato*: se define su relación a contrato, así como sus campos con las siguientes restricciones: los montos de cobro no pueden ser menores que 0, y el día de cobro debe ser un número en el rango entre el 1 y el 30, ambos inclusive.
- *Arrendatario*: se define su relación con contrato activo, así como una propiedad calculada para obtener su nombre completo. Se agrega la validación de formato al campo de correo, así como validaciones para no permitir cambiar el contrato activo en caso de que ya exista uno, que este contrato esté además en la lista de contratos que contienen al arrendatario, que no se pueda eliminar si es que tiene algún contrato asociado y que no puede haber otro arrendatario que tenga el mismo contrato activo.
- *CuotaGarantía*: se define su relación con contrato y el mínimo de 0 para el monto.
- *Cobro*: se define su relación con contrato, los valores mínimos para monto (1 para el caso de pesos chilenos, 0.01 para UF), campos calculados que entregan el valor en pesos chilenos (realizando la conversión si es necesario), si es que el cobro está vencido o en estado pendiente pero con observaciones realizadas.
- *RecargoCobro* y *DescuentoCobro*: se define su relación con cobro así como el valor mínimo de 0 para su monto, así como una propiedad calculada que entrega el motivo formateado para ser mostrado en frontend.
- *Pago*: se define su relación con cobro, además de un campo calculado que retorna si es que el pago fue hecho con atraso.
- *ObservacionCobro*: se define su relación con cobro y usuario.

- *ComprobantePago*: se establece su relación con pago y la restricción de que solo puede haber un comprobante por cada pago.
- *DocumentoContrato*: se define su relación con contrato.

El resto de la lógica de la aplicación se encuentra implementada en el archivo de *views*. Dentro de éste se encuentra el *viewset* que maneja el CRUD de arrendatario, otorgando las 4 acciones disponibles. Se crea además un controlador para listar los arrendatarios que no tengan ningún contrato activo actualmente, lo cual el frontend utiliza para la vista de crear/editar contrato. Se crea también el *viewset* de contrato, el cual tiene lógica especial para dos de sus acciones:

- *Crear contrato*: debido a limitaciones de los *serializers* de *DRF* se implementa a mano la lógica para verificar que en el *payload* de creación se incluya al menos una propiedad, bodega y/o estacionamiento, además de verificar que los datos de cuotas de garantía pagadas cuadre con las cuotas totales elegidas. Una vez pasadas las validaciones se crea el contrato y se le asocian la propiedad y las bodegas/estacionamientos correspondientes. A continuación, se crea el detalle de contrato y se asocia al contrato recién creado, así como también se asocian los documentos enviados en el *payload*. Finalmente, se crean las cuotas de garantía según la información ingresada, y se asocian también al contrato. En la sección 3.1.2 se menciona entre los requisitos el poder ingresar los datos de un contrato a partir de un archivo externo, pero finalmente esto no fue implementado en la plataforma por razones de tiempo.
- *Editar contrato*: se hacen las mismas validaciones iniciales que en crear contrato, pero además se revisa, en caso de que se esté intentando editar un contrato en estado vigente, que no se haya cambiado ninguno de los campos que no se permiten editar en ese estado. Una vez pasada esta validación se asocia la propiedad y las bodegas/estacionamientos correspondientes (si está en vigente y pasó la validación anterior, estos cambios son irrelevantes), se crea un nuevo detalle de contrato y se establece como el detalle de contrato activo, se modifican las cuotas de garantía según sea necesario, se eliminan los documentos de contrato que hayan sido marcados para eliminar (tanto de la relación como de la base de datos) y se asocian los nuevos.

Luego se implementa el controlador para la acción de pasar un contrato a estado vigente, lo que involucra validar que el contrato esté en estado pendiente y que ninguna de sus entidades relacionadas tenga un contrato activo previamente. Una vez confirmada la validez de la acción se cambia el estado de éste, se asigna el contrato como el activo para la propiedad, estacionamientos, bodegas y arrendatario correspondientes y se crean los primeros cobros de éste.

En el caso de no tener cuotas de garantía impagas, se crea simplemente el cobro correspondiente al mes siguiente, de lo contrario se crean tantos cobros como cuotas de garantía impagas hayan sido ingresadas al momento de crear el contrato. Cada uno de estos cobros corresponde a un mes distinto y se genera como un cobro por el monto de arriendo con un recargo correspondiente a la cuota de garantía. Esto se realiza así para evitar revisar periódicamente los contratos nuevos y generar cobros con la garantía correspondiente, así como permitir a los arrendatarios pagar sus primeros meses de arriendo de forma adelantada (lo que el cliente señala como un suceso común).

La acción de cerrar contrato se implementa con la lógica de quitar la asociación de contrato activo a las entidades que estaban en el contrato, cambiar el estado de éste a cerrado (habiendo validado anteriormente que estuviese en vigencia) y marcar todos los cobros no pagados como anulados.

Se crea el controlador para listar cobros, permitiendo filtrar por contrato y estado mediante *query params*. Aparte a este se crea la acción para listar cobros pendientes según un rut de arrendatario, para lo cual se busca la existencia de algún arrendatario con este dato y en caso de existir, se buscan sus cobros con estado pendiente, devolviendo el resultado de esta *query*. En ambas acciones se implementa además la lógica para el cálculo de los intereses del cobro. En vez de revisar día a día si es que corresponde sumarle un día de penalización a cada cobro, esto se calcula solamente cuando se accede al recurso desde la base de datos, independiente de en qué sección de la plataforma se realiza este acceso. Entonces, un cobro puede mantenerse estático por un tiempo, pero apenas se accede a éste en la aplicación (ya sea directamente o anidado en una relación) se calcula y aplica el interés correspondiente si es que no ha sido calculado para el día actual con anterioridad. De esta forma se evita tener una acción recurrente y se delega esta lógica al momento en que el usuario debe interactuar con este.

Para calcular la generación de intereses, primero se revisa que éste se encuentre en estado pendiente y no haya sido excluido del cálculo de intereses y a continuación se calculan los días de atraso con respecto a la fecha de vencimiento. De haber un atraso, se revisa si ya existe el recargo por intereses con anterioridad para editarlo o crearlo según sea necesario. El monto se calcula como el 1% del monto de arriendo, multiplicado por tantos días de atraso haya.

Se implementan a continuación los *viewsets* de recargo y descuento de cobro, implementando para ambos lógica que evita su creación/edición/eliminación si es que su cobro asociado o por asociar no se encuentra en estado pendiente.

El controlador para la subida de comprobantes de pago, así como para documentos de contrato funcionan de manera similar: ambos reciben los datos del archivo en formato binario y lo almacenan en la aplicación, con la ligera diferencia que para los documentos de contrato se almacena además el nombre original del archivo subido, mientras que para el comprobante se genera un identificador aleatorio. Cabe notar que esta es la solución para el desarrollo local, y que en un ambiente de producción real esto debe reemplazarse por algún método de almacenamiento escalable (por ejemplo: S3 de *Amazon Web Services*). De igual manera se implementan los controladores para descargar ambos tipos de archivo, los cuales retornan una URL que el frontend utiliza para realizar la descarga.

Para el documento de contrato se crean además una serie de controladores para realizar la acción de subir un documento a un contrato específico en un solo paso obteniendo el ID del contrato desde la URL asociada al controlador, uno para eliminar un documento específico y uno para listar los documentos de un contrato. Todos estos son usados en la administración de un contrato en vigencia.

Existiendo ya la posibilidad de subir comprobantes de pago se implementa la acción de realizar un pago manual. Para esto se crean dos controladores: uno para que un arrendatario realice el pago, y otro para que un usuario de administración lo haga. Ambos funcionan bajo la misma lógica: se valida que el cobro se encuentre en estado pendiente y se cambia éste a en revisión. Se crea el pago marcándolo como tipo manual y asociando el comprobante correspondiente (subido con anterioridad). Para el pago desde administración se agrega además el crear una observación para esta acción, evidenciado así que se realizó un pago por fuera del conducto normal. Finalmente se envía al arrendatario un correo avisando que su pago ha pasado a revisión.

Se implementan también las acciones realizables sobre el cobro una vez ha sido pasado a revisión: aprobar, observar, devolver a “pendiente” y marcar como conflictivo. Todas funcionan de manera similar: se revisa que el cobro esté en el estado correcto para realizar la acción y luego actualizarlo, y luego se crea una observación para esa acción con el fin de mantener el *timeline* de acciones sobre el pago. El flujo posible para un cobro se ilustra en la figura 34.

Las diferencias radican en que, para la acción de aprobar se marcan como completados los pagos que estuvieran pendientes para ese cobro; para marcar como conflictivo los pagos pasan a estar en estado “en revisión”, para devolver a pendiente estos pasan a estar rechazados, y para observar no se realiza ninguna acción sobre estos. Además, para cada una de estas acciones se envía un correo al arrendatario notificando el cambio de estado.



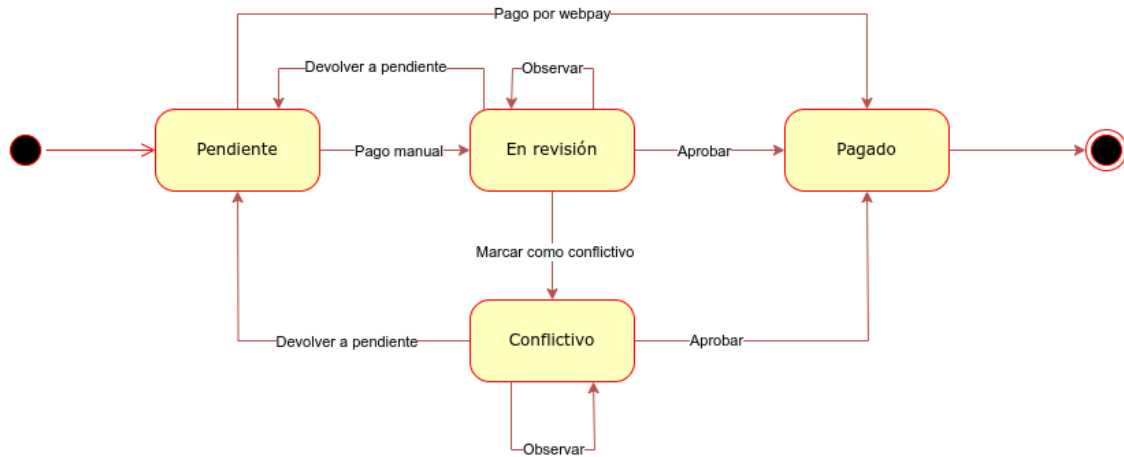


Figura 34. Estados de un cobro.

Para el pago mediante *Webpay* se crean dos controladores: uno se encarga de crear el link de pago según los datos de un cobro mediante el SDK de *Transbank* [8], que permite usar métodos descriptivos para comunicarse con su API sin tener que escribir las *requests* manualmente. Se crea una transacción pasando a este método el monto y la URL de retorno que será usada una vez se termine la transacción para volver a la plataforma de Efirent.

El segundo controlador corresponde a la verificación de la transacción: se recibe un *token* que identifica a ésta y se consulta nuevamente mediante el SDK su estado, enviando esta respuesta al frontend para que se le muestre al usuario la vista correspondiente.

Aparte de los controladores, en el archivo *tasks.py* se definen las tareas para enviar correos según los cambios de estado. Cada vez que se hace alguna de estas acciones el correo se encola para ser enviado cuando sea posible. En un ambiente de desarrollo local esto es instantáneo pero en un ambiente de producción con una gran cantidad de usuarios puede demorar unos minutos.

Se agregan también en el archivo de *tasks* las tareas recurrentes para enviar mensajes de recordatorio a los arrendatarios según la cantidad de días faltantes para el vencimiento de su siguiente cobro. Se implementan avisos 5 días antes, 1 día antes, y luego un aviso diario por cada día de atraso. Esto se realiza todos los días a las 12:00 PM.

Para enviar estos avisos se utiliza la *Whatsapp Business Platform* de *Meta* [9] la cual entrega acceso a una API para enviar mensajes por *Whatsapp*. Para poder enviar un

mensaje primero es necesario subir la plantilla del mensaje para revisión, y una vez es aprobada se permite entonces enviar mensajes con esa plantilla (Figura 35).

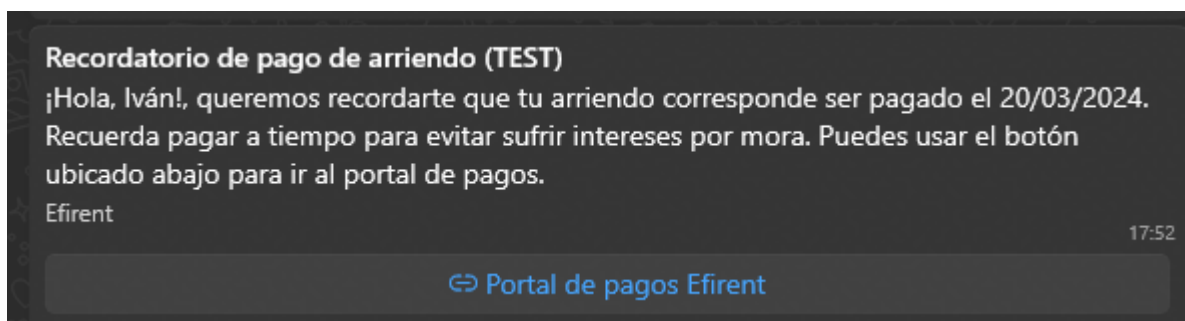


Figura 35. Mensaje de prueba para el envío de mensajes de *Whatsapp*.

Adicionalmente se crea la tarea recurrente para generar los cobros correspondientes al mes siguiente, para cada contrato vigente, lo que se realiza todos los días 10 de cada mes a las 6:00 AM. Esta tarea obtiene el mes y año actual, calcula cuál es el mes/año que le sigue y, para cada contrato vigente, revisa si es que existe ya un cobro para tal mes. Si no existe, entonces genera el cobro correspondiente para ese mes.

#### 4.3.3. Aplicación de monedas

Esta aplicación sigue al igual que las anteriores la estructura estándar de *Django*. Se utiliza exclusivamente para almacenar el valor diario del cambio de UF en un lugar de fácil acceso, permitiendo contar con esta información de manera inmediata en el resto del proyecto.

Se implementa un método, utilizado por el resto de aplicaciones del sistema, que pide el valor de conversión de UF a CLP para un día específico. Este valor se busca primero en base de datos, y de no encontrarse, se utiliza la API de *mindicador.cl*<sup>26</sup> para obtener el valor, el cual es entonces almacenado para su posterior uso. Así, la primera vez que se consulta el valor de la UF para un día, se debe conseguir de una fuente externa, pero todas las consultas futuras pueden utilizar el valor almacenado anteriormente, evitando *requests* innecesarios.

#### 4.3.4. Manejo de usuarios

Los usuarios de la plataforma fueron representados a través de un modelo simple de *User* con correo y contraseña. Por el momento no se implementa un sistema de roles, por lo que se distingue entre usuarios de administración y arrendatarios mediante restricciones en los *endpoints* de la API.

---

<sup>26</sup> <https://mindicador.cl>

Si un arrendatario intenta acceder a cualquiera de las vistas del sistema que no le corresponde, éste lo hará sin un usuario y por ende el sistema no enviará el *token* que la mayoría de las vistas requieren para autenticar. Solo una pequeña colección de *endpoints* pensados para uso de arrendatarios son públicos y permiten acceso sin una cuenta.

Por el momento, la creación de cuentas es a mano mediante las herramientas de desarrollo que entrega *Django*. Se identifica como una mejora a futuro, el implementar un sistema de roles y para permitir que arrendatarios posean cuentas con permisos distintos a aquellos para los usuarios de administración.

En los *mockups* de la interfaz descritos en el Anexo A se hace alusión a un sistema de códigos de autorización para permitir el realizar ciertas acciones a usuarios con permisos distintos, así como en la sección de objetivos específicos del proyecto. La idea de esta implementación se abandona luego de que el cliente identifica como un problema menor para el sistema. Si bien era un objetivo mencionado en la planificación, luego se dejó fuera del alcance del proyecto por solicitud del cliente.

## 4.4. Descripción de la API

La API cuenta con los siguientes *endpoints*, divididos por *app* de *Django*:

### 4.4.1. Aplicación de propiedades

Todos los *endpoints* de esta *app* comienzan con “*properties*”. A continuación se muestran por racks, según el objeto de datos al cual acceden:

- *GET /inmobiliarias*: retorna un listado de inmobiliarias (*status 200*).
- *GET /inmobiliarias/<id>*: retorna la inmobiliaria con el id específico (*status 200*).
- *POST /inmobiliarias*: crea una inmobiliaria (*status 201*).
- *PUT/PATCH /inmobiliarias/<id>*: edita la inmobiliaria con el id específico (*status 200*).
- *DELETE /inmobiliarias/<id>*: elimina la inmobiliaria con el id específico (*status 204*).
  
- *GET /edificios*: retorna un listado de edificios (*status 200*).
- *GET /edificios/<id>*: retorna el edificio con el id específico (*status 200*).
- *POST /edificios*: crea un edificio (*status 201*).
- *PUT/PATCH /edificios/<id>*: edita el edificio con el id específico (*status 200*).
- *DELETE /edificios/<id>*: elimina el edificio con el id específico (*status 204*).
  
- *GET /propietarios*: retorna un listado de propietarios (*status 200*).
- *GET /propietarios/<id>*: retorna el propietario con el id específico (*status 200*).

- *POST /propietarios*: crea un propietario (*status 201*).
- *PUT/PATCH /propietarios/<id>*: edita el propietario con el id específico (*status 200*).
- *DELETE /propietarios/<id>*: elimina el propietario con el id específico (*status 204*).
  
- *GET /propiedades*: retorna un listado de propiedades (*status 200*). Recibe *query param "inmobiliaria"* para filtrar por la inmobiliaria de la propiedad.
- *GET /propiedades/arrendables*: retorna la lista de propiedades disponibles para arriendo (*status 200*).
- *GET /propiedades/<id>*: retorna la propiedad con el id específico (*status 200*).
- *POST /propiedades*: crea una propiedad (*status 201*).
- *PUT/PATCH /propiedades/<id>*: edita la propiedad con el id específico (*status 200*).
- *DELETE /propiedades/<id>*: elimina la propiedad con el id específico (*status 204*).
  
- *GET /estacionamientos*: retorna un listado de estacionamientos (*status 200*).
- *GET /estacionamientos/arrendables*: retorna la lista de estacionamientos disponibles para arriendo (*status 200*).
- *GET /estacionamientos/<id>*: retorna el estacionamiento con el id específico (*status 200*).
- *POST /estacionamientos*: crea un estacionamiento (*status 201*).
- *PUT/PATCH /estacionamientos/<id>*: edita el estacionamiento con el id específico (*status 200*).
- *DELETE /estacionamientos/<id>*: elimina el estacionamiento con el id específico (*status 204*).
  
- *GET /bodegas*: retorna un listado de bodegas (*status 200*).
- *GET /bodegas/arrendables*: retorna la lista de bodegas disponibles para arriendo (*status 200*).
- *GET /bodegas/<id>*: retorna la bodega con el id específico (*status 200*).
- *POST /bodegas*: crea una bodega (*status 201*).
- *PUT/PATCH /bodegas/<id>*: edita la bodega con el id específico (*status 200*).
- *DELETE /bodegas/<id>*: elimina la bodega con el id específico (*status 204*).
  
- *GET /documentos-propiedad/<id>*: retorna el documento de propiedad con el id específico (*status 200*).
- *POST /documentos-propiedad/upload*: sube un documento de propiedad (*status 201*).
- *DELETE /documentos-propiedad/<id>*: elimina el documento de propiedad con el id específico (*status 204*).

#### 4.4.2. Aplicación de contratos

Todos los *endpoints* de esta app comienzan con “*contracts/*”. De manera similar al anterior, a continuación se detallan los endpoints por tracks.

- *GET /arrendatarios*: retorna un listado de arrendatarios (*status 200*).
- *GET /arrendatarios/sin-contrato*: retorna un listado de arrendatarios sin contrato de arriendo activo (*status 200*).
- *GET /arrendatarios/cobros-pendientes*: retorna un listado de cobros pendientes, según el *query param* “*rut*” del arrendatario (*status 200*).
- *GET /arrendatarios/<id>*: retorna el arrendatario con el id específico (*status 200*).
- *POST /arrendatarios*: crea un arrendatario (*status 201*).
- *PUT/PATCH /arrendatarios/<id>*: edita el arrendatario con el id específico (*status 200*).
- *DELETE /arrendatarios/<id>*: elimina el arrendatario con el id específico (*status 204*).
  
- *GET /contratos*: retorna un listado de contratos (*status 200*).
- *GET /contratos/<id>*: retorna el contrato con el id específico (*status 200*).
- *GET /contratos/<id>/cobros*: retorna los cobros del contrato con el id específico (*status 200*). Recibe *query param* “*estado*” para filtrar por estado del cobro.
- *GET /contratos/<id>/documentos-contrato*: retorna los documentos del contrato con el id específico (*status 200*).
- *POST /contratos*: crea un contrato (*status 201*).
- *POST /contratos/pasar-a-vigente*: pasa un contrato a estado vigente (*status 200*).
- *POST /contratos/cerrar*: cierra un contrato vigente (*status 200*).
- *POST /contratos/<id>/upload-documento-contrato*: sube un documento al contrato con el id específico (*status 201*).
- *PUT/PATCH /contratos/<id>*: edita el contrato con el id específico (*status 200*).
- *DELETE /contratos/<id>*: elimina el contrato con el id específico (*status 204*).
  
- *POST /recargos*: crea un recargo de cobro (*status 201*).
- *PUT/PATCH /recargos/<id>*: edita el recargo de cobro con el id específico (*status 200*).
- *DELETE /recargos/<id>*: elimina el recargo de cobro con el id específico (*status 204*).
  
- *POST /descuentos*: crea un descuento de cobro (*status 201*).
- *PUT/PATCH /descuentos/<id>*: edita el descuento de cobro con el id específico (*status 200*).
- *DELETE /descuentos/<id>*: elimina el descuento de cobro con el id específico (*status 204*).

- *GET /documentos-contrato/<id>*: retorna el documento de contrato con el id específico (*status 200*).
- *POST /documentos-contrato/upload*: sube un documento de contrato (*status 201*).
- *PUT/PATCH /edificios/<id>*: edita el edificio con el id específico (*status 200*).
- *DELETE /documentos-contrato/<id>*: elimina el documento de contrato con el id específico (*status 204*).
  
- *GET /cobros*: retorna un listado de cobros (*status 200*). Permite *query params* “contrato” y “estado” para filtrar por estos campos.
- *POST /cobros/<id>/pagar*: realiza un pago manual para el cobro (*status 200*).
- *POST /cobros/<id>/aprobar*: aprueba un cobro y lo marca como pagado (*status 200*).
- *POST /cobros/<id>/marcar-conflictivo*: marca un cobro como conflictivo (*status 200*).
- *POST /cobros/<id>/observar*: realiza una observación sobre un cobro (*status 200*).
- *POST /cobros/<id>/devolver-a-pendiente*: devuelve un cobro a pendiente (*status 200*).
- *POST /cobros/<id>/pago-admin*: realiza un pago manual para el cobro desde el panel de administración (*status 200*).
- *POST /cobros/<id>/generar-link-pago*: genera un link de pago de *Webpay* para el cobro (*status 200*).
- *POST /cobros/<id>/confirmar-pago/*: confirma el estado del pago en *Webpay* para el cobro (*status 200*).
- *PUT/PATCH /edificios/<id>*: edita el edificio con el id específico (*status 200*).
- *DELETE /edificios/<id>*: elimina el edificio con el id específico (*status 204*).
  
- *GET /comprobantes-pago/<id>*: retorna el comprobante de pago con id específico (*status 200*).
- *POST /comprobantes-pago/upload*: sube un comprobante de pago (*status 201*).

#### 4.4.3. Usuarios

Existe un endpoint utilizado relacionado a usuarios, que es el que se indica a continuación:

- *POST /users/login*: retorna el token de sesión para el usuario que se intenta loguear (*status 200*).

#### 4.4.4. Aplicación de monedas

Existe un endpoint para pedir el cambio de UF, el cual es el siguiente:

- *GET /currencies/uf*: retorna el valor de cambio de la UF para un cierto día (*status 200*). Admite un *query param "date"* para elegir la fecha de cambio. Si no se envía por defecto es hoy.

## 5. Evaluación de la solución

La solución fue evaluada por tres usuarios: dos usuarios expertos del sistema legado (trabajadores de Efirent) con amplio conocimientos del negocio y su necesidad, uno de ellos representando al perfil de ejecutivo y el otro al de supervisor de ejecutivos, y un tercer usuario ajeno al sistema representando al perfil de arrendatario. A cada usuario se le realizó una exposición de las funcionalidades de la plataforma (pertinentes a su perfil de usuario) seguida de un ejercicio en donde se pidió al usuario realizar diversas tareas. Luego de este ejercicio, se les aplicaron las encuestas mencionadas en la sección 1.4.

### 5.1. Usuarios expertos

A ambos usuarios expertos se les realizó la misma evaluación debido a que actualmente no existen diferencias en el funcionamiento de la plataforma para ambos perfiles.

#### 5.1.1. Ejercicio realizado

Se le pidió a los usuarios realizar una serie de tareas que recorrieran el flujo completo de la plataforma. En primer lugar, se les solicitó iniciar sesión con un perfil de usuario de administración y crear un contrato de arriendo para una propiedad, con un estacionamiento y una bodega. Esta operación involucra la creación de todas las entidades necesarias para esto (inmobiliaria, edificio, torres propiedad, propietario y arrendatario).

Una vez creado el contrato, se les pide ingresar como un arrendatario y realizar un pago manual. Luego, se pide volver a ingresar como administrador y mandar el pago a “conflictivo” (marcarlo como conflictivo), observarlo (agregar observaciones), devolverlo a “pendiente”, pagarlo nuevamente, y finalmente aprobarlo. Seguido de esto se pide realizar un pago por webpay (como arrendatario).

Para finalizar, se les pide a los usuarios (como administradores) subir y descargar documentos de una propiedad, y luego realizar lo mismo, pero con documentos requeridos para el contrato de la nueva propiedad.

#### 5.1.2. Evaluación de usabilidad

La evaluación de la usabilidad de la solución se realizó luego de llevar a cabo el ejercicio antes descrito. Esta evaluación implica el uso de la encuesta SUS (System Usability Scale), que cuenta de los 10 ítems mostrados en la Tabla 1. En cada ítem de



la encuesta el usuario debe indicar su nivel de acuerdo, usando una escala de Likert de 5 puntos, donde 1 es “totalmente en desacuerdo” y el 5 “totalmente de acuerdo”. Los ítems están puestos de forma intercalada, en sentido positivo y negativo, para no influenciar las respuestas del evaluador. Los resultados para cada usuario se muestran en las tablas 1 y 2.

Tabla 1. Enunciados de la encuesta SUS para el usuario ejecutivo.

Enunciado	1	2	3	4	5
Creo que me gustaría utilizar este sistema con frecuencia.				X	
Encontré el sistema innecesariamente complejo.	X				
Pensé que el sistema era fácil de usar.				X	
Creo que necesitaría el apoyo de un técnico para poder utilizar este sistema.		X			
Encontré que las diversas funciones de este sistema estaban bien integradas.				X	
Pensé que había demasiada inconsistencia en este sistema.	X				
Me imagino que la mayoría de la gente aprendería a utilizar este sistema muy rápidamente.				X	
Encontré el sistema muy complicado de usar.	X				
Me sentí muy seguro usando el sistema			X		
Necesitaba aprender muchas cosas antes de empezar con este sistema.			X		

Siguiendo el procedimiento para calcular el resultado final de la encuesta, se debe realizar lo siguiente:

Suma de los enunciados impares:  $S_i = 19$

Suma de los enunciados pares:  $S_p = 8$

Luego se aplica la siguiente fórmula:

$$2.5 \cdot ((S_i - 5) + (25 - S_p)) = 77.5$$

Se obtiene entonces que el resultado es de 77.5 puntos de 100 posibles para el usuario de perfil ejecutivo.

Tabla 2. Enunciados de la encuesta SUS para el usuario supervisor.

Enunciado	1	2	3	4	5
Creo que me gustaría utilizar este sistema con frecuencia.				X	
Encontré el sistema innecesariamente complejo.	X				
Pensé que el sistema era fácil de usar.					X
Creo que necesitaría el apoyo de un técnico para poder utilizar este sistema.	X				
Encontré que las diversas funciones de este sistema estaban bien integradas.				X	
Pensé que había demasiada inconsistencia en este sistema.		X			
Me imagino que la mayoría de la gente aprendería a utilizar este sistema muy rápidamente.					X
Encontré el sistema muy complicado de usar.	X				
Me sentí muy seguro usando el sistema				X	
Necesitaba aprender muchas cosas antes de empezar con este sistema.		X			

Siguiendo el procedimiento para calcular el resultado final de la encuesta, se debe realizar lo siguiente:

Suma de los enunciados impares:  $S_i = 22$

Suma de los enunciados pares:  $S_p = 7$

Luego se aplica la siguiente fórmula:

$$2.5 \cdot ((S_i - 5) + (25 - S_p)) = 87.5$$

Se obtiene entonces que el resultado es de 87.5 puntos de 100 posibles para el usuario de perfil supervisor.

Esto arroja una puntuación de usabilidad promedio entre ambos usuarios de 82.5 puntos de 100 posibles.

Según la literatura, el puntaje de usabilidad mínimo promedio para páginas web (medida con SUS) corresponde a 68 puntos. Esto indica que el nuevo sistema está por sobre el promedio en términos de usabilidad, aunque también se reconoce que el número de usuarios que evaluaron la aplicación debe ser aumentado para sacar conclusiones más duraderas.

Se conoce que un puntaje de 75 significa que el resultado está en el percentil 73<sup>27</sup>, lo que quiere decir que la solución fue mejor evaluada que al menos el 73% de las evaluaciones realizadas con esta escala (que se conocen). Esto quiere decir que, en términos de usabilidad, el sistema implementado recibe una valoración favorable.

Es importante notar, de todos modos, que los usuarios que recibieron esta encuesta corresponden a usuarios expertos, por lo que sus apreciaciones de la facilidad de uso pueden resultar sesgadas en comparación a otros usuarios. De todos modos, este resultado es esperable, pues el sistema apunta a un nicho de usuarios con un conocimiento de su proceso actual y una dolencia identificada. Debido a eso, la curva de aprendizaje para estos usuarios es menor de la que podría ser para un usuario externo (por ejemplo un arrendatario). Es importante tener esto en cuenta a futuro, pues si bien en el presente los usuarios de la plataforma corresponden en su mayoría a usuarios expertos (empleados de Efirent), esto puede cambiar en el futuro con la llegada de nuevo personal.

También es importante notar que la evaluación del usuario considera el proceso de pagos que en la práctica no sería realizado por un usuario de administración, lo que nuevamente puede introducir un sesgo en la evaluación, pues un arrendatario no siempre va a poseer el nivel de conocimiento del sistema y familiaridad con el uso de plataformas digitales que posee el usuario experto. Es debido a esto que se evalúa también el sistema con un usuario externo a la plataforma.

### 5.1.3. Evaluación de la utilidad percibida

Para evaluar la utilidad percibida se utiliza una versión reducida del cuestionario de TAM (*Technology Acceptance Model*) [2], manteniendo solo las preguntas pertinentes a la utilidad percibida. La evaluación de esta dimensión del software involucra cuatro ítems, donde el evaluador indica su nivel de acuerdo utilizando una escala Likert de 7 puntos: 1 - totalmente en desacuerdo, y 7 - totalmente de acuerdo:

Los ítems para evaluar la utilidad percibida y sus resultados por usuario se muestran en las tablas 3 y 4.

---

<sup>27</sup> <https://measuringu.com/interpret-sus-score/>

Tabla 3. Enunciados de TAM para evaluar la utilidad percibida, con respuestas del usuario ejecutivo.

	1	2	3	4	5	6	7
1. Usar este producto en mi trabajo me permitiría cumplir tareas más rápidamente.							X
2. Usar este producto mejoraría mi desempeño en el trabajo.						X	
3. Usar este producto en el trabajo mejoraría mi productividad.					X		
4. Usar este producto mejoraría mi efectividad en el trabajo.					X		
5. Usar este producto haría más fácil mi trabajo.							X
6. Encontraría útil este producto en mi trabajo.							X

El valor final de la utilidad percibida se calcula como el promedio simple entre los ítems [11], lo que para este set de respuestas corresponde a aproximadamente 6,1. Un valor por encima de 4 indica que el sistema es percibido como útil.

Tabla 4. Enunciados de TAM para evaluar la utilidad percibida, con respuestas del usuario supervisor.

	1	2	3	4	5	6	7
1. Usar este producto en mi trabajo me permitiría cumplir tareas más rápidamente.							X
2. Usar este producto mejoraría mi desempeño en el trabajo.						X	
3. Usar este producto en el trabajo mejoraría mi productividad.						X	
4. Usar este producto mejoraría mi							X

efectividad en el trabajo.							
5. Usar este producto haría más fácil mi trabajo.							X
6. Encontraría útil este producto en mi trabajo.						X	

Calculando el promedio de los ítems se obtiene un valor de utilidad de 6,5. Al igual que para el usuario ejecutivo, este sistema se percibe como útil. De ambas evaluaciones se obtiene un promedio de 6,3, lo que sigue estando por sobre el valor de 4. Esto implica que los usuarios evaluados consideran útil la plataforma, pero es necesario y se recoge como punto de mejora el engrosar la evaluación actual con más usuarios expertos para obtener resultados más concluyentes.

#### 5.1.4. Comparación de tiempos

Como siguiente punto de evaluación se tiene la comparación entre el tiempo requerido para realizar el cuadro de los pagos en el sistema actual y el sistema implementado. Según lo reportado por el cliente, el proceso de cuadro de pagos en la actualidad toma, en promedio, 10 minutos por pago. Esto incluye la recepción del comprobante, verificación de los datos, creación y manipulación de la planilla de cálculo y notificación al cliente. Esto puede aumentar a 20 minutos por pago en periodos con alta actividad, principalmente los últimos y primeros días del mes.

Cada pago con algún problema detectado aumenta la duración promedio del proceso, y en el sistema manual es necesario rastrear el proceso de cuadro a mano, lo que también aumenta la posibilidad de cometer errores.

Para la plataforma implementada, luego de hacer los experimentos con cada uno de los procesos, se obtiene que, para un pago hecho a mano (o sea, el arrendatario sube el comprobante), el tiempo promedio que se dedica a realizar el cuadro es de aproximadamente 2-3 minutos por pago, en el caso en que éste no presente problemas.

Para pagos con información errónea o hechos de manera incorrecta, esta duración aumenta debido a la necesidad de pasar por más estados del proceso. La duración real del problema depende del tiempo que demore el ejecutivo o el arrendatario en solucionar el error, pero la interacción con la plataforma requerida para marcar estas acciones (y poner los cobros en el estado correspondiente) nunca es superior a los 5 minutos. No es necesario ingresar ni mover datos a mano; basta con aplicar las acciones entregadas al usuario, esperar a que la parte correspondiente arregle lo que corresponda, y luego aprobar el pago.

Para un pago con *Webpay* esta duración se reduce a lo que sea que demore el arrendatario en terminar el pago con la plataforma externa, lo que depende completamente del usuario y/o factores externos (conexión, estado de la plataforma de *Webpay*, etc). Si hay algún error con *Webpay* el tiempo para pagar aumenta, pero esta penalización no afecta al proceso del negocio, pues tan solo entra en juego la administración una vez *Webpay* acepta un pago. En este punto, el cuadro de la información es automático, así que no se requiere de tiempo de los usuarios de administración.

Entonces, el principal ahorro de tiempo corresponde a que para la administración ya no es necesario manipular datos de forma manual para manejar un pago. La plataforma ahora ofrece las acciones necesarias, y la estructura requerida para que la información cuadre con su contrato correspondiente. Queda entonces a responsabilidad del usuario solamente descargar el comprobante, confirmar el monto y tomar las acciones correspondientes.

#### 5.1.5. Opiniones de los usuarios

Finalmente, se recoge *feedback* de los usuarios con respecto a la funcionalidad de la plataforma. El usuario ejecutivo menciona los siguientes puntos:

- El sistema funciona bien y la idea es buena, pero le falta completitud para que sea un mayor aporte al negocio. Su potencial para resolver los problemas de gran gasto de tiempo es alto.
- La interfaz es fácil de usar y sigue una lógica simple, pero la elección de textos para las acciones a realizar no siempre es la más clara. Hace falta una revisión de estos textos para que su intención y las consecuencias de las acciones queden claramente transmitidas a los usuarios.
- Se considera una mejora deseada el poder ver la información de forma más centralizada, pudiendo desde una entidad ver los datos de sus relaciones de manera inmediata en vez de que estos datos se encuentren solamente en sus vistas correspondientes (se intenta aplicar este *feedback* en vistas como el detalle de propiedad o la administración de contrato).

El usuario supervisor menciona los siguientes puntos:

- El sistema cumple con todo lo esperado y se encuentra conforme con el funcionamiento del sistema de pagos.
- Las interfaces de listado de elementos son simples de utilizar pero podrían verse mejoradas con filtros para cada tabla. Los ya existentes (por texto y en el caso de las propiedades, por inmobiliaria) funcionan bien pero sería valioso poder filtrar por otros atributos.

- El sistema de notificaciones por correo funciona bien, y se considera deseable además notificar los pagos a la administración, ya sea con mensajes destinados a eso o agregando en copia a un correo de administración todo correo enviado por plataforma.
- Como pasos siguientes se identifica agregar reportería a la plataforma como el más importante, seguido de un sistema de permisos por perfiles de usuario. Además se hace énfasis en el proceso futuro de integración de la plataforma con lo ya existente.

## 5.2. Usuario externo

Para representar al perfil de usuario de arrendatario se realizó la evaluación de la plataforma con un usuario externo al negocio y sin conocimientos previos del funcionamiento de éste.

### 5.2.1. Ejercicio realizado

Se le pidió al usuario realizar una serie de tareas que representan la interacción de un arrendatario con la plataforma. En primer lugar se le solicitó buscar sus pagos pendientes en el portal de arrendatarios, para luego realizar un pago manual (con subida de comprobante) del primero de la lista. Éste pago es luego rechazado, y se le pide identificar la razón del rechazo y realizar nuevamente un intento de pago manual, el cual es prontamente aprobado.

A continuación se requirió que el usuario realice un pago para el siguiente mes disponible, pero esta vez mediante *Webpay*, primero ingresando mal los datos para causar una transacción rechazada, y luego reintentando el pago con los datos correctos. Durante todo este proceso el encuestador toma el rol de administración, rechazando/aprobando los pagos cuando es necesario.

### 5.2.2. Evaluación de usabilidad

Al usuario externo se le realizó únicamente la evaluación de usabilidad, debido a que los enunciados de la evaluación de utilidad van enfocados al uso de la plataforma en un contexto de administración/laboral y no corresponden al uso del usuario externo. Los resultados del usuario externo se exponen en la tabla 5.

Tabla 5. Enunciados de la encuesta SUS para el usuario externo.

Enunciado	1	2	3	4	5
Creo que me gustaría utilizar este sistema con			X		

frecuencia.					
Encontré el sistema innecesariamente complejo.	X				
Pensé que el sistema era fácil de usar.				X	
Creo que necesitaría el apoyo de un técnico para poder utilizar este sistema.	X				
Encontré que las diversas funciones de este sistema estaban bien integradas.					X
Pensé que había demasiada inconsistencia en este sistema.	X				
Me imagino que la mayoría de la gente aprendería a utilizar este sistema muy rápidamente.				X	
Encontré el sistema muy complicado de usar.	X				
Me sentí muy seguro usando el sistema				X	
Necesitaba aprender muchas cosas antes de empezar con este sistema.	X				

Siguiendo el procedimiento para calcular el resultado final de la encuesta, se debe realizar lo siguiente:

Suma de los enunciados impares:  $S_i = 20$

Suma de los enunciados pares:  $S_p = 5$

Luego se aplica la siguiente fórmula:

$$2.5 \cdot ((S_i - 5) + (25 - S_p)) = 87.5$$

Se obtiene entonces que el resultado es de 87.5 puntos de 100 posibles para el usuario externo. Esto igualmente se ubica por sobre el mínimo de 68 puntos para un sistema considerado usable, pero se identifica igualmente que con la evaluación de expertos que es necesario un número mayor de usuarios encuestados para obtener resultados concluyentes.

### 5.2.3. Opinión del usuario

El *feedback* recogido del usuario externo fue el siguiente:



- La plataforma es sencilla de ocupar. El flujo para realizar el pago es directo, simple de acceder y claro en sus resultados.
- Es bueno que la interfaz se adecue a distintos tamaños de pantalla, pues es mucho más probable que este proceso lo realice en un dispositivo móvil en vez de en un computador.
- El tamaño del texto puede ser un problema para las personas con problemas de visión.

## 6. Conclusiones y trabajo a futuro

Este trabajo de título se desarrolló como un proyecto para Efirent, una empresa que ofrece el servicio de administración inmobiliaria a dueños de propiedades. Entre sus labores se encuentra la captación de arrendatarios, manejo de contratos de arriendo y sus pagos mensuales. Su sistema actual (legado) consiste en un proceso completamente manual basado en hojas de cálculo.

La naturaleza manual del proceso tiene como consecuencia que, para recibir los pagos de arriendo, se debe hacer un cuadro de los comprobantes recibidos con los contratos correspondientes. Actualmente, no se cuenta con soporte para plataformas de pago que simplifiquen el proceso, tanto para arrendatarios como para los ejecutivos de Efirent. Esto resulta en una operación con un alto costo en tiempo, sobre todo en periodos de alta actividad (fin y principio de mes).

Dado lo anterior, el objetivo general del trabajo de memoria consistió en desarrollar una plataforma que permita manejar la recepción de pagos, permitiendo asociar estos directamente a sus contratos y entregando acceso a una plataforma externa de pagos para automatizar su realización. Éste fue cumplido en su totalidad, y la mayoría de los objetivos específicos fueron alcanzados satisfactoriamente, con la excepción del manejo de usuarios y sus roles/permisos asociados, el cual tuvo que ser simplificado para priorizar otras secciones de la plataforma.

Para cumplir estos objetivos se desarrolló un sistema que consiste de una aplicación de *frontend* y una de *backend*. La de *frontend* se desarrolló en el *framework* de *JavaScript Vue*, usando además el *framework Quasar* para obtener componentes útiles para la interfaz. Esta aplicación cuenta con diversas interfaces para el ingreso y visualización de datos de propiedades, contratos y el manejo de cobros. Dentro de ésta se incluye el portal de arrendatarios, en donde un inquilino puede realizar el pago de su arriendo manualmente o mediante la integración con *Webpay*.

El *backend* del sistema corresponde a una aplicación de *Django (python)* con la extensión de *Django Rest Framework* para dar soporte a desarrollar una *API REST*. Esta aplicación se encarga del manejo de las entidades ingresadas mediante el *frontend*, incluyendo implementar la lógica detrás de todas las acciones tomadas en la plataforma, la generación de contratos y de cobros, y la comunicación con *Webpay* para la generación y confirmación de pagos.

El sistema implementado fue evaluado por dos usuarios expertos en el proceso de Efirent. Luego de cumplir una serie de tareas dentro de la plataforma, estos expertos fueron encuestados para conocer sus evaluaciones de usabilidad y utilidad percibidas de la plataforma.

Además, se recibió su *feedback* directo sobre las funcionalidades, y se hizo una comparación de tiempos para el cuadro de pagos entre las hojas de cálculo actuales y el sistema implementado. La evaluación entregó resultados positivos, con observaciones que apuntan a la necesidad de continuar el trabajo para obtener un sistema más completo. Particularmente, se identificaron como aspectos de mejora a la claridad en el texto presentado al usuario en ciertas acciones, y la necesidad de crear interfaces que entreguen un acceso más centralizado a la información.

Adicionalmente se evaluó la plataforma con un usuario externo, el cual siguió un proceso simplificado limitado a la sección de arrendatarios y con una encuesta enfocada solo en la usabilidad del sistema, además de recibir *feedback* directo sobre su experiencia. Esta evaluación también resulta positiva, con énfasis en la facilidad de uso de la plataforma y recomendaciones para considerar usuarios con problemas de visión.

Como trabajo a futuro, se considera de prioridad lo necesario para cumplir el objetivo de tener usuarios con roles gobernando lo que pueden hacer en la plataforma. Se considera también importante generar más interfaces en las que recopilar datos de distintas entidades, así como modificar los textos de la aplicación para que sean más claros para los usuarios.

Para alcanzar una mayor completitud en la funcionalidad de la plataforma, se considera de importancia enfocar el desarrollo futuro a finalizar el flujo del pago de arriendos. Además, se requiere implementar el soporte para manejar el pago de este dinero a los propietarios correspondientes, entregando así un flujo “de principio a fin” para el dinero del arriendo. También se debe trabajar en preparar la integración con el resto de plataformas del ecosistema de Efirent, para así aprovechar la principal ventaja de desarrollar una plataforma propia.

# Bibliografía

- [1] Brooke, John. (1995). SUS: A quick and dirty usability scale. *Usability Eval. Ind.* 189.
- [2] Davis, F. D. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly* 13 (3): 319-340, doi:10.2307/249008.
- [3] What is REST - REST API Tutorial. (2022, 7 de Abril). REST API Tutorial. <https://restfulapi.net/>. Último acceso: Marzo de 2024.
- [4] Django. (2023). Django Project. <https://docs.djangoproject.com/en/4.2/>. Último acceso: Marzo de 2024.
- [5] Christie, T. (n.d.). Home - Django REST framework. <https://www.django-rest-framework.org/>. Último acceso: Marzo de 2024.
- [6] MVC - MDN Web Docs Glossary: Definitions of Web-related terms | MDN. <https://developer.mozilla.org/en-US/docs/Glossary/MVC>. Último acceso: Marzo de 2024.
- [7] Kashan Ali, Kim Freimann. (2021). Applying the Technology Acceptance Model to AI decisions in the Swedish Telecom Industry. MBA Thesis, Department of Industrial Economics. Blekinge Tekniska Högskola Forskningsrapport. Sweden.
- [8] Cumbregroup. (s.f.). *tbk. | DEVELOPERS - Referencia Api*. Tbk. | DEVELOPERS. <https://www.transbankdevelopers.cl/referencia/webpay>
- [9] *WhatsApp Business Platform*. Documentation - Meta for Developers. (n.d.). <https://developers.facebook.com/docs/whatsapp/>
- [10] Sauro, J. (s. f.). *Measuring usability with the System Usability Scale (SUS)*. <https://www.userfocus.co.uk/articles/measuring-usability-with-the-SUS.html#:~:text=What%20is%20a%20good%20SUS,through%20a%20process%20called%20normalizing>.
- [11] James R. Lewis. (2019, 30 agosto). Comparison of Four TAM Item Formats: Effect of Response Option Labels and Order - JUX. *JUX - The Journal of User Experience*, Volume 14, Issue 4, August 2019. <https://uxpajournal.org/tam-formats-effect-response-labels-order/>

# Anexos

## Anexo A: Mockups de Interfaces de Usuario

En esta sección se presentan algunos de los mockups desarrollados durante la memoria, y que sirvieron para acordar la apariencia y funcionalidad de los formularios que se debían desarrollar.

### A.1. Diseño de interfaces de usuario

Tal como se indicó antes, se usó el software *Balsamiq*<sup>28</sup> para la creación de mockups que representan la interfaz del sistema a grandes rasgos. A continuación se muestra una selección relevante de estos, que incluyen la vista de datos en forma de tabla, la que se repetirá a lo largo de la plataforma (Fig. 36).

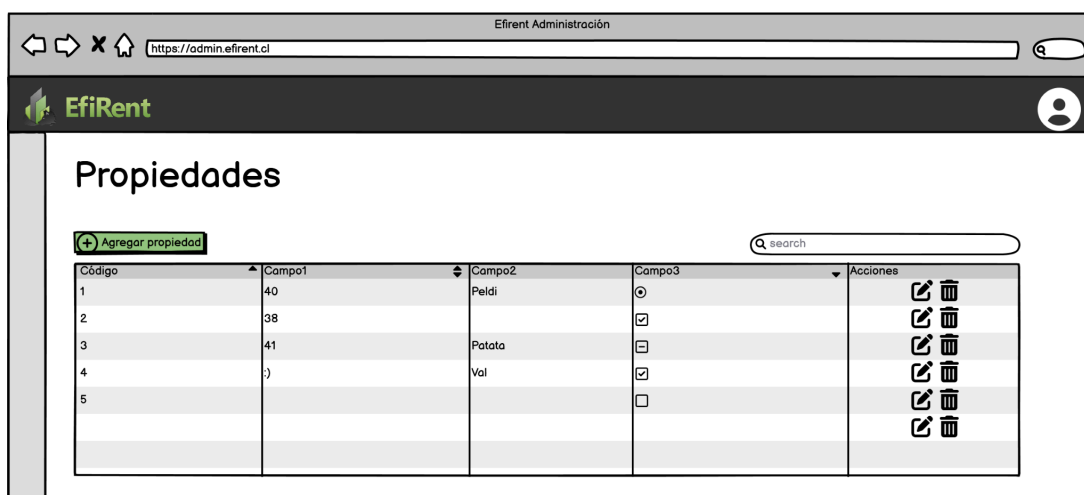


Figura 36. Vista de la tabla de datos (por ej. para mostrar propiedades).

Dentro de esta vista de datos, está la acción para agregar un nuevo dato (Fig. 37), que permite abrir una interfaz para ingresar la información. Cada fila de la tabla tendrá acciones a realizar, las cuales pueden ser permitidas o no dependiendo del perfil de usuario.

<sup>28</sup> <https://balsamiq.com>

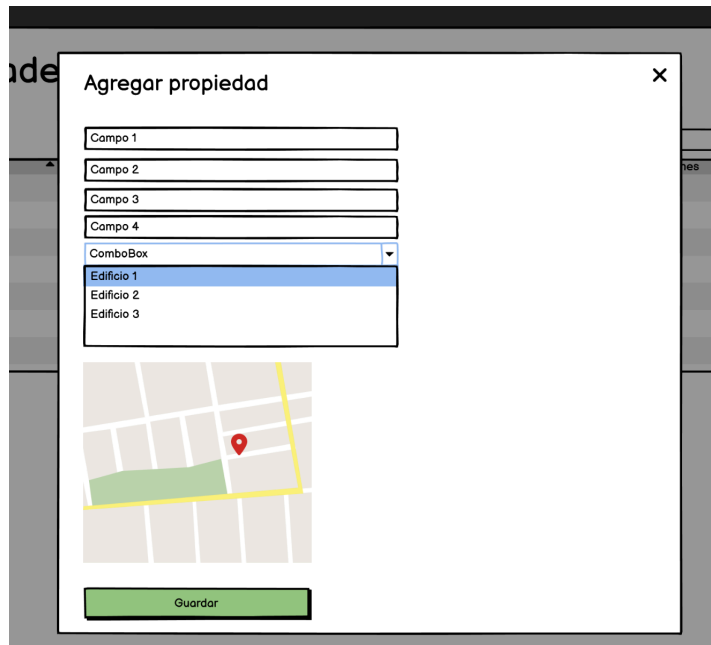


Figura 37. Vista de la acción de agregar un nuevo dato.

En caso de no ser permitidas, se puede mostrar una interfaz indicando que se requiere autorización de un supervisor. Para ello, se debe ingresar un código de autorización, como se aprecia en la figura 38.

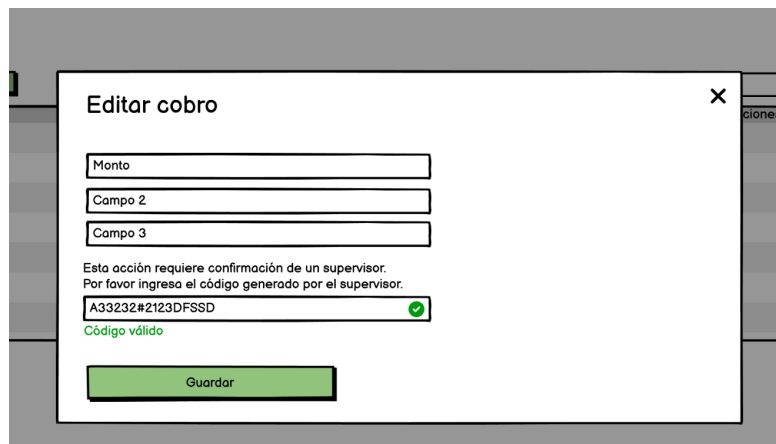


Figura 38. Vista de editar un cobro con autorización satisfactoria.

La generación del código se realiza con una sesión de supervisor, utilizando una interfaz diseñada para esto (Fig. 39 y 40).



Figura 39. Vista de opciones del supervisor.

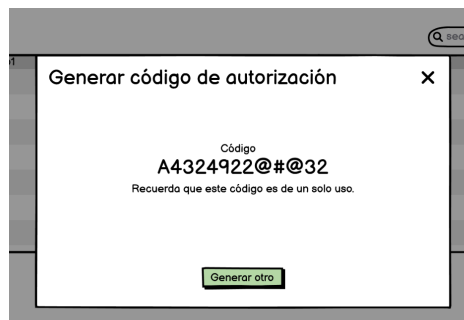


Figura 40. Vista de generación del código de autorización del supervisor.

Para navegar dentro del sistema se usará un menú lateral con las secciones de la aplicación (Fig. 41).

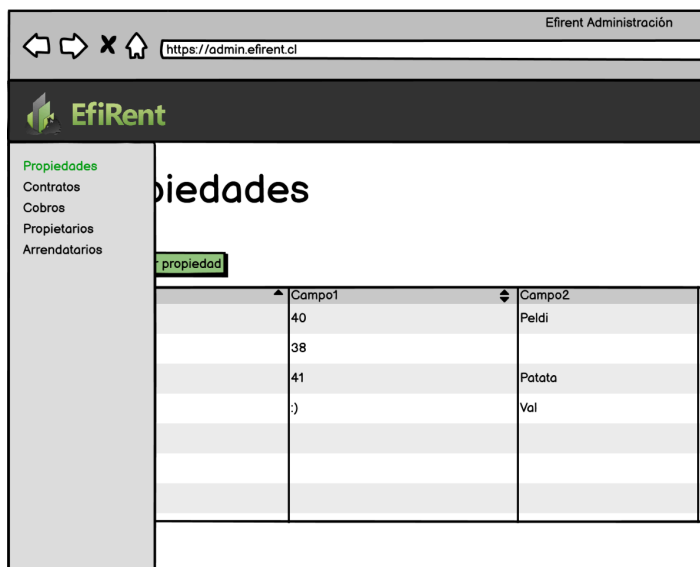


Figura 41. Menú lateral.

Para la vista de los arrendatarios se crearon dos mock-ups, uno en vista de escritorio con un buscador por RUT, y otro para la lista de pagos y su información correspondiente. Al elegir un pago pendiente, se muestran las opciones de pago (Fig. 42).

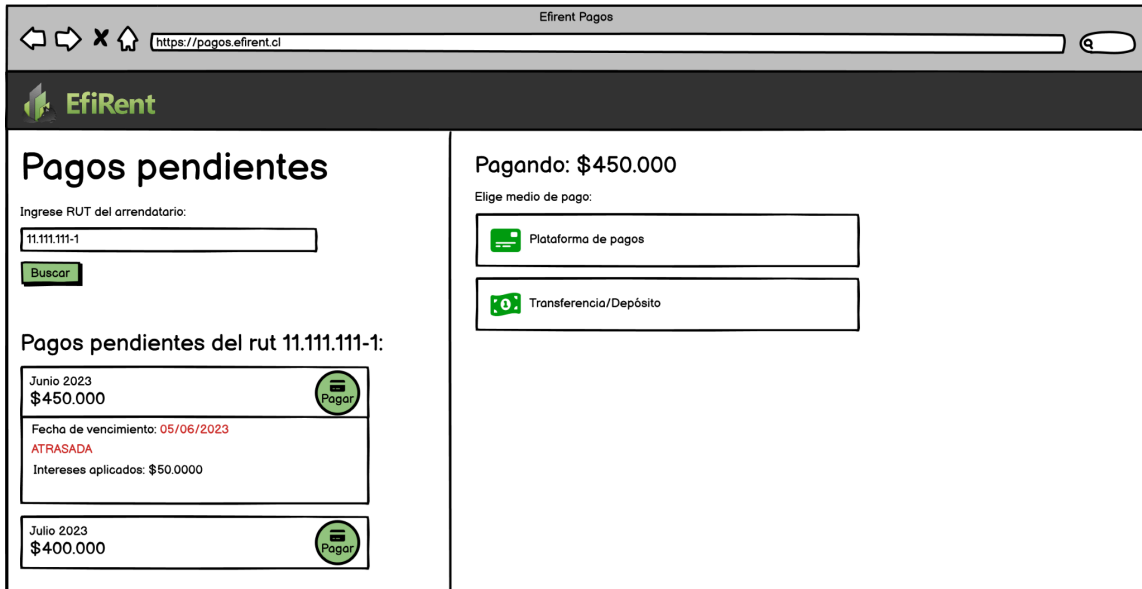


Figura 42. Vista del arrendatario luego de buscar, con un pago atrasado.

Para esta vista en particular se crearon además mockups en dispositivos móviles, pues la interfaz de esta funcionalidad debe facilitar el pago en cualquier dispositivo sin requerir de un computador (Fig. 43).





Figura 43. Versión móvil de la vista del arrendatario.

## Anexo B: Schema de datos

El *schema* de los datos corresponde a la estructura real de la base de datos en el entorno la aplicación. Se utiliza un visualizador de bases de datos para generar el siguiente diagrama:

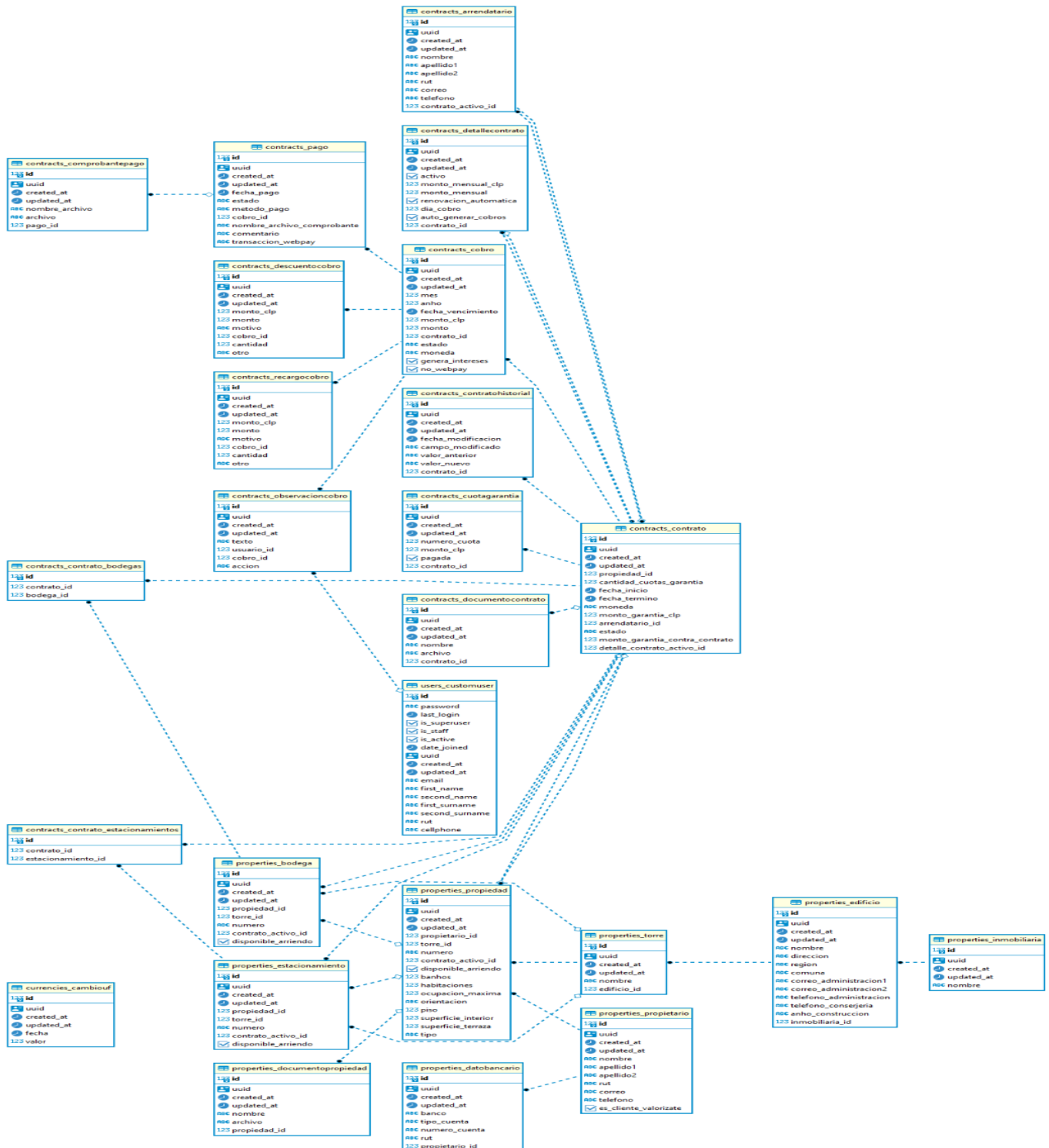


Figura 44. Schema de la base de datos.