72.45 - Proyecto Final

Instituto Tecnológico de Buenos Aires - ITBA

escuela de ingeniería y gestión

Trabajo final presentado para la obtención del título de

Ingeniera en informática, Julio 2021

Ponte API-v2

Bianca Ritorto - 57082

Clara Guzzetti - 57100

Tutora: Marcela Guerrero

Fecha: –/07/2021

Índice

1. Introducción **3**

2. Estado del arte **5**

2.1. Atributos de calidad 5

2.1.1. Reliability 6

2.1.2. Maintainability 6

2.2. Arquitectura 6

2.3. Diseño 9

2.4. Seguridad 10

3. Extensión y alcance del proyecto **11**

3.1. Limitaciones 12

3.1.1. Amazon Web Services 12

3.1.2. Heroku 14

4. Producto desarrollado **15**

4.1. Atributos de calidad 15

4.2. Arquitectura de los servicios 16

4.3. Estructura del código 19

4.4. Lógica de negocio 20

4.4.1. Baja lógica 21

4.4.2. Creación de donaciones 21

4.4.3. Máquina de estados de donaciones 23

4.5. Seguridad 25

4.5.1. Spring Security 25

4.5.2. JSON Web Token 25

4.6. Mailing 26

4.7. Documentación de los endpoints 26

4.8. Integración con front-end 28

4.8.1. Borrado de código innecesario 28

4.8.2. Voluntarios 28

4.8.3. Envío de mails 28

4.8.4. Instituciones 28
4.8.5. Mejoras generales al código del front-end 29
4.9. Pruebas 29
4.9.1. Pruebas unitarias 29
4.9.2. Pruebas en ambientes de desarrollo 29

5. Proceso **30**
5.1. Organización de las tareas 30

6. Conclusiones y trabajo futuro **32**
6.1. Conclusiones y aprendizajes 32
6.2. Trabajo futuro 33

Referencias **34**

1. Introducción

"Ponte es una plataforma que permite conectar instituciones que realizan pedidos, usuarios que desean donar y voluntarios que se ofrecen para transportar la donación del usuario a la Institución."

Ponte es un proyecto creado originalmente por Juan Sebastián Vera y Noelia Belén López como su trabajo final en 2019. José Carlos Noriega Defferrari desarrolló aplicación Android, y Constanza De Rienzo junto con Alejandro Santoflaminio la versión en iOS.

El objetivo del proyecto Ponte API-v2 es el de proveer una arquitectura más robusta a la plataforma, reestructurando el back-end junto con los servicios externos utilizados. Si bien la aplicación web desarrollada en 2019 es funcional, los autores originales se concentraron en el desarrollo del front-end y la usabilidad de la aplicación. El back-end implementado presenta muchas aristas de mejora que son las que se explotan y atacan en la segunda versión de la API REST, explorada en el presente informe.

Esta nueva iteración del back-end de la aplicación se considera crucial para el futuro lanzamiento a producción de la plataforma. Sin la implementación de seguridad sobre los endpoints que hoy en día utilizan tanto la aplicación mobile de iOS como la de Android y también la versión web, sería muy riesgoso presentar el proyecto al público general. Por otro lado, Ponte API v-2 está diseñada para poder escalar fácilmente en términos de código y de interdependencia de servicios externos. Se reducen estos servicios, facilitando el manejo de la aplicación y la inclusión de nuevos contribuyentes al proyecto.

Así, los objetivos para el proyecto final Ponte API-v-2 quedan reducidos a:

- Construir un back-end en forma de API REST adecuado a la problemática y amoldado a la solución original planteada.
- Segurizar los endpoints de dicha API para impedir la exposición de la base de datos.
- Reducir al mínimo indispensable la dependencia con servicios externos, para mejor manejo de la aplicación en su conjunto. Esto implica integrar algunos servicios al back-end, e implementarlos de cero.

- Dejar una versión de la aplicación web junto con el nuevo back-end funcionando y desplegado en algún servidor de conveniencia.

2. Estado del arte

A partir del proyecto final de Sebastián y Noelia en 2019 [1] surgen varios repositorios con código que debieron ser analizados para la presente iteración de Ponte:

- **ponte-mailer**
- **ponte-api**
- **ponte-web**

Por otro lado, cabe destacar las versiones móviles del proyecto que se debieron tener en cuenta en el transcurso del desarrollo, para poder brindar una solución que aportara tanto a la web como a ambas plataformas móviles:

- **Ponte Android** [2]
- **Ponte iOS** [3]

El relevamiento detallado de la versión original del back-end de Ponte (**ponte-api**) detectó los siguientes puntos de mejora, separados en distintas áreas del producto.

2.1. Atributos de calidad

Los atributos planteados para la primera versión de Ponte eran los siguientes. Cabe aclarar que serán presentados en el idioma inglés ya que se considera que en una escala global, estos atributos son entendidos en su totalidad bajo los significados de las palabras en inglés, y su traducción al español a veces resulta errónea y poco precisa.

1. Reliability
2. Availability
3. Security
4. Maintainability
5. Scalability

A continuación se detalla el análisis sobre los atributos que se consideraron conflictivos con el proyecto.

2.1.1. Reliability

Por su definición, la aplicación debe mantenerse funcionando a lo largo del tiempo y las diferentes interacciones dentro de la misma no deben poner en juego que el sistema se mantenga operacional. En caso de que el sistema tenga una falla, debe ser capaz de reconocerla y compensarla permaneciendo operacional.

No se cree que este sea un atributo de calidad principal para Ponte, pues no se producen transacciones o procesos con alta probabilidad de fallo, como sería por ejemplo el caso de manejo de divisas, donde, de no contemplar la precisión y el posible acarreo de errores, se correría el riesgo de que el sistema no se mantenga operacional en el tiempo.

2.1.2. Maintainability

Considerando que este atributo hace referencia a la facilidad con la que un producto es mantenido, no se considera que sea un atributo clave en este caso ya que la aplicación no está pensada específicamente para que se expanda de la idea principal. Es decir, la misma tiene un alcance reducido de eventos que puede llegar a soportar. No es probable que en el futuro se quiera incrementar significativamente el tamaño del código.

2.2. Arquitectura

La arquitectura original de Ponte consistía en los componentes que se pueden observar en la Figura 1. A partir de la misma se detectaron los siguientes puntos de mejora:

- Se utilizaban dos servicios de *hosting* diferentes para el front-end y para el back-end, AWS y Heroku respectivamente. Esto implicaba la mantención de dos cuentas distintas para realizar una misma tarea.
- El front-end era responsable de gran parte de la lógica de negocios y el back-end se reducía a una interfaz para modificar la base de datos.

- La responsabilidad del servicio de email recaía en el front-end.
- Se utilizaba un servicio externo (Firebase) que requería mantenimiento y configuración manual únicamente para el manejo de autenticación.

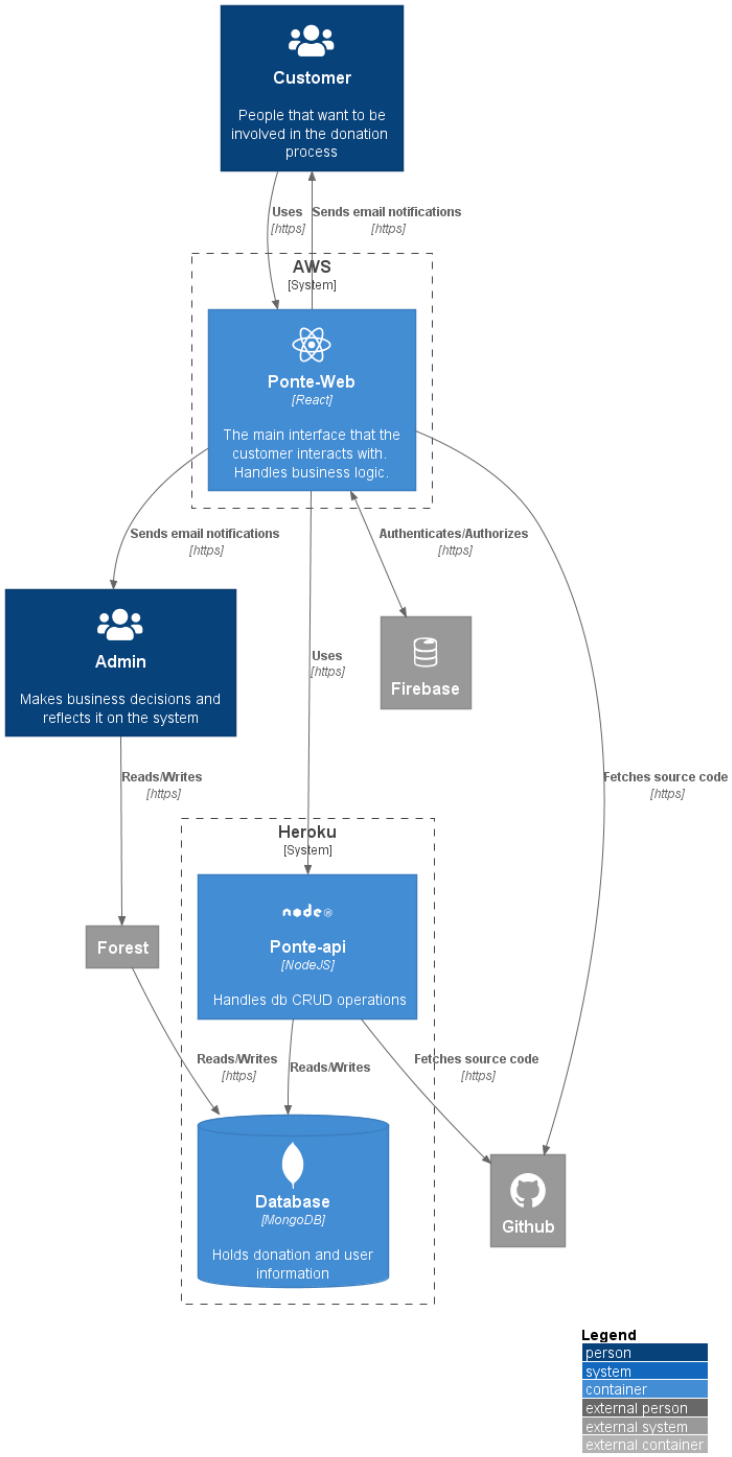

Figura 1: Arquitectura de Ponte1

2.3. Diseño

En cuanto al diseño y modelado de clases, se identificaron mejoras relacionadas a los siguientes aspectos:

- Problemas de nomenclatura tanto de endpoints como de clases dentro del código. En ninguno de los dos casos se seguían las mejores prácticas de diseño de APIs [4] lo que dificultaba el entendimiento del código para futuras iteraciones.
- El campo email no se encontraba restringido como *unique* en la creación de usuarios. Es decir, una persona podía crearse dos usuarios con el mismo email.
- Los objetos existían contenidos entre sí en lugar de referenciarse por un identificador.
    - Específicamente, se estaba utilizando una técnica de referencia de MongoDB llamada DBRef. La misma documentación de Mongo [5] dice que a menos que haya una razón explícita para utilizarla, es mejor referenciar manualmente los objetos y delegar a la lógica dentro de la aplicación el buscarlos o no si los necesita. Esto se debe a que DBRef anida objetos enteros en vez de referenciarlos, lo que le agrega tiempo innecesario de computación a la base de datos.
- No existía encapsulamiento de objetos necesarios para la representación de los ítems asociados a donaciones, ofertas y pedidos (*Weight*, *Good* y *Value*, entre otros). Dichos objetos eran únicamente representados por el agrupamiento de campos arbitrarios.
- Los distintos tipos de roles que puede tener un usuario dentro de la aplicación estaban mal representados. No deberían existir *User* (donante) y *Volunteer* como clases excluyentes ya que un usuario voluntario, por definición, es también un usuario. Ambos *Donor* y *Volunteer* como roles son usuarios y por eso comparten la información general de usuario y en todo caso deberían tener información agregada según su rol lo indique. Por esta razón, ambos se deberían instanciar como *User* con su información general y contener además *VolunteerInfo* respectivamente para que el modelo de datos esté correctamente representado.

2.4. Seguridad

Uno de los puntos de mejora más grandes que se encontraron al analizar el estado del arte al comienzo del proyecto fue la desprotección que presenta la base de datos actual.

La primer versión de Ponte API era una aplicación en Java Script que se encargaba básicamente de comunicarse con la base de datos y devolver el resultado de la consulta. Se reducía a un ABM (alta, baja y modificación) de objetos. No contaba con un sistema de seguridad y los endpoints estaban accesibles a pegadas de usuarios no autenticados. Esto se puede observar fácilmente en el repositorio mencionado previamente.

Mientras que la plataforma no fuera expuesta al público, esto no presentaba un problema. Con el desarrollo de las aplicaciones móviles y con la integración de nuevos colaboradores al proyecto, se presentó la oportunidad de comenzar a pensar en una posible salida a producción. Para que esto pudiera ser una realidad se necesitaba que la aplicación sea segura, y esto implicaba implementar algún tipo de seguridad sobre los endpoints para que la base de datos no pueda ser modificada de forma errónea o maliciosa.

Se necesitaba, principalmente, implementar seguridad basada en tokens de acceso. Puntualmente lograr que cada llamada a la API esté autenticada y autorizada de acuerdo a los recursos a los que se quiere acceder, y de acuerdo a quién está haciendo la llamada. También se necesitaba un servicio de recuperación de contraseña y de verificación de la cuenta creada.

3. Extensión y alcance del proyecto

A partir del estado actual de Ponte y de la identificación de los puntos con potencial de mejora desarrollados en la sección anterior, el alcance del proyecto consistió en las siguientes contribuciones que representaron el plan de mejora:

- En una primera instancia, se analizaron en profundidad los puntos débiles de la versión original del back-end. Se tuvo en cuenta el primer análisis de Sebastían y Noelia en cuanto a requerimientos funcionales y atributos de calidad, iterando sobre ellos para encontrar mejoras aplicables.

- Se diseñó a partir del análisis mencionado una nueva arquitectura que agrega robustez y escalabilidad a la aplicación. Se reescribió el back-end en Java, separando la funcionalidad existente en capas de persistencia, dominio y servicios según fue requerido.

- Se redujo la interdependencia de servicios externos como Firebase, para incluir sus funcionalidades de autenticación de usuarios y *mailing* nativamente en la aplicación.

- Se buscó unificar la gestión de los múltiples servicios que actualmente se encuentran separados para facilitar el manejo de tanto los ambientes de desarrollo y producción, como las bases de datos y el back-office, entre otros. El objetivo fue simplificar el desarrollo de la aplicación a futuro.

Cabe mencionar que la definición del plan de mejora fue asesorada por Sebastián Vera, miembro del primer equipo de Ponte, con quien se realizaron varios encuentros virtuales para definir el alcance del proyecto actual y consultar cuestiones del proyecto original.

Una vez definido el alcance, también se concretaron reuniones con él para explicar y discutir decisiones que afectarían a los modelos de la aplicación y a parte de la lógica de negocio. Se realizaron estas interacciones de la misma forma que se realizarían reuniones con el área de producto de una empresa de software.

3.1. Limitaciones

A la hora de evaluar las opciones posibles para resolver la unificación del *hosting* de ambos el front-end y el back-end, se tuvieron en cuenta diferentes alternativas como *AWS* y *Heroku* entre otros. Este aspecto del proyecto presentó varias limitaciones que afectaron a la estimación original y al alcance respectivamente. A continuación se detallan las consideraciones que se tuvieron y las decisiones que se tomaron en relación a los obstáculos que surgieron.

3.1.1. Amazon Web Services

Con el objetivo de utilizar los créditos educativos proporcionados por *AWS* se intentó hacer un despliegue de prueba de una API REST escrita en Java utilizando el *framework* de Spring Boot. En esta prueba se utilizó Docker para generar una imagen liviana, auto-conentida y aislada, ECR (Elastic Container Registry) para disponibilizar en *AWS* esa imagen en un contenedor de Docker y ECS (Elastic Container Service) para desplegarla.

La limitación encontrada en esta instancia estuvo dada por la falta de acceso al usuario ROOT en la cuenta educativa de *AWS*. A diferencia de las cuentas personales, en esta no se contaba con este nivel de privilegio en usuarios por lo que no fue posible usar las tecnologías mencionadas previamente.

Un segundo intento con *AWS* implicó desplegar una API Serverless usando *Lambda* y *API Gateway* [6]. Esta alternativa fue sugerida a través de una consulta con el Lic. Luis Paolini, profesor de la materia Cloud Computing (82.08). El beneficio principal que fomentó el uso de esta alternativa fue el hecho de que dichas tecnologías pertenecen al *Free Tier* de *AWS* en cualquier tipo de cuenta además de que proveen un modelo *serverless*.

Las ventajas de una API Serverless son varias, y se quiso hacer uso de las siguientes para la aplicación de Ponte:

- Escala a cero. Es decir, cuando nadie está usando el servicio, el costo es nulo ya que está apagado.
- La plataforma de *hosting* maneja todo lo relacionado a la configuración, prendido y apagado de la aplicación. También escala automáticamente.

Se intentó realizar la migración haciendo uso de SAM (Serverless Application Model), una herramienta para testear localmente y desplegar a AWS. SAM dockeriza la aplicación para simular el entorno en el que va a existir en la nube.

Durante el proceso de migración, se presentaron distintos obstáculos relacionados con el uso de Java en conjunto con Spring Boot con un modelo serverless. Java es especialmente interesante como lenguaje debido a su madurez, su comunidad y su *codebase*. Sin embargo, a continuación se detallan las razones por las cuales se concluyó que no sería eficiente utilizar Java para una migración a una arquitectura *serverless*.

- El problema con Lambda, el servicio de Function as a Service de AWS, es que una instancia de lambda se enfría después de 15-30 minutos [7], lo que significa que la aplicación se reiniciará pasado ese tiempo. Esto puede tomar varios segundos para una aplicación de Spring Boot, por lo que el tiempo de inicio perjudica el tiempo de respuesta de una pegada a la API. Lambda funciona mejor con sistemas de bajo tiempo de inicio, o "cold starts", escritos en lenguajes más livianos como son Go, Python o JavaScript.

- Por otro lado, al tener avanzado en un porcentaje considerable el desarrollo del nuevo backend para el momento que se planteó esta alternativa, y considerando que ciertos *frameworks* como Spring Security no son compatibles con una migración de API Rest al servicio de API Gateway, se iba a tener que descartar gran parte del trabajo que ya había sido realizado. Se encontraron varias recomendaciones de la comunidad en contra de utilizar el servicio de AWS Lambda en conjunto con Spring Security [8].

- Finalmente, cabe destacar que la integración con MongoDB no resultaría fácil ya que el servicio está optimizado para utilizar DynamoDB [9] y por lo tanto significaba que además se debería migrar también la base de datos a DynamoDB, modificando o descartando gran parte de la capa de persistencia.

Las aplicaciones de Spring Boot están diseñadas para permanecer activas, no reiniciarse por solicitud, lo que las hace más adecuadas para un servicio como AWS Fargate [10]. Este también es un servicio *serverless* donde simplemente se envía una imagen de Docker a AWS ECR y Fargate se encarga del resto. Es un "serverless de

larga duración". El inconveniente en este caso es que este servicio no pertenece a la *Free Tier* de *AWS* y no entraba en el alcance del proyecto costearlo personalmente. Esto llevó a la conclusión de que lo mejor sería ir por un modelo clásico donde se *hostería* de manera junta o separada el front-end y el back-end en un servidor idealmente gratis.

3.1.2. Heroku

Como consecuencia de la sección anterior, se prosiguió con un despliegue clásico en la plataforma *Heroku* [11], que provee varios servicios, de los cuales se utilizaron:

- Contenedores con 1000 horas por mes de uso gratis. Esto sirivó para contener tanto el back-end como el front-end de Ponte.
- Despliegues automáticos conectados con el repositorio de Github que utilizamos en el desarrollo.
- Fácil integración de otros usuarios como contribuidores a la configuración central del proyecto.

Al día de la fecha de entrega de este informe, el proyecto en su totalidad se encontraba andando en la siguiente url: `https://ponte-web-v2.herokuapp.com/`

4. Producto desarrollado

4.1. Atributos de calidad

A partir de los atributos de calidad originales mencionados en la sección 2, se reevaluaron los mismos, y para Ponte API-v2, se propusieron y justificaron los siguientes como se detalla a continuación:

1. Usability
    - Usuarios con poca experiencia en el uso de este tipo de tecnologías deben poder usarla para realizar donaciones.
    - Instituciones chicas, sin gente dedicada a manejar sistemas de tecnología de la información, deben poder ingresar pedidos fácil y rápidamente.
    - La aplicación en sí está pensada para facilitar el intercambio de productos entre personas. Si resulta más difícil de usar que un contacto personal, entonces falla completamente en su objetivo.
2. Security
    - Los datos de los donantes y las instituciones como fuera domicilio, teléfono, y especialmente qué donan o qué reciben como donación son información personal y confidencial.
    - El hecho de que las donaciones puedan ser de cualquier tipo implica que puedan existir algunas de alto valor. El sistema se encarga del seguimiento y transacción de las donaciones por lo que debe ser seguro para preservar las mismas.
3. Manageability
    - Se tiene que poder manejar fácilmente por los administradores porque requiere de validaciones que solo puede realizar una persona física.
4. Availability
    - La aplicación debe estar disponible a todo momento, más aún si se tiene que atender emergencias y alguien necesite donaciones de comida, ropa, etc ante una catástrofe natural.

5. Scalability

    - La aplicación debe poder escalar adecuadamente en el caso de una catástrofe natural.

Además de modificarse el orden de los atributos originales, los atributos que fueron reemplazados con respecto a la primera versión de Ponte son: Usability en lugar de Reliability y Manageability en lugar de Maintainability.

4.2. Arquitectura de los servicios

Como se describió anteriormente, uno de los objetivos del proyecto Ponte API-v2 era el de mejorar y facilitar el manejo de la aplicación de Ponte web como un todo. Para esto, se tomaron los siguientes pasos:

- La nueva arquitectura de Ponte unifica el *hosting* de ambos front-end y back-end en el mismo servicio, en este caso en *Heroku*. Esto facilita agregar y manejar colaboradores, ya que en la implementación inicial se necesitaban varias credenciales y servicios (AWS, Heroku, Firebase, Forest) para poder administrar la aplicación entera.

- Se migró la base de datos al servicio que provee Atlas, la plataforma a la que MongoDB le da soporte oficialmente. Se cree que esto facilitará cualquier problema que pueda haber en un futuro, ya que la documentación siempre estará actualizada y la comunidad atenta a resolver dudas.

    - Cabe destacar en este punto que la base de datos antigua que estaba soportando tanto Ponte web como las aplicaciones móviles quedó *hosteada* en un servicio que fue deprecado unos meses luego de que se comenzara la implementación del presente proyecto. Esto quiere decir que se perdieron los datos y el acceso a tal base de datos, por ende la misma no se encontraba productiva. La nueva base que deja implementada Ponte API-v2 era no solo ventajosa sino necesaria y servirá para dar soporte a todas las aplicaciones que se integren a ella en un futuro.

- Se eliminó por completo la necesidad de los servicios de Firebase y Forest, simplificando otra vez la coordinación de datos.

- Firebase fue reemplazado por la integración de los servicios de autenticación y envío de mails en el código de la API en sí.
- Como se mencionó anteriormente, se tuvo que crear una nueva base de datos en Atlas. El *Free tier* que ofrece el servicio no es compatible con Forest. Sin embargo la API está preparada para que se construya el back-office integrado en la misma plataforma para no tener que seguir dependiendo del servicio externo que brindaba Forest. Hoy en día todo lo que se podía hacer desde ahí se puede hacer desde la interfaz que brinda Atlas.
- Cabe aclarar que si en el futuro se necesita dividir la funcionalidad del back-end para, por ejemplo, tener los servicios de autenticación y envío de mails por separado, la forma en la que está diseñado el código hará esta tarea relativamente fácil.

En la Figura 2 más abajo se puede ver explicitado el nuevo diseño de la arquitectura de Ponte (iteración) 4.

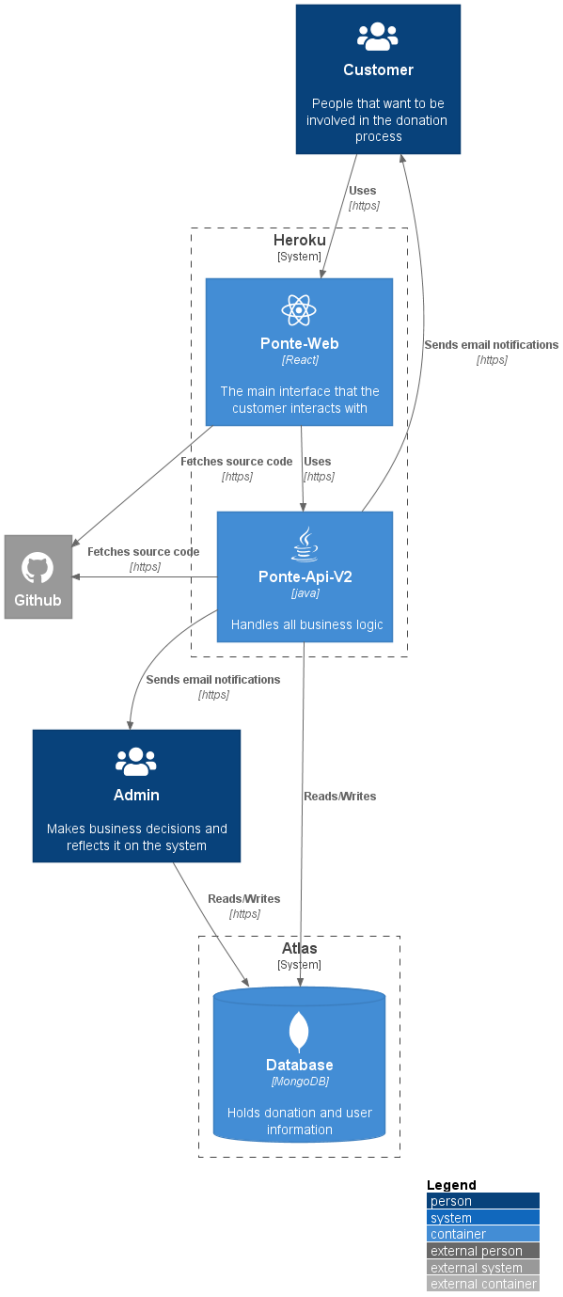

Figura 2: Arquitectura de Ponte4

4.3. Estructura del código

En el transcurso del desarrollo se intentaron seguir los principios SOLID [12] del diseño en el paradigma orientado a objetos. En esta sección se explica cómo se intentó cumplir con las buenas prácticas asociadas a dichos principios.

Se trabajó con una división del código en capas muy precisa y de acuerdo con las mejores prácticas del paradigma.

- En cuanto a modelado de datos: Se tienen **modelos** y **DTO**s correspondientes para cada entidad necesaria.
- En cuanto a separación de responsabilidades: Se tienen **controladores**, **servicios**, y capa de **persistencia** en un intento de explicitar y separar correctamente el código de acuerdo al principio de *Single responsability* en el diseño del paradigma orientado a objetos.

En cuanto a respetar el principio *Dependency inversion*, se utilizaron las herramientas que provee Spring y el uso de interfaces sobre todos los servicios y la capa de persistencia para cumplir con el objetivo de lograr que las clases no dependan de implementaciones concretas, sino de abstractas. De esta forma si se quisiera, por ejemplo, cambiar el tipo de base de datos, se debería únicamente cambiar el *Data Access Object* implementado.

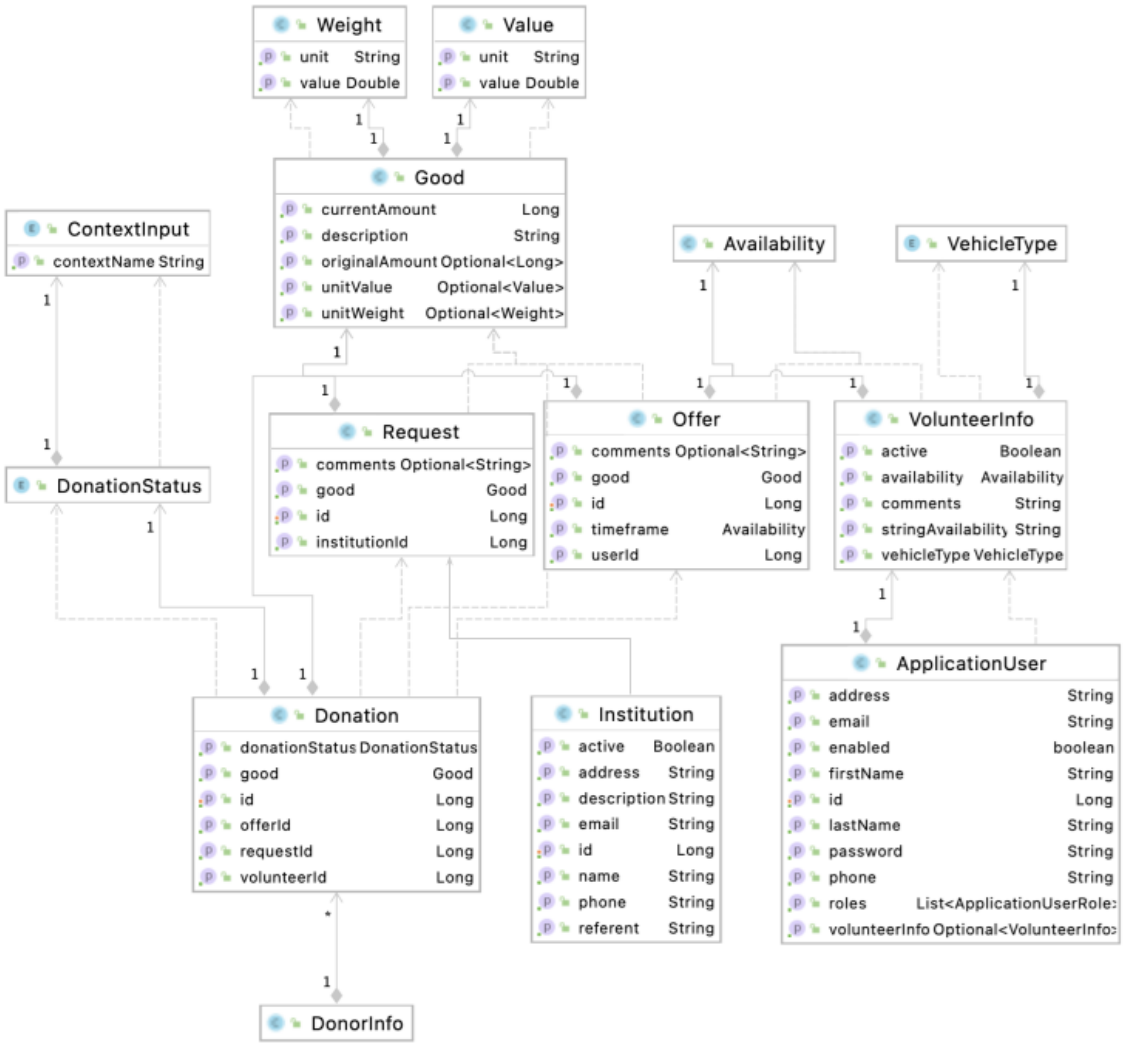

Figura 3: Diagrama de clases

4.4. Lógica de negocio

Como se detalló anteriormente, Ponte API en su primer versión se resumía en disponibilizar endpoints de crear, eliminar y actualizar objetos. Para que Ponte API-v2 sea considerada realmente una mejora sobre su predecesora, además de los puntos descriptos más arriba, se integró gran parte de la lógica de negocio que se estaba procesando manualmente por los administradores de la página. Esta sección describe los cambios más significativos a los endpoints más importantes.

4.4.1. Baja lógica

Todos los DELETE se implementaron como bajas lógicas. Se agregó un campo en las entidades: **visible**, y que al crear un nuevo objeto se lo guarda como *true*, y al eliminarlo, como *false*. Las solicitudes GET que no sean por endpoint de administrador solo traerán las entidades que tengan este campo en *true*.

4.4.2. Creación de donaciones

Para concretar donaciones, en la versión ideada originalmente por Sebastián y Noelia se habían modelado tres entidades principales: pedidos (*Requests*), ofertas (*Offers*) y donaciones (*Donations*). La lógica principal recaía en que para que una donación se considere como tal debían haber una oferta y un pedido para emparejar. Sin embargo, se notó que no siempre se seguía esta restricción, y para la nueva versión de Ponte API se intentó delimitar las posibles situaciones correctamente y de acuerdo a lo planteado originalmente. El diagrama presentado debajo es una simplificación de lo que se implementó en el back-end.

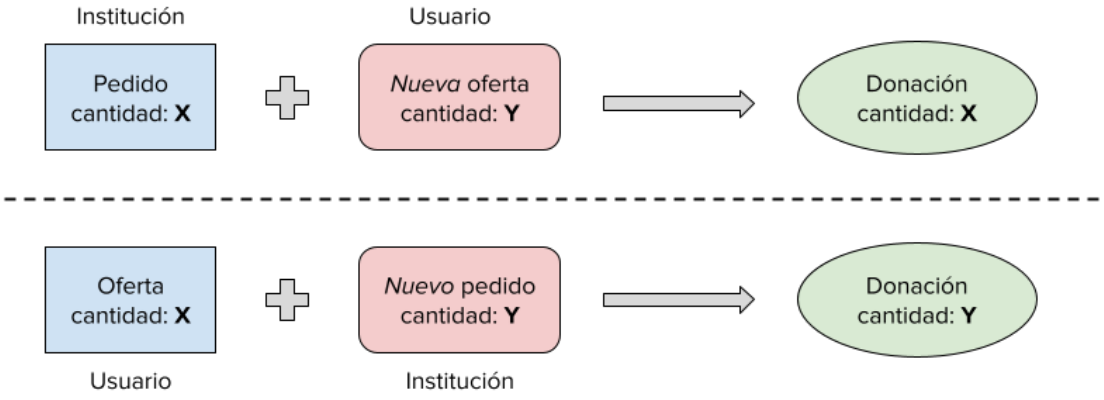

Figura 4: Creación de una donación

En la versión anterior de la API, cuando se creaba una *Offer* programáticamente, para que la misma se convirtiera en una *Donation* tenían que suceder 2 cosas:

- Se debía crear una *Request* dentro de la *Offer* en el back-office (Forest).
- Se debía crear una *Donation* en el back-office (Forest)

En ambos casos, se necesitaba intervención de un administrador para que el flujo siguiera su camino y efectivamente se publicase dicha oferta o pedido. Esto causaba que la base de datos pudiera ser inconsistente y que un error de ingreso manual de información rompiera el modelo de negocios original.

Por otro lado, cuando un usuario se mostraba interesado en donar algo que una institución estaba pidiendo (en otras palabras, cuando se creaba una *Offer* a partir de una *Request* existente), se generaba una llamada a un endpoint que creaba la oferta silenciosamente y asociaba esa oferta y el pedido original con la nueva entidad de *Donation* que generaba. Esta era la única manera de crear una entidad donación, y requería de un ingreso manual. En la segunda versión de la API se desarrollaron endpoints que abarcan todos los escenarios posibles en la creación de una donación. Los endpoints en cuestión son:

```
POST /donation/offer/{requestId}
POST /donation/request/{offerId}
```

El objetivo de estos endpoints, y que de esta forma se pueda crear una donación fue el de corregir y alinear la idea original de una donación (un encuentro entre una *Request* y una *Offer*) al funcionamiento de la API. A partir de Ponte API-v2, las únicas formas válidas de crear una donación son:

- A través del primer endpoint. Se planeta el escenario en el que existe un pedido realizado por una institución y un donante elige donar a dicho pedido. Al endpoint se le envía el identificador del pedido que eligió y los datos necesarios para saber qué es lo que el donante quiere aportar, creando así una donación propiamente dicha.

- A través del segundo endpoint. Hay una oferta de un donante, y una institución "elige" dicha oferta porque necesita la donación. Al endpoint se le envía el identificador de la oferta que eligió, y los datos necesarios para crear una donación.

En ambos endpoints se actualizan los datos de todos los objetos pertinentes y se valida que el emparejamiento sea correcto. Específicamente se valida todo lo relacionado con la cantidad de objetos pedidos y la cantidad de objetos ofertados.

- Por ejemplo, si un usuario quiere donar 8 objetos a un pedido que tiene 10 objetos, entonces el pedido luego quedará con 2 objetos. Para manejar esto se agregó un campo a la *Request* y a la *Offer* llamado **originalAmount** para poder tener un conteo de este valor todo el tiempo.

- Si un usuario dona el pedido completo, se hace un cambio de estado, del pedido para que se muestre como no disponible. Sin embargo, si hay una cancelación posterior de la donación, se vuelve a levantar el pedido original automáticamente.

En conclusión, los endpoints implementados sirven para plasmar correctamente la lógica de negocio presentada en la idea original. Se cierran todos los flujos adecuadamente, y se delimita algo muy importante para el correcto funcionamiento de la aplicación: una donación es un emparejamiento entre un pedido y una oferta. No se puede crear de ninguna otra forma.

Queda pendiente analizar el último caso, que todavía no está soportado por ninguna plataforma de Ponte y para el que se necesitará un análisis funcional y de producto para saber cómo manejarlo: el caso en que se cree una donación de un pedido y una oferta, ambas ya preexistentes.

4.4.3. Máquina de estados de donaciones

En relación a los posibles estados de una donación, si bien el concepto de los estados en sí siguen siendo los mismos en la nueva versión de la API, cabe mencionar las siguientes modificaciones relacionadas a las mejoras planteadas:

- Tal como se mencionó anteriormente, el nuevo diagrama de estados de la Figura 6 refleja la creación de una donación como el encuentro entre una *Request* y una *Offer*. Esto implica la eliminación del estado *PENDING INSTITUTION* ya que en el nuevo modelo no puede existir una donación hasta que haya una institución asociada a la *Request* correspondiente. De esta manera, el primer estado unívoco de una donación es *PENDING VOLUNTEER*.

- Considerando que ahora la lógica pertenece únicamente al back-end, cada modificación al estado de una donación existe como un endpoint separado cuyo

acceso está restringido dependiendo del rol del usuario y su relación con la entidad.

En las figuras 5 y 6 se pueden observar el diagrama anterior y el nuevo diagrama de estados de una donación. En la figura 6 particularmente se puede observar el detalle de los estados y transiciones de una donación con los roles necesarios para efectuar cambios en la misma.

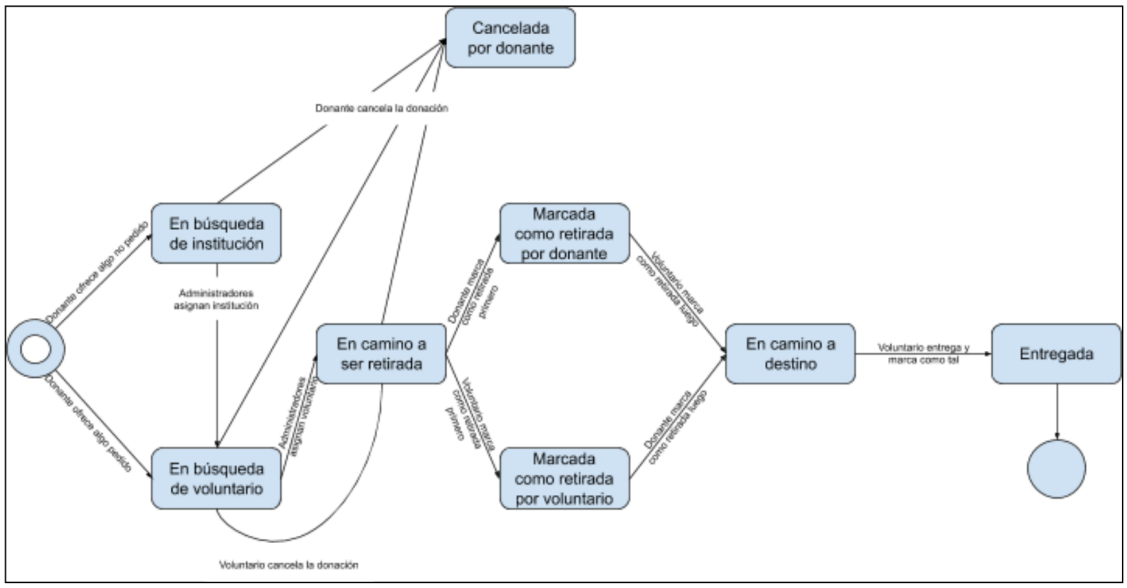

Figura 5: Antiguo diagrama de estados de una donación

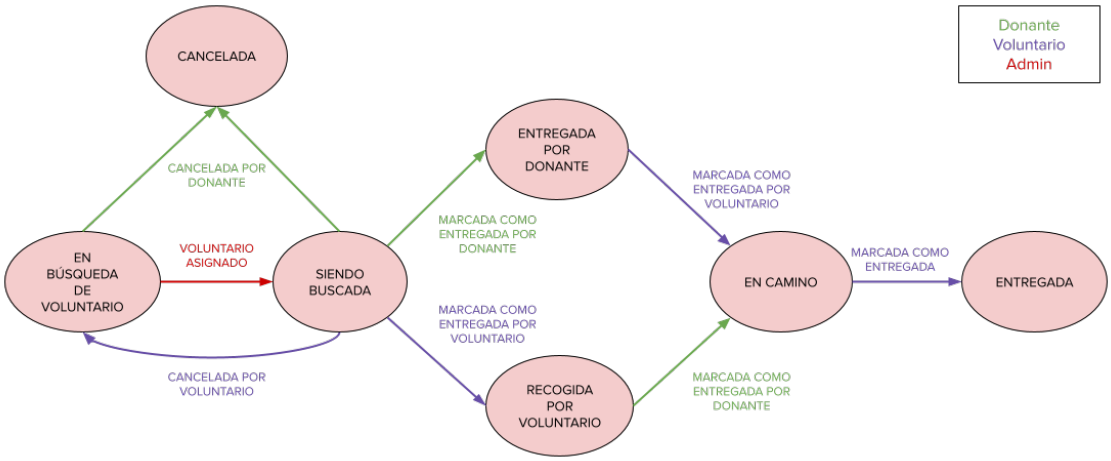

Figura 6: Nuevo diagrama de estados de una donación

4.5. Seguridad

Con el objetivo de enforzar el segundo atributo de calidad de la nueva versión de Ponte, se utilizó Spring Security junto con JWT para manejar la autenticación y autorización de los usuarios. De esta manera se reemplazó el uso de Firebase nativamente y se permitió una mejor definición, configuración y segmentación de permisos de manera programática.

Cabe mencionar que se utilizaron a su vez estas tecnologías para implementar la funcionalidad de restablecer contraseña, uno de los flujos faltantes de la versión original de Ponte.

4.5.1. Spring Security

Spring Security es un *framework* de control de acceso y autenticación potente y altamente personalizable. Es el estándar de facto para proteger las aplicaciones basadas en Spring.

Spring Security se centra en proporcionar autenticación y autorización a las aplicaciones Java. Como todos los proyectos de Spring, el verdadero poder de Spring Security se encuentra en la facilidad con la que se puede extender para cumplir con los requisitos personalizados.

Algunas características destacables son:

- Soporte completo y extensible tanto para autenticación como para autorización.
- Protección contra ataques como fijación de sesiones, clickjacking, falsificación de solicitudes entre sitios, etc.
- Integración de API de servlet.
- Integración opcional con Spring Web MVC.

4.5.2. JSON Web Token

JSON Web Token (JWT) es un estándar abierto (RFC 7519 [13]) que define una forma compacta y autocontenida para transmitir información de forma segura entre las partes a través de un objeto JSON. Esta información se puede verificar y

es confiable porque está firmada digitalmente. A continuación se expanden algunos conceptos de la definición.

Compacta: debido a su tamaño más pequeño, los JWT se pueden enviar a través de una URL, un parámetro POST o dentro de un encabezado HTTP. Además, el tamaño más pequeño significa que la transmisión es rápida.

Autocontenida: el *payload* contiene toda la información requerida sobre el usuario, evitando la necesidad de consultar la base de datos más de una vez.

El uso de JWTs resulta especialmente útil cuando se trata de procesos de autenticación o de intercambio de información, y son justamente esos escenarios en los que se implementó dicho estándar en Ponte 4.

4.6. Mailing

Originalmente la responsabilidad del envío de mails caía sobre Firebase para la confirmación de cuenta, y sobre el front-end para el resto de las notificaciones de este tipo.

En la nueva versión de Ponte, el envío de mails sucede completamente en el back-end y está implementado con el uso de Spring Email, un *framework* de envío de mails extensible soportado dentro de Spring.

4.7. Documentación de los endpoints

Se escribió un documento **.yaml** documentando todos los endpoints de la aplicación. Se utilizó la plataforma Swagger siguiendo las especificaciones de OpenAPI [14]. Para poder visualizar correctamente el documento se necesita abrirlo con algún editor de texto que soporte el formato **.yaml**. Al ser el archivo interactivo, se recomienda al lector del informe que visualice la documentación completa utilizando, por ejemplo, el editor online de Swagger: `https://editor.swagger.io/`. A continuación se presentan un par de capturas de pantalla del mismo.

Figura 7: Endpoints de donaciones

Figura 8: Modelo de donación

4.8. Integración con front-end

4.8.1. Borrado de código innecesario

Casi toda la lógica que se manejaba en el front-end en cuanto a donaciones ahora se maneja en el back-end. Por lo tanto se pudo eliminar bastante código.

- Cuando un usuario quiere donar algo que una institución pide explícitamente (es decir, hay un pedido publicado), la validación de si esto es posible y la actualización del número de elementos que siguen estando publicados para ser donados se maneja en el back-end.
- No es necesario hacer varios llamados a la base de datos para obtener las ofertas y luego las donaciones asociadas a esas ofertas de un usuario, ya que se dispone de un endpoint que devuelve todas las donaciones pertinentes a tal usuario directamente. Sucede lo mismo pero con los pedidos de una institución.

4.8.2. Voluntarios

Un usuario puede en la versión nueva de la API postularse como voluntario y ser automáticamente categorizado como tal además de como donante. Se tomó esta decisión porque se consideró que se necesitan más voluntarios siempre y significaba ralentizar el proceso de donaciones si los voluntarios debían estar verificados manualmente. Simplemente se le agrega el rol correspondiente al usuario en el back-end.

4.8.3. Envío de mails

Ya no es necesaria la aplicación de mensajería que hay en el front-end. Se eliminaron todas las llamadas que enviaban los mails desde ahí ya que ahora se realizan en un hilo aparte en el back-end. Se redujo así el tiempo de espera y se agregó la posibilidad de reintentos si el envío fuera a fallar, directamente desde el back-end.

4.8.4. Instituciones

El backend está preparado para aceptar usuarios con rol de institución. Es decir, usuarios que puedan acceder a todo lo que ahora está en el back-office, para poder integrar eso a la plataforma eventualmente.

Por ahora maneja instituciones como entidades aparte para poder adaptarse al front-end existente pero sería deseable migrar a la otra forma de hacer las cosas en un futuro cercano. Existe un endpoint para crear pedidos en nombre de una institución y estos son los que se muestran en la sección de "Pedidos"de la página web. Solo un administrador puede crear este tipo de pedidos.

4.8.5. Mejoras generales al código del front-end

En la nueva versión del front-end las sesiones del usuario perduran en el browser. Antes, si el usuario cerraba el browser o la pestaña en la que estaba navegando Ponte se le cerraba la sesión iniciada. Ahora como se utiliza la variable **localStorage** en vez de **sessionStorage**, no sucede esto.

Se encontraron varias oportunidades de mejora en el código existente del front-end. Una de los cambios significativos es que todas las pegadas al back-end se hacen con métodos asincrónicos, mejorando el rendimiento de la página web. Se recomienda migrar todo el front-end a *React hooks*. Se dejó el componente de *Login* migrado como ejemplo para el futuro.

4.9. Pruebas

4.9.1. Pruebas unitarias

Para las pruebas unitarias se utilizó el framework de JUnit 5 y las utilidades que brinda Mockito. Se implementaron mejores prácticas de *testing* como la nomenclatura *given, when, then* y la correcta separación de responsabilidades de cada prueba, es decir, que cada método se aísle y se pruebe correspondientemente.

4.9.2. Pruebas en ambientes de desarrollo

Se le dedicó un tiempo considerable a utilizar la nueva versión de la página web para comprobar su correcto funcionamiento. Se puede observar en la base de datos varios objetos de prueba utilizados para este fin. También se entregó con anticipación el link correspondiente a la tutora del proyecto para que pudiese comprobar la página en paralelo.

5. Proceso

5.1. Organización de las tareas

Se utilizó la herramienta Trello para la organización general del proyecto. Se dividió el trabajo acorde a como se dividiría cuando uno planea Sprints, dentro de la modalidad Scrum. Se tuvieron reuniones semanales o bi-semanales (dependiendo de la disponibilidad del equipo) en las que se hacía un recuento de las tareas completadas y se agregaban las que surgieran luego. Cada tarea llevaba una etiqueta correspondiente dependiendo de la categoría asignada. También se utilizó una herramienta de calendario agregada a Trello para poder tener una mejor visibilidad de las fechas pautadas para la finalización de las tareas.

Se pueden ver una captura de pantalla más abajo en el informe. La Figura 9 muestra el estado del proyecto y de sus tareas con sus correspondientes etiquetas y responsables asignados un mes antes de la entrega.

Asimismo nos resulta importante destacar que el equipo no pudo juntarse más de un par de veces presencialmente debido a la situación de pandemia mundial que se vio en el año 2020 y el 2021. Este contexto dificultó bastante el trabajo en equipo, más que nada porque el compañerismo y la presencialidad siempre acelera considerablemente el desarrollo y la colaboración de cualquier proyecto.

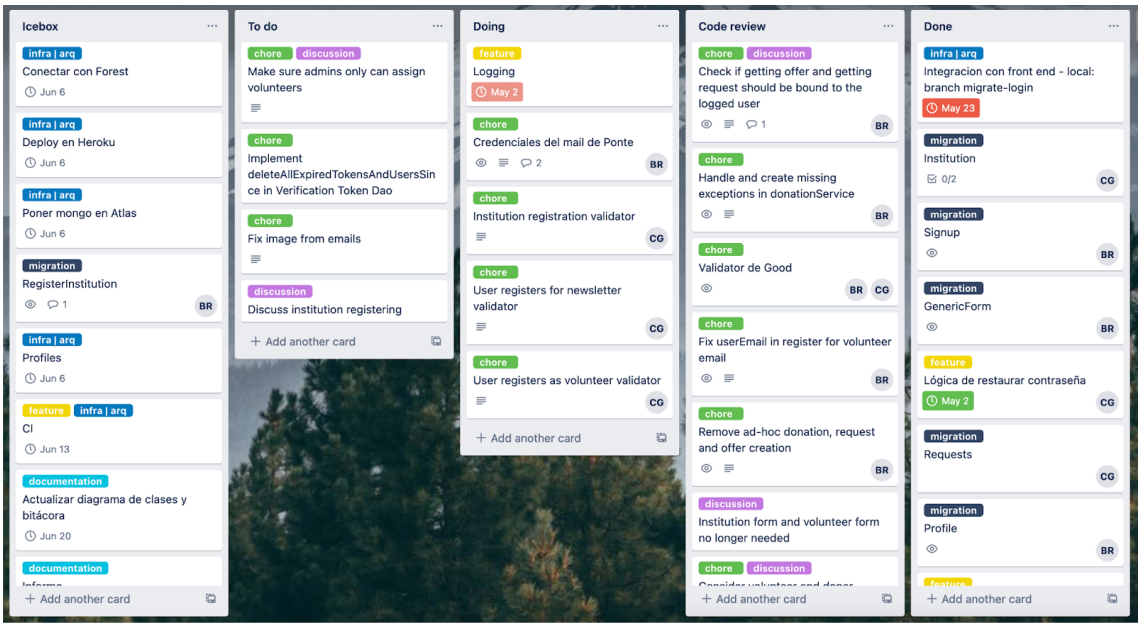

Figura 9: Panel de Trello. Un mes antes de la entrega

6. Conclusiones y trabajo futuro

6.1. Conclusiones y aprendizajes

Ponte API-v2 se planteó desde el principio que sería un proyecto interesante para llevar a cabo por varias razones. Una de ellas era que iba a ser similar a un proyecto real a implementar en una empresa que quiere analizar los puntos de falla de lo que ya tiene desarrollado y a partir de eso crear algo que mejor se adapte al negocio y más que nada, que escale fácilmente en un futuro. Se intentó pensar el proyecto justamente como el trabajo de un ingeniero en una empresa real, poniendo énfasis en que la nueva versión de algo ya existente debe ser compatible con el resto de los sistemas acoplados y que también debía estar desarrollado en paralelo, ya que no se puede simplemente descartar lo que está en producción y empezar de cero.

Fue posible seguir esta linea de desarrollo durante un año y se puede afirmar que la nueva versión de la API fue implementada sin perjudicar ni las versiones móviles de Ponte ni la versión web. Se atribuye este logro al planteo inicial de desarrollo paralelo.

Se concluye que con la presente iteración de Ponte (API-v2), la plataforma se encuentra en condiciones de ser desplegada en ambientes de producción y presentada al público general. Es necesario, para poder disponibilizar las aplicaciones móviles, que estas se integren al nuevo back-end desarrollado. Se dejó el front-end web integrado como ejemplo a seguir.

Como aprendizajes en el ámbito profesional cabe destacar la mayor dificultad encontrada y lo que surgió de ella:

- De intentar servir la aplicación utilizando la herramienta de AWS Lambda surge el aprendizaje de que se debe incluir en el plan inicial de cualquier proyecto el análisis del *hosting*. No es algo que se pueda dejar para evaluar más adelante, ya que seguramente determinará o ayudará a determinar el mejor lenguaje y las mejores tecnologías a utilizar en el proyecto.

- Del tiempo perdido intentando adaptar la aplicación para que sea compatible con AWS Lambda surge el aprendizaje de en el futuro intentar determinar claras fechas límites para trabajar en una tarea abierta como esta, y así no desperdiciar el tiempo si parece que no va a ser posible lograr lo propuesto.

6.2. Trabajo futuro

Quedan varios puntos a trabajar que surgen del trabajo realizado en Ponte API-v2:

- Una mejora futura sería, como ya se mencionó anteriormente, migrar las aplicaciones móviles tanto de Android como de iOS al nuevo back-end de Ponte.
- Otra mejora futura sería el desarrollo de un front-end que pueda hacer uso de lo que está preparado en la API en cuanto al back-office de la aplicación.
- El back-end también quedó preparado para cuando una institución pueda iniciar sesión como un usuario, y tenga a su disposición el back-office mencionado en el item anterior. Por lo tanto una mejora futura sería el manejo de instituciones como usuarios del lado del front-end y de las aplicaciones móviles.

Referencias

[1] Vera , Juan Sebastián, López, Noelia Belén. 2019. Ponte. Disponible en: https://ri.itba.edu.ar/handle/123456789/1667

[2] Noriega Defferrari, José Carlos. 2020. Ponte 2: androide mobile. Disponible en: https://ri.itba.edu.ar/handle/123456789/3395

[3] De Rienzo, Constanza, Santoflaminio, Alejandro. 2020. Ponte ios. Disponible en: https://ri.itba.edu.ar/handle/123456789/3388

[4] Restful API.net. 2018. REST Resource Naming Guide. Disponible en: https://restfulapi.net/resource-naming/ [Visitado el 4 de Julio de 2021]

[5] MongoDB documentation. 2021. Database References. Disponible en: https://docs.mongodb.com/manual/reference/database-references/ [Visitado el 4 de Julio de 2021]

[6] AWS documentation. 2021. AWS Lambda. Disponible en: https://aws.amazon.com/lambda/ [Visitado el 4 de Julio de 2021]

[7] Nathan Malishev. 2019. AWS Lambda Cold Start Language Comparisons, 2019 edition. *Level Up Coding*. Disponible en: https://levelup.gitconnected.com/aws-lambda-cold-start-language-comparisons-2019-edition-%EF%B8%8F-1946d32a0244 [Visitado el 4 de Julio de 2021]

[8] Foro Stackoverflow. 2016. AWS Lambda, AWS API Gateway and Securing REST endpoints. Disponible en: https://stackoverflow.com/questions/36653414/aws-lambda-aws-api-gateway-and-securing-rest-endpoints [Visitado el 4 de Julio de 2021]

[9] Lou Bichard. 2020. Can AWS Lambda Access A Database? And The Considerations You Should Be Taking. Disponible en: https://openupthecloud.com/can-aws-lambda-access-database/ [Visitado el 4 de Julio de 2021]

[10] AWS documentation. 2021. Fargate. Disponible en: https://aws.amazon.com/fargate/ [Visitado el 4 de Julio de 2021]

[11] Heroku documentation. 2021. Pricing. Disponible en: https://www.heroku.com/pricing [Visitado el 4 de Julio de 2021]

[12] Digital Ocean community. 2020. SOLID: The First 5 Principles of Object Oriented Design. Disponible en: https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design [Visitado el 4 de Julio de 2021]

[13] M. Jones, Microsoft, J. Bradley, Ping Identity, N. Sakimura. 2015. JSON Web Token (JWT). https://datatracker.ietf.org/doc/html/rfc7519 [Visitado el 4 de Julio de 2021]

[14] Swagger documentation. 2021. OpenAPI Specification. Disponible en: https://swagger.io/specification/ [Visitado el 4 de Julio de 2021]