UNMDP - Facultad de Ingeniería
Departamento de Informática

Sistema automatizado de riego de kiwi
Subsistema nube

Alumnos:

- Gros, Mariquena Sofía
- Porzio, Pablo

Director:

- Hinojal, Hernán

Co-Director:

- Finochietto, José Mariano

Proyecto final para optar al grado de Ingeniero/a en Informática
Mar del Plata, 31 de octubre de 2022

UNMDP - Facultad de Ingeniería
Departamento de Informática

Sistema automatizado de riego de kiwi
Subsistema nube

Alumnos:

- Gros, Mariquena Sofía
- Porzio, Pablo

Director:

- Hinojal, Hernán

Co-Director:

- Finochietto, José Mariano

Proyecto final para optar al grado de Ingeniero/a en Informática
Mar del Plata, 31 de octubre de 2022

Agradecimientos

A nuestros padres por el cariño, soporte y paciencia dados.
A los docentes por la formación y conocimientos recibidos.
A nuestros compañeros por la colaboración y momentos compartidos.
A Ponce AgTech por la oportunidad y confianza depositadas.

Índice

Resumen

El presente documento describe la realización del proyecto del subsistema nube para un sistema automatizado de riego de kiwi.

Debido a la magnitud global del sistema, el proyecto original se dividió en dos trabajos finales. Por un lado, los alumnos Ivan Aprea y Martín Casas realizaron el subsistema campo. Por otro lado, el subsistema en la nube (aquí descripto) estuvo a cargo de los estudiantes Mariquena Sofía Gros y Pablo Porzio. Aún así, la comunicación entre los grupos fue permanente, especialmente durante las etapas de relevamiento, análisis y diseño.

El sistema global cuenta con una sección en campo, conformada por una unidad de control y el conjunto de bombas y válvulas, y una sección en la nube, encargada de almacenar los datos y gestionar la comunicación remota con el campo. El objetivo global era analizar, diseñar e implementar un sistema basado en Internet de las Cosas (*Internet of Things - IoT*) que le permita al productor de kiwi, tanto de forma local como remota, monitorear y controlar bombas y válvulas, así como establecer programas de riego. De esta manera, se logra reducir la supervisión, registrar eventos en campo, minimizar errores, tomar decisiones y ejecutarlas en el momento.

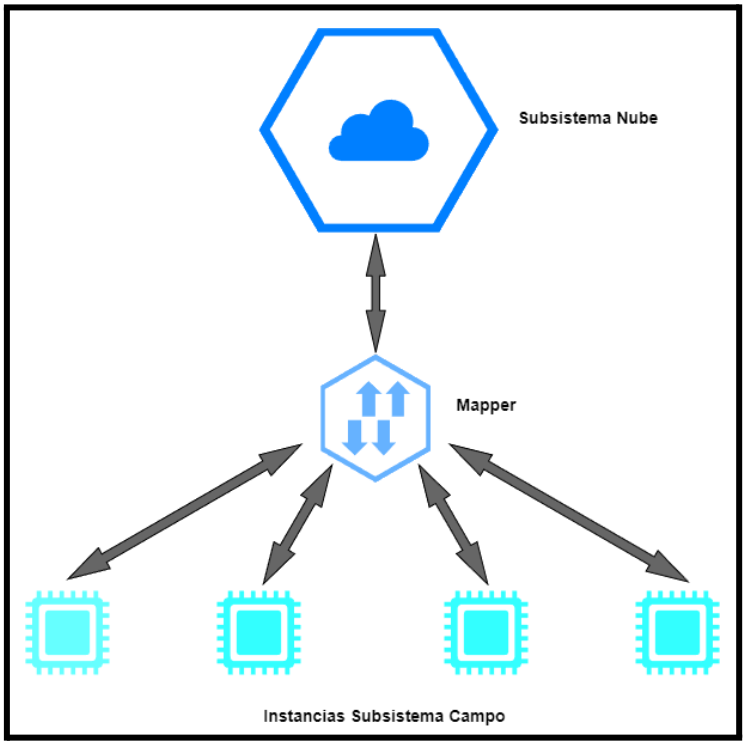

Figura 1 - Vista general del sistema.

Se incluye aquí la descripción del análisis, diseño e implementación realizados en colaboración con la empresa demandante Ponce AgTech. Además se presentan estimaciones de tiempos iniciales, una comparación con la duración real y reflexiones sobre la realización de un proyecto final de grado en equipo, con los puntos positivos y negativos que involucró.

1 - Introducción

Para crecer satisfactoriamente, el kiwi necesita alta humedad ambiental y un régimen de lluvias abundante y relativamente frecuente. El productor recurre a sistemas de riego que permiten conducir el agua mediante una red de tuberías y aplicarla a los cultivos a través de emisores que entregan pequeños volúmenes en forma periódica. El agua se distribuye en forma de goteo por medio de goteros o en forma de lluvia a través de difusores denominados microaspersores y microjets. Actualmente, muchas plantaciones se riegan de forma manual, lo que requiere que operadores en campo prendan y apaguen el sistema en determinados horarios establecidos en un plan de riego por los asesores de riego. Estos horarios pueden ser actualizados según el estado del cultivo y del clima.

Esta modalidad presenta algunos inconvenientes. Primero, no siempre los operadores están disponibles a horario para las tareas involucradas debido a imprevistos que puedan surgir. Esto es importante para cumplir los planes de riego establecidos y que el cultivo crezca saludablemente. También lo es para mitigar el efecto de las heladas y para evitar que el riego funcione durante horarios pico, donde el precio de la electricidad aumenta significativamente con respecto al resto del día.

En segundo lugar, los asesores no pueden controlar que el plan de riego se esté siguiendo correctamente y no tienen información confiable que los ayude a tomar decisiones. Por último, si ocurre algún desperfecto en el sistema, los operadores no tienen forma de darse cuenta hasta que se encuentran en el lugar. Un desperfecto no atendido a tiempo puede provocar un daño en la plantación y un desperdicio de recursos energéticos.

Ponce AgTech es una empresa especializada en el monitoreo de sistemas de riego. Cuenta con un sistema específico para riego de kiwi, pero con escasa funcionalidad y soporte para un sólo campo por instancia. Además, está desactualizado respecto al resto de su línea de productos, por lo que se dificulta su integración con ella. Así surgió su participación como demandante en el proyecto, con el objetivo de tercerizar la actualización del sistema. La nueva versión debe satisfacer tanto las necesidades de los productores como los lineamientos actuales de la organización.

2 - Objetivos del Proyecto

2.1 - Objetivo Global

El objetivo principal es compartido por ambos proyectos: *impactar positivamente en la región*. Localmente, no existen herramientas similares. Tiene un potencial considerable debido a que la mayoría de los productores de kiwi se encuentra en el cinturón frutihortícola de Mar del Plata. Los usuarios se beneficiarán al reducir las rondas de supervisión, minimizar errores de operación en campo, tomar decisiones y ejecutarlas en el momento de forma remota, reducir el gasto energético, y en consecuencia, sus costos.

2.2 - Objetivos Específicos

Los objetivos que se listan a continuación son propios del proyecto del subsistema nube:

- *Diseñar una nueva versión del subsistema que satisfaga los requerimientos funcionales de productores y los no funcionales del demandante*: a partir de la comprensión del dominio, se diseñó una arquitectura en la nube que cubre las necesidades de los productores y que sigue los lineamientos técnicos indicados por Ponce AgTech.

- *Implementar el subsistema*: se construyó un subsistema en la nube que permite a los usuarios gestionar el sistema de riego de manera remota. Además, se utilizaron tecnologías compatibles con los requerimientos no funcionales.

- *Diseñar la solución de forma tal que se facilite su evolución por parte del demandante*: además de construirlo siguiendo los lineamientos de Ponce AgTech, se hizo énfasis en la documentación de la arquitectura, código, Application Programming Interfaces (APIs), etc. De esta manera, se simplifica la continuación del proyecto por parte del equipo de desarrollo del demandante.

2.3 - Alcance

El objetivo global de ambos subsistemas (proyectos finales) es obtener un sistema IoT de riego integrado que le permita al productor, tanto de forma local como remota, monitorear y controlar bombas y válvulas, así como establecer programas de riego.

En particular, el subsistema aquí presentado tiene como finalidad desarrollar la funcionalidad “en la nube” para la gestión remota. No se incluye en este proyecto el monitoreo en el campo (función del otro subsistema), pero se trabajó en forma coordinada para comunicarse correctamente y resolver toda la funcionalidad de forma completa.

Los entregables generados fueron:

- Listado de requerimientos funcionales y no funcionales esperados.
- Diagramas de arquitectura del sistema.
- Documentación de APIs.
- Documentación de tests unitarios del servidor.
- Código fuente de tests unitarios del servidor.
- Código fuente del sistema:
    - Aplicación web para que los usuarios puedan acceder desde cualquier lugar del mundo, y con un diseño compatible tanto con móviles como equipos de escritorio.
    - Aplicación web para administradores del sistema.
    - Servidor encargado de la gestión de múltiples sistemas de riego de manera remota.
    - Servidor intermediario encargado de la comunicación entre el servidor remoto y los servidores en cada campo.
    - Base de datos remota para almacenar datos de todos los campos, usuarios, sistemas de riego, y el estado de los componentes de cada uno.

3 - Estimaciones Iniciales

3.1 - Planificación

Previo a la ejecución del proyecto, se relevó el dominio del problema y los requerimientos a cumplir. Esta etapa de análisis permitió estimar la duración y cronograma de tareas necesarias para desarrollar el sistema. Obtener un resultado preciso, aún al considerar un margen de error por imprevistos como es usual, fue difícil debido a la falta de experiencia previa en este tipo de estimaciones. Los pasos a nivel general resultaban claros, pero no así su duración.

El cronograma original seguía una metodología de cascada, donde primero ocurre el análisis del sistema completo, luego su diseño, implementación y revisión. A pesar de ser poco flexible ante cambios en las necesidades del cliente, fue adecuado por tratarse de un Proyecto Final con un alcance limitado. Al trabajar en cascada, se analizó completamente el problema al inicio y se definió cuáles requerimientos se implementarían. Luego, se dedicaron varias reuniones seguidas a plantear la arquitectura con suficiente detalle y validarla con miembros de Ponce AgTech.

Una vez diseñado el borrador de la arquitectura se pasó a una metodología ágil, donde a la implementación de cada característica o módulo le siguió un período de revisión, que incluía testing y validación con la empresa. Así, se redujo el desvío del sistema implementado respecto del proyectado, ajustándose a las expectativas del cliente.

El diagrama de Gantt fue construído antes de empezar con el borrador de la arquitectura, por lo que en base a los requerimientos se pensó en tres módulos: acceso de usuarios, monitoreo de válvulas y bombas, y control de estos dispositivos y programas de riego.

Aunque formalmente ambos subsistemas (nube y campo) representaban proyectos diferentes, era necesario coordinar las actividades entre ambos equipos. Esto era especialmente importante para las tareas de diseño, pues debían obtenerse interfaces claras y completas que permitieran desacoplar el desarrollo de los dos grupos. La idea desde el primer momento fue trabajar siempre en paralelo, pero definiendo de manera clara los límites, contratos y mensajes entre ambos subsistemas.

Al momento de la planificación, tres de los cuatro integrantes del grupo trabajaban a jornada completa y todos cursaban materias de la carrera. Por ende, se consideró que por cada miembro se podrían dedicar entre ocho y diez horas semanales, repartidas como dispusiera cada uno. Es por esto que se graficó el cronograma en términos de semanas y no en horas.

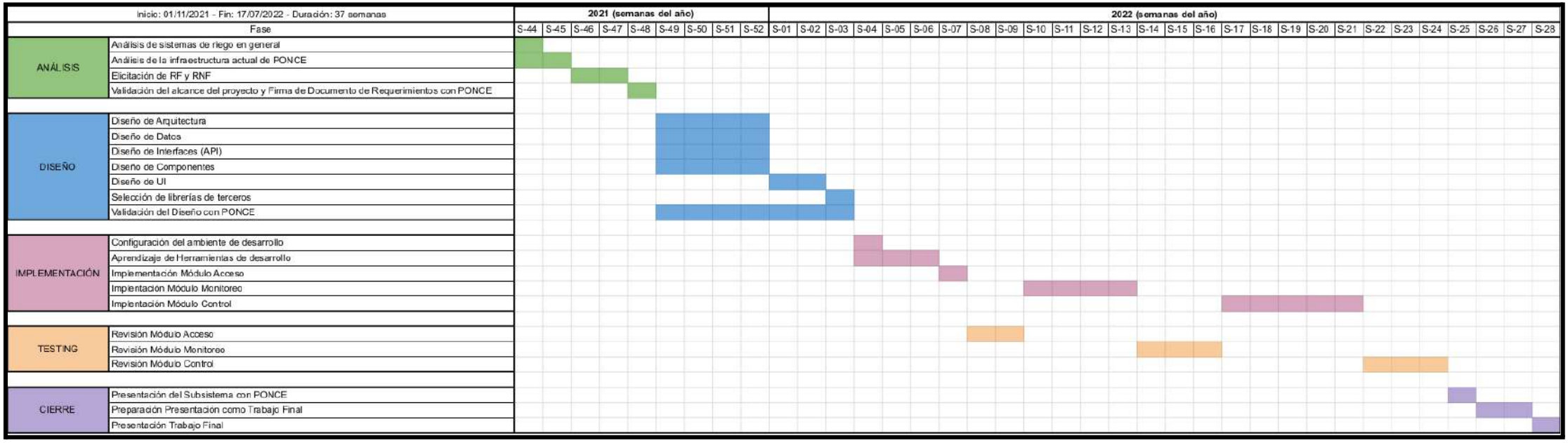

Figura 2 - Diagrama de Gantt del proyecto.

3.2 - Fortalezas, Oportunidades, Debilidades y Amenazas (FODA)

Fortalezas

- Disponibilidad de un producto previo de referencia, útil en el análisis de requerimientos.
- Fortaleza del grupo de desarrollo en tecnologías como JavaScript, Node.js, entre otras.
- Motivación del grupo en desarrollar un área de interés personal como son sistemas distribuidos y servicios en la nube.

Oportunidades

- Asistencia y recursos de Ponce AgTech en sistemas de riego y gestión de proyectos de software.
- Competencia en la región prácticamente nula.
- Mercado potencial importante.

Debilidades

- Inexperiencia del grupo en hardware de sistemas de riego o sistemas embebidos.
- Disponibilidad horaria acotada de todos los integrantes.

Amenazas

- Aparición de una tecnología de riego superior a la de goteo/microaspersión, que dejara al producto obsoleto.
- Cambios en Ponce AgTech, que vuelvan incompatible lo desarrollado con el resto de productos.
- Cierre de Ponce AgTech, perdiéndose la asistencia y los recursos para el proyecto.

3.3 - Análisis de Riesgos

Además del análisis FODA, se consideraron factores de riesgo que pudieran afectar el avance del proyecto, e incluso su finalización. Fueron ponderados en base a su probabilidad de ocurrencia (Po) y su impacto (I), ambas variables con valores entre uno y tres. Si el peso (P = Po * I) es mayor o igual a seis, entonces para ese factor de riesgo debe considerarse un plan de contingencia, pues es muy probable que suceda y/o que impacte notoriamente.

Riesgo	Consecuencia	Po	I	P
Cierre de Ponce AgTech	Cancelación del proyecto	1	3	3
Integrante abandona proyecto	Pérdida de mano de obra y conocimiento	1	3	3
Desconocimiento sobre tecnologías, sistemas de riego	Posibles demoras, dudas sobre dónde se originan errores	1	2	2
Imposibilidad de coordinar reuniones con demandante	Retraso en validaciones, cambios en requerimientos inoportunos	2	3	**6**
Mala integración de componentes	Demora por retrabajo	1	3	3
Librerías de terceros desactualizadas durante desarrollo	Producto con errores, vulnerabilidades y sin novedades útiles de versiones recientes	3	2	**6**
Ausencia momentánea de integrante por razones personales	Pérdida de mano de obra y conocimiento	2	3	**6**
Pérdida de interés del demandante en el proyecto	Cancelación del proyecto	1	3	3

Figura 3 - Análisis de riesgos del proyecto.

Hay algunos riesgos con peso igual a seis, que requieren de un plan de contingencia:

- *Imposibilidad de coordinar reuniones con demandante*: tener más de un referente para realizar consultas. Pactar de antemano un cronograma de reuniones periódicas con el compromiso de todos los integrantes y del demandante.
- *Librerías de terceros desactualizadas durante el desarrollo*: actualizar librerías en otra rama del proyecto y utilizar testing automático para no desperdiciar horas de trabajo.
- *Ausencia momentánea de integrante por razones personales*: cubrir a la persona.

Un riesgo que no figura en el cuadro pues fue detectado y resuelto apenas iniciado el proyecto fue la cuarentena por COVID-19. Si bien repercutía principalmente en el proyecto de campo, pues requerían de hardware para sus pruebas, también afectaba al subsistema de nube. Por la dependencia entre ambos equipos, se decidió solicitar el préstamo del equipo necesario al demandante, conservándolo durante todo el proyecto. De esta manera, se evitaron retrasos.

4 - Partes Interesadas en el Proyecto

Un factor distintivo del proyecto fue la presencia de diferentes actores con distintos intereses en el sistema. A continuación, se presenta cada uno y cómo fue su participación.

Product Owner (Referente Funcional) del Demandante

Es el representante de las necesidades del cliente y quien mejor entiende cómo debe ser el producto. Su participación era sumamente importante porque de él se obtendrían los requerimientos funcionales del sistema a crear. Se trata de un perfil no técnico por lo que el diálogo debía ser de alto nivel.

Su incorporación al proyecto se planificó a través de una serie de reuniones: al principio, durante la etapa de análisis, la interacción fue constante para relevar los requerimientos; luego, su participación fue para validar el avance.

Equipo de Desarrollo del Demandante

No se interactuó con cada miembro del equipo, sino que sus intereses fueron representados por el líder técnico y principalmente por el director de tecnología. Juntos impusieron restricciones técnicas, en especial tecnologías que el sistema debía utilizar. Aún así, surgieron negociaciones en donde se recomendaron otras herramientas que finalmente fueron aceptadas.

Su incorporación al proyecto se planificó a través de una serie de reuniones: al principio, durante la etapa de análisis, la interacción fue constante para relevar los requerimientos no funcionales; luego, su participación fue para validar el diseño propuesto antes de cada implementación.

Cliente Actual del Demandante

Originalmente no estaba planeada una reunión con el cliente pero durante la etapa de análisis surgió la oportunidad de visitarlo al campo. El objetivo era conocer el sistema en persona y validar con él lo relevado hasta el momento. Además, indicó una serie de sugerencias e ideas extra que fueron registradas como requerimientos deseados, al quedar fuera del alcance del proyecto. Por último, ayudó a comprender aún más el dominio del problema al explicar muchas de las dificultades que impactan en el riego.

Equipo del Subsistema Campo

La interacción con el equipo del subsistema campo fue la más delicada del proyecto, en tanto que requería de la mayor atención. Ambos subsistemas debían ser compatibles entre sí y responder a las necesidades del cliente y del demandante. Entonces, fue muy importante comprender juntos el dominio del problema. Por otro lado, a la hora del diseño, fue necesario discutir y conciliar las interfaces con el objetivo de lograr un “todo” transparente y eficaz.

5 - Metodología Aplicada

Como se mencionó en la sección anterior, inicialmente se había considerado una metodología de cascada, para hacer énfasis en las etapas de análisis y diseño, y reducir errores de arquitectura por la falta de experiencia. Una vez obtenido un borrador completo del sistema, se tomó la decisión de implementarlo siguiendo una metodología ágil.

En particular, se eligió trabajar con *Scrum*. Esta metodología divide el tiempo en períodos en general de dos semanas, llamados *sprints*. Antes del inicio de cada uno, se estima el tamaño de las próximas tareas a encarar (registradas en un listado llamado *backlog*). Con la estimación de cada uno, se decide en equipo qué se trabajará el próximo sprint, tratando de cubrir todo el tiempo disponible sin excederlo. Las tareas tienen estados que describen su avance. Comienzan en *para hacer*, y una vez que alguien empieza a realizarlas pasan a *en curso*. Luego pasan a *en revisión*, etapa durante la cual el código es revisado por otro miembro para detectar errores, y el referente funcional valida si lo conseguido satisface los requerimientos. Luego de cada implementación se obtiene retroalimentación para corregir lo que fuese necesario. Si la tarea está lista, entonces puede fusionarse con la rama de desarrollo.

Figura 4 - Diagrama de estados de una tarea en Jira.

Además de ser una metodología simple de aplicar y ser un estándar en la industria, Scrum era conocida por los miembros de ambos equipos, pues ya la habían aplicado profesionalmente. Era también usada en Ponce AgTech, por lo que era más simple acoplar el desarrollo del proyecto con los otros de la empresa.

Se validó con el referente no sólo lo implementado, sino también el diseño antes de llevarlo a código. Además, para cada implementación a nivel de back-end se construyeron tests unitarios, y se agregaron los cambios a la documentación de las APIs correspondientes. Al tratarse de un sistema que sería continuado por otro equipo, era importante documentar en gran detalle, lo que facilitaría la transición y comprensión.

A continuación se describe en mayor detalle la metodología de cada etapa.

5.1 - Análisis

El proceso de análisis de requerimientos se desarrolló en su mayoría al inicio del proyecto. Comenzó con la presentación del sistema esperado por Ponce AgTech al equipo y terminó con la entrega del Protocolo del Trabajo Final. Durante ese período se concretaron varias reuniones con el referente funcional y el director de tecnología.

Luego de cada reunión se conformaba un listado de requerimientos, distinguiendo entre *esperados* y *deseados*. Los primeros eran los que sí serían realizados como parte del proyecto, mientras que los otros eran opcionales. Estos fueron relevados pero no eran prioritarios. Para mantener el alcance del proyecto acotado, su implementación quedó como trabajo futuro para el demandante. Por otra parte, en ambos grupos había requerimientos *funcionales*, que indican qué debe hacer el sistema, y *no funcionales*, que describen cómo debe hacerlo (tecnologías y protocolos que deben usarse principalmente).

El listado de requerimientos era refinado en cada reunión, por lo que se realizaron iteraciones hasta obtener un resultado completo y sin ambigüedades. A partir de la versión final, se pactó con el demandante el alcance del proyecto. Ponce AgTech obtendría un sistema completo que podría extenderse en el futuro, y el equipo podría realizarlo en un plazo adecuado para un trabajo final de carrera.

Por otro lado, gracias a la invitación de Ponce AgTech, se tuvo una reunión en campo con un cliente del sistema actual. La interacción con la interfaz de usuario permitió no sólo comprender de manera gráfica las funcionalidades del sistema, si no también observar al sistema de riego en funcionamiento. La visita proporcionó información adicional al conocer cuáles eran las expectativas del cliente.

5.2 - Diseño

Una gran parte del tiempo dedicado al diseño del sistema también sucedió previo a la presentación del Protocolo. Mediante diferentes reuniones con los integrantes del equipo del subsistema de campo se construyó un diagrama borrador de la arquitectura, incluyendo los protocolos de comunicación entre cada componente. Cada requerimiento funcional se tradujo en un camino a seguir en el sistema, definiendo de esta manera responsabilidades, mensajes e interfaces.

Durante esta etapa tuvo una importante participación el director de tecnología, ya que por su rol dentro del área de desarrollo en Ponce AgTech pudo validar la arquitectura. Además, propuso algunas tecnologías que podrían aplicarse al proyecto, que luego fueron analizadas y comparadas por los miembros del proyecto para elegir las más adecuadas.

Un objetivo importante de esta etapa era poder dimensionar el sistema de manera completa y decidir cómo sería la división entre ambos subsistemas. En este contexto, surge la creación de un componente llamado *mapper*, encargado de comunicar ambos subsistemas. Además de ofrecer ventajas desde lo técnico, permitía una clara división del campo y la nube, a modo de frontera entre ambos.

Previo a la entrega del Protocolo, surgieron los entregables de diagrama de arquitectura, el listado de tecnologías a utilizar y el diagrama de entidad-relación. Este último se creó a partir de detectar en los requerimientos las entidades que participarían del sistema, sus características y las relaciones entre ellas. Representa en gran parte el diseño de la base de datos del sistema.

Una vez iniciada la implementación, el diseño continuó en todos los *sprints*. Con cada nueva funcionalidad agregada, se diseñaban entre ambos equipos las APIs y mensajes a enviarse entre ambas partes.

5.3 - Implementación

Al tener todos los miembros experiencia profesional como programadores fue simple establecer el flujo que seguirían las tareas a la hora de llevarlas a código. El proceso se inicia cuando una tarea es asignada a alguien para realizarla durante un sprint. El responsable la descompone en las subtareas que considere necesarias.

Cada repositorio de código cuenta con una rama principal y una de desarrollo. En la primera sólo está el código estable y testeado que resulta de todo el trabajo de un sprint. En cambio en desarrollo se agregan todos los cambios hechos durante el sprint. Sin embargo, cada tarea no es desarrollada directamente en esta rama, sino que se hace sobre una exclusiva que se ramifica de ella. Así el desarrollador puede trabajar de manera aislada pero sin perder la posibilidad de resguardar y compartir su trabajo. El nombre de la rama corresponde con el ID y título de la tarea.

Cuando el programador termina de desarrollar la tarea, crea una solicitud de revisión de código. Completa un formulario explicando los cambios realizados y otros detalles, como indicaciones de si se han agregado nuevas librerías al proyecto, o imágenes de las modificaciones.

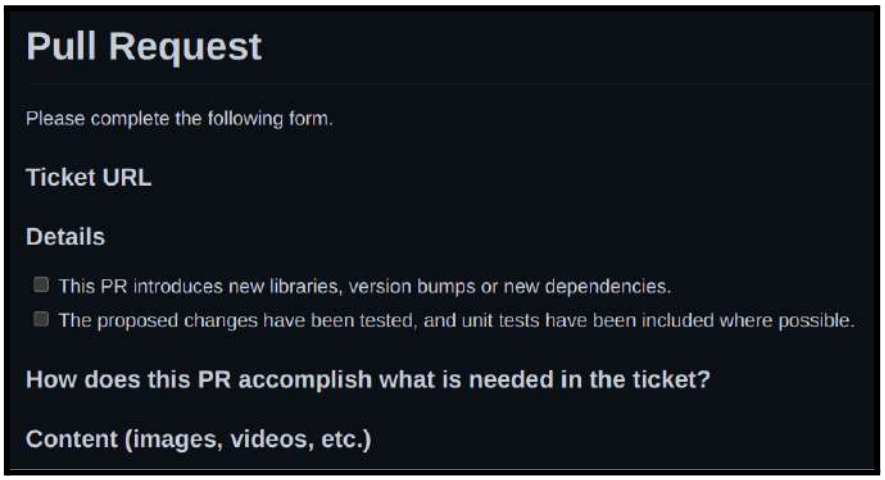

Figura 5 - Formulario que completa un desarrollador al solicitar una revisión de código.

Otro desarrollador se encarga de revisar los cambios, prestando especial atención a errores o mejoras posibles. Si no hay que hacer ningún cambio, el nuevo código se fusiona con la rama de desarrollo.

Figura 6 - Fragmento de una revisión de código.

Cuando se completa el sprint, la rama de desarrollo se fusiona con la principal, y así ambas quedan sincronizadas. En general, la rama principal es la que corre en producción. En este proyecto, fue útil para ir mostrando avances al demandante y validar que el proyecto siga el curso correcto.

Para reducir los errores en el código y asegurar un estilo similar entre los programadores, se aplicaron herramientas automáticas y *linting* usando las reconocidas *Reglas de AirBnB*. De esta manera, se siguen estilos aceptados por toda la comunidad de los lenguajes utilizados y el código resulta más claro para los próximos desarrolladores.

A pesar de no contar con el presupuesto para la utilización de herramientas de integración continua, se buscó lograr un resultado similar de manera gratuita. Se personalizaron los editores de código para que lo analicen y corrijan, y se configuró *git* para que controle los cambios antes de subirlos al repositorio.

5.4 - Testing

En cada sprint se invirtió aproximadamente la misma cantidad de tiempo en la implementación como en su testing correspondiente. El objetivo era asegurar la calidad del código, y disponer de tests que pudieran validar los próximos cambios a aplicar sobre lo establecido. Las pruebas creadas fueron documentadas en tablas donde se agrupaban por entidades o componentes y describían cuáles eran sus entradas y salidas correspondientes. Dichas tablas se encuentran en el Apéndice para su consulta.

Para mantener el proyecto acotado a la duración de un trabajo final se decidió sólo implementar tests unitarios de caja negra, dónde se corrobora el cumplimiento de los contratos de las entidades y piezas de código sin depender de su implementación. Estos tests no incluyen la comunicación real entre los subsistemas o componentes de mayor tamaño, lo que corresponde a pruebas de integración.

En los casos donde las pruebas utilizaran código que consultara otro subsistema de la arquitectura su API fue simulada. Aunque la comunicación no es real, se puede controlar que los componentes llamen a los métodos de otros cuando corresponda y con los parámetros correctos. En este sentido la tecnología de servidor elegida provee una conexión a la base de datos específica para pruebas agilizando el desarrollo de los tests, por lo que fue de gran utilidad.

Al igual que para el *linting* del código, no era posible establecer un proceso de integración continua, pero en base a la experiencia profesional se implementó un entorno de testing dentro del mismo repositorio. Este se ejecuta de manera manual cuando el desarrollador lo necesite, pero también lo hace *git* antes de subir el código. Si fallan los tests, no se envían los cambios, logrando así reducir en gran medida los errores introducidos en las ramas estables. Además, se agregó la medición automática de la cobertura del código fuente. Los tests cubren diferentes líneas, con el objetivo de lograr testear la mayor cantidad posible y no dejar afuera ninguna fuente de error.

No fueron testeadas las interfaces gráficas *web* por varias razones. Las aplicaciones fueron pensadas con el objetivo de que sean lo más simples posibles, sin ningún tipo de lógica que involucre reglas de negocios. Sólo se validan entradas, y se envían, reciben y muestran datos. Por ende, no tenían ningún contrato o responsabilidad importante a asegurar en cuanto a integridad de datos y seguimiento de reglas. Como la mayoría de esta lógica se ubicaba en el servidor, ahí se concentró el *testing*. Además, una de las interfaces es construida de manera automática por la tecnología elegida para el servidor. Como se trata de un proveedor reconocido en el ámbito, se asume que ha sido probado y su calidad está asegurada.

6 - Problema a Resolver

6.1 - Dominio del Problema

Riego de Kiwi

Para crecer satisfactoriamente, el kiwi necesita alta humedad ambiental y un régimen de lluvias abundante y relativamente frecuente. Para esto el productor recurre a sistemas de riego que permiten conducir el agua mediante una red de tuberías y aplicarla a los cultivos a través de emisores que entregan pequeños volúmenes en forma periódica.

El agua se distribuye en forma de goteo por medio de goteros o en forma de lluvia a través de difusores denominados microaspersores y microjets. En lo que concierne al sistema de software, los componentes que interesan son las bombas y válvulas, pues son las que se controlarán para administrar el flujo de agua y ejercer el riego.

Figuras 7 y 8 - Microaspersor y gotero, respectivamente.

Producción de Kiwi en Argentina

La mayoría de la producción de kiwi se encuentra en el sudeste de la provincia de Buenos Aires, principalmente en Mar del Plata y Sierra de los Padres [(0)]. Según estadísticas de la Cámara de Productores de Kiwi de Mar del Plata, en 2020 se produjeron en el país alrededor de 11 millones de kilos de los cuales el 75% perteneció al sudeste bonaerense. En segundo lugar, se encuentran algunas localidades del noreste de la provincia, y en tercer lugar, el centro de Córdoba.

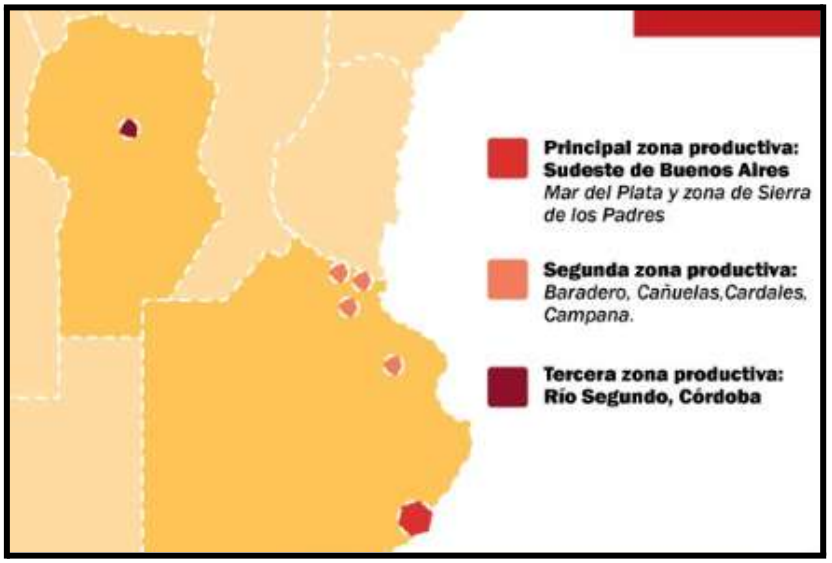

Figura 9 - Zonas productivas de kiwi en Argentina.

Además de producirse para consumo interno, el kiwi argentino es vendido en otros países. En 2020, se exportaron aproximadamente 3000 toneladas desde Buenos Aires. Incluso, existen variedades que se comercializan sólo en el mercado internacional. Para poder ser aceptado, el fruto debe cumplir una serie de certificaciones, que incluyen parámetros como los milímetros regados.

Dificultades del Riego

Muchas plantaciones se riegan de forma manual, lo que requiere que operadores en campo prendan y apaguen el sistema en determinados horarios establecidos en un plan diseñado por asesores de riego. Estos horarios pueden ser actualizados según el estado del cultivo y del clima.

Esta modalidad presenta algunos inconvenientes. Primero, no siempre los operadores están disponibles a horario para las tareas involucradas debido a imprevistos que puedan surgir. Esto es importante para cumplir los planes de riego establecidos y que el cultivo crezca saludablemente. También lo es para mitigar el efecto de las heladas y para evitar que el riego funcione durante horarios pico, donde el precio de la electricidad aumenta significativamente con respecto al resto del día.

En segundo lugar, los asesores no pueden controlar que el plan de riego se esté siguiendo correctamente y no tienen información confiable que los ayude a tomar decisiones. Por último, si ocurre algún desperfecto en el sistema, los operadores no tienen forma de darse cuenta hasta que se encuentran en el lugar. Un desperfecto no atendido a tiempo puede provocar un daño en la plantación y un desperdicio de recursos energéticos.

Unidades de Medida

En el contexto del riego de kiwi existen diferentes magnitudes a tener en cuenta:

Magnitud	Unidad
Latitud - Longitud	Grados decimales
Caudal teórico	Metros cúbicos / hora
Presión	Bares
Área	Hectáreas
Hora de inicio de programa o paso	hh:mm (formato 24 horas)
Duración de paso de programa	Minutos

Figura 10 - Magnitudes involucradas en el riego de kiwi.

6.2 - Entidades del Dominio

Campo

Es una parcela de tierra apta para el cultivo. Puede ser de pequeña, mediana o gran superficie.

Lote

Es un sector del campo regado por un subconjunto de válvulas del sistema de riego.

Válvula

Es un dispositivo que permite iniciar o detener la circulación del agua por el sistema de riego mediante una pieza movible que abre o cierra el conducto. Tiene dos estados posibles: encendido o apagado. Se puede accionar manualmente en el lugar de instalación o a distancia con mandos hidráulicos o eléctricos. En un sistema existen varias válvulas.

Figura 11 - Válvulas instaladas en tuberías [1].

Bomba

Es un dispositivo que suministra el agua desde un depósito hacia el sistema de riego. Su tamaño y potencia dependen de la superficie a regar. El dimensionamiento debe ser tal que la presión sea suficiente para vencer las diferencias de cota y las pérdidas de carga de todo el sistema. Tiene dos estados posibles: encendido o apagado. Las más utilizadas son de acción centrífuga abastecidas por energía eléctrica o motores a explosión. En un sistema puede haber una o más bombas.

Figura 12 - Bombas centrífugas instaladas en paralelo [(2)].

Programa de Riego

Es una serie de pasos que representan instrucciones para regar un lote o el campo completo. Cada paso indica qué dispositivo debe prenderse o apagarse y en qué momento. Los programas tienen una hora de inicio. En el sistema automatizado serán ejecutados sin intervención del usuario. En un mismo campo pueden existir diferentes programas de riego para distintas situaciones, pero sólo uno puede estar activo en un momento dado.

Organización

Es una empresa, cooperativa u otro tipo de grupo de personas que poseen o están a cargo de uno o más campos.

Usuarios

Son las personas que tendrán acceso al sistema automatizado para revisar el estado del riego y gestionarlo. Existen diferentes roles:

- *Regador*: es el operario de la máquina de riego, encargado de prender, apagar y atender el equipo ante una falla. Puede realizar tareas de mantenimiento menores y le reporta tanto al encargado de campo como al productor.
- *Encargado de Campo* o *Gerente*: es en general un ingeniero agrónomo a cargo de tomar las decisiones que impacten en la producción.
- *Productor*: es el dueño del campo o representante de la organización dueña del campo. Toma las decisiones pensando en el negocio, y tiene una visión más general. Trabaja en equipo con el encargado de campo.

Un usuario pertenece a una determinada organización, pero puede a la vez tener acceso a los campos pertenecientes a otra, de acuerdo a los convenios y contratos entre éstas.

6.3 - Acerca del Demandante

El problema descrito anteriormente representa una oportunidad de negocio para el demandante de este proyecto, Ponce AgTech. Es una *start-up* fundada en 2017 en la ciudad de Mar del Plata, Argentina. Tiene como misión ayudar a los productores agrícolas a hacer un uso más eficiente de los recursos involucrados en sistemas de riego. Para ello, ofrece un servicio de monitoreo basado en *IoT* y sistemas en la nube. Actualmente, comercializa soluciones para riego por pivot, avance frontal y enrollador. Busca desarrollar un nuevo producto para riego por microaspersión y goteo.

Figura 13 - Logo de Ponce AgTech.

6.4 - Elicitación de Requerimientos

En esta sección, se resumen los requerimientos solicitados por Ponce AgTech. En el Apéndice se puede consultar el listado completo.

Requerimientos Funcionales Esperados

- **Monitoreo**: el sistema deberá permitir al usuario monitorear el estado (encendido/apagado) de las bombas y válvulas, la presión de trabajo de las bombas y el estado de la conectividad con el subsistema en campo.
- **Control**: el sistema deberá permitir al usuario encender/apagar remotamente las bombas y válvulas, y ofrecer la creación, ejecución, modificación y eliminación de programas de riego. El sistema también deberá permitir al usuario agrupar diferentes válvulas conformando lotes.
- **Acceso**: el sistema permitirá al usuario acceder a la plataforma web remota mediante el ingreso de nombre de usuario y contraseña. El sistema deberá permitir al administrador realizar todas las acciones de un usuario normal, agregando: altas, bajas y modificaciones de usuarios, organizaciones, campos, bombas y válvulas.

Requerimientos No Funcionales Esperados

Se incluyen ciertos requerimientos no funcionales que influyen principalmente en el subsistema campo, pero que permiten comprender mejor el sistema de manera completa.

- El sistema debe ser construido de manera tal que soporte varios subsistemas de campo en simultáneo, cada uno con su hardware propio: una Raspberry Pi como unidad central y un conjunto de placas Arduino esclavas. El sistema deberá ser dimensionado de manera tal que no se vea limitada la cantidad de válvulas a usar.
- Los usuarios podrán acceder al sistema desde diferentes aplicaciones. Por un lado, desde cualquier lugar del mundo y cualquier dispositivo a través de una interfaz web (subsistema nube) que ofrecerá la funcionalidad completa del sistema. Por otro, podrá utilizarse desde una web específica para el campo. En este caso, podrá gestionar válvulas y bombas, al igual que apagar el programa activo, pero no crear, editar ni eliminar lotes o programas.
- El sistema deberá ser implementado usando Vue.js para interfaces web, PostgreSQL para base de datos y Node.js para servidores.
- La comunicación entre los subsistemas de campo y la nube debe realizarse a través de un protocolo lo suficientemente liviano, en términos de bytes por mensaje, para que en un futuro, fuera del alcance de este proyecto, se pueda implementar comunicación satelital. De momento, se hará por Protocolo de Control de Transmisión (*Transmission Control Protocol - TCP*) en capa de transporte.
- El sistema llevará un *log* de lo acontecido en los servidores, existiendo diferentes tipos: errores (fallas en el sistema), información (acciones de usuarios) y debug (como información pero más detallado para resolver errores).

6.5 - Flujo de Trabajo

El uso del sistema automatizado de riego por parte de los usuarios regadores y encargados de campo puede resumirse en un sólo flujo de acciones representable mediante un Modelo y Notación de Procesos de Negocio (*Business Process Model and Notation - BPMN*).

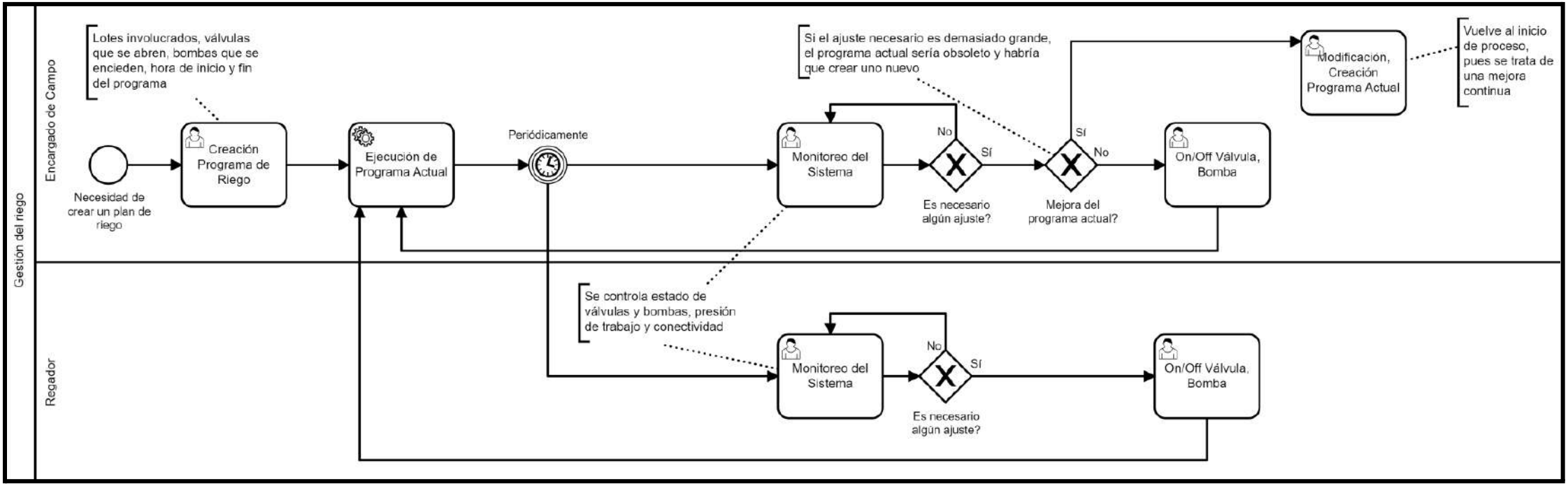

Figura 14 - BPMN principal.

7 - Diseño del Sistema

7.1 - Arquitectura General

Para la comprensión del sistema, se presenta primero un esquema general que incluye al subsistema nube interactuando con diferentes instancias del subsistema campo.

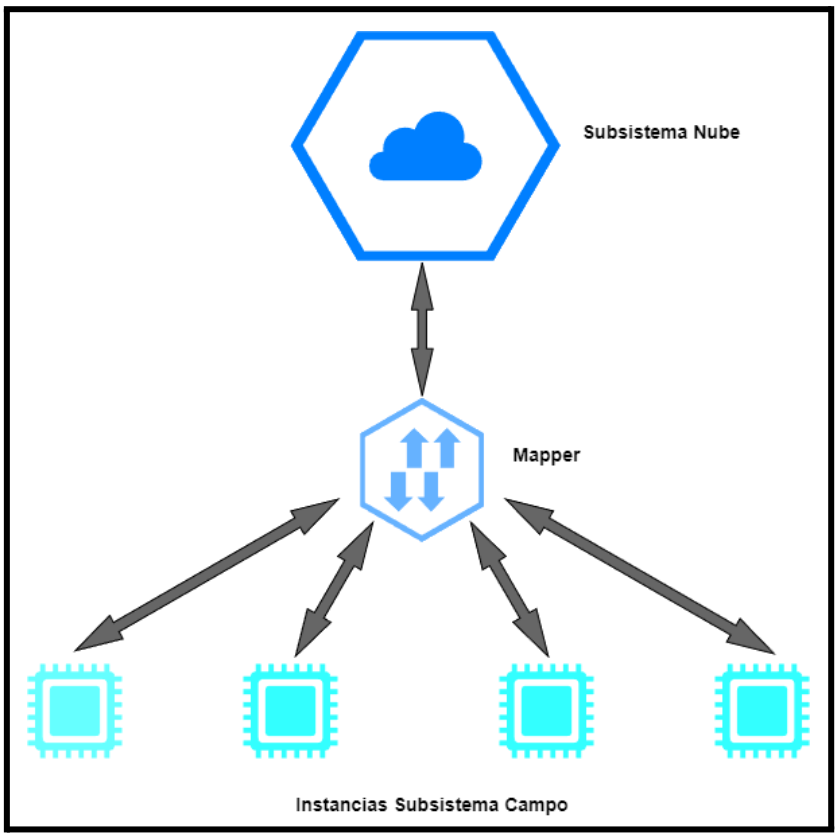

Figura 15 - Vista general del sistema.

Como se puede observar, existe una única instancia central del subsistema nube que interactúa con múltiples instancias del subsistema campo. Cada campo tiene un equipo que gestiona el sistema de riego local, permite cierto control en el lugar y envía datos de estado al de la nube. Este almacena el estado de todos los sistemas de riego, y permite controlarlos de manera remota. En la frontera entre ambos niveles se encuentra el Mapper, componente encargado de traducir los mensajes entre ambos subsistemas, y que permite desacoplarlos ante cambios futuros en el tipo de comunicación usado por cada uno.

7.2 - Arquitectura Subsistema Nube

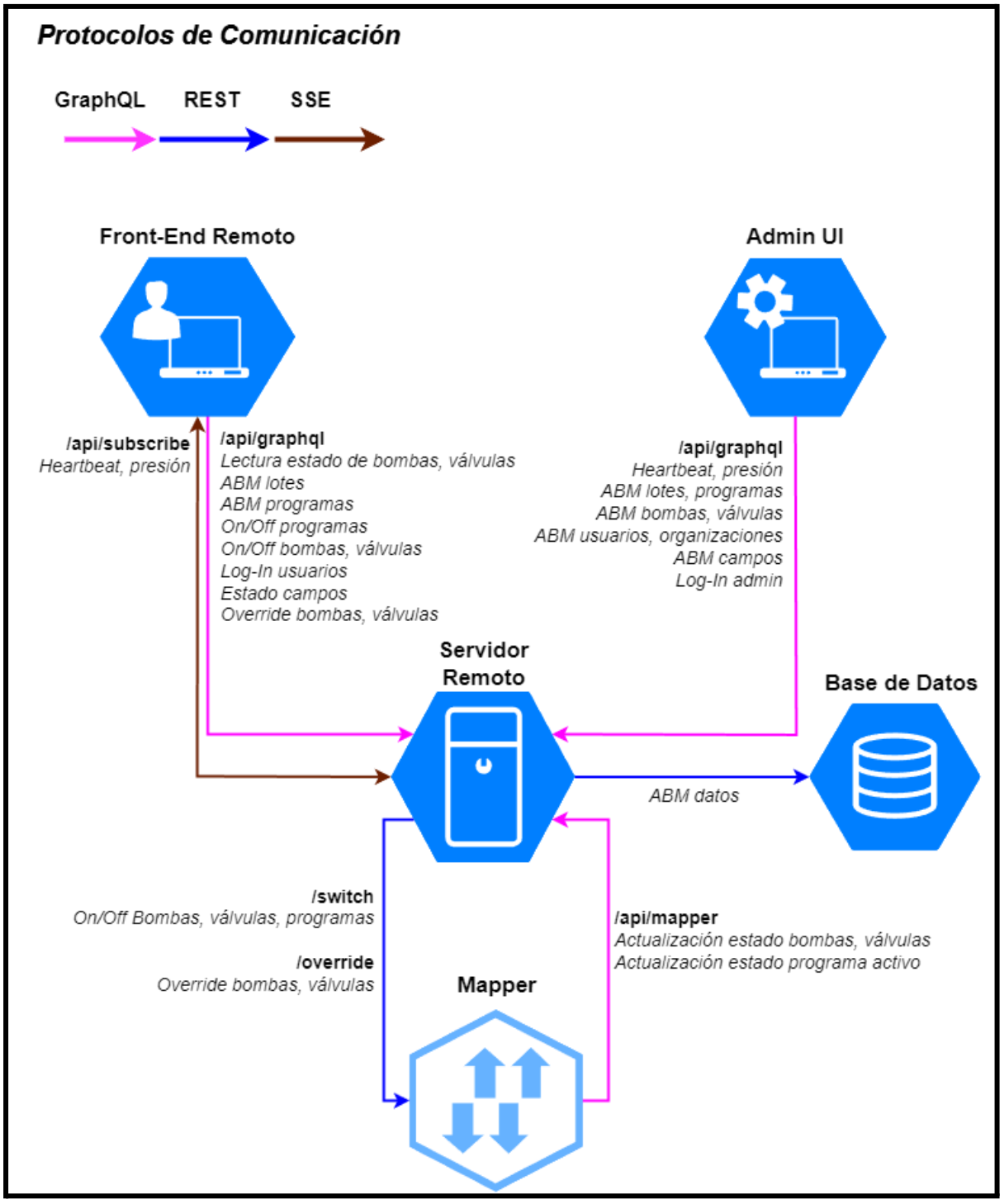

Figura 16 - Componentes del subsistema nube e interacciones entre ellos.

Servidor Remoto

Se trata del componente central del subsistema, encargado de gestionar todos los eventos de manera tal que se respeten las reglas de negocio. Administra alta, baja y modificación de todas las entidades del sistema, y el control de acceso, distinguiendo entre usuarios normales y administradores.

Para la mayoría de operaciones responde a solicitudes emitidas desde alguna de las aplicaciones web o desde el Mapper, que se comunican a través de una API GraphQL. Además, controla periódicamente el *heartbeat* de cada subsistema campo, para detectar cambios en su conexión. En caso de que haya cambios de estado o reciba un nuevo *heartbeat*, el servidor emite un evento SSE (*Server-Sent Events*) para notificar a los usuarios correspondientes del Front-End Remoto. Los SSE se utilizan también para que los usuarios tengan siempre actualizada la presión de las bombas, ya que es un parámetro crítico del sistema. En el Apéndice B se desarrolla en profundidad el diseño del envío del estado de campo.

Base de Datos

El subsistema almacena la información de todas las entidades en la misma base de datos, de tipo relacional. El Servidor Remoto se puede comunicar con ella a través de Internet, en caso de que sea remota, o de manera local, si se encuentran instalados en la misma instancia. Al usar Mapeo Objeto-Relacional (*Object-Relational Mapping - ORM*), se puede trabajar en el servidor con objetos y almacenarlos en tablas de manera transparente y simple.

Mapper

Se trata de un componente compartido entre ambos subsistemas, que cumple el papel de traductor entre ambos. Su razón de ser se debe a la falta de infraestructura de red en zonas remotas como son los campos. En general, la conexión a *Internet* es pobre e inestable. Por ende, el subsistema campo emite sus mensajes mediante Llamadas de Procedimiento Remoto (*Google Remote Procedure Calls - gRPC*), un protocolo que consume pocos recursos. Aunque esta primera versión del sistema lo utiliza sobre TCP, es importante facilitar la transición en un futuro a una comunicación mediante señal satelital, como utilizan otros productos de Ponce AgTech.

Por ende, no tiene sentido que el Servidor Remoto implemente ese tipo de comunicación directamente, pues quedaría demasiado acoplado al subsistema de campo actual. El sistema sería poco flexible a cualquier cambio que dependa del hardware del campo.

El Mapper aparece entonces como intermediario. Es un servidor que no almacena ningún estado, sino que simplemente expone APIs en todos los protocolos de comunicación utilizados. Su función es traducir (o *mapear*) los mensajes enviados desde el campo para que el Servidor Remoto pueda procesarlos, y viceversa. Es fácil de modificar y por su simpleza consume pocos recursos. Incluso, si existiesen diferentes versiones del subsistema campo, con un hardware y/o protocolo de comunicación diferente, podría haber varios tipos de Mappers corriendo, cada uno encargado de traducir una sintaxis diferente.

Esto no sólo es cierto para productos de Ponce AgTech, sino que potencialmente se podría integrar con automatizaciones en campo ofrecidas por los propios fabricantes. Así, se incrementan las posibilidades comerciales del sistema propuesto.

Admin UI

Se trata de una aplicación web de escritorio cuya interfaz es construída de manera automática por una de las tecnologías elegidas para el Servidor Remoto. Su utilización es a modo de *backoffice*, es decir, está pensada para los empleados del servicio.

La aplicación ofrece las funcionalidades del Front-End Remoto pero también permite a los administradores del sistema gestionar la alta, baja y modificación (ABM) de los datos. En la empresa demandante, el manejo de los usuarios y sus activos es realizado por el área comercial. Por esta razón, el ABM sólo se implementó en el Admin UI. Sin embargo, si en un futuro se ofrecieran las funcionalidades de ABM a los usuarios directamente, el Admin UI seguiría siendo útil para el soporte al cliente y la administración del sistema de una manera simple y gráfica.

El diseño del Admin UI es genérico, difícil de adaptar a la imagen de Ponce AgTech y poco usable para un usuario final. Por estas razones, se decidió implementar el front-end remoto con una interfaz y experiencia de usuario adaptadas a los lineamientos de Ponce AgTech.

Front-End Remoto

Es la aplicación web utilizada por los clientes para administrar sus campos. Los usuarios podrán visualizar y gestionar los programas y dispositivos pertenecientes a cada campo. Es decir, tendrán la capacidad de encender, apagar o consultar su estado. Si el usuario que accede tiene el rol de administrador, podrá ver todos los campos registrados en el sistema.

7.3 - Entidades del Dominio

En el siguiente diagrama se ilustran las entidades del sistema, cómo se relacionan entre sí y los atributos pertenecientes a cada una de ellas. Además, se incluyen las participaciones y cardinalidad de cada entidad en las relaciones.

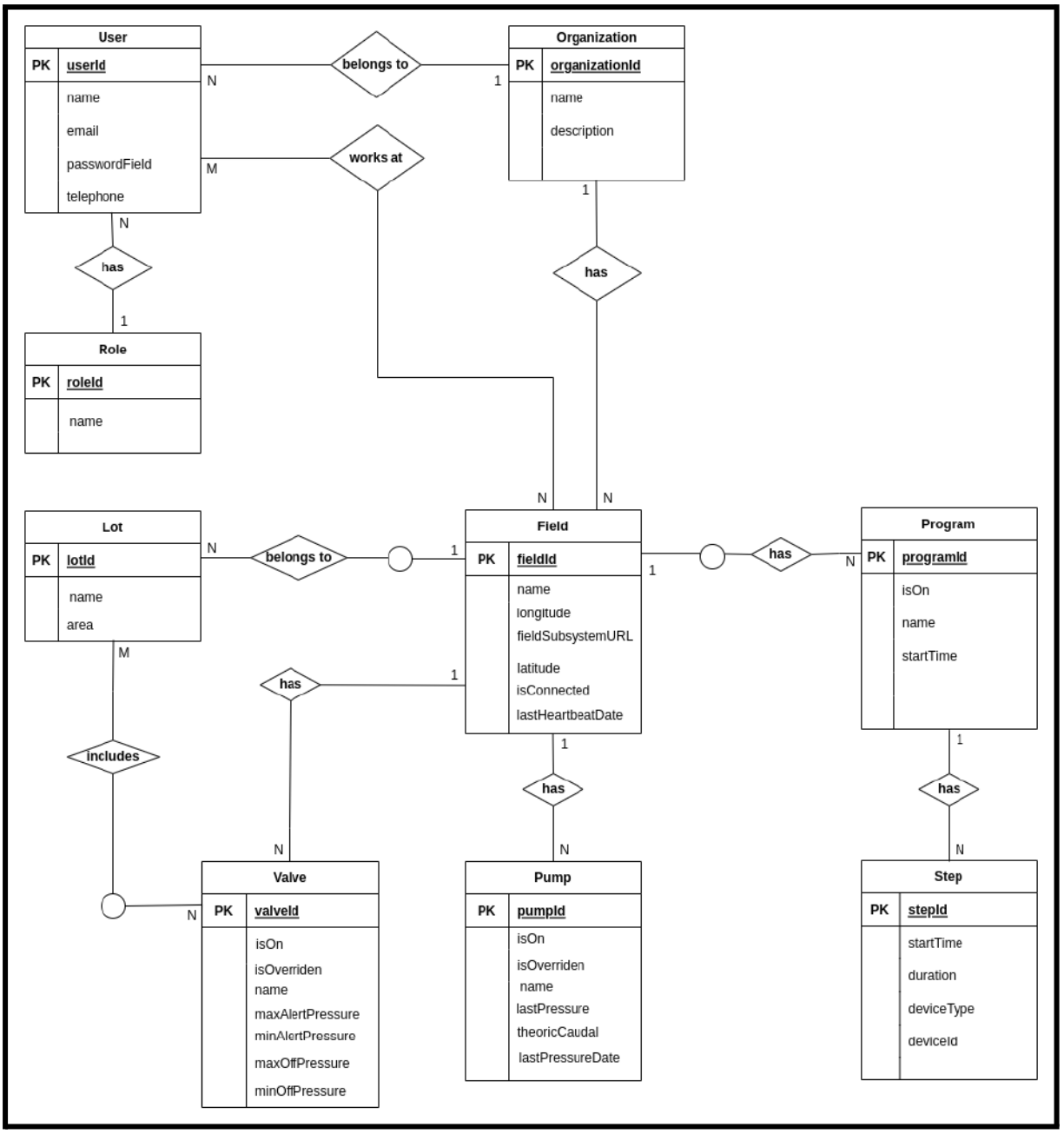

Figura 17 - Diagrama entidad-relación del sistema.

Como se puede observar en el diagrama, existe una entidad Usuario que se identifica con un determinado Rol, pertenece a una Organización y trabaja en uno o varios Campos. Cada Campo pertenece a una única Organización y posee Bombas y Válvulas. Estas últimas pueden o no, formar parte de un Lote, entidad que representa un conjunto de válvulas y pertenece a un único Campo. Por último, todo Campo puede tener asociados varios Programas, que se componen de uno o varios Pasos.

En definitiva, se trata de un modelo de datos bien definido y estable, que describe el problema de manera completa. Esta cualidad influye en la elección de la tecnología de base de datos para el sistema, como se detalla en la próxima sección.

7.4 - Tecnologías Seleccionadas

En esta sección se describen las tecnologías usadas para el proyecto. La empresa demandante tuvo un papel decisivo en la selección, al solicitar que se utilicen varias herramientas con las que sus otros productos están construidos.

Lenguaje de Programación

Uno de los primeros requerimientos no funcionales de Ponce AgTech fue utilizar Node.js para todos los componentes que oficien de servidor, en ambos subsistemas. En la actualidad es una de las principales tecnologías para implementar la lógica de *back-end*. Cuenta con una enorme comunidad de desarrolladores y documentación, al igual que librerías de terceros. Por otro lado, los integrantes de ambos proyectos (subsistemas) cuentan con experiencia en desarrollos con Node.js, por lo que no había curva de aprendizaje.

Node.js funciona únicamente con JavaScript, por lo que no hay otra opción para escribir el código. Nuevamente, cuenta con una gran popularidad y soporte, por lo que es ideal. Además, es un lenguaje originalmente diseñado para front-end, donde sigue siendo muy utilizado. Se decidió entonces también seleccionarlo para las aplicaciones web, logrando así tener un único lenguaje apto para ambos contextos. Esto simplifica el desarrollo y la incorporación de nuevos programadores en el futuro.

Base de Datos

Para el almacenamiento y consulta de datos, el principal punto a tener en cuenta era si debía utilizarse una base de datos relacional, o una no relacional. Estas últimas no determinan una estructura fija, sino que son flexibles al almacenar datos en diferentes colecciones de documentos. Esto las hace especialmente útiles cuando se trabaja con requerimientos cambiantes, datos muy poco relacionados, o las operaciones se dan en el orden de millones por segundo, entre otros casos de uso.

En cambio, una base de datos relacional almacena la información en tablas normalizadas donde hay integridad referencial. Son útiles cuando justamente el problema se puede estructurar de dicha manera, no cambia demasiado sus características en el tiempo, o se trabaja con consultas complejas y reportes.

Para el sistema propuesto se adapta mejor una base de datos relacional, pues las entidades del problema y sus relaciones están bien definidas, sin demasiados cambios en el futuro. La cantidad de operaciones por segundo es baja, ya que en el corto/mediano plazo no tendrá una enorme cantidad de clientes el producto por el volumen que maneja el demandante. En cambio, sí es posible que se incorporen reportes del riego para los usuarios, al igual que se hace con sus otros sistemas comercializados. Entonces, SQL será apropiado para responder a consultas históricas.

Ponce AgTech solicitó utilizar PostgreSQL, pues ya aplica la tecnología en otros servicios. Como se trata de una base de datos muy utilizada, hay variedad de soporte, extensiones, proveedores, etc., y es constantemente actualizada y testeada.

Framework Back-End

Para agilizar el desarrollo del Servidor Remoto y reducir los errores, se decidió buscar un framework de back-end popular que resolviera la mayoría de operaciones comunes y fuese fiable. Las herramientas más populares para Node.js son sistemas de gestión de contenido (*Content Management System - CMS*) que tienen como casos de uso principales el comercio online o *blogs*, entre otros. Aun así, son lo suficientemente generales para resolver muchas otras situaciones. Ofrecen ventajas como:

- Mapeo Objeto-Relacional (ORM): no es necesario programar las consultas a la base de datos en SQL, sino que esto sucede de manera automática. Los objetos que se manejan en la lógica son convertidos a entradas en tabla y viceversa de manera transparente. En el Apéndice hay una sección llamada “KeystoneJS Relationships y Relaciones M:N” donde se describe un ejemplo.
- Sintaxis simple y completa para definir las entidades del sistema, sus propiedades y reglas de control de acceso a ellas. Todas las operaciones de ABM se crean automáticamente. También ofrece filtrado, paginado y orden de los datos de serie.
- Generación de interfaz de usuario para administradores automática: permite realizar el ABM de todas las entidades del sistema. Así no es necesario crear el Admin UI desde cero, pero puede modificarse y extenderse si es necesario.
- Hooks para extender y adaptar lógica genérica ya resuelta de serie al caso de uso.
- Manejo de sesiones de usuario, personalización y extensión del servidor web.

Ningún integrante había trabajado con un CMS previamente, pero ofrecían un gran potencial. Desde Ponce AgTech se sugirió utilizar KeystoneJS, pues ya se encuentra en varios sistemas propios de la empresa. Antes de elegirlo directamente, se decidió buscar otras opciones, compararlas y luego seleccionar la más adecuada.

CMS	KeystoneJS	supabase (3)	strapi (4)
BD Soportadas	PostgreSQL, SQLite	PostgreSQL	MySQL, MongoDB, PostgreSQL, SQLite
Estado Proyecto	Disponibilidad General	Beta Pública	Producción
API	GraphQL, REST	REST	GraphQL, REST
Hosting y Precio	Libre y Gratis	Planes para hosting en Supabase, uno sólo es gratis pero limitado. Libre y gratis mediante Docker	Libre, pero mediante planes. Hay uno gratis pero sólo permite tres tipos de roles de usuario.
Admin UI	Sí, en Next.js (React)	Sí, si se hostea en Supabase	Sí, en React

Figura 18 - CMS considerados para el Servidor Remoto. Las características de cada uno corresponden al período de diciembre de 2021, cuando se seleccionaron las diferentes tecnologías del proyecto.

Como puede observarse en el cuadro anterior, todas las opciones ofrecen soporte para PostgreSQL, por lo que empatan en ese sentido. Supabase se descartó rápidamente por no encontrarse lista para producción. Al momento de seleccionar las tecnologías, KeystoneJS recién estaba disponible de manera estable, pero Strapi llevaba más tiempo en producción. Sin embargo, este producto ofrece un plan gratuito limitado a sólo tres tipos de usuarios, cantidad que podría no ser suficiente en el futuro del sistema. Un último punto a tener en cuenta es que Supabase no ofrece un Admin UI, salvo si se elige contratar su hosting. Las otras opciones sí lo ofrecen, pero ninguna utiliza Vue.js, sino React o derivados.

Finalmente, se seleccionó KeystoneJS. Se trata de un producto estable, completamente gratuito y libre. No está tan orientado al comercio online y *blogs* como Strapi, sino que soporta un caso de uso más general, y por ende, se adapta al sistema propuesto. Es utilizado por el demandante en otros sistemas, por lo que facilita la continuación del proyecto por su equipo de desarrolladores, y soporta la base de datos sugerida por él.

Framework Front-End

Inicialmente, el demandante había decidido utilizar Vue.js para las aplicaciones web, como en sus otros sistemas. Sin embargo, los integrantes de ambos proyectos tienen mucha experiencia utilizando React, por lo que fue sugerido. Además de agilizar el desarrollo ya que no habría curva de aprendizaje, ofrece varias ventajas independientemente del programador.

Por un lado, cuenta con una comunidad mucho más grande y es soportado oficialmente por Facebook, a diferencia de Vue.js que es administrado por un equipo con menos recursos. Además, hay más cantidad y variedad de librerías compatibles. Por otro lado, su sintaxis es más simple que la de Vue.js, facilitando el mantenimiento del código. No mezcla HTML con JavaScript, sino que ofrece directivas propias más sofisticadas y cercanas a un lenguaje de programación que a uno de marcado a través de JSX. El pasaje de propiedades entre componentes es más fácil, ya que la API es mínima pero más flexible que la de Vue.js, que requiere gestionar más directivas.

Otra ventaja de React sobre Vue.js se debe a la elección de KeystoneJS. La interfaz de administrador que ofrece dicha tecnología es construida con React, por lo que es preferible tener dos aplicaciones web con el mismo framework para su mantenimiento y evolución. Ninguna de las tecnologías de back-end comparadas ofrece una interfaz realizada con Vue.js, por lo que fue preferible para el demandante utilizar una ya hecha en React que tener que construir otra desde cero.

Por último, al momento de iniciar el desarrollo, la última versión de Vue.js todavía no era soportada por la librería de componentes Vuetify. Este módulo es usado en todos las interfaces de usuario de Ponce AgTech. Si se utilizaba Vuetify, el Front-End Remoto debería ser construido con una versión anterior de Vue.js o con una versión no estable de la librería, dificultando su mantenimiento de cualquier manera.

Finalmente, debido a esta situación y las ventajas mencionadas, el demandante aceptó la sugerencia de utilizar React. De esta manera, podrá evaluar la tecnología en mayor profundidad y decidir si la extiende a otros productos.

Protocolos de Comunicación

Para el subsistema nube se decidió utilizar diferentes protocolos de comunicación, en vez de uno sólo para todos los componentes. Esto se debe a que existen diferentes situaciones de intercambio de mensajes, por lo que se buscó el más adecuado para cada una.

GraphQL
Al elegir KeystoneJS como framework de back-end, había dos posibilidades para realizar las operaciones generales de creación, lectura, edición y eliminación de entidades. El CMS crea de serie todo lo necesario en GraphQL, pero también permite extender el servidor con REST. Esto ya es un punto a favor para elegir GraphQL, pues al definir las entidades y sus atributos, no es necesario ningún esfuerzo adicional para poder realizar consultas. Más allá del ahorro de recursos que esto implica, el lenguaje ofrece ventajas inherentes.

Por un lado, no es necesario definir múltiples endpoints por cada tipo de operación a realizarse. Toda consulta se realiza a un único endpoint general, y lo que se necesita se especifica en el cuerpo. La sintaxis es sumamente simple y clara, permitiendo indicar cuáles atributos se quiere obtener, y cómo se deben filtrar, ordenar y/o paginar los resultados. Además, se puede trabajar sobre datos relacionados en una misma consulta. Todo esto resulta en una mayor eficiencia en la comunicación, y en un código más legible y menos propenso a errores.

Por estas razones, GraphQL fue elegido para todo lo relacionado a operaciones de ABM, que representan la gran mayoría de mensajes del sistema.

REST
Aunque GraphQL ofrece muchas ventajas cuando se trabaja con ABM de datos, no resulta práctico para disparar flujos en otros sistemas, pues fue diseñado como lenguaje de consulta. Es por ello que para enviar órdenes al subsistema de campo se implementaron endpoints con REST en el Mapper. Cuando el usuario cambia el estado de una válvula, bomba o programa, el Servidor Remoto envía una orden a un endpoint que representa dicha acción en el Mapper. Como no es necesario obtener datos, sino simplemente un mensaje de éxito o error, se puede construir un pequeño servidor HTTP en el Mapper. Éste recibe entradas, las valida y envía la orden al subsistema de campo correspondiente. Construir algo así con GraphQL implicaría más trabajo, y la respuesta a dar no corresponde con el objetivo de la tecnología.

SSE
En el sistema propuesto se informa en tiempo real los valores de presión de las bombas y el estado de la conectividad con el campo. Aunque GraphQL ofrece suscripción a eventos, KeystoneJS todavía no ofrece soporte para ello. Esto representa una de las principales desventajas del framework de back-end, pues es necesario implementar otra solución para

ese requerimiento específico. Realizar esa suscripción desde cero es complejo y no se justifica para los pocos datos que se manejan en tiempo real.

Descartada esa posibilidad, podría usarse un WebSocket [5] [6] o un SSE. El primero tiene como caso de uso la comunicación bi-direccional, por ejemplo un chat en vivo. Por ende, su implementación es algo más compleja que la de un SSE, pensado para casos unidireccionales. Justamente, notificar al usuario o mantenerlo actualizado sobre un dato es el caso de uso de esta última tecnología. Su implementación es simple y flexible, pues sólo es necesario ofrecer un endpoint HTTP para que los clientes se suscriban, y los mensajes únicamente tienen que definir un tipo, con el cuerpo libre. Así, se puede crear un formato de mensaje para cada clase de evento. Además, los SSEs implementan reconexión automáticamente al endpoint de suscripción, y en general dan menos problemas a la hora de pasar por un firewall. Por último, JavaScript, tanto del lado del navegador como del servidor, ofrece soporte nativo, sin necesidad de depender de librerías de terceros.

Testing

Para los tests unitarios del Servidor Remoto se decidió trabajar con Jest. Este framework de testing es muy popular en proyectos hechos con JavaScript. De hecho, KeystoneJS ofrece soporte para Jest, por lo que resulta natural utilizarlo. Si en el futuro se desea testear las interfaces de usuario también puede aplicarse la misma tecnología, ya que está diseñada para cualquier contexto. Por otro lado, la herramienta ofrece la posibilidad de simular APIs y métodos de componentes que no forman parte del test en sí, lo que es muy útil para aquellos sectores de código donde el Servidor Remoto debe enviar una solicitud al Mapper, o viceversa. Otros puntos a favor son la experiencia profesional de los integrantes del equipo utilizando el framework, y que Ponce AgTech también lo aplica en otros proyectos.

Por último, para medir la cobertura del código se eligió Majestic. Esta herramienta es compatible con Jest, y construye automáticamente una interfaz de usuario que indica la cobertura del código en cada archivo, y permite correr los tests con sólo un click.

7.5 - Diseño de Seguridad

Implementación General

Para la implementación del subsistema nube se propone instalar todos los componentes dentro de una red no conectada a *Internet* pública, ofreciendo un mayor nivel de seguridad. Al tratarse de un entorno aislado donde todos los elementos son conocidos, se logra un mayor control sobre la comunicación entre ellos. El único contacto con *Internet* es a través del Firewall y del Proxy Reverso, por lo que se reduce la posibilidad de ataques externos.

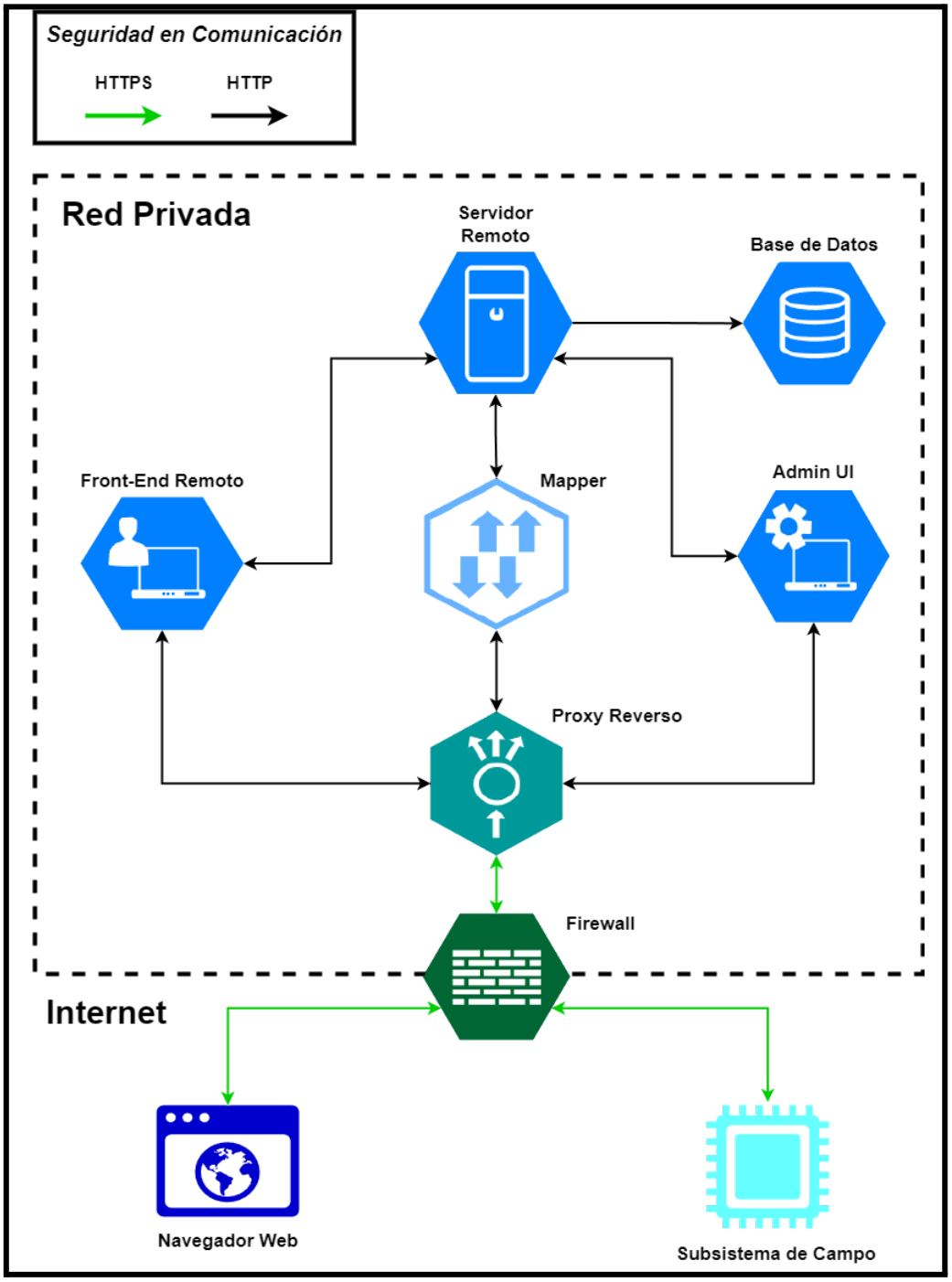

Figura 19 - Implementación del subsistema de nube dentro de una red privada.

Mediante el Firewall se configuran y aplican reglas que habilitan sólo ciertos puertos dentro de la red. En este caso, los únicos puertos que se abren a Internet son los que necesite el Proxy Reverso. Los otros componentes también exponen algunas entradas, pero únicamente dentro de la red privada, ya que se indica en cada regla cuáles direcciones de Protocolo Internet (*Internet Protocol - IP*) podrán utilizarlos. Son las direcciones IP estáticas y conocidas de los otros elementos que consumen sus servicios. Por ejemplo, la Base de Datos tendrá un puerto abierto, pero que sólo podrá ser accedido desde la dirección IP del Servidor Remoto.

El Proxy Reverso se coloca *delante* de todos los demás servidores y envía las solicitudes que llegan a ellos, según corresponda. Así, se oculta la existencia de los componentes reales del sistema a los agentes externos. Además, se los desliga de la responsabilidad de implementar comunicación encriptada. Como el Proxy Reverso es el punto de contacto con *Internet*, se comunica con el exterior usando Protocolo de Transferencia de Hipertexto Seguro (*Hypertext Transfer Protocol Secure - HTTPS*). De esta manera, todo dato sensible es encriptado, lo que dificulta su lectura por cualquier atacante que lo intercepte. El resto de componentes del subsistema de nube, al estar *detrás* del Proxy Reverso y dentro de la red privada, no necesitan encriptar los mensajes.

Autenticación y Permisos de Usuarios

Todos los usuarios del Front-End Remoto y el Admin UI son identificados unívocamente por su dirección de correo electrónico, que utilizan para iniciar sesión junto a una contraseña. Este valor no se almacena en la Base de Datos, sino que se registra su hash, limitando la posibilidad de que un atacante lo descubra.

Por otro lado, todo usuario tiene uno de dos roles posibles. Puede ser un administrador, que tiene acceso a todas las aplicaciones web y es capaz de gestionar todos los campos, o un usuario normal, que sólo puede gestionar sus campos a través del Front-End Remoto. Este control fue implementado mediante la creación de permisos de acceso a los recursos que ofrece KeystoneJS [(7)] [(8)]. En el futuro, se podrían extender de manera simple con nuevos roles que surjan.

La sesión de cada usuario es sin estado (*stateless*) [(9)]. Esto significa que no hay datos de sesión en el Servidor Remoto, sino que todo se almacena en un token que se recibe al iniciar sesión y que debe usarse en cada solicitud. El token contiene el ID y rol del usuario, por lo que el Servidor puede aceptar o rechazar cada solicitud, en función de si se trata de alguien autenticado y con los permisos suficientes.

Como el Servidor Remoto no almacena datos de sesión, se simplifica su diseño e implementación. No hay que realizar ningún trabajo extra para lograr la consistencia de los datos, y en caso de querer escalar el sistema a múltiples servidores, el usuario puede utilizar el mismo token en todos. Esto es especialmente útil para transacciones distribuidas, balanceo de cargas o en caso de que uno de los servidores falle y deba ser reemplazado por otro. La principal desventaja de una sesión stateless es que se genera un mayor consumo de red al repetir la misma información en cada solicitud, pero es despreciable. Sí

es importante ocultar los datos a atacantes que intercepten las comunicaciones, que se soluciona mediante la red privada y el uso de HTTPS.

Usuarios del Subsistema Campo

En conjunto con los encargados del subsistema de campo se decidió no implementar autenticación de usuarios a través del Servidor Remoto, debido a las características generales de todo campo.

El acceso a la red es inestable, lo que dificulta que los usuarios puedan autenticarse contra una base de datos remota. Si se utiliza una combinación de base de datos local y remota, se complejiza el sistema por la gestión de datos para evitar inconsistencias. Si se utiliza una base de datos local solamente, aún sería necesario contar con acceso a la red para que los administradores puedan actualizar credenciales de manera remota, por ejemplo, si alguien olvidó su contraseña.

Además, por la misma razón de la inestabilidad de la red, se decidió que el subsistema de campo ofrezca funcionalidad reducida, orientada principalmente a tareas operativas o de emergencia. Aunque se puede apagar y encender válvulas o bombas, y detener el programa actual, no es posible encender programas almacenados en la base de datos ni modificarlos. Por ende, un intruso tiene menos posibilidades de generar un daño a través de software, y si quisiera sabotear el sistema podría directamente desconectar o romper los componentes físicos del sistema de riego.

De hecho, la seguridad en el lugar no es responsabilidad de ninguno de los dos subsistemas. Tanto el perímetro del campo como la casilla donde se encuentra el hardware se encuentran protegidos de manera física, y sólo pueden ser accedidos por personal autorizado.

Autenticación del Mapper

El Mapper, en su rol de intermediario entre ambos subsistemas, funciona para algunas operaciones como cliente del Servidor Remoto. Dicha interacción es mediante GraphQL, al igual que sucede con ambas aplicaciones web. Sin embargo, utiliza un endpoint específico, que tiene como objetivo autenticar sus solicitudes de manera diferente a las de un usuario normal.

Como no se trata de un humano interactuando a través de un navegador, no tiene sentido que deba autenticarse y manejar sesiones como tal. Si se utilizara el mismo endpoint, una opción sería implementar algún tipo de middleware que reconociera las solicitudes del Mapper por alguna característica (dirección IP por ejemplo) para concederle los permisos suficientes. Por ese middleware pasaría toda solicitud, incluso las de los front-end, agregando una sobrecarga innecesaria.

Por ello se usa un endpoint aparte, que ofrece la misma funcionalidad pero con diferente autenticación. Mediante el header Authorization de HTTP se envía un identificador y contraseña específicos. Una vez autenticado, se ejecuta la solicitud como siempre. Gracias a la extensibilidad de KeystoneJS, se pueden agregar endpoints específicos y acceder en

ellos a la lógica de GraphQL en su totalidad. Incluso el formato de las solicitudes y de las respuestas es prácticamente igual, por lo que no es necesario conocer dos APIs totalmente diferentes.

Intercambio de Recursos de Origen Cruzado

Como se observa en el esquema de la arquitectura, las aplicaciones web y el Servidor Remoto se encuentran cada uno en un origen diferente. Normalmente, los navegadores sólo permiten que las páginas realicen solicitudes hacia su propio servidor. Mediante el Intercambio de Recursos de Origen Cruzado (*Cross-Origin Resource Sharing - CORS*), se indica a los navegadores que deben autorizar las solicitudes que ambas aplicaciones realicen hacia el Servidor Remoto, a pesar de encontrarse en otra ubicación.

Al mismo tiempo, se mantiene bloqueado cualquier otro origen al que las aplicaciones intenten enviar solicitudes. Esto es importante para reducir el impacto que pueda tener un atacante sobre el sistema. Por ejemplo, si lograra introducir en una de las aplicaciones un script para enviar datos sensibles a un servidor suyo, CORS bloquearía las solicitudes hacia allí por no encontrarse dentro de los orígenes autorizados.

Auditoría

Tanto el Servidor Remoto como el Mapper están configurados para llevar una bitácora de eventos del sistema. Esto es útil para monitorear el estado de los componentes, pero también para auditar qué ha sucedido en caso de un error, brecha de seguridad u otro incidente.

La bitácora se puede configurar al momento de iniciar el sistema. Se puede personalizar el formato de nombre de cada archivo, la frecuencia con la que son creados, su tamaño máximo, la cantidad máxima de archivos a conservar y si se deben comprimir automáticamente al completarse. De los parámetros mencionados, los más importantes son la frecuencia y la cantidad máxima de archivos, pues influyen en el acuerdo de nivel de servicio (*Service Level Agreement - SLA*). Para dar soporte por un cierto período de tiempo, es necesario que se registre lo acontecido durante al menos ese plazo. Con esa información se podrán resolver errores más rápido, reduciendo parámetros cruciales como el tiempo medio de reparación (*Mean Time To Repair - MTTR*). Debido a que es un sistema de control del que depende una inversión alta como es el cultivo, es importante a la hora de la comercialización ofrecer un SLA adecuado.

Los mensajes de bitácora almacenados poseen un nivel, una marca de tiempo en la hora local del servidor y un contenido que es libre. Existen tres niveles: errores (fallas en el sistema), información (acciones de usuarios) y debug (como información pero más detallado para resolver errores).

8 - Producto

A través de la solución implementada se ha logrado conseguir un producto que cumple con los requerimientos funcionales, no funcionales y restricciones identificadas. Por un lado, el Front-End Remoto ofrece a los usuarios el monitoreo y control de sus dispositivos, lotes y programas. Por otro lado, a través del Admin UI, a las funcionalidades antes mencionadas, se agrega la gestión de las bombas, campos, usuarios y válvulas en cuanto a su creación, modificación o eliminación.

8.1 - Características del Front-End Remoto

El sistema cuenta con dos tipos de usuarios que pueden acceder al mismo. Se distinguen el usuario común y el administrador. Ambos pueden ingresar al sistema a través de la pantalla de log-in, mediante su email y contraseña.

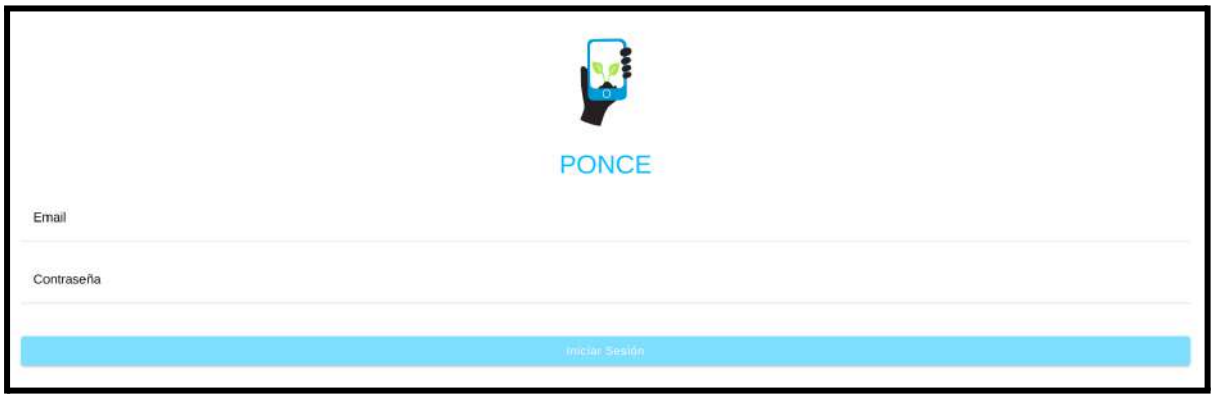

Figura 20 - Inicio de sesión.

Una vez que el usuario ingresa al sistema, podrá visualizar en el menú lateral la lista de los campos a los que tiene acceso. Si el usuario es administrador se listarán todos los campos presentes en el sistema. Al hacer click en alguno, será redirigido a sus detalles. De esta manera, ya no es necesario que el cliente vaya físicamente a cada uno de sus campos para regar y revisar el sistema. Esto supone un enorme ahorro de tiempo y de costos, ya que puede responder más rápido a los cambios en las condiciones climáticas, problemas, etc.

Figura 21 - Detalle de un campo.

Al seleccionar un tipo de dispositivo, se muestran los datos de las bombas o válvulas, según corresponda. Desde allí, se puede monitorear y gestionar su encendido y apagado, como así también *forzarlas*.

Figura 22 - Sección válvulas.

Al forzado de un dispositivo también se lo denomina *override.* Se utiliza principalmente para la detención de su funcionamiento debido a roturas o emergencias. Al indicar que una bomba o válvula se fuerza, pasa a control manual y si forma parte de algún programa, no se seguirán las instrucciones hasta que se quite el *override*.

A diferencia de las bombas y válvulas, cuya creación es posible sólo desde el Admin UI, los lotes y programas pueden gestionarse desde el Front-End Remoto. La división del campo mediante lotes facilita un riego localizado según diferentes criterios. El usuario puede agrupar las válvulas con cualquier criterio, pero generalmente lo hará según la localización.

Figura 23 - Detalle de un lote.

Ante la necesidad de crear un plan de riego, el usuario podrá generar un nuevo programa ingresando nombre y fecha de inicio. El mismo se compone de una serie de pasos, que poseen hora de inicio, duración y cuál es la entidad asociada: lote, válvula o bomba. De esta manera, no es necesario estar pendiente del sistema, sino que este ejecutará regularmente las instrucciones.

Un usuario puede diseñar un programa con cualquier criterio. Por ejemplo, puede crear uno específico para períodos de heladas, donde se riega para contrarrestar el efecto del frío sobre el fruto. Otro caso podría ser para que durante los momentos del día donde el costo de electricidad es mayor sólo se mantengan prendidas las válvulas y bombas vitales. En vez de tener que prender y apagar el resto todo el tiempo, el usuario puede confiar en el sistema automatizado, y dedicar su tiempo a otras tareas.

Figura 24 - Edición de un programa de riego.

Como se ve en la imagen, si es necesario realizar algún ajuste, el usuario puede modificar los pasos, la información básica del programa e incluso eliminarlo, si el mismo queda obsoleto.

8.2 - Características del Admin UI

La gestión de todas las entidades del sistema se encuentra disponible en el Admin UI. A diferencia del Front-End Remoto, desde aquí pueden darse de alta, modificar y eliminar campos, válvulas, bombas, usuarios y organizaciones. Otra diferencia a destacar es que sólo los administradores tienen acceso a la plataforma.

Esta herramienta es útil para varios roles dentro de Ponce AgTech. Por un lado, alguien del área comercial la utilizará para dar de alta una nueva organización, sus campos y usuarios. También ofrece valor para el área de soporte al cliente. Por ejemplo, si necesita ayuda para gestionar su sistema de riego.

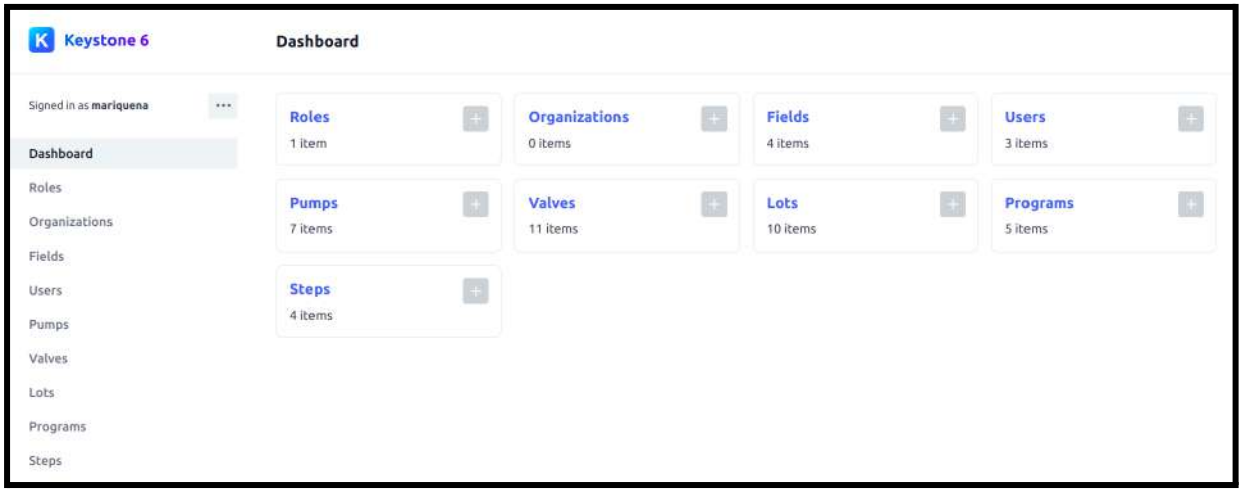

Figura 25 - Dashboard del Admin UI.

8.3 - Comparación con Producto Existente

El sistema propuesto supone mejoras sustanciales para el cliente por los siguientes motivos:

- *Mayor libertad en el manejo de válvulas.* No hay limitaciones en la cantidad de válvulas que soporta el sistema y se pueden repetir dentro de un mismo programa en diferentes pasos.
- *Gestión centralizada de varios campos*. En el sistema previo, cada campo tenía su propia interfaz web y conjunto de usuarios. Por ende, el cliente tenía que iniciar sesión varias veces para administrar cada uno. En el sistema propuesto, un usuario puede controlar desde la misma interfaz todos los campos a los que se le concedió acceso.
- *Mejor experiencia de usuario.* La interfaz es más simple e intuitiva y se adapta a pantallas de distintos tamaños, ya sea móvil o de escritorio.
- *Actualización en tiempo real de la presión y la conectividad*. El usuario puede detectar cambios en la presión de las bombas y de la conectividad con el campo sin necesidad de refrescar manualmente la interfaz web.

8.4 - Benchmarking

En el mercado de sistemas de riego por microaspersión y goteo, los principales fabricantes son Rainbird [(10)] y Hunter Industries [(11)], ambos de Estados Unidos. Rainbird se especializa en el aspecto mecánico de los sistemas de riego (tuberías, bombas, válvulas, etc.) y ofrece como accesorio un sistema de control remoto. En el caso de Hunter Industries, sí se ofrecen sistemas automatizados como producto principal. Aunque son compatibles con uso agrícola, tienen como caso de uso campos de golf, parques y sistemas hogareños, por lo que no resultan ideales al ser sistemas de menores dimensiones y con una mecánica diferente.

Ambos fabricantes ofrecen productos que se integran únicamente con sistemas de fabricación propia, mientras que el sistema propuesto está pensado para funcionar con cualquiera. En cuanto al soporte de plataformas, la propuesta de Rainbird sólo corre sobre Windows, mientras que Hunter Industries ofrece versiones web y nativas para iOS y Android.

El cliente objetivo del demandante busca definir programas de riego en función de grupos de válvulas y de bombas con horarios fijos de encendido y apagado. Los controladores de Rainbird determinan los programas en función de la cantidad de agua regada y/o las horas de funcionamiento de cada válvula. Esto provoca que el cliente tenga que estar controlando el riego frecuentemente, para evitar que se desplace a horarios pico, donde el proveedor eléctrico cobra cuantiosas multas. Los productos de Hunter Industries sí permiten personalizar los programas en función de horarios fijos, pero no están pensados para la agricultura.

En cuanto al soporte, ambos competidores son extranjeros. Esto provoca un mayor costo por impuestos y plazos de tiempo más largos por los trámites de importación. Además, no existe una filial local de ambos fabricantes, sino que son representados por proveedores genéricos de sistemas de riego y de jardinería localizados en la Ciudad Autónoma de Buenos Aires. Teniendo en cuenta que la mayoría de la producción de kiwi se realiza en Mar del Plata y Sierra de los Padres, Ponce AgTech puede ofrecer un soporte inmediato, especializado, de menor costo y a medida de cada cliente.

Por último, se desconoce el precio de los competidores. El área comercial de Ponce AgTech tiene previsto ofrecer el sistema en forma de servicio por un costo de 200 USD mensuales. Los sensores son propiedad de Ponce y son entregados en comodato durante la duración del contrato.

Fabricante	Hunter	RAIN BIRD	PONCE
Origen	California, Estados Unidos	California, Estados Unidos	Mar del Plata, Argentina
Producto	Sistemas de riego para campos de golf, parques y hogares, con automatización y control	Sistemas de riego agrícola mecánicos, sistema de control como accesorio	Sistema de control de riego para uso agrícola
Compatibilidad	Sólo con productos de Hunter	Sólo con productos de Rainbird	Cualquier sistema de riego por microaspersión o goteo
Plataformas	Android, iOS, Web	Windows	Web
Programas de Riego	Por horarios fijos, orientados a campos de golf, parques y hogares	Por cantidad de agua regada y funcionamiento de válvulas, orientado al uso agrícola	Por horarios fijos, orientados al uso agrícola
Soporte	A través de proveedor genérico, productos importados	A través de proveedor genérico, productos importados	Local, directo, a medida

Figura 26 - Comparación de competidores con el sistema propuesto.

9 - Memoria del Proyecto

9.1 - Objetivos

A continuación, se listan los objetivos planteados en el proyecto. Por cada uno, se especifica cuál fue el grado de cumplimiento obtenido, de qué manera se logró, o por el contrario, los motivos por los cuales no se alcanzó con lo esperado.

Objetivo Global

- *Impactar positivamente en la región*: objetivo aún no cumplido. Debido a que la solución no ha sido validada en campo todavía, no se puede decir que impacta positivamente en la región. De todas formas, tiene el potencial para lograrlo una vez que salga a producción.

Objetivos Específicos

- *Diseñar una nueva versión del subsistema que satisfaga los requerimientos funcionales de productores y los no funcionales del demandante*: la arquitectura propuesta permite implementar las funcionalidades esperadas de manera simple y escalable. Las tecnologías demandadas por Ponce AgTech fueron utilizadas excepto Vue.js, que fue reemplazada por React debido a las ventajas nombradas en la sección *Tecnologías seleccionadas*. El cambio resultó en un aporte significativo al proyecto y bien recibido por el demandante, por lo que el objetivo fue cumplido.
- *Implementar el subsistema*: objetivo cumplido. Luego del análisis y diseño, prosiguió la implementación del subsistema. Durante esta etapa, se realizaron pruebas en profundidad para asegurar la calidad del código y su correcto funcionamiento. Se logró obtener una solución que le permite al usuario realizar todas las funciones esperadas: prender y apagar válvulas y bombas remotamente, ejecutar programas de riego de manera autónoma, controlar la presión y el estado general de los dispositivos.
- *Diseñar la solución de forma tal que se facilite su evolución por parte del demandante*: objetivo cumplido debido a la utilización de tecnologías previamente aplicadas por Ponce AgTech, junto con la amplia documentación del sistema generada. Los tests creados podrán usarse en el futuro para controlar que cada cambio hecho respete las reglas de negocios ya aplicadas.

9.2 - Trabajo en Equipo

Inicialmente, al presentarse la idea del trabajo por parte de Ponce AgTech, se decidió conformar un equipo de cuatro integrantes. Sin embargo, debido a la magnitud del sistema, el proyecto se dividió en dos trabajos finales: subsistema campo y subsistema nube. Cada equipo centraría sus esfuerzos en la sección correspondiente, logrando así un producto más completo.

Durante el análisis y el diseño del sistema, el trabajo en equipo fue crucial y de gran ayuda para las siguientes etapas. Relevar y definir la lista de requerimientos con el demandante llevó varias reuniones en las que ambos grupos participaron activamente. Diseñar una solución junto a otro equipo aportó un valor agregado a la experiencia del proyecto. El obtener diferentes enfoques y discutir cómo lograr la mejor solución posible enriqueció a los integrantes al enfrentarlos a circunstancias que ocurren de manera frecuente en el ámbito laboral.

Previo a la implementación de una nueva funcionalidad, a través de una llamada grupal, se tomaba del backlog una tarea y entre ambos equipos se desglosaban las subtareas necesarias para llevarla a cabo. De esta manera, cada grupo podía implementar su solución por separado y a su vez, conocer el avance del otro. Trabajar en paralelo permitió que ante dudas en la implementación se pudiera no sólo consultar al integrante del propio equipo si no también a los demás. Finalmente, en la revisión de código se corregían errores o malas prácticas con el objetivo de mejorar el producto final.

En cuanto al testing del código, se generaron tablas donde se definieron entre los integrantes del equipo del subsistema nube qué casos serían probados. Al igual que en las etapas antes mencionadas, se debatió en conjunto cómo proseguir, buscando siempre ser lo más eficiente posible.

Si bien la articulación con otro equipo agregó complejidad al proyecto, se puede concluir que la experiencia resultó muy enriquecedora y formadora para los integrantes.

9.3 - Métricas y Análisis de las Etapas

Como se mencionó en la sección 3, el cronograma se planteó en formato de cascada principalmente, considerando etapas de análisis, diseño, implementación, testing y cierre. Debido a que la mayoría de los integrantes trabajaban y todos cursaban por lo menos una materia, era difícil precisar una cantidad de horas diarias para el proyecto. A modo de estimación, se calculó que se podrían dedicar entre ocho y diez horas por semana por persona. Por ello, el Gantt se presentó en semanas y no en horas o días. De esta manera, el proyecto se estimó con una duración de entre 592 y 740 horas, repartidas a lo largo de 37 semanas. Esta última cifra contempla un margen de error por imprevistos, enfermedad, etc. Los números corresponden a los dos miembros del subsistema nube.

Finalmente, la forma en que se desarrolló el trabajo fue más cercana a una metodología ágil. Al inicio se realizó todo el análisis y gran parte del diseño. Luego, se resolvió cada funcionalidad importante mediante iteraciones de diseño, implementación y testing. Se siguieron los lineamientos de Scrum. Como Ponce AgTech utiliza la misma metodología, el proyecto se pudo acoplar adecuadamente con su plan de trabajo.

La fecha de inicio fue la proyectada originalmente, pero el trabajo concluyó el 23 de septiembre cuando se entregó la Nota de Finalización. En total, se invirtieron 757 horas a lo largo de 46 semanas.

A continuación, se presentan ambos cronogramas. Los números dentro de cada barra representan la cantidad de horas que se invirtieron durante las semanas que abarca. El detalle de lo acontecido en cada metodología se encuentra luego de los gráficos.

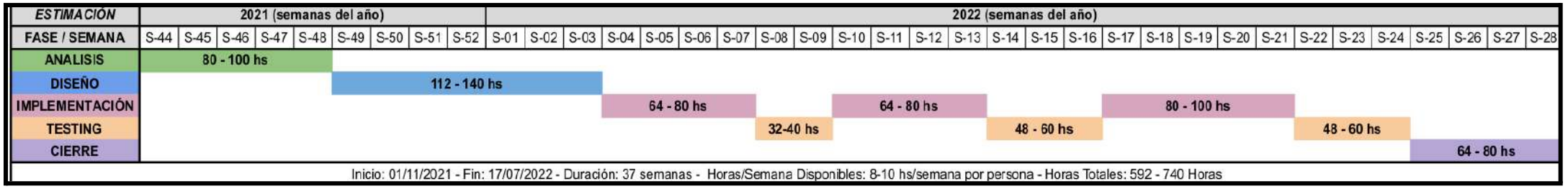

Figura 27 - Cronograma proyectado.

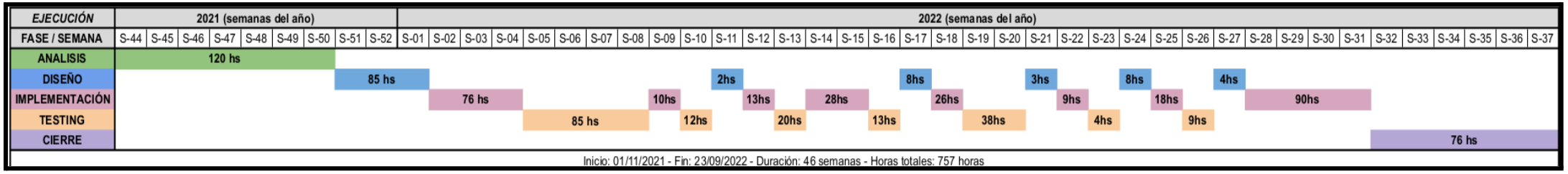

Figura 28 - Cronograma ejecutado.

Análisis

La totalidad del análisis se realizó al inicio del proyecto, en colaboración con los miembros del subsistema campo. En total se invirtieron 120 horas, algo más de lo estimado. Esto se debió principalmente a la dificultad para lograr un listado de requerimientos completo, sin ambigüedades y viable. Se realizaron varias reuniones con el referente funcional, que a veces eran difíciles de concretar por su agenda. Además, fue necesario delimitar el alcance, para lograr una funcionalidad interesante, útil y realizable. Por último, surgió la oportunidad de realizar una reunión con un cliente actual en el campo. Esto implicó algunas horas extras no contempladas por el viaje hasta la locación, pero fue sumamente productivo, al poder ver el sistema de riego en persona y conversar directamente con un usuario.

Diseño

La mayoría del diseño se concretó una vez finalizado el análisis. Luego, a medida que se implementaban ciertas funcionalidades, se produjeron otras discusiones, para determinar APIs y responsabilidades de ambos subsistemas, entre otras cuestiones. Incluso se pudo concretar una reunión con un líder técnico de la empresa demandante, para validar el diseño general y poder avanzar con mayor seguridad. Esto repercutió en poco retrabajo durante el resto del proyecto.

En total, se destinaron 110 horas, algo menos del mínimo estimado. Al haber comprendido la totalidad del problema y los requerimientos esperados, se planteó una arquitectura general sin mayores dificultades. La elección de las tecnologías fue la tarea que más tiempo consumió dentro del diseño, en especial al investigar sobre CMS por la falta de experiencia. Aun así, si en el futuro se trabaja en otro proyecto con herramientas similares, el conocimiento adquirido ayudará a tomar una decisión en este ámbito en menos tiempo.

El retraso en el lanzamiento de Vuetify para Vue 3 no generó demoras en el proyecto, pues se pospuso la implementación del Front-End Remoto para ganar tiempo. Como al final la herramienta no estaba lista todavía, se seleccionó React. La discusión con el demandante fue muy positiva, pues se interesó en la tecnología y sus ventajas.

Implementación

Se había estimado que la etapa de implementación sería la más larga y así fue, pero llevó 270 horas, un poco más de lo proyectado. En general, no hubo mayores dificultades, excepto en el primer tramo. Puede observarse que tuvo una mayor duración, producto de que los integrantes no estaban familiarizados con KeystoneJS. Además, al ir trabajando con esta herramienta se encontraron algunas desventajas. Aunque no supusieron una enorme demora, sí fueron cuestiones no contempladas que requirieron algo de tiempo. El listado completo se encuentra en el Apéndice B.

Otra razón de retraso al inicio fue la configuración del entorno de desarrollo, en especial en cuanto a la construcción del flujo de las tareas. Aun así, esta demora inicial repercutió en un trabajo más ágil durante el resto del proyecto.

En todo momento se buscó la mayor reutilización de código posible. Por ejemplo, el desarrollo de lo relacionado a las bombas implicó más tiempo que el dedicado a válvulas, pero justamente lo implementado para las primeras ahorró ese tiempo para las segundas por su similitud.

La última etapa de implementación fue para el desarrollo del Front-End Remoto, que se pospuso para esperar a que Vuetify diera soporte a Vue 3. Como esto no sucedió, se trabajó con React. Al ser una herramienta ya conocida por los integrantes, se pudo construir la interfaz de usuario en poco tiempo.

Testing

El testing del Servidor Remoto se desarrolló en paralelo con su implementación: cada funcionalidad agregada se testeaba. En total, se consumieron 181 horas, más de lo esperado. Lo más llamativo no fue esto, sino que se dedicó prácticamente el mismo tiempo a implementarlo como a testearlo. Si se restan las 90 horas dedicadas a desarrollar el Front-End Remoto, quedan 180 horas de programación, contra 181 de testing.

El haber dedicado suficiente tiempo a las pruebas aumentó la calidad del resultado notoriamente, y es una buena práctica para todo proyecto. A priori, algunos tests de entradas podrían parecer poco valiosos, pero ayudan a su validación, lo que representa el primer refuerzo en pos de la seguridad.

Muchas veces el testing es poco valorado y dejado para el final de un proyecto. El hecho de haberle dedicado tiempo de manera constante permitió apreciar su importancia y utilidad. Además, queda como aprendizaje para un próximo proyecto el tiempo que realmente es necesario, y que debe ser intercalado en mayor medida con las etapas de desarrollo.

Al igual que con la implementación, la primera etapa de pruebas fue la más larga. Esto se debió a que era necesario configurar las herramientas requeridas. Luego el trabajo fue más ágil gracias a lo construido.

Cierre

La etapa de cierre consistió por un lado en la redacción del informe del proyecto. Esto se realizó relativamente rápido pues ya se contaba con todo lo necesario en diferentes documentos. Durante las horas de esta etapa se recopiló y ordenó esa información. Por último, se realizó una demostración del sistema al demandante, especialmente al equipo de desarrollo para facilitar la transición. En total, se invirtieron 76 horas, algo menos del máximo estimado.

Resumen

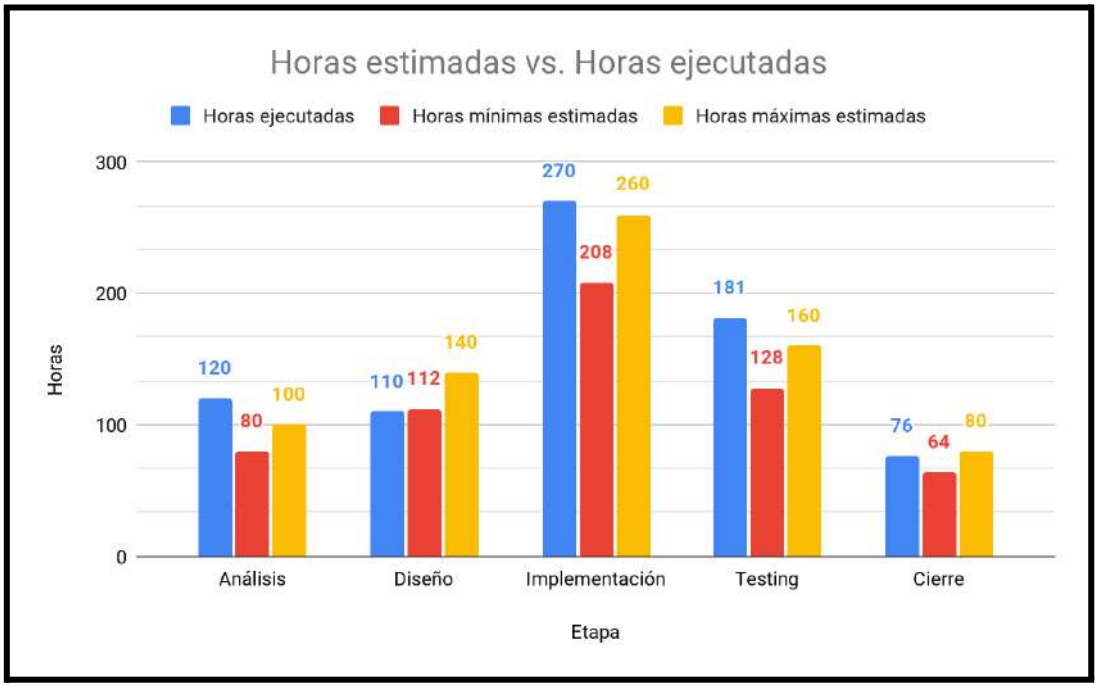

Figura 29 - Comparación de tiempos por etapas.

En general, cada etapa consumió algo más de tiempo que lo estimado, salvo la de diseño que duró menos. Como se describió anteriormente, surgieron algunos imprevistos que alargaron las etapas, sumado a la inexperiencia por tratarse del primer proyecto. Además, hubo varios momentos donde el trabajo tuvo que esperar, por los estudios, empleos y cuestiones personales de cada integrante, lo que hizo que se desarrollara durante más semanas de lo esperado.

Aun así, el resultado es positivo, pues se trata del primer proyecto realizado completamente por los integrantes. Sin dedicación a tiempo completo, el trabajo se pudo llevar adelante sin mayores dificultades y la estimación inicial no estuvo tan alejada de la realidad.

Trabajar con bandas de mínimos y máximos de tiempos para cada etapa fue una buena estrategia para estimar la duración del proyecto. Es una práctica a conservar en el futuro, pero tratando de reducir lo más posible su diferencia. De esta manera, los presupuestos y la entrega del producto generan menos incertidumbre en el cliente, y hay menos riesgos económicos. Por ejemplo, si se promete cumplir con los tiempos mínimos, y finalmente estos se superan, se pierde dinero. En cambio, si se promete cumplir con los máximos, quizá resultaría inviable para el cliente.

9.4 - Resultados de Testing

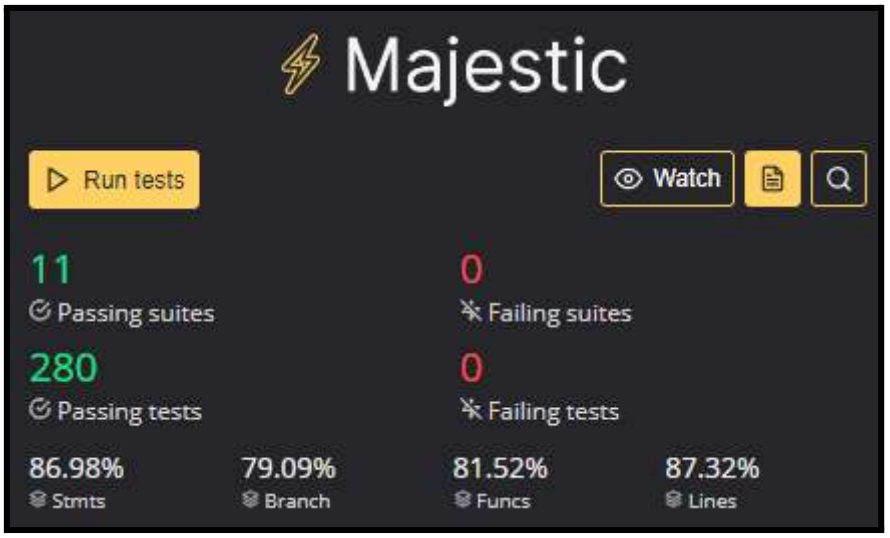

Figura 30 - Cobertura del código del Servidor Remoto.

Como se mencionó antes, prácticamente por cada hora de implementación a nivel back-end se realizó una hora de testing. Sobre el código del Servidor Remoto se construyeron 280 tests unitarios en total. El listado completo puede consultarse en el Apéndice.

Este esfuerzo trajo sus frutos: con cada funcionalidad testeada, se encontraron y resolvieron varios errores apenas surgieron. Además, los tests implementados quedan disponibles para el futuro del proyecto, pues cada vez que se modifique algo validarán que se respeten las reglas de negocio. Las pruebas generaron una cobertura de aproximadamente el 87% de las líneas y declaraciones. Es una cifra más que aceptable y en concordancia con el estándar del demandante en sus otros sistemas.

Las líneas que quedaron sin cubrir corresponden a la lógica del heartbeat y de los eventos SSE. Esto se debe al error en el manejo de la caché de Node.js por parte de KeystoneJS, explicado en el Apéndice B.

10 - Conclusiones

El proyecto presentado tenía como objetivos analizar la problemática de la automatización del riego de kiwi, diseñar una solución, e implementar un producto que tuviera el potencial de impactar positivamente en la región. Además, el sistema debía seguir los lineamientos indicados por Ponce AgTech, empresa demandante del proyecto, que luego continuaría el desarrollo. Estos objetivos fueron desarrollados según lo descrito en el informe y se cumplieron satisfactoriamente.

Debido a la magnitud del sistema, su desarrollo requirió de dos duplas. A pesar de que esto suponía un esfuerzo adicional de coordinación y comunicación, el trabajo se pudo llevar adelante sin mayores dificultades, y se vió beneficiado de contar con diferentes puntos de vista para su diseño.

Además de poner en práctica conocimientos y tecnologías vistos durante la carrera, se incorporaron otros nuevos, en especial aquellos no técnicos. Un factor distintivo fue la interacción con diferentes partes interesadas. Se experimentó el diálogo con clientes, el asesoramiento sobre tecnologías, la negociación y la gestión integral de un proyecto. Estas son habilidades fundamentales de un Ingeniero que van más allá de lo técnico y que son cruciales en la industria.

Lo más difícil del proyecto fue estimar su duración. En el total de horas, el resultado no estuvo tan alejado de lo proyectado, pero sí terminó más tarde de lo esperado. Es algo propio de la falta de experiencia, pues se trató de la primera vez que se ponía en práctica.

En conclusión, el resultado fue exitoso. El producto desarrollado facilita la labor de riego del kiwi, al disminuir la presencia de personal en campo, reducir el tiempo de respuesta ante fallas y lograr un riego más eficiente. El sistema impactará notoriamente en las plantaciones de la región una vez que salga a producción, a la vez que podrá continuar siendo desarrollado por el demandante. Desde el punto de vista personal, el aprendizaje fue muy importante: se experimentó por primera vez la gestión de un proyecto, y se integraron muchos de los conocimientos vistos en la carrera.

10.1 - Trabajos futuros

Algunas funcionalidades no fueron agregadas al producto final pues no eran prioritarias y era necesario lograr un alcance adecuado al contexto de un trabajo final. Aún así, los requerimientos fueron relevados y clasificados, por lo que podrían ser implementados en el futuro. A continuación, se detallan las funcionalidades pendientes:

- *Acceso*.
  Incorporar autenticación de usuarios al subsistema de campo, al igual que en el subsistema nube. A pesar de ser una responsabilidad que recae principalmente sobre el primero, es necesario que los datos sean consistentes entre ambos.

- *Alertas*.
  Agregar notificaciones de alerta al usuario ante eventos de interés ocurridos en el sistema, como pueden ser presiones por encima/debajo de límites establecidos o el inicio/detención de programas de riego. También la capacidad de desactivar las alarmas si así lo desea el usuario. Como el sistema ya almacena presiones mínimas y máximas, recibe las presiones de cada bomba periódicamente, y trabaja con programas, se podría extender esta lógica para disparar los mensajes a través de una API de mensajería.

- *Control.*
  Detección de presiones del sistema, con el objetivo de mantenerlo dentro de los límites definidos. Para ello, el sistema debería ser capaz de abrir válvulas de alivio, apagarse y/o enviar mensajes de alerta, como se nombró en el punto anterior. Además, se podría incorporar la capacidad de activar varios programas de riego simultáneamente.

- *Disponibilidad*.
  Capacidad de almacenar y reenviar mensajes entre el campo y la nube, una vez restaurada la conexión, en caso de perder conectividad a Internet. Si bien es una responsabilidad que recae principalmente en el subsistema campo, requiere de colaboración del subsistema nube. Este deberá ofrecer un tratamiento especial para los mensajes de mayor antigüedad.

- *Monitoreo*.
  Detener un programa de riego ante la detección de lluvia, al sobrepasar un mínimo establecido. Podría tratarse de un nuevo tipo de evento enviado desde campo, que mediante el uso de hooks de KeystoneJS sea detectado y se responda en consecuencia.

- *Reportes*.
  Registro y persistencia de los acontecimientos históricos del sistema. Estos datos podrán ser descargados por el usuario, o consultados a través de un dashboard de métricas. Hoy en día ya se registran ciertos eventos, como cambios de presión, conectividad o estados de dispositivos. Mediante hooks, se podrían guardar en una base de datos de históricos, y ser consultados a través de una nueva API.

Apéndices

Apéndice A - Glosario

- API: del inglés Application Programming Interface [12], son mecanismos que permiten a dos componentes de software comunicarse entre sí mediante un conjunto de definiciones y protocolos. La interfaz puede considerarse como un contrato de servicio entre dos aplicaciones. Define cómo se comunican entre sí mediante solicitudes y respuestas.
- Backlog: lista de trabajo ordenada por prioridades para el equipo de desarrollo que se obtiene de la hoja de ruta y sus requisitos [13].
- BPMN: del inglés Business Process Model and Notation [14], es una notación gráfica que brinda a las empresas la capacidad de comprender sus procedimientos comerciales internos en una notación gráfica y brinda a las organizaciones la capacidad de comunicar estos procedimientos de manera estándar.
- CMS: del inglés Content Management System [15], es una aplicación de software que permite a los usuarios crear y administrar el contenido de un sitio web a través de una interfaz de usuario sencilla, sin tener que codificarlo desde cero.
- Git: sistema de control de versiones distribuido de código abierto y gratis [16].
- GraphQL: lenguaje de consulta y entorno de ejecución del lado del servidor para el desarrollo de APIs [17]. Su particularidad y fuerte radica en que el cliente puede solicitar exactamente las propiedades o datos que necesita dentro del recurso en cuestión.
- Heartbeat: señal periódica generada por hardware o software para indicar su normal funcionamiento o para sincronizar otras partes de un sistema informático [18]. Por lo general, se envía un heartbeat entre máquinas a intervalos regulares de tiempo. Si el endpoint no recibe un heartbeat durante un determinado plazo, se supone que la máquina que lo debería haber enviado ha fallado.
- HTTP: del inglés Hypertext Transfer Protocol [19], es un protocolo de capa de aplicación diseñado para transferir información entre dispositivos en red. Un flujo típico a través de HTTP implica que una máquina cliente realice una solicitud a un servidor, que luego envía un mensaje de respuesta.
- HTTPS: del inglés Hypertext Transfer Protocol Secure [20], es la versión segura de HTTP: está encriptado para aumentar la seguridad de la transferencia de datos.
- IoT: del inglés Internet of Things [21], se refiere a la red colectiva de dispositivos conectados y a la tecnología que facilita la comunicación entre los dispositivos y la nube, así como entre los propios dispositivos. El Internet de las Cosas integra las “cosas” de uso diario con Internet.
- IP: del inglés Internet Protocol [22], es un conjunto de reglas que rigen el formato de la comunicación a través de Internet o la red local. Una dirección IP identifica a una red o dispositivo en Internet.
- Jira: software de gestión de proyectos, ampliamente utilizado por equipos de desarrollo ágiles para rastrear errores, historias, épicas y otras tareas [23].

- KeystoneJS: CMS de código libre [24]. A partir de la descripción de las entidades del dominio y sus relaciones, se obtiene una API GraphQL y una interfaz de usuario de administración.
- Linting: herramienta de análisis de código utilizada para marcar errores de estilo, de programación o bugs [25].
- ORM: del inglés Object Relational Mapping [26], es una técnica que permite consultar y manipular datos de una base de datos usando un paradigma orientado a objetos y no realizar consultas SQL.
- Reglas de AirBnB: guía de estilo más popular para la codificación en React [27].
- SQL: del inglés Structured Query Language [28], es el lenguaje estándar para los sistemas de gestión de bases de datos relacionales.
- SSE: del inglés Server Sent Events [29] [30] [31], es una tecnología que permite enviar notificaciones, mensajes o eventos desde el servidor al cliente a través de una conexión HTTP unidireccional.
- TCP: del inglés Transport Control Protocol [32], es un estándar que define cómo establecer y mantener una conexión entre dos hosts e intercambiar flujos de datos. Garantiza la entrega de datos y que los paquetes se entregarán en el mismo orden en que se enviaron.

Apéndice B - Requerimientos

A continuación se listan todos los requerimientos para el sistema que fueron relevados durante la etapa de análisis. Sólo los esperados fueron implementados, mientras que el resto quedan como trabajo futuro.

Requerimientos Funcionales Esperados

Monitoreo

- RFM1: el sistema deberá permitir al Usuario monitorear el estado de la conectividad con el subsistema en campo.
- RFM2: el sistema deberá permitir al Usuario monitorear el estado (ON/OFF) de las bombas eléctricas.
- RFM3: el sistema deberá permitir al Usuario monitorear el estado (ON/OFF) de las válvulas.
- RFM4: el sistema deberá permitir al Usuario monitorear la presión de trabajo de riego, correspondiente a la bomba.

Control

- RFC1: el sistema deberá permitir al Usuario encender/apagar remotamente las bombas.
- RFC2: el sistema deberá permitir al Usuario encender/apagar remotamente las válvulas.
- RFC3: el sistema deberá permitir al Usuario la creación, modificación y eliminación de programas de riego.
    - El sistema deberá permitir al Usuario definir el tiempo de apertura y cierre de cada válvula y bomba en un programa.
    - El sistema deberá permitir al Usuario agrupar diferentes válvulas conformando lotes, cada uno con un nombre asociado. Las válvulas pueden estar en más de un lote a la vez.
    - El sistema deberá permitir al Usuario utilizar la misma válvula o bomba más de una vez en el mismo programa, con el fin de crear programas de mayor duración.
- RFC4: el sistema deberá permitir al Usuario la selección del programa de riego a ejecutar, definido previamente.
- RFC5: el sistema deberá encenderse de acuerdo al horario de inicio del programa seleccionado, y pararse de acuerdo al horario de parada del programa.
- RFC6: el sistema deberá permitir al Usuario forzar (*override*) una válvula o bomba: siendo parte de un programa de riego, debe pasar a control manual y ya no seguir los pasos.

Acceso

- RFA1: el sistema permitirá al Usuario acceder a la plataforma web remota mediante el ingreso de nombre de usuario y contraseña. En cambio, el Usuario accederá directamente a la web del campo y la casilla, sin necesidad de autenticarse.
- RFA2: el sistema registrará para cada usuario los siguientes datos: nombre, apellido, email, teléfono y organización.

- RFA3: el sistema deberá permitir al administrador realizar todas las acciones de un usuario normal, pudiendo también realizar:
    - Altas, bajas y modificaciones de usuarios, campos y organizaciones.
    - Altas, bajas y modificaciones de bombas y válvulas.

Requerimientos No Funcionales Esperados

- RNF1: el sistema constará de una plataforma web dividida en tres instancias:
    - El sistema podrá ser accedido desde cualquier lugar del mundo y cualquier dispositivo.
    - El sistema podrá ser accedido desde el *hotspot* (red WiFi) propia de la placa Raspberry Pi, dentro de la casilla de bombas del campo. Esta versión de la plataforma se servirá desde un servidor web en la misma placa.
    - El sistema podrá ser accedido desde la red local del campo, mediante cualquier dispositivo. Se debe proveer la plataforma del punto anterior conectando la Raspberry Pi a una red entre la casilla de bombas y el campo, la cual queda fuera del alcance de este proyecto.
- RNF2: la UI de la plataforma web será adaptable a cualquier dispositivo móvil y de escritorio.
- RNF3: el sistema deberá ser implementado con las siguientes tecnologías:
    - Para interfaces web se usará Vue.js
    - Para base de datos se usará PostgreSQL.
    - Para servidores se usará Node.js.
- RNF4: el sistema deberá utilizar en cuanto a hardware:
    - Una Raspberry Pi como unidad central, provista por Ponce AgTech.
    - Un conjunto de placas Arduino que serán esclavos de la Raspberry Pi, provisto por Ponce AgTech.
    - El protocolo de comunicación entre la Raspberry Pi y las Arduino será sobre serial, utilizando uno existente o uno hecho a medida.
    - Las Arduino usarán *shields* hechos a medida y provistos por Ponce AgTech.
- RNF5: el sistema deberá ser dimensionado de manera tal que no se vea limitada la cantidad de válvulas a usar.
- RNF6: el monitoreo de la conectividad del sistema en campo se hará mediante la técnica de *Heartbeat*.
- RNF7: la comunicación entre los sistemas y la nube debe realizarse a través de un protocolo lo suficientemente liviano, en términos de bytes por mensaje, para que en un futuro, fuera del alcance de este proyecto, se pueda implementar comunicación satelital. De momento, se hará por TCP en capa de transporte.
- RNF8: el mapeo de válvulas y bombas a puertos deberá estar localizado en un archivo de configuración almacenado en la Raspberry Pi. Este podrá ser modificado utilizando el puerto serie de la placa o mediante SSH.
- RNF9: las válvulas de alivio no se mostrarán al usuario desde la interfaz, de manera de no poder apagarlas/prenderlas.
- RNF10: el sistema llevará un log de lo acontecido en el back-end, con las siguientes características:
    - Habrá un log de errores para fallas en el sistema.

    - Habrá un log de información sobre las acciones de los usuarios, que ayudará en la comprensión de los errores.
    - Habrá un log de debug, que se escribiría al mismo tiempo que el de información, pero con más detalles sobre las acciones de los usuarios.
    - El log de debug no funciona por defecto, sino que debería poder activarse sólo cuando se necesita mediante una variable específica para ello.
    - Los archivos de log deben ser comprimidos una vez finalizado el día.
- RNF11: el sistema debe ser construido de manera tal que soporte varios sistemas de riego en simultáneo, cada uno con su hardware propio.

Requerimientos Funcionales Deseados

Disponibilidad

- RFDD1: el sistema deberá, en caso de perder conectividad a Internet, almacenar y reenviar los mensajes de comunicación general entre el campo y la nube, una vez restaurada la conexión. Se deben mantener los mensajes hasta que puedan ser enviados o durante una campaña de riego, lo que suceda primero.

Monitoreo

- RFDM1: el sistema deberá detectar cuando comienza a llover. Si en ese momento el sistema estaba regando, y se supera un cierto valor mínimo de lluvia, se detendrá el riego.

Control

- RFDC1: el sistema deberá permitir al Usuario definir límites de presión por cada lote.
    - Presión máxima admitida: si el equipo sobrepasa esta presión deberá apagarse.
    - Presión mínima admitida: si el equipo se encuentra por debajo de esta presión deberá notificar al usuario.
    - Presión de alerta: Es un valor menor a la presión máxima admitida y dispara un mensaje de alerta al usuario para avisarle que la presión es demasiado alta.
    - El sistema deberá proceder a detener el equipo, apagando las bombas, y abriendo las válvulas de alivio, si detecta que el valor de presión sobrepasó el límite definido.
    - El sistema deberá permitir al Usuario deshabilitar este control.
- RFDC2: el sistema permitirá al Usuario seleccionar simultáneamente más de un programa, no solapados temporalmente. El sistema ejecutará cada uno de los programas seleccionados en el momento correspondiente.

Alarmas

- RFDA1: el sistema deberá notificar al Usuario cuando se detenga involuntariamente antes de finalizar el programa de riego seteado, debido a fallas en las bombas, en la alimentación eléctrica o cuando se alcance la presión máxima admitida (RFDC1).
- RFDA2: el sistema deberá notificar al Usuario cuando el sistema de riego sea iniciado o detenido localmente.

- RFDA3: el sistema deberá notificar al Usuario cuando la presión de un lote se encuentre por debajo de un límite inferior definido o cuando se alcance la presión de alerta (RFDC1).
- RFDA4: el sistema deberá notificar al Usuario cuando se inicie un programa de riego.
- RFDA5: el sistema deberá permitir al Usuario desactivar las alarmas según los tipos definidos en los requerimientos anteriores.

Reportes

- RFDR1: el sistema deberá llevar un registro de los siguientes datos, teniendo cada valor una marca de tiempo asociada: cuándo se abren y cierran las válvulas, las presiones reportadas, hora local de la Raspberry Pi, cuándo se activa un programa, cuánta agua fue regada y, en caso de lluvia, cuánta agua cayó.
- RFDR2: el sistema deberá permitir al usuario descargar los datos de RFDR1.
- RFDR3: el sistema deberá permitir al usuario borrar los datos almacenados, pero si no son borrados por él deben ser persistentes en el tiempo, sin tiempo máximo.

Acceso

- RFDAC1: el sistema deberá permitir al usuario añadir uno o más números de celular y elegir de esa lista a cuáles se notificará.
- RFDAC2: el sistema permitirá al Usuario acceder a la plataforma web del campo y casilla mediante el ingreso de nombre de usuario y contraseña.

Requerimientos No Funcionales Deseados

- RNFD1: las notificaciones serán enviadas por mensajes SMS o por WhatsApp.
- RNFD2: los datos para reportes se almacenarán en una base de datos de serie de tiempo.
- RNFD3: el sistema deberá ofrecer la descarga de RFDR2 en formato .csv o .xlsx.
- RNFD4: el sistema deberá permitir que mediante la interfaz web de la casilla se pueda reemplazar el archivo de configuración nombrado en RNF8.
- RNFD5: los componentes de la interfaz de usuarios serán desarrollados en forma de librería propia, utilizando *Storybook* para su presentación y *npm* para su administración.
- RNFD6: la gestión de usuarios será independiente para el front-end remoto y los de casilla/campo. Para el primero será administrado por un servicio en la nube, mientras que para el segundo se administrará localmente desde la Raspberry Pi. Aún así, existirá cierta sincronización que permitirá al administrador agregar y eliminar usuarios de casilla/campo remotamente.

Apéndice C - Detalles de Diseño

Envío de Estado de Campo

Al tratarse de la primera versión del sistema sólo fue necesario medir e informar la presión de las bombas y el *heartbeat* del subsistema de campo para detectar cambios en la conexión a la red. Inicialmente, se consideró unificar los eventos en uno sólo debido a que ambos resultaban ser mensajes periódicos. Es decir, al recibir notificaciones de presión se considerarían como *heartbeats*. A pesar de ofrecer cierto ahorro en el consumo de red, la mejora es despreciable y aumenta el acoplamiento entre los eventos. No se trata de las mismas magnitudes, y cada mensaje podría tener un período diferente.

Por las razones previamente mencionadas, se decidió emitir un mensaje específico para cada tipo de evento. Esta solución es mucho más extensible ante cambios, sobre todo si se agregaran nuevas notificaciones. En caso de incorporar los sensores adecuados, se podría informar la temperatura del campo o los milímetros de lluvia, entre otras magnitudes.

Por otro lado, como no se trabaja con reportes y datos históricos, sólo se almacena el último valor de presión de cada bomba y el último *heartbeat* de cada campo. De esta manera, si hace tiempo que no se reciben novedades, el usuario puede ver qué fue lo último que sucedió. Aun así, si en el futuro se decidiera almacenar datos históricos, los *hooks* de KeystoneJS permitirían agregar la lógica necesaria de manera simple.

De acuerdo con la teoría detrás de los *heartbeats*, el intervalo de desconexión debe ser mayor al intervalo de envío, que a su vez debe ser mayor a la latencia de la red. Por ejemplo, si este último valor es de 20 milisegundos (ms), los heartbeats pueden enviarse cada 100 ms, y el servidor revisa los valores de cada campo luego de 1000 ms. De esta forma, se deja suficiente margen para que múltiples mensajes puedan ser enviados y no haya falsos negativos. Si no se recibe nada en el intervalo mayor, entonces se considera al campo como desconectado.

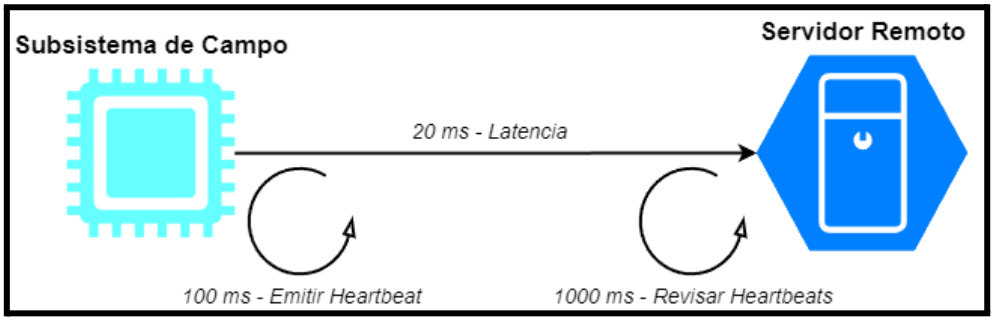

Figura 31 - Esquema básico de heartbeat.

Para la primera versión del sistema, se supone que todos los campos tienen la misma latencia. Sin embargo, la realidad es que este valor varía campo a campo, pues podrían estar distribuidos por todo el mundo. En el futuro, se podría modificar la lógica para que el intervalo de desconexión dependa de un valor de latencia que se calcule automáticamente a partir de los valores anteriores, o al menos se pueda cargar como dato del campo, o en función de la distancia.

KeystoneJS Relationships y Relaciones M:N

Por ser la primera vez que los integrantes del equipo trabajaban con un ORM, tenían curiosidad sobre cómo era el mapeo. En particular, si la herramienta respetaba las prácticas de normalización e integridad referencial vistas durante la cursada de Base de Datos. A modo de ejemplo, se tomaron las entidades User y Field, que en el DER se conectan con la relación Works At. Un campo puede ser gestionado por varios usuarios, y estos a su vez pueden trabajar en diferentes campos al mismo tiempo. Por ende, entre ambas entidades existe una relación con cardinalidad M:N.

Sin un ORM, sólo habría un back-end y una base de datos conectados de manera directa. Entonces sería necesario transformar dicha relación en una tabla intermedia que almacena una tupla por cada conexión entre dos de User y Field. Es decir, cada vez que un usuario comienza a trabajar en un campo, aparece una fila nueva que tiene dos columnas: una clave foránea que corresponde con User, y otra con Field. De esta manera, al querer consultar por los usuarios de un campo, o viceversa, habría que escribir consultas SQL complejas con uniones. Es la manera correcta de hacerlo para obtener un diseño normalizado, con integridad referencial, sin inconsistencias y relativamente eficiente.

Con las Relationships de KeystoneJS, la implementación descrita anteriormente sigue produciéndose, pero de manera simple y transparente al programador: sólo se declara una entidad como atributo de la otra, y se indica que se trata de una relación M:N a través de la propiedad *many*.

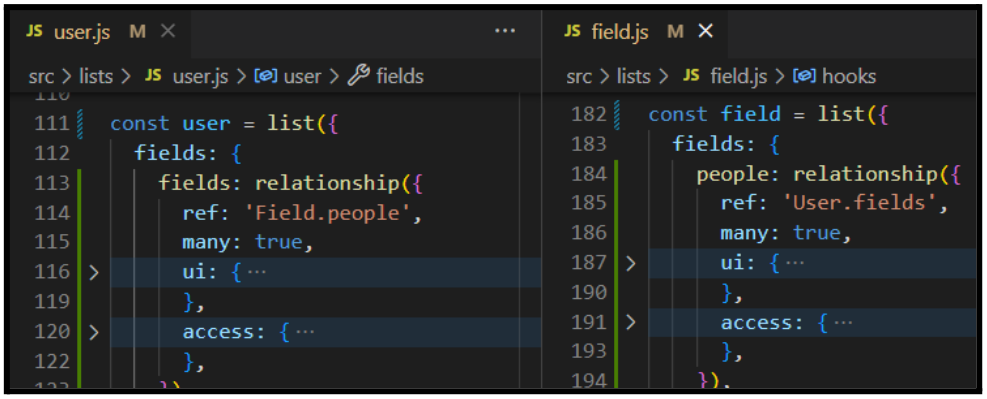

Figura 32 - Creación y conexión de ambas entidades utilizando la API de KeystoneJS.

Entonces, KeystoneJS, a través de Prisma, crea una tabla intermedia, pero la consulta es totalmente transparente gracias al ORM y a GraphQL. Aunque siguen generándose consultas, la sintaxis que debe utilizar el Servidor Remoto y sus clientes son muy simples y descriptivas.

```
-- Table: public._Field_people

-- DROP TABLE IF EXISTS public."_Field_people";

CREATE TABLE IF NOT EXISTS public."_Field_people"
(
    "A" uuid NOT NULL,
    "B" uuid NOT NULL,
    CONSTRAINT "_Field_people_A_fkey" FOREIGN KEY ("A")
        REFERENCES public."Field" (id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    CONSTRAINT "_Field_people_B_fkey" FOREIGN KEY ("B")
        REFERENCES public."User" (id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE
)
```

Figura 33 - Tabla intermedia creada automáticamente por KeystoneJS. Pueden observarse ambas claves foráneas.

En conclusión, al utilizar KeystoneJS se conservan las buenas prácticas a la hora de diseñar la base de datos. Pero lo hace de manera mucho más simple para el programador, tanto a la hora de construirla como al momento de realizar consultas.

```
query GET_USER_DATA($userId: ID) {
  user(where: { id: $userId }) {
    name
    id
    email
    organization {
      id
      name
    }
    fields {
      id
      name
    }
    role {
      name
    }
  }
}
```

Figura 34 - Consulta de GraphQL para obtener los datos de un usuario, incluido el listado de campos a los que tiene acceso.

Problemas encontrados al usar KeystoneJS

No permite campos de texto *case insensitive*

Existe un *issue* en el repositorio oficial [33], en donde el equipo de desarrollo admitió que es deuda técnica pero que no se puede resolver actualmente: PostgreSQL y SQLite trabajan con los strings de manera diferente, por lo que no hay una solución unificada todavía. Además existe una dependencia respecto a cómo gestiona Prisma los textos, lo que imposibilita avanzar con el desarrollo hasta que el ORM sea actualizado.

La solución alterna fue crear un hook propio: valida que un nuevo valor ya no esté siendo usado por otra entrada, sin distinguir mayúsculas de minúsculas. Fue diseñado para que funcione con cualquier campo de texto de cualquier lista.

No permite tener como atributo de una lista una relación con múltiples entidades

Por ejemplo, un paso de un programa puede activar una bomba, lote o válvula, y quizá en el futuro otros dispositivos. Hoy en día, las relaciones son sólo entre dos entidades, por lo que no es posible establecer algo así. Una opción es crear a mano una tabla Dispositivo, y otras llamadas Bomba, Lote y Válvula, que se extenderían de la primera. Entonces un paso se relacionaría solamente con entradas de la tabla Dispositivo. Sin embargo esto es difícil de implementar y rompe con la abstracción propuesta por KeystoneJS.

Por ende, se eligió tener dos atributos para los pasos: tipo de dispositivo e identificador de dispositivo. En el Admin UI los usuarios deben llenarlos a mano, lo cual es incómodo pero no es algo que usarán frecuentemente. En cambio en el Front-End Remoto, los clientes no notarán la dificultad, pues pueden seleccionar de manera simple el dispositivo y agregarlo al paso. Aun así, se debió implementar un hook que valide la existencia del dispositivo para asegurar la integridad referencial.

No permite actualizar entradas y filtrarlas al mismo tiempo por un atributo no único

Al encender un nuevo programa, debe detenerse cualquier otro que esté en ejecución. Aunque KeystoneJS crea una API que permite actualizar una o varias entradas a la vez, solo permite filtrar por campos que las identifiquen unívocamente, sin poder armar una consulta sobre cualquier campo como en SQL. Una solución sería primero ejecutar una búsqueda de todas aquellas entradas que cumplen la condición deseada. Obtenidos sus IDs, habría que ejecutar una actualización. Esto es poco eficiente porque requiere dos llamadas a la base de datos y a toda la lógica intermedia.

Sin embargo, KeystoneJS expone el ORM utilizado (Prisma) en su API, que sí ofrece un método que filtra por cualquier campo y actualiza las entradas resultantes al mismo tiempo [34]. Resulta llamativo que no ofrezca esa opción directamente [35], cuando el ORM usado en una capa inferior sí lo hace. Al tener que interactuar directamente con Prisma, se pierde parte de la abstracción que justamente define al framework.

No ofrece suscripciones mediante GraphQL (todavía)

La característica está contemplada en el *roadmap* [(36)], pero tiene por delante otras de más prioridad y no hay una fecha definida. En el sistema de riego, se necesitan suscripciones para poder notificar a los usuarios ante cambios de presión o de conexión con el campo. Para aun así poder trabajar con dicha característica, se decidió implementar SSE. Aunque es simple y del lado del usuario no se necesitan librerías extras, se obtiene una API de back-end menos homogénea, porque una parte está realizada con estos eventos y la mayoría mediante GraphQL.

Admin UI muestra las fechas en la zona horaria del usuario sin avisarle

KeystoneJS ofrece un tipo de dato de fecha que la almacena en UTC en la base de datos, pero el Admin UI la muestra en la zona horaria del usuario. No es completamente incorrecto, pero no lo indica explícitamente y no permite configurar una zona horaria default. En el sistema planteado, es especialmente confuso porque al ser remoto quizá el usuario está en una zona y el campo en otra. Entonces, la fecha de presión o heartbeat es en realidad la del campo, pero se muestra en la zona horaria del usuario. Hay un *issue* abierto [(37)], pero como solución alterna se decidió avisar al usuario que los campos de fechas los está viendo en su zona horaria. Otra opción hubiera sido crear/agregar un input propio hecho a medida que permita elegir la zona horaria o mostrarla por default en UTC. Esto requiere más trabajo y es más propenso a errores.

En modo de desarrollo no funciona correctamente la caché de Node.js

Este problema no aparece documentado en ningún lado, sino que se descubrió al implementar el listado de usuarios que se encuentran conectados como una única instancia global. En teoría, al importar por segunda vez un objeto de otro archivo, Node.js no vuelve a crearlo, sino que lo toma de su caché. Esto permite en cada importación compartir la misma instancia.

Sin embargo, por algún proceso interno de KeystoneJS, lo descrito no funciona en modo de desarrollo. En una prueba realizada en Windows 10 Pro y Ubuntu 20.04, se pudo observar que cada archivo que realiza la importación no está recibiendo el mismo estado que el resto necesariamente. En general, esto sucede en Node.js cuando los archivos son leídos por procesos diferentes, cada uno con su caché [(38)]. En la lista de procesos de los sistemas operativos se pudo observar que KeystoneJS aparece en dos entradas diferentes, pero con el mismo PID, lo cual resulta confuso.

Este problema no pudo ser resuelto debido a la falta de documentación y de pistas para orientarse. Al no suceder en modo de producción se decidió seguir adelante.

No permite hacer que los atributos basados en relaciones sean requeridos

Es otro caso donde existe un *issue* oficial [(39)], y en donde el equipo de desarrollo admite que es deuda técnica. Como solución alternativa también se sugieren hooks validadores. Esta falta es perjudicial para los usuarios no administradores: por ejemplo, podrían crear un lote o programa sin asignarle un campo, y perderían acceso a él.

No permite establecer ciertas reglas de negocio de manera simple y centralizada

Existen casos donde se necesita validar que se mantenga cierta consistencia al crear, eliminar o editar entidades relacionadas entre sí. Lograr el cumplimiento de estas reglas requiere de múltiples hooks repartidos sobre varias listas que contemplen demasiadas situaciones, por lo que son vulnerables a errores y se dificulta su mantenimiento.

Por ejemplo, en el sistema planteado debe cumplirse lo siguiente en todo momento: *Un **lote** es un conjunto de **válvulas** que pertenecen al mismo **campo**.* Para asegurar esto con KeystoneJS se deben agregar hooks para operaciones de creación, modificación y eliminación en cada lista involucrada, y no se pueden reutilizar, sino que habrá tres en cada una.

Este caso demandaba una alta complejidad y por los tiempos acotados que se disponían, se decidió no resolver el problema. Sin embargo, se analizaron las relaciones y restricciones entre ellas y se diseñó una solución, que se adjunta en la siguiente sección.

Reglas de Negocio: Campo, Lote y Válvulas

El Servidor Remoto debe asegurar que en todo momento se cumpla la siguiente regla de negocio:

*“Un **lote** es un conjunto de **válvulas** que pertenecen al mismo **campo**”*

Es decir, todas las válvulas están en el mismo campo, pero más importante, el lote debe pertenecer a ese campo y no a otro. Físicamente, esto es evidente, pero el sistema debe asegurar que la base de datos sea consistente. Por ende, son necesarias validaciones extra cuando se crean, editan o modifican las entidades. En KeystoneJS pueden implementarse en forma de hooks. A continuación se describe cada entidad con los hooks asociados a cada operación que puede realizarse.

Hooks para entidad Válvula

Una válvula tiene, entre otros atributos, un único campo y una lista de lotes a los que pertenece.

- **Creación**: se debe controlar que al asignarse un campo y uno o más lotes, estos pertenezcan a dicho campo. Si es así, se permite la operación y si no se rechaza. No se pueden asignar lotes sin un campo, pero sí al revés. Por ende, el campo es requerido.

- **Modificación**: si no se modifica el campo o los lotes no hay nada que revisar. Si se cambia el campo, no deben quedar lotes asignados del campo anterior, pero sí pueden aceptarse otros del nuevo campo. Por ende, se debe controlar que los lotes del arreglo resultante, pertenezcan al campo. Si se quita el campo, no debe quedar ningún lote asociado a la válvula. Si no se modifica el campo pero sí los lotes, debe controlarse que cada uno de ellos pertenezca al campo que ya tiene asignada la válvula.
- **Eliminación**: no hay que validar nada, ya que KeystoneJS automáticamente remueve la válvula del campo y de los lotes a los que estaba asociada.

Hooks para entidad Campo

Un campo tiene, entre otros atributos, una lista de válvulas y una de lotes.

- **Creación**: es necesario rechazar las creaciones en donde se agregue un lote o válvula con un campo previamente asignado, porque al ejecutarlas se estaría cambiando el campo de ellas y podrían quedar inconsistencias. Al rechazar la creación, se obliga al usuario a editar previamente el lote o válvula, lo que se controla con los otros hooks.
- **Modificación**: si se agrega una válvula o lote, debe seguirse la misma lógica mencionada para la creación de un campo. No se pueden quitar lotes o válvulas pues quedarían sin campo, y esto es requerido en esas entidades.
- **Eliminación**: al ser una entidad requerida en las válvulas y los lotes, una vez que se elimina un campo, todas las válvulas y lotes asociadas a él deberían desvincularse y eliminarse.

Hooks para entidad Lote

Un lote tiene, entre otros atributos, un único campo y un conjunto de válvulas.

- **Creación**: si no se asignan válvulas no se debe validar nada. Si no se asigna un campo, se debe rechazar la creación, pues es requerido. Si se asignan ambas entidades, se debe validar que cada válvula pertenezca al campo elegido.
- **Modificación**: si no se modifican campos ni válvulas no se debe validar nada. Si se modifica alguno de los dos, se debe validar que todas las válvulas del arreglo resultante (de incorporar las nuevas, conservar las previas y remover las indicadas) pertenezcan al campo actual o al nuevo, en caso de que también se modifique. No puede dejarse al lote sin campo pues es requerido, pero sí sin válvulas.
- **Eliminación**: no hay nada que validar ya que KeystoneJS desvincula las válvulas del lote y a este del campo.

Apéndice D - APIs

API del Servidor Remoto

Características generales

- Utiliza GraphQL y REST.
- Toda entidad de la base de datos cuenta con un identificador **uuid**.
- Existen 3 endpoints:
  - **/api/graphql**: utilizado por el Front-End Remoto y el Admin UI. Utiliza sesiones stateless, en donde el cliente envía en cada solicitud un token entregado al autenticarse.
  - **/api/mapper**: utilizado por el Mapper. Ofrece lo mismo que **/api/graphql** pero cambia la autenticación.
  - **/api/subscribe**: utilizado por el Front-End Remoto para suscribirse a SSE.
- El acceso a los recursos se encuentra regulado por reglas de control de acceso que distinguen entre usuarios administradores y normales. El mapper tiene permisos de administrador. Los usuarios no autenticados no podrán ejecutar ninguna solicitud, por lo que no pueden acceder a ningún recurso.
- La documentación específica de los parámetros y retornos de cada query y mutation pueden consultarse en el Playground del servidor en **/api/graphql**.

Errores comunes devueltos por /api/graphql y /api/mapper

En los bloques de código, *operationName* podría, por ejemplo, ser *createUser*. El objeto *error* tiene otros atributos pero los que están a continuación son los importantes a verificar:

Acceso denegado: al intentar ejecutar una query o mutation para la que no se cuenta con los permisos necesarios.

```
{
     data: {
         operationName: null
     },
     error: [
        {
            path: operationName,
            extensions: {
                code: 'KS_ACCESS_DENIED'
            }
        }
     ]
}
```

Fallo de validación: al ejecutar alguna mutation donde uno o más campos no cumplen con las validaciones requeridas.

```
{
     data: {
         operationName: null
     },
     error: [
        {
            path: operationName,
            extensions: {
                code: 'KS_VALIDATION_FAILURE'
            }
        }
     ]
}
```

Fallo de unicidad: al ejecutar una mutation en donde se le asigna a un campo un valor ya utilizado, que no puede repetirse.

```
{
     data: {
         operationName: null
     },
     error: [
        {
            path: operationName,
            extensions: {
                code: 'KS_PRISMA_ERROR'
            }
        }
     ]
}
```

Fallo de relación: al ejecutar una mutation e intentar relacionar a una entidad con otra a través de un id inexistente.

```
{
     data: {
         operationName: null
     },
     error: [
        {
            path: operationName,
            extensions: {
                code: 'KS_RELATIONSHIP_ERROR'
            }
        }
     ]
}
```

Fallo de extensión: ocurre debido a un error en un hook o en una función de control de acceso.

```
{
     data: {
         operationName: null
     },
     error: [
        {
            path: [ operationName ],
            extensions: {
                code: 'KS_EXTENSION_ERROR'
            }
        }
     ]
}
```

Fallo de input: el input de la operación es sintácticamente correcto, pero los valores son inválidos.

```
{
     data: {
         operationName: null
     },
     error: [
        {
            path: [ operationName ],
            extensions: {
                code: 'KS_USER_INPUT_ERROR'
            }
        }
     ]
}
```

Además de los errores anteriores, en el caso de **/api/mapper** existen otros posibles:

- Si el Mapper no está autenticado el código de status será 404 y el atributo *errors* de la respuesta contendrá directamente el siguiente string: *"Resource or action not found, or user not authenticated".*
- Si se produce algún error en el Servidor Remoto (fuera de los anteriores) el código de status será 505 y el atributo *errors* de la respuesta contendrá directamente el siguiente string: *"Internal server error while processing the request".*

Operación REST subyacente en requests
Ambos endpoints esperan requests de GraphQL enviadas mediante **POST**. El body será en formato JSON, y contendrá dos atributos: **query** y **variables**. **query** contendrá el string de la query/mutation a ejecutar contra el schema, y **variables** será un objeto con los parámetros esperados en la query/mutation.

Autenticación para /api/graphql
Los usuarios deberán iniciar sesión utilizando las mutations correspondientes para ello, y obtendrán como respuesta un **sessionToken** que enviarán con cada request subsecuente. También existe una mutation para cerrar sesión.

Entidades
A continuación, se describen las reglas de control de acceso de las entidades y las particularidades de ciertos campos que no son detectadas directamente por la documentación del GraphQL Playground.

Organization

Particularidades de campos:

- **name**: nombre de la organización. Es requerido para crear una, no se puede repetir y no se permiten strings vacíos ni con únicamente espacios en blanco. En estos casos se obtendrá un mensaje de error: "*name cannot have whitespaces only.*"
- **people**: listado de miembros de la organización, correspondientes a la entidad **User**. En caso de no tener miembros todavía se obtendrá un arreglo vacío. Si no se tienen permisos para ver este campo se devolverá **null**. Un **User** sólo puede pertenecer a una **Organization** a la vez.
- **fields**: listado de campos de la organización, correspondientes a la entidad **Field**. En caso de no tener campos todavía se obtendrá un arreglo vacío. Si no se tienen permisos para ver este campo se devolverá **null**. Un **Field** sólo puede pertenecer a una **Organization** a la vez.

Reglas de control de acceso:

- Solo un usuario administrador puede crear, actualizar o eliminar organizaciones. También puede consultar todas las organizaciones y ver los miembros de cada una.
- Un usuario no administrador puede solamente consultar los datos de la organización a la cual pertenece, excepto el campo **people**.

Role

Particularidades de campos:

- **name**: nombre del rol. Es requerido para crear uno, no se puede repetir y no se permiten strings vacíos ni con únicamente espacios en blanco. En estos casos se obtendrá un mensaje de error: "*name cannot have whitespaces only.*"
- **people**: listado de usuarios con este rol, correspondientes a la entidad **User**. En caso de no tener miembros todavía se obtendrá un arreglo vacío. Si no se tienen permisos para ver este campo se devolverá **null**. Un **User** sólo puede estar asignado a un **Role** a la vez.

Reglas de control de acceso:

- Solo un usuario administrador puede crear, actualizar o eliminar roles. También puede consultar todos los roles y ver los miembros de cada uno.
- Un usuario no administrador puede solamente consultar los datos del rol que tiene asignado, excepto el campo **people**.

Particularidades:

- El back-end crea automáticamente un **Role** con name *"ADMIN"* al crear el primer usuario del sistema mediante la mutation **createInitialUser**. Este rol debe ser asignado a todo usuario que sea administrador para poder contar con los privilegios correspondientes.

User

Particularidades de campos:

- **name**: nombre del usuario. Es requerido para crear uno, no se puede repetir y no se permiten strings vacíos ni con únicamente espacios en blanco. En estos casos se obtendrá un mensaje de error: "*name cannot have whitespaces only.*"
- **email**: email del usuario. Es requerido para crear uno, no se puede repetir y no se permiten strings vacíos ni con únicamente espacios en blanco. En estos casos se obtendrá un mensaje de error: "*email cannot have whitespaces only.*" Además, debe ser de formato *caracteres@caracteres.caracteres*, pudiendo contener letras, símbolos o números. Si no se respeta el formato se obtendrá el siguiente mensaje de error *"The email provided is not valid."*
- **password**: contraseña del usuario. Es requerida al crear un uno y al iniciar sesión. No se permiten strings vacíos ni únicamente espacios en blanco. Debe tener entre 8 y 20 caracteres, y al menos una mayúscula, minúscula y número. En los casos donde no se cumpla se obtendrá un mensaje de error: "*Password must be between 8-20 characters and include at least 1 uppercase, 1 lowercase and 1 number characters.*"
- **telephone**: teléfono del usuario. Es requerida al crear un uno. No se permiten strings vacíos ni únicamente espacios en blanco. Sólo se permiten números separados por espacios o guiones, pudiendo tener el símbolo "+" como prefijo. En los casos donde no se cumpla se obtendrá un mensaje de error: *"Telephone number can only include digits, spaces, '-' and can start with '+'."*
- **organization**: organización a la que pertenece el usuario. Ver entidad **Organization**.
- **role**: rol asignado al usuario. Ver entidad **Role**.
- **fields**: listado de campos a los que está asignado el Usuario, correspondientes a la entidad **Field**. En caso de no tener campos todavía se obtendrá un arreglo vacío. Si no se tienen permisos para ver este campo se devolverá **null**. Un usuario puede ser asignado a campos que no son de la organización a la cual pertenece.

Reglas de control de acceso:

- Solo un usuario administrador puede crear o eliminar usuarios.
- Un usuario no administrador puede solamente consultar sus datos. Un administrador puede consultar los datos de todos los usuarios.
- Un usuario no administrador sólo puede actualizar los campos email, password y telephone de su perfil. Un administrador puede actualizar todos los campos de todos los usuarios.
- El campo **password** no puede obtenerse de ninguna mutation o query, ni siquiera por el propio usuario o por un administrador.

Log In, Out y Up:

- Un usuario debe iniciar sesión utilizando **authenticateUserWithPassword**.
- Para cerrar sesión debe utilizar **endSession**.
- Los usuarios no pueden darse de alta en el sistema por su propia cuenta. Sólo un administrador puede crearles una cuenta utilizando **createUser**.
- El primer administrador del sistema puede crear su cuenta usando **createInitialUser**. Una vez ya creado el primer usuario dicha mutation deja de estar disponible.

Field

Particularidades de campos:

- **name**: nombre del campo. Es requerido para crear uno, no se puede repetir y no se permiten strings vacíos ni con únicamente espacios en blanco. En estos casos se obtendrá un mensaje de error: "*name cannot have whitespaces only.*" En caso de repetirse se obtendrá el siguiente mensaje de error: "*value already in use*".
- **latitude**: latitud de las coordenadas del campo. La unidad de medida es en grados decimales. Es requerido y debe ser un número entre -90.0 y 90.0.
- **longitude**: longitud de las coordenadas del campo. La unidad de medida es en grados decimales. Es requerido y debe ser un número entre -180.0 y 180.0.
- **fieldSubsystemURL**: URL del endpoint de la Raspberry Pi colocada en campo que recibirá las requests del Mapper. Es requerido, no se puede repetir y no se permiten strings vacíos ni con únicamente espacios en blanco. En estos casos se obtendrá un mensaje de error: "*name cannot have whitespaces only.*" En caso de repetirse se obtendrá el siguiente mensaje de error: "*value already in use*".
- **organization**: organización a la que pertenece el campo. Ver entidad **Organization**. Un **Field** sólo puede pertenecer a una organización a la vez.
- **people**: listado de usuarios asignados al campo, correspondientes a la entidad **User**. En caso de no tener miembros todavía se obtendrá un arreglo vacío. Si no se tienen permisos para ver este campo se devolverá **null**.
- **pumps**: listado de bombas asignadas al campo, correspondientes a la entidad **Pump**. En caso de no tener bombas todavía se obtendrá un arreglo vacío. Si no se tienen permisos para ver este campo se devolverá **null**.
- **valves**: listado de válvulas asignadas al campo, correspondientes a la entidad **Valve**. En caso de no tener válvulas todavía se obtendrá un arreglo vacío. Si no se tienen permisos para ver este campo se devolverá **null**.
- **lots**: listado de lotes del campo, correspondientes a la entidad **Lot**. En caso de no tener lotes todavía se obtendrá un arreglo vacío. Si no se tienen permisos para ver este campo se devolverá **null**.
- **programs**: listado de programas del campo, correspondientes a la entidad **Program**. En caso de no tener programas todavía se obtendrá un arreglo vacío. Si no se tienen permisos para ver este campo se devolverá **null**.
- **isConnected**: flag que indica si el campo está conectado con el Servidor Remoto o no. Depende de su heartbeat.
- **lastHeartbeatDate**: timestamp asociado a **isConnected**. Está en zona horaria UTC, y es un string en formato ISO. Indica la fecha del último heartbeat recibido desde el campo.

Reglas de control de acceso:

- Solo un usuario administrador puede crear, editar o eliminar campos.
- Un usuario no administrador puede solamente consultar los datos de los campos que tiene asignado, excepto el campo **people**.
- Un usuario puede ser asignado a campos que no son de la organización a la cual pertenece.

Particularidades:

- No se puede crear un campo agregando programas encendidos, ni puede eliminarse si contiene programas encendidos. Un campo no puede actualizarse para agregar/quitar programas si estos están activos.
- **lastHeartbeatDate** e **isConnected** no están diseñados para ser actualizados manualmente por ningún tipo de usuario, sino que se actualizan automáticamente para tener el último valor disponible.

Pump

Particularidades de campos:

- **theoricCaudal**: caudal teórico de la bomba. La unidad de medida es en m3/hora. Es requerido y debe ser un número mayor o igual a 0.0.
- **field**: campo al que pertenece la bomba. Ver entidad **Field**. Un **Pump** sólo puede pertenecer a un campo a la vez.
- **isOn**: flag que indica si la bomba está encendida o apagada. No se puede usar en una mutation de creación, sólo de actualización.
- **isOverriden**: flag que indica si la bomba está forzada, lo cual significa que pasa a control manual y no se siguen las instrucciones del programa hasta que se quite el forzado. No se puede usar en una mutation de creación, sólo de actualización.
- **lastPressureValue**: último valor de presión informado desde el campo. Es un número de coma flotante sin valores mínimo o máximos, para permitir valores imposibles que podrían indicar una falla en el sensor. No se puede usar en una mutation de creación, sólo de actualización.
- **lastPressureDate**: timestamp asociado al lastPressureValue. Está en zona horaria UTC, y es un string en formato ISO.

Reglas de control de acceso:

- Solo un usuario administrador puede crear o eliminar bombas.
- Un usuario no administrador puede solamente consultar los datos de bombas que pertenezcan a campos a los cuales esté asignado. Un administrador puede consultar cualquier bomba.
- Un usuario no administrador sólo puede actualizar **isOn** e **isOverriden**. Un administrador puede actualizar todos los campos.
- **lastPressureDate** y **lastPressureValue** no están diseñados para ser actualizados manualmente por ningún tipo de usuario, sino que se actualizan automáticamente desde el campo para tener el último valor disponible.

Valve

Particularidades de campos:

- **minAlertPressure**: presión mínima de alerta. La unidad de medida es en bares. Es requerido y debe ser un número mayor o igual a 0.0.
- **maxAlertPressure**: presión máxima de alerta. La unidad de medida es en bares. Es requerido y debe ser un número mayor o igual a 0.0.
- **minOffPressure**: presión mínima de apagado. La unidad de medida es en bares. Es requerido y debe ser un número mayor o igual a 0.0.
- **maxOffPressure**: presión máxima de apagado. La unidad de medida es en bares. Es requerido y debe ser un número mayor o igual a 0.0.

- **field**: campo al que pertenece la válvula. Ver entidad **Field**. Un **Valve** sólo puede pertenecer a un campo a la vez.
- **isOn**: flag que indica si está encendida o apagada.
- **isOverriden**: flag que indica si la válvula está forzada, lo cual significa que pasa a control manual y no se siguen las instrucciones del programa hasta que se quite el forzado. No se puede usar en una mutation de creación, sólo de actualización.
- **lot**: lote al que pertenece la válvula. Ver entidad **Lot**. Un **Valve** puede pertenecer a varios lotes a la vez, todos en el mismo campo que la válvula.

Reglas de control de acceso:

- Solo un usuario administrador puede crear o eliminar válvulas.
- Un usuario no administrador puede solamente consultar los datos de válvulas que pertenezcan a campos a los cuales esté asignado. Un administrador puede consultar cualquiera.
- Un usuario no administrador sólo puede actualizar **isOn**, **isOverriden** y asignar/desasignar lotes. Un administrador puede actualizar todos los campos.

Lot

Particularidades de campos:

- **name**: nombre del lote. Es requerido para crear uno, no se puede repetir y no se permiten strings vacíos ni con únicamente espacios en blanco. En estos casos se obtendrá un mensaje de error: "*name cannot have whitespaces only.*" En caso de repetirse se obtendrá el siguiente mensaje de error: "*value already in use*".
- **area**: área del lote en hectáreas. Es requerido y debe ser un número mayor o igual a 0.0.
- **field**: campo al que pertenece el lote. Ver entidad **Field**. Un **Lot** sólo puede pertenecer a un campo a la vez.
- **valves**: listado de válvulas incluídas en el lote, correspondientes a la entidad **Valve**. En caso de no tener válvulas todavía, se obtendrá un arreglo vacío.

Reglas de control de acceso:

- Un usuario administrador puede crear o eliminar lotes de cualquier campo. Un usuario no administrador puede crear o eliminar lotes que pertenezcan a campos a los cuales esté asignado y asignarle válvulas que pertenezcan al mismo.
- Un usuario no administrador puede solamente consultar los datos de lotes que pertenezcan a campos a los cuales esté asignado. Un administrador puede consultar cualquiera.
- Tanto un usuario no administrador como un administrador puede actualizar todos los atributos de los lotes. Un usuario no administrador solamente puede editar lotes que pertenezcan a campos a los cuales esté asignado.

Program

Particularidades de campos:

- **name**: nombre del programa. Es requerido para crear uno, se puede repetir pero no se permiten strings vacíos ni con únicamente espacios en blanco. En estos casos se obtendrá un mensaje de error: "*name cannot have whitespaces only.*".
- **isOn**: flag que indica si está encendido o apagado.
- **startTime**: string requerido que indica la hora de inicio del programa, en formato hh:mm (24 horas). Debe ser menor o igual a la hora de inicio del primer paso del programa.
- **field**: campo al que pertenece el programa. Ver entidad **Field**. Un **Program** sólo puede pertenecer a un campo a la vez.
- **steps**: listado de pasos del programa, correspondientes a la entidad **Step**. En caso de no tener pasos todavía, se obtendrá un arreglo vacío.

Reglas de control de acceso:

- Un usuario administrador puede crear o eliminar programas de cualquier campo. Un usuario no administrador puede crear o eliminar programas que pertenezcan a campos a los cuales esté asignado.
- Un usuario no administrador puede solamente consultar los datos de programas que pertenezcan a campos a los cuales esté asignado. Un administrador puede consultar cualquiera.
- Tanto un usuario no administrador como un administrador puede actualizar todos los atributos de los programas. Un usuario no administrador solamente puede editar programas que pertenezcan a campos a los cuales esté asignado.

Particularidades:

- Un programa sólo puede ser actualizado o eliminado mientras no esté encendido. Mientras esté encendido, sólo puede ser apagado.
- En un campo, sólo un programa puede estar activo, por lo que al encender uno, si hay otro activo se apagará.
- Al crear un programa, siempre estará apagado.
- Al eliminar un programa, todos sus pasos se eliminarán.
- Para encender un programa, debe tener al menos un paso. En caso contrario, se emitirá un error.

Step

Particularidades de campos:

- **startTime**: string requerido que indica la hora de inicio del step, en formato hh:mm (24 horas). Debe ser mayor o igual a la hora de inicio del programa al que pertenece.
- **program**: programa al que pertenece el paso. Ver entidad **Program**. Un **Step** sólo puede pertenecer a un programa a la vez.
- **duration**: duración del step en minutos. Es un número requerido mayor o igual a 1 minuto.
- **deviceType**: string que indica el tipo de dispositivo sobre el que actúa el paso. Es requerido y puede ser una de las siguientes opciones: *"lot"*, *"valve"* o *"pump"*.
- **deviceId**: string en formato uuid que corresponde al ID del dispositivo sobre el que actúa el paso. Debe corresponder a un dispositivo de tipo permitido en **deviceType** que ya esté registrado.

Reglas de control de acceso:

- Un usuario administrador puede crear, actualizar o eliminar pasos de cualquier programa. Un usuario no administrador puede crear, actualizar o eliminar pasos de programas que pertenezcan a campos a los cuales esté asignado.
- Un usuario no administrador puede solamente consultar los datos de pasos de programas que pertenezcan a campos a los cuales esté asignado. Un administrador puede consultar cualquiera.
- Un usuario no administrador sólo puede asociar a un paso un lote, válvula o bomba existente que pertenezca a un campo al cual esté asignado el usuario. Un usuario administrador puede hacerlo con cualquier campo.

Particularidades:

- Un paso puede ser agregado a un programa, quitado de un programa o actualizado sólo si el programa al que pertenece y/o pertenecerá están apagados.
- Cuando un programa es eliminado, todos sus pasos son eliminados.
- Al encender un programa, los pasos de tipo lote son convertidos a N pasos de tipo válvula, siendo N la cantidad de válvulas del lote.

API REST de suscripción a SSE

Para recibir actualizaciones acerca de las presiones de las bombas y de la conectividad con el campo existe una API REST que permite suscribirse a SSE. A continuación, se indican los detalles a tener en cuenta.

Solicitud de Suscripción

- URL: */api/subscribe?id=userId*
- Método: *GET*
- Query: id de usuario registrado en el sistema

Respuesta

En caso de éxito, el Servidor Remoto mantendrá la conexión abierta hasta que alguno de los dos la cierre. Se configurarán los siguientes headers en el cliente:

- *Content-Type: text/event-stream*
- *Cache-Control: no-cache*
- *Connection: keep-alive*

A partir de entonces, el cliente recibirá SSE en formato texto con la siguiente estructura: *event: tipo-evento\ndata: datos\n\n*. Tener en cuenta los saltos de líneas. Los datos dependen del tipo de evento, pero siempre son en formato JSON, por lo que deben ser parseados para recuperar un objeto:

Datos de evento: heartbeat:

```
{
      fieldId: string, id del campo al que corresponde el dato
      isConnected: booleano, indica si campo está conectado a Internet
      date: string, timestamp ISO UTC último heartbeat recibido
}
```

Tener en cuenta que la fecha puede ser nula si nunca se emitió un heartbeat.

Datos de evento: pressure:

```
{
      id: string, id de la bomba a la que corresponde el dato,
      pressure: número, valor de presión actual de la bomba
      pressureDate: string, timestamp ISO UTC de última presión recibida
}
```

Errores

El servidor puede responder con un mensaje de error en formato JSON al momento de suscribirse. Luego, con la suscripción en curso, no enviará errores.

Al momento de la suscripción, si no se envía un ID de usuario en la solicitud, o el ID no está registrado en la base de datos, se enviará el siguiente error con status 400:

```
{
      errors: "user with id id-enviado not registered"
}
```

Si se produce algún error interno en el servidor, se responderá con status 500 y el siguiente mensaje:

```
{
      errors: "Internal server error while processing the request"
}
```

API REST del Mapper

En este apartado se describe la API correspondiente a la comunicación entre el Mapper y el Servidor Remoto que actúa como cliente. No se describe la API gRPC entre el Mapper y el subsistema campo.

Solicitudes
Cuando se utilice POST, el cuerpo de la solicitud debe estar en formato JSON. En este caso, se debe setear el header *Content-Type* a *application/json*.

Respuestas
Todas las respuestas del Mapper se envían en formato JSON, con dos atributos: **error**, que es un string, y **result**, que es un booleano. El header *Content-Type* será *application/json*. El resultado dependerá de cada endpoint. El código de status de la respuesta indicará si esta fue exitosa o no:

- 200: fue exitosa y habrá un objeto como resultado.
- 400: solicitud incorrecta. El mensaje de error indicará que sucedió.
- 403 - 404: hubo un error de control de acceso o el recurso solicitado no existe.
- 500: error interno del Mapper.

En ningún caso de error habrá un objeto de resultado. Si la operación fue exitosa no habrá mensaje de error, y el resultado será *true*.

Endpoints

Encender/apagar bomba, programa o válvula

URL: /switch
Método: PATCH
Cuerpo de la solicitud:

```
{
      "type": "PUMP" | "VALVE" | "PROGRAM",
      "id": string uuid, id del programa, bomba o válvula,
      "url": string url del subsistema de campo,
      "newState": "ON" | "OFF",
      "programData": {
            "name": string,
            "startTime": string hh:mm,
            "steps": [
                  {
                        "startTime": string hh:mm,
                        "duration": número,
                        "deviceType": "PUMP" | "VALVE",
                        "deviceId": string uuid
                  },
                  {...}
            ]
      }
}
```

programData sólo debe enviarse si **type** equivale a *"PROGRAM"*.

Forzado de bomba o válvula

URL: /override
Método: PATCH
Cuerpo de la solicitud:

```
{
      type: "PUMP" | "VALVE",
      "id": string uuid, id de la bomba o válvula,
      "url": string url del subsistema de campo,
      "override": booleano, (true === override),
}
```

Apéndice E - Tests Unitarios

A continuación se listan los tests implementados para el Servidor Remoto. Se organizaron de la siguiente manera:

- Cada una de las tablas corresponde a un archivo *nombre.spec.js*.
- Cada cabecera azul corresponde a una sección dentro del archivo *nombre.spec.js*.
- Cada una de las filas corresponde a un test dentro de la sección correspondiente.
- En la columna de resultado esperado la celda es verde si debería ser un resultado del caso feliz, o rojo si debiera ser de error o rechazo.
- Algunos errores están descritos en mayor detalle en la API del Servidor Remoto, disponibles en el Apéndice.

Organization

CÓDIGO	**CONTEXTO**	*Usuario no autenticado*
O1	**TEST**	**RESULTADO ESPERADO**
O1T1	Crear una Organización	Acceso denegado para createOrganization
O1T2	Actualizar una Organización (todos los campos válidos)	Acceso denegado para updateOrganization
O1T3	Eliminar una Organización	Acceso denegado para deleteOrganization
O1T4	Consultar Organizaciones	{ data: { organizations: [ ] } }
CÓDIGO	**CONTEXTO**	*Usuario no admin*
O2	**TEST**	**RESULTADO ESPERADO**
O2T1	Crear una Organización	Acceso denegado para createOrganization
O2T2	Actualizar una Organización (todos los campos válidos)	Acceso denegado para updateOrganization
O2T3	Eliminar una Organización	Acceso denegado para deleteOrganization
O2T4	Consultar Organizaciones	{ data: { organizations: { ...datos org del usuario, people: null } }}
CÓDIGO	**CONTEXTO**	*Usuario admin*
O3	**TEST**	**RESULTADO ESPERADO**
O3T1	Crear una Organización con Nombre y Descripción no vacíos, agregar un par de usuarios, un field	{ data: { organization: { ...datos ingresados al crearla } }
O3T2	Crear una Organización con Nombre y Descripción no vacíos, sin usuarios ni fields	{ data: { organization: { ...datos ingresados al crearla, people: [ ] } }
O3T3	Crear una Organización con Nombre vacío, descripción no vacía, sin usuarios ni fields	Falla de Validación para createOrganization
O3T4	Crear una Organización con Nombre repetido, descripción no vacía, sin usuarios ni fields	Falla de Unicidad para createOrganization
O3T5	Crear una Organización con Nombre sólo espacios, descripción no vacía, sin usuarios ni fields	Falla de Validación para createOrganization

O3T6	Crear una Organización con Nombre válido, descripción vacía, sin usuarios ni fields	{ data: { organization: { name: 'algo', description: '', people: [ ] } }
O3T7	Actualizar una Organización con Nombre válido, descripción válida, agregar usuario y field	{ data: { organization: { name: 'nuevo', description: 'nuevo', people: [usuarios existentes, nuevo], fields: [fields existentes, nuevo], id: 'igual que antes' } }
O3T8	Actualizar una Organización con descripción vacía	{ data: { organization: { ...datos igual que antes, description: "" }
O3T9	Actualizar una Organización con Nombre vacío	Falla de Validación para updateOrganization
O3T10	Actualizar una Organización con Nombre sólo espacios	Falla de Validación para updateOrganization
O3T11	Actualizar una Organización con Nombre repetido	Falla de Unicidad para createOrganization
O3T12	Actualizar una Organización quitando un usuario y un field	{ data: { organization: { ...datos ingresados al crearla, people: [igual que antes, excepto el eliminado], fields: [igual que antes, excepto el eliminado] } }
O3T13	Eliminar una Organización	{ data: null } al realizar query con id de la organización eliminada
O3T14	Consultar Organizaciones	Todas las organizaciones, incluidos los miembros y fields de cada una

Figura 35 - Test implementados para la entidad Organization.

Role

CÓDIGO	**CONTEXTO**	*Usuario no autenticado*
R1	**TEST**	**RESULTADO ESPERADO**
R1T1	Crear un Rol	Acceso denegado
R1T2	Actualizar un Rol (todos los campos válidos)	Acceso denegado
R1T3	Eliminar un Rol	Acceso denegado
R1T4	Consultar Roles	Acceso denegado
CÓDIGO	**CONTEXTO**	*Usuario no admin*
R2	**TEST**	**RESULTADO ESPERADO**
R2T1	Crear un Rol	Acceso denegado
R2T2	Actualizar un Rol (todos los campos válidos)	Acceso denegado
R2T3	Eliminar un Rol	Acceso denegado
R2T4	Consultar Roles	Sólo su field, pero sin la lista del resto de usuarios con igual field
CÓDIGO	**CONTEXTO**	*Usuario admin*
R3	**TEST**	**RESULTADO ESPERADO**
R3T1	Crear un Rol con nombre válido, agregar algún usuario	Éxito
R3T2	Crear un Rol con nombre válido, sin agregar usuarios	Éxito
R3T3	Crear un Rol con nombre vacío	Error
R3T4	Crear un Rol con nombre sólo espacios	Error
R3T5	Crear un Rol con nombre repetido	Error
R3T6	Actualizar un Rol con nombre válido, agregar usuario	Éxito
R3T7	Actualizar un Rol quitando un usuario	Éxito

R3T8	Actualizar un Rol con nombre vacío	Error
R3T9	Actualizar un Rol con nombre sólo espacios	Error
R3T10	Actualizar un Rol con nombre repetido	Error
R3T11	Eliminar un Rol	Éxito
R3T12	Consultar Roles	Todas los fields, incluidos los usuarios de cada field

Figura 36 - Test implementados para la entidad Role.

User

CÓDIGO	**CONTEXTO**	*Usuario no autenticado*
U1	**TEST**	**RESULTADO ESPERADO**
U1T1	Crear un Usuario	Acceso denegado para createUser
U1T2	Actualizar un Usuario (todos los campos válidos)	Acceso denegado para updateUser
U1T3	Eliminar un Usuario	Acceso denegado para deleteUser
U1T4	Consultar Usuarios	{ data: { users: [ ] } }
CÓDIGO	**CONTEXTO**	*Usuario no admin*
U2	**TEST**	**RESULTADO ESPERADO**
U2T1	Crear un Usuario	Acceso denegado para createUser
U2T2	Actualizar un Usuario (todos los campos válidos)	Acceso denegado para updateUser
U2T3	Actualizar su Usuario con nuevo email, contraseña y teléfono	{ data: updateUser: { datos propios del usuario } }
U2T4	Actualizar su Usuario con nuevo nombre	Acceso denegado para updateUser
U2T5	Actualizar su Usuario con otro role	Acceso denegado para updateUser
U2T6	Actualizar su Usuario con otra organización	Acceso denegado para updateUser
U2T7	Actualizar su Usuario con otro field	Acceso denegado para updateUser
U2T8	Eliminar un Usuario	Acceso denegado para deleteUser
U2T9	Consultar Usuarios	{ data: users: [{ datos propios del usuario }] }
CÓDIGO	**CONTEXTO**	*Usuario admin*
U3	**TEST**	**RESULTADO ESPERADO**
U3T1	Crear un Usuario con Nombre, Email, Contraseña, Teléfono, Fields, Rol y Organización válidos	{ data: createUser: { datos del usuario creado, password: { isSet } } }
U3T2	Crear un Usuario con Nombre vacío, Email, Contraseña y Teléfono válidos	Falla de Validación para createUser

U3T3	Crear un Usuario con Nombre solo espacios, Email, Contraseña y Teléfono válidos	Falla de Validación para createUser
U3T4	Crear un Usuario con Email vacío, Nombre, Contraseña y Teléfono válidos	Falla de Validación para createUser
U3T5	Crear un Usuario con Email solo espacios, Nombre, Contraseña y Teléfono válidos	Falla de Validación para createUser
U3T6	Crear un Usuario con un Email ya registrado, Nombre, Contraseña y Teléfono válidos	Falla de Unicidad para createUser
U3T7	Crear un Usuario con Contraseña vacía, Nombre, Email y Teléfono válidos	Falla de Validación para createUser
U3T8	Crear un Usuario con Contraseña de menor longitud que lo permitido, Nombre, Email y Teléfono válidos	Falla de Validación para createUser
U3T9	Crear un Usuario con Contraseña de mayor longitud que lo permitido, Nombre, Email y Teléfono válidos	Falla de Validación para createUser
U3T10	Crear un Usuario con Contraseña sin mayúscula, Nombre, Email y Teléfono válidos	Falla de Validación para createUser
U3T11	Crear un Usuario con Contraseña sin minúscula, Nombre, Email y Teléfono válidos	Falla de Validación para createUser
U3T12	Crear un Usuario con Contraseña sin número , Nombre, Email y Teléfono válidos	Falla de Validación para createUser
U3T13	Crear un Usuario con Teléfono vacío, Nombre, Email y Contraseña válidos	Falla de Validación para createUser
U3T14	Crear un Usuario con Teléfono solo espacios, Nombre, Email y Contraseña válidos	Falla de Validación para createUser
U3T15	Crear un Usuario con Teléfono "-", Nombre, Email y Contraseña válidos	Falla de Validación para createUser
U3T16	Crear un Usuario con Teléfono "+", Nombre, Email y Contraseña válidos	Falla de Validación para createUser
U3T17	Crear un Usuario con Teléfono con caracteres no permitidos (letras por ejemplo), Nombre, Email y Contraseña válidos	Falla de Validación para createUser
U3T18	Actualizar un Usuario con nuevo Nombre, Email, Teléfono, Organización, Fields y Rol válidos	{ data: updateUser: { datos del usuario actualizados } }
U3T19	Eliminar un Usuario	{ data: null } al realizar query con id del usuario eliminado
U3T20	Consultar Usuarios	{ data: users: [datos de todos los usuarios incluido admin] }

Figura 37 - Tests implementados para la entidad User.

Field

CÓDIGO	**CONTEXTO**	*Usuario no autenticado*
F1	**TEST**	**RESULTADO ESPERADO**
F1T1	Crear un field	Acceso denegado para createField
F1T2	Actualizar un field (todos los campos válidos)	Acceso denegado para updateField
F1T3	Eliminar un field	Acceso denegado para deleteField
F1T4	Consultar fields	{ data: { fields: [ ] } }
CÓDIGO	**CONTEXTO**	*Usuario no admin*
F2	**TEST**	**RESULTADO ESPERADO**
F2T1	Crear un field	Acceso denegado para createField
F2T2	Actualizar un field (todos los campos válidos)	Acceso denegado para updateField
F2T3	Eliminar un field	Acceso denegado para deleteField
F2T4	Consultar fields (debe poder ver bombas, lotes, válvulas, heartbeat, si está conectado y programas)	{ data: { fields: [{ ...datos de fields del usuario, people: null }, ...] } }
CÓDIGO	**CONTEXTO**	*Usuario admin*
F3	**TEST**	**RESULTADO ESPERADO**
F3T1	Crear un field con nombre, url, latitud y longitud válidos, agregar algún usuario, bomba, lote, válvula, programa y org	{ data: { createField: ...todos los datos del field } }
F3T2	Crear un field con nombre, url, latitud y longitud válidos, sin agregar usuario, bomba, lote, válvula, programa u org	{ data: { createField: ...todos los datos del field } }
F3T3	Crear un field con nombre vacío	Falla de validación para createField
F3T4	Crear un field con nombre sólo espacios	Falla de validación para createField
F3T5	Crear un field con nombre repetido case insenitive	Falla de validación para createField

F3T6	Crear un field con url vacía	Falla de validación para createField
F3T7	Crear un field con url sólo espacios	Falla de validación para createField
F3T8	Crear un field con url repetida case insenitive	Falla de validación para createField
F3T9	Crear un field con latitud menor al mínimo	Falla de validación para createField
F3T10	Crear un field con latitud mayor al máximo	Falla de validación para createField
F3T11	Crear un field con latitud con caracteres no numéricos	{ errors: [GraphQLError] }
F3T12	Crear un field con longitud menor al mínimo	Falla de validación para createField
F3T13	Crear un field con longitud mayor al máximo	Falla de validación para createField
F3T14	Crear un field con longitud con caracteres no numéricos	{ errors: [GraphQLError] }
F3T15	Crear un field con un programa activo	Falla de validación para createField
F3T16	Eliminar un field con un programa activo	Falla de validación para createField
F3T17	Actualizar un field agregando un programa activo y quitando otro activo	Falla de validación para createField, msj error con id ambos programas
F3T18	Actualizar un field agregando un programa activo y quitando otro activo usando sintaxis `set`	Falla de validación para createField, msj error con id ambos programas
F3T19	Actualizar un field con nombre, latitud y longitud válidos, agregar usuario, lote, válvula, programa y bomba, cambiar org	{ data: { ...nuevos datos, people: [usuario anterior, nuevo usuario], pumps: [bomba anterior, nueva bomba], lots: [lote anterior, nuevo lote], valves: [válvula anterior, nueva válvula], programs: [programa anterior, nuevo programa] } }
F3T20	Actualizar un field quitando los usuarios, bombas, lotes, válvulas, programas y organización	{ data: { ...nuevos datos, people: [ ], pumps: [ ], lots: [ ], valves: [ ], organization: null, programs: [ ] } }
F3T21	Actualizar un field con nombre vacío	Falla de validación para updateField
F3T22	Actualizar un field con nombre sólo espacios	Falla de validación para updateField
F3T23	Actualizar un field con nombre repetido case insensitive	Falla de validación para updateField

F3T24	Actualizar un field con url vacía	Falla de validación para updateField
F3T25	Actualizar un field con url sólo espacios	Falla de validación para updateField
F3T26	Actualizar un field con url repetida case insensitive	Falla de validación para updateField
F3T27	Actualizar un field con latitud menor al mínimo	Falla de validación para updateField
F3T28	Actualizar un field con latitud mayor al máximo	Falla de validación para updateField
F3T29	Actualizar un field con latitud con caracteres no numéricos	{ errors: [GraphQLError] }
F3T30	Actualizar un field con longitud menor al mínimo	Falla de validación para updateField
F3T31	Actualizar un field con longitud mayor al máximo	Falla de validación para updateField
F3T32	Actualizar un field con longitud con caracteres no numéricos	Falla de validación para updateField
F3T33	Eliminar un field	{ data: null } al realizar query con id del field eliminado
F3T34	Consultar fields	Todos los fields, incluidos los usuarios de cada field

Figura 38 - Tests implementados para la entidad Field.

Pump

CÓDIGO	**CONTEXTO**	*Usuario no autenticado*
P1	**TEST**	**RESULTADO ESPERADO**
P1T1	Crear una bomba	Acceso denegado para createPump
P1T2	Actualizar una bomba (todos los campos válidos)	Acceso denegado para updatePump
P1T3	Eliminar una bomba	Acceso denegado para deletePump
P1T4	Consultar bombas	{ data: { pumps: [ ] } }
CÓDIGO	**CONTEXTO**	*Usuario no admin*
P2	**TEST**	**RESULTADO ESPERADO**
P2T1	Crear una bomba	Acceso denegado para createPump
P2T2	Actualizar una bomba de un field al cual no está asignado (todos los campos válidos)	Acceso denegado para updatePump
P2T3	Actualizar el caudal teórico de una bomba de un field al cual está asignado	Acceso denegado para updatePump
P2T4	Actualizar el field de una bomba de un field al cual está asignado	Acceso denegado para updatePump
P2T5	Eliminar una bomba	Acceso denegado para deletePump
P2T6	Consultar bombas	{ data: { pumps: [...datos de bombas de campos del usuario] } }
CÓDIGO	**CONTEXTO**	*Usuario admin*
P3	**TEST**	**RESULTADO ESPERADO**
P3T1	Crear una bomba con caudal teórico y field válidos	{ data: { createPump: ...datos bomba, isOn: false, isOverriden: false } }
P3T2	Crear una bomba con caudal teórico y sin field	{ data: { createPump: ...datos bomba, isOn: false, field: null, isOverriden: false } }
P3T3	Crear una bomba con caudal teórico menor al mínimo	Falla de validación para createPump
P3T4	Crear una bomba con caudal teórico con caracteres no numéricos	{ errors: [GraphQLError] }

P3T5	Actualizar una bomba con caudal teórico válido, cambiar field	{ data: { updatePump: { ...nuevos datos bomba } } }
P3T6	Actualizar una bomba quitando field	{ data: { updatePump: { ...nuevos datos bomba, field: null } } }
P3T7	Actualizar una bomba con caudal teórico menor al mínimo	Falla de validación para updatePump
P3T8	Actualizar un bomba con caudal teórico con caracteres no numéricos	{ errors: [GraphQLError] }
P3T9	Eliminar una bomba	{ data: null } al realizar query con id de bomba eliminada
P3T10	Consultar bombas	{ data: { pumps: [...datos de todas las bombas] } }
CÓDIGO	**CONTEXTO**	*Cambio de estado de la bomba incluyendo llamada al mapper con usuario admin*
P4	**TEST**	**RESULTADO ESPERADO**
P4T1	Actualizar el estado de una bomba simulando error de conexión con el mapper	Falla de extensión para updatePump
P4T2	Actualizar el estado de una bomba que tiene un field asignado	{ data: { updatePump: { ...datos bomba, nuevo estado } } } + llamada Axios
P4T3	Actualizar el estado de una bomba al mismo tiempo que se actualiza el field al que pertenece	{ data: { updatePump: { ...datos bomba, nuevo estado, nuevo field } } } + llamada Axios
P4T4	Actualizar el estado de una bomba al mismo tiempo que se le asigna un field (sin tener uno previo)	{ data: { updatePump: { ...datos bomba, nuevo estado, nuevo field } } } + llamada Axios
P4T5	Actualizar el estado de una bomba que no tiene un field asignado	Falla de input del usuario para updatePump y no hay llamada a Axios
P4T6	Actualizar el estado de una bomba al mismo tiempo que se elimina el field al que pertenece	Falla de extensión para updatePump y no hay llamada a Axios
CÓDIGO	**CONTEXTO**	*Cambio de override de la bomba incluyendo llamada al mapper con usuario admin*
P5	**TEST**	**RESULTADO ESPERADO**
P5T1	Actualizar override de una bomba que tiene un field asignado y forma parte de un programa	{ data: { updatePump: { ...datos bomba, isOverriden: true } } } + llamada Axios

P5T2	Actualizar override de una bomba que tiene un field asignado y no forma parte de un programa	Falla de extensión para updatePump y no hay llamada a Axios
P5T3	Actualizar override de una bomba que no tiene un field asignado y forma parte de un programa	Falla de extensión para updatePump y no hay llamada a Axios

Figura 39 - Tests implementados para la entidad Pump.

Program

CÓDIGO	**CONTEXTO**	*Usuario no autenticado*
PR1	**TEST**	**RESULTADO ESPERADO**
PR1T1	Crear un programa	Acceso denegado para createProgram
PR1T2	Actualizar un programa (todos los campos válidos)	Acceso denegado para updateProgram
PR1T3	Eliminar un programa	Acceso denegado para deleteProgram
PR1T4	Consultar programas	{ data: { programs: [ ] } }
CÓDIGO	**CONTEXTO**	*Usuario no admin, atributo isOn no se utiliza en ninguna mutation excepto cuando se aclara*
PR2	**TEST**	**RESULTADO ESPERADO**
PR2T1	Consultar programas	{ data: { programs: [...datos de programas de campos del usuario] } }
PR2T2	Crear un programa con todos los campos válidos	{ data: { createProgram: ...datos programa } }
PR2T3	Actualizar un programa de un field al cual no está asignado el usuario (campos válidos)	Acceso denegado para updateProgram
PR2T4	Actualizar un programa apagado de un field al cual está asignado el usuario (campos válidos)	{ data: { updateProgram: { ...nuevos datos } }
PR2T5	Actualizar un programa apagado de un field al cual está asignado el usuario cambiando el field	Acceso denegado para updateProgram
PR2T6	Eliminar un programa de un field al cual no está asignado el usuario	Acceso denegado para deleteProgram
PR2T7	Eliminar un programa apagado de un field al cual está asignado el usuario	{ data: null } al realizar query con id de programa eliminado
CÓDIGO	**CONTEXTO**	*Usuario admin, atributo isOn no se utiliza en ninguna mutation excepto cuando se aclara*
PR3	**TEST**	**RESULTADO ESPERADO**
PR3T1	Consultar programas	{ data: { programs: [...datos de todos los programas] } }

PR3T2	Crear un programa con todos los campos válidos	{ data: { createProgram: ...datos programa } }
PR3T3	Crear un programa con todos los campos válidos, isOn en true	Error porque no aparece en el GraphQL schema
PR3T4	Crear un programa con todos los campos válidos pero sin steps ni field	{ data: { createProgram: ...datos programa, steps: [ ], field: null } }
PR3T5	Crear un programa con nombre vacío	Falla de validación para createProgram
PR3T6	Crear un programa con nombre sólo espacios	Falla de validación para createProgram
PR3T7	Crear un programa con startTime no en formato hh:mm de 24 horas	Falla de validación para createProgram
PR3T8	Actualizar un programa apagado con campos válidos, cambiar field y steps	{ data: { updateProgram: { ...nuevos datos programa } } }
PR3T9	Actualizar un programa apagado quitando field y steps	{ data: { updateProgram: { ...datos programa, field: null, steps: [ ] } } }
PR3T10	Actualizar un programa apagado con nombre vacío	Falla de validación para updateProgram
PR3T11	Actualizar un programa apagado con nombre sólo espacios	Falla de validación para updateProgram
PR3T12	Actualizar un programa apagado con startTime no en formato hh:mm de 24 horas	Falla de validación para updateProgram
PR3T13	Actualizar un programa encendido con campos válidos: agregando step	Falla de validación para updateProgram
PR3T14	Eliminar un programa encendido	Falla de validación para deleteProgram
PR3T15	Eliminar un programa sin steps y que está apagado	{ data: null } al realizar query con id de programa eliminado
PR3T16	Eliminar un programa que posee step y que está apagado	{ data: null } al realizar query con id de programa eliminado { data: null } al realizar query con id de step eliminado
CÓDIGO	**CONTEXTO**	*Cambio de estado del programa incluyendo llamada al mapper con usuario admin*
PR4	**TEST**	**RESULTADO ESPERADO**
PR4T1	Actualizar el estado de un programa existiendo dos programas encendidos en ese momento. Uno perteneciente al mismo campo, que debería apagarse y otro en otro campo, que debería seguir funcionando.	{ data: { updateProgram: { ...datos programa, isOn: true } } } + llamada Axios + programa del mismo campo { isOn: false } + programa de otro campo { isOn: true }
PR4T2	Actualizar el estado de un programa (encenderlo) que no tenga steps	Falla de extensión para updateProgram y no hay llamada a Axios

PR4T3	Actualizar el estado de un programa (encenderlo) que tenga un step tipo lote con 2 válvulas	{ data: { updateProgram: { ...datos programa, isOn: true } } } + llamada Axios con 2 steps en programData, correspondiente a las 2 válvulas del lote.
PR4T4	Actualizar el estado de un programa (encenderlo) que tenga un step tipo lote sin válvulas	Falla de extensión para updateProgram y no hay llamada a Axios

Figura 40 - Tests implementados para entidad Program.

Step

CÓDIGO	**CONTEXTO**	*Usuario no autenticado*
S1	**TEST**	**RESULTADO ESPERADO**
S1T1	Crear un step	Acceso denegado para createStep
S1T2	Actualizar un step (todos los campos válidos)	Acceso denegado para updateStep
S1T3	Eliminar un step	Acceso denegado para deleteStep
S1T4	Consultar steps	{ data: { steps: [ ] } }
CÓDIGO	**CONTEXTO**	*Usuario no admin*
S2	**TEST**	**RESULTADO ESPERADO**
S2T1	Consultar steps	{ data: { steps: [...datos de steps de programas de campos del usuario] } }
S2T2	Crear un step con todos los campos válidos	{ data: { createStep: ...datos step } }
S2T3	Actualizar un step de un programa de un field al cual no está asignado el usuario (campos válidos)	Acceso denegado para updateStep
S2T4	Actualizar un step de un programa apagado de un field al cual está asignado el usuario (campos válidos)	{ data: { updateStep: { ...nuevos datos } }
S2T5	Eliminar un step de un programa de un field al cual no está asignado el usuario	Acceso denegado para deleteStep
S2T6	Eliminar un step de un programa apagado de un field al cual está asignado el usuario	{ data: null } al realizar query con id de step eliminado
S2T7	Crear un step con dispositivo perteneciente a un campo al cual no está asignado el usuario	Falla de validación para createStep
CÓDIGO	**CONTEXTO**	*Usuario admin*
S3	**TEST**	**RESULTADO ESPERADO**
S3T1	Consultar steps	{ data: { steps: [...datos de todos los steps] } }
S3T2	Crear un step con todos los campos válidos, para cada uno de los tipos de dispositivo	{ data: { createStep: ...datos step } }

S3T3	Crear un step con todos los campos válidos, pero asociándolo a un programa activo	Falla de validación para createStep
S3T4	Crear un step con hora de inicio no en formato hh:mm de 24 horas	Falla de validación para createStep
S3T5	Crear un step con duración menor al mínimo	Falla de validación para createStep
S3T6	Crear un step con ID de dispositivo no en formato uuid	Falla de validación para createStep
S3T7	Crear un step con ID de dispositivo no registrado	Falla de validación para createStep
S3T8	Crear un step con tipo de dispositivo inexistente	Falla de validación para createStep
S3T9	Actualizar un step de un programa apagado con campos válidos	{ data: { updateStep: { ...nuevos datos step } } }
S3T10	Actualizar un step de un programa apagado pero sin cambiar dispositivo (hook debería terminar ok)	{ data: { updateStep: { ...nuevos datos, deviceId: "sin cambio", deviceType: "sin cambio" } } }
S3T11	Actualizar un step de un programa apagado con campos inválidos	Falla de validación para updateStep
S3T12	Actualizar un step de un programa activo con campos válidos	Falla de validación para createStep
S3T13	Actualizar un step de un programa apagado cambiandolo por otro activo	Falla de validación para createStep
S3T14	Eliminar un step de un programa activo	Falla de validación para createStep
S3T15	Eliminar un step de un programa apagado	{ data: null } al realizar query con id de step eliminado

Figura 41 - Tests implementados para la entidad Step.

Valve

CÓDIGO	**CONTEXTO**	*Usuario no autenticado*
V1	**TEST**	**RESULTADO ESPERADO**
V1T1	Crear una válvula	Acceso denegado para createValve
V1T2	Actualizar una válvula (todos los campos válidos)	Acceso denegado para updateValve
V1T3	Eliminar una válvula	Acceso denegado para deleteValve
V1T4	Consultar válvulas	{ data: { valves: [ ] } }
CÓDIGO	**CONTEXTO**	*Usuario no admin*
V2	**TEST**	**RESULTADO ESPERADO**
V2T1	Crear una válvula	Acceso denegado para createValve
V2T2	Actualizar una válvula de un field al cual no está asignado (todos los campos válidos)	Acceso denegado para updateValve
V2T3	Actualizar la presión mínima de alerta de una válvula de un field al cual está asignado	Acceso denegado para updateValve
V2T4	Actualizar la presión máxima de alerta de una válvula de un field al cual está asignado	Acceso denegado para updateValve
V2T5	Actualizar la presión mínima de corte de una válvula de un field al cual está asignado	Acceso denegado para updateValve
V2T6	Actualizar la presión máxima de alerta de una válvula de un field al cual está asignado	Acceso denegado para updateValve
V2T7	Actualizar el field de una válvula de un field al cual está asignado	Acceso denegado para updateValve
V2T8	Actualizar el/los lote/s de una válvula de un field al cual está asignado	{ data: { updateValve: { ...nuevos datos válvulas } } }
V2T9	Eliminar una válvula	Acceso denegado para deleteValve
V2T10	Consultar válvulas	{ data: { valves: [...datos de válvulas de campos del usuario] } }
CÓDIGO	**CONTEXTO**	*Usuario admin*
V3	**TEST**	**RESULTADO ESPERADO**
V3T1	Crear una válvula con presiones, field y lote válidos	{ data: { createValve: ...datos válvula, isOn: false, isOverriden:

		false } }
V3T2	Crear una válvula con presiones y lote válidos y sin field	{ data: { createValve: ...datos válvula, isOn: false, isOverriden: false , field: null } }
V3T3	Crear una válvula con presiones y field válidos y sin lote	{ data: { createValve: ...datos válvula, isOn: false, isOverriden: false , lots: [ ] } }
V3T4	Crear una válvula con presión mínima de alerta menor al mínimo	Falla de validación para createValve
V3T5	Crear una válvula con presión máxima de alerta menor al mínimo	Falla de validación para createValve
V3T6	Crear una válvula con presión mínima de corte menor al mínimo	Falla de validación para createValve
V3T7	Crear una válvula con presión máxima de corte menor al mínimo	Falla de validación para createValve
V3T8	Crear una válvula con las presiones de alerta y corte con caracteres no numéricos	{ errors: [GraphQLError] }
V3T9	Actualizar una válvula con nuevas presiones, lote y cambiar field	{ data: { updateValve: { ...nuevos datos válvula } } }
V3T10	Actualizar una válvula quitando field	{ data: { updateValve: { ...nuevos datos válvula, field: null } } }
V3T11	Actualizar una válvula quitando lot	{ data: { updateValve: { ...nuevos datos válvula, lots: [ ] } } }
V3T12	Actualizar una válvula con presión mínima de alerta menor al mínimo	Falla de validación para updateValve
V3T13	Actualizar un válvula con presión mínima de alerta con caracteres no numéricos	{ errors: [GraphQLError] }
V3T14	Eliminar una válvula	{ data: null } al realizar query con id de válvula eliminada
V3T15	Consultar válvulas	{ data: { valves: [...datos de todas las válvulas] } }
CÓDIGO	**CONTEXTO**	*Cambio de estado de la válvula incluyendo llamada al mapper con usuario admin*
V4	**TEST**	**RESULTADO ESPERADO**
V4T1	Actualizar el estado de una válvula simulando error de conexión con el mapper	Falla de extensión para updateValve
V4T2	Actualizar el estado de una válvula que tiene un field asignado	{ data: { updateValve: { ...datos válvula, nuevo estado } } } + llamada Axios
V4T3	Actualizar el estado de una válvula al mismo tiempo que se actualiza el field al que pertenece	{ data: { updateValve: { ...datos válvula, nuevo estado, nuevo

		field } } } + llamada Axios
V4T4	Actualizar el estado de una válvula al mismo tiempo que se le asigna un field (sin tener uno previo)	{ data: { updateValve: { ...datos válvula, nuevo estado, nuevo field } } } + llamada Axios
V4T5	Actualizar el estado de una válvula que no tiene un field asignado	Falla de extensión para updateValve y no hay llamada a Axios
V4T6	Actualizar el estado de una válvula al mismo tiempo que se elimina el field al que pertenece	Falla de extensión para updateValve y no hay llamada a Axios
V5	**TEST**	**RESULTADO ESPERADO**
V5T1	Actualizar override de una válvula que tiene un field asignado y forma parte de un programa	{ data: { updateValve: { ...datos válvula, isOverriden: true } } } + llamada Axios
V5T2	Actualizar override de una válvula que tiene un field asignado y no forma parte de un programa	Falla de extensión para updateValve y no hay llamada a Axios
V5T3	Actualizar override de una válvula que no tiene un field asignado y forma parte de un programa	Falla de extensión para updateValve y no hay llamada a Axios

Figura 42 - Tests implementados para la entidad Valve.

Lot

CÓDIGO	**CONTEXTO**	*Usuario no autenticado*
LT1	**TEST**	**RESULTADO ESPERADO**
LT1T1	Crear un lote	Acceso denegado para createLot
LT1T2	Actualizar un lote (todos los campos válidos)	Acceso denegado para updateLot
LT1T3	Eliminar un lote	Acceso denegado para deleteLot
LT1T4	Consultar lotes	{ data: { lots: [ ] } }
CÓDIGO	**CONTEXTO**	*Usuario no admin*
LT2	**TEST**	**RESULTADO ESPERADO**
LT2T1	Consultar lotes	{ data: { lots: [...datos de lotes de campos del usuario] } }
LT2T2	Crear un lote con todos los campos válidos	{ data: { createLot: ...datos lote } }
LT2T3	Actualizar un lote de un field al cual no está asignado (todos los campos válidos)	Acceso denegado para updateLot
LT2T4	Actualizar un lote de un field al cual está asignado (todos los campos válidos)	{ data: { updateLot: { nuevos datos } }
LT2T5	Eliminar un lote de un field al cual está asignado	{ data: null } al realizar query con id de lote eliminado
LT2T6	Eliminar un lote de un field al cual no está asignado	Acceso denegado para deleteLot
CÓDIGO	**CONTEXTO**	*Usuario admin*
LT3	**TEST**	**RESULTADO ESPERADO**
LT3T1	Crear un lote con todos los campos válidos	{ data: { createLot: ...datos lote } }
LT3T2	Crear un lote con campos válidos pero sin válvulas ni field	{ data: { createLot: ...datos lote, valves: [ ], field: null } }
LT3T3	Crear un lote con nombre vacío	Falla de validación para createLot
LT3T4	Crear un lote con nombre sólo espacios	Falla de validación para createLot
LT3T5	Crear un lote con nombre repetido case insenitive	Falla de validación para createLot

LT3T6	Crear un lote con área menor al mínimo	Falla de validación para createLot
LT3T7	Crear un lote con área con caracteres no numéricos	{ errors: [GraphQLError] }
LT3T8	Actualizar un lote con campos válidos, cambiar field y válvulas	{ data: { updateLot: { ...nuevos datos lote } } }
LT3T9	Actualizar un lote quitando field y válvulas	{ data: { updateLot: { ...nuevos datos lote, field: null, valves: [ ] } } }
LT3T10	Actualizar un lote con nombre vacío	Falla de validación para updateLot
LT3T11	Actualizar un lote con nombre sólo espacios	Falla de validación para updateLot
LT3T12	Actualizar un lote con nombre repetido case insenitive	Falla de validación para updateLot
LT3T13	Actualizar un lote con área menor al mínimo	Falla de validación para updateLot
LT3T14	Actualizar un lote con área con caracteres no numéricos	{ errors: [GraphQLError] }
LT3T15	Eliminar un lote	{ data: null } al realizar query con id de lote eliminado
LT3T16	Consultar lotes	{ data: { lots: [...datos de todos los lotes] } }

Figura 43 - Tests implementados para entidad Lot.

Inicio de Sesión y Creación del Primer Usuario

CÓDIGO	**CONTEXTO**	*Inicio de sesión*
L1	**TEST**	**RESULTADO ESPERADO**

L1T1	Log-In de usuario con email registrado y contraseña correcta	{
		"data": {
		"authenticateUserWithPassword": {
		"__typename":"UserAuthenticationWithPasswordSuccess",
		"item": {
		"id": "id del usuario"
		}
		}
		}
		}

L1T2	Log-In de usuario con email registrado y contraseña incorrecta	{
		"data": {
		"authenticateUserWithPassword": {
		"__typename":
		"UserAuthenticationWithPasswordFailure",
		"message": "Authentication failed."
		}
		}
		}

L1T3	Log-In de usuario con email no registrado	{
		"data": {
		"authenticateUserWithPassword": {
		"__typename":
		"UserAuthenticationWithPasswordFailure",
		"message": "Authentication failed."
		}
		}

		}

L1T4	Log-In sin enviar campos	{
		"errors": [
		{ extensions: { code: 'BAD_USER_INPUT' } },
		{ extensions: { code: 'BAD_USER_INPUT' } }
		]
		}

CÓDIGO	**CONTEXTO**	*Creación de 1er Usuario*
L2	**TEST**	**RESULTADO ESPERADO**
L2T1	Creación 1er Usuario con campos válidos	Ejemplo exitoso createInitialUser
L2T2	Creación 1er Usuario con campos válidos, pero ya existiendo un usuario	Falla de validación para createInitialUser

Figura 44 - Tests implementados para operaciones de inicio de sesión y creación del primer usuario.

/api/mapper

CÓDIGO	**CONTEXTO**	*Mapper no autenticado*
M1	**TEST**	**RESULTADO ESPERADO**
M1T1	Realizar mutation con campos válidos, por ejemplo createUser	{ errors: "Resource or action not found, or user not authenticated" }
M1T2	Realizar query, por ejemplo: { query { users { ...campos de User } } }	{ errors: "Resource or action not found, or user not authenticated" }
CÓDIGO	**CONTEXTO**	*Mapper autenticado*
M2	**TEST**	**RESULTADO ESPERADO**
M2T1	Realizar mutation con campos válidos, por ejemplo createUser	{ data: { createUser: { ...datos del usuario } } }
M2T2	Realizar query, por ejemplo: { query { users { ...campos de User } } }	{ data: { users: [{ ...datos del usuario }] } }
CÓDIGO	**CONTEXTO**	*Mapper autenticado envía actualizaciones de estado de dispositivos*
M3	**TEST**	**RESULTADO ESPERADO**
M3T1	Actualizar el estado de una bomba enviando la solicitud desde el mapper	{ data: { updatePump: { ...datos bomba, nuevo estado } } } pero sin llamar mapper

Figura 45 - Tests implementados para la API que consume el mapper.

Bibliografía

(0) *Kiwi: una producción que crece a paso firme*. Revista Internos. Recuperado el 12/11/2021 de https://www.revistainternos.com.ar/2021/04/kiwi-una-produccion-que-crece-a-paso-firme/.

(1) (2) Liotta A. Mario, *Los sistemas de riego por goteo y microaspersión*. Recuperado el 13/11/2021 de https://inta.gob.ar/sites/default/files/script-tmp-articulo_riego_presurizado.pdf.

(3) *Supabase*. Supabase. Recuperado el 20/12/2022 de https://supabase.com/.

(4) *Strapi*. Strapi. Recuperado el 20/12/2022 de https://strapi.io/.

(5) *WebSockets vs. Server-Sent events/EventSource*. Stack Overflow. Recuperado el 26/05/2022 de https://stackoverflow.com/questions/5195452/websockets-vs-server-sent-events-eventsource.

(6) *WebSockets vs Server-Sent Events*. Telerik. Recuperado el 26/05/2022 de https://www.telerik.com/blogs/websockets-vs-server-sent-events.

(7) *Authentication and Access Control*. KeystoneJS. Recuperado el 22/12/2022 de https://keystonejs.com/docs/guides/auth-and-access-control.

(8) *Access Control API*. KeystoneJS. Recuperado el 22/12/2022 de https://keystonejs.com/docs/apis/access-control.

(9) *Stateful vs Stateless Architecture: Why Stateless Won*. Virtasant. Recuperado el 22/12/2022 de https://www.virtasant.com/blog/stateful-vs-stateless-architecture-why-stateless-won.

(10) *Maxicom v4.5*. Rainbird. Recuperado el 17/08/2022 de https://www.rainbird.com/es/products/maxicom-v45.

(11) *Water Management Software*. Hunter Industries. Recuperado el 17/08/2022 de https://www.hunterindustries.com/en-metric/product-line/water-management-software.

(12) *Understanding APIs*. Amazon AWS. Recuperado el 06/09/2022 de https://aws.amazon.com/what-is/api/.

(13) *El backlog del producto: la lista de tareas pendientes definitiva*. Atlassian. Recuperado el 06/09/2022 de https://www.atlassian.com/es/agile/scrum/backlogs.

(14) *BPMN Specification*. BPMN. Recuperado el 06/09/2022 de https://www.bpmn.org/.

(15) *What Is a CMS and Why Should You Care?* HubSpot. Recuperado el 06/09/2022 de https://blog.hubspot.com/blog/tabid/6307/bid/7969/what-is-a-cms-and-why-should-you-care.aspx.

(16) *Git*. Git. Recuperado el 06/09/2022 de https://git-scm.com/.

(17) *GraphQL: A query language for your API*. GraphQL. Recuperado el 06/09/2022 de https://graphql.org/.

(18) Fowler Martin, *Heartbeat*. Recuperado el 06/09/2022 de https://martinfowler.com/articles/patterns-of-distributed-systems/heartbeat.html.

(19) *What is HTTP?* Cloudflare. Recuperado el 06/09/2022 de https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/.

(20) *What is HTTPS?* Cloudflare. Recuperado el 06/09/2022 de https://www.cloudflare.com/learning/ssl/what-is-https/.

(21) *What is IoT?* Amazon AWS. Recuperado el 06/09/2022 de https://aws.amazon.com/what-is/iot/.

(22) *What is an IP address?* Avast. Recuperado el 06/09/2022 de https://www.avast.com/c-what-is-an-ip-address.

(23) *Jira Software*. Atlassian. Recuperado el 06/09/2022 de https://www.atlassian.com/es/software/jira.

(24) *KeystoneJS*. KeystoneJS. Recuperado el 20/12/2022 de https://keystonejs.com/.

(25) *Lint*. Wikipedia. Recuperado el 06/09/2022 de https://en.wikipedia.org/wiki/Lint_(software).

(26) *What is an ORM, how does it work and how should I use one?* Stack Overflow. Recuperado el 07/09/2022 de https://stackoverflow.com/questions/1279613/what-is-an-orm-how-does-it-work-and-how-should-i-use-one.

(27) *AirBnB JavaScript Style Guide*. AirBnB. Recuperado el 06/09/2022 de https://airbnb.io/javascript/react/.

(28) *What is SQL?* SQL Course. Recuperado el 07/09/2022 de https://www.sqlcourse.com.

(29) *What is Server-Sent Events (SSE) and how to implement it?* Medium. Recuperado el 07/09/2022 de https://medium.com/yemeksepeti-teknoloji/what-is-server-sent-events-sse-and-how-to-implement-it-904938bffd73.

(30) *Downside of using Server-Sent events for bidirectional client-server communication*. Stack Overflow. Recuperado el 07/09/2022 de https://stackoverflow.com/questions/13278365/downside-of-using-server-sent-events-for-bidirectional-client-server-communicati.

(31) *Using server-sent events*. Mozilla. Recuperado el 07/09/2022 de https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events.

(32) *Guide to Industrial Control Systems (ICS) Security*. NIST. Recuperado el 07/09/2022 de https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82r2.pdf.

(33) *identityField is Case Sensitive for Authentication*. GitHub. Recuperado el 15/01/2022 https://github.com/keystonejs/keystone/issues/6844.

(34) *Prisma Client API reference*. Prisma. Recuperado el 08/04/2022 de https://www.prisma.io/docs/reference/api-reference/prisma-client-reference#updatemany.

(35) *Query API*. KeystoneJS. Recuperado el 08/04/2022 de https://keystonejs.com/docs/apis/query#update-many.

(36) *Roadmap*. KeystoneJS. Recuperado el 01/06/2022 de https://keystonejs.com/updates/roadmap.

(37) *Customizable timezone for the timestamp field type*. GitHub. Recuperado el 05/05/2022 de https://github.com/keystonejs/keystone/discussions/7327.

(38) *JS share class between files*. Stack Overflow. Recuperado el 15/06/2022 de https://stackoverflow.com/questions/47365388/js-share-class-instance-between-files.

(39) *IsRequired support for Relationship field*. GitHub. Recuperado el 02/03/2022 de https://github.com/keystonejs/keystone/discussions/7310.