Desarrollo Cooperativo de Herramientas TIC para la Democracia Directa

Alberto Miedes Garcés

Grado en Ingeniería de Computadores

Facultad de Informática

Universidad Complutense de Madrid

Trabajo de Fin de Grado

Madrid, junio 2017

Directores:

Antonio Tapiador del Dujo

Sara Román Navarro

Autorización de Difusión y Uso

Autorizo a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, tanto la propia memoria, como el código, la documentación y el software desarrollado.

Alberto Miedes Garcés

Madrid, Junio de 2017

Agradecimientos

Me gustaría expresar mi agradecimiento a todas las personas que han ayudado a la realización de este trabajo:

- A Enrique García Cota, Raimond García, Juanjo Bazán y Alberto García, desarrolladores de Consul, por el trabajo que están haciendo, los consejos y el feedback que me han dado y el tiempo que me han dedicado. En especial me gustaría dar las gracias a Enrique por la cantidad enorme de dudas que me ha resuelto y consejos que me ha dado durante todo el desarrollo de este proyecto, de carácter técnico y no técnico, no sólo en relación con este proyecto sino en diferentes ámbitos.

- A los directores de este proyecto, Sara Román Navarro y Antonio Tapiador del Dujo, cuya ayuda me ha sido enormemente útil en las fases inicial y final de este trabajo.

- A Miguel Arana Catania, Pedro Álvarez González y Yago Bermejo Abati, que han participado en el proceso de definición de la API y me han echado una mano en diversas ocasiones.

- A Pablo Moreno Ger, por su interés en este proyecto y la invitación para que diese una charla en la Semana de la Informática de la facultad.

- Finalmente, a mis padres, cuyo soporte ha sido vital para mí durante todos los años de la carrera.

Índice

Resumen

En los últimos años estamos viendo un crecimiento en el número y calidad de las iniciativas que fomentan la participación e implicación de la ciudadanía en la toma de decisiones que tradicionalmente habían sido delegadas completamente en los representantes políticos electos.

Las nuevas tecnologías están desempeñando un papel fundamental en la creación de herramientas destinadas a facilitar y abrir nuevos caminos a la participación a través de Internet: filtrado colaborativo, deliberación y discusión, toma de decisiones, procesos y estrategias de participación, redacción colaborativa de legislación, etc.

En España, Madrid ha sido una de las ciudades pioneras en llevar a cabo este tipo de iniciativas.

Uno de los proyectos que más repercusión ha tenido es Consul, la plataforma en la que se basa Decide Madrid, el portal de participación ciudadana del Ayuntamiento de Madrid. Consul aglutina mecanismos de propuestas ciudadanas, espacios de debates, y presupuestos participativos, entre otros. Además goza de una gran popularidad, pues ha sido y está siendo adoptado por multitud de ciudades de España, Europa y América Latina. Getafe, Oviedo, Valencia y París, entre otros.

El objetivo de este proyecto es implementar una API para Consul, capaz de dotar a esta plataforma de un método eficaz destinado a liberar la información de interés público almacenada en esta plataforma a todo aquel que esté interesado en hacer uso de ella. Durante el desarrollo de este proyecto se va a trabajar colaborativamente tanto con el Ayuntamiento de Madrid, los desarrolladores de Consul y otros actores relevantes.

El objetivo de este proyecto es que la API sea incorporada al repositorio principal de Consul y desplegada en producción en Decide Madrid. Por este motivo una parte muy importante del desarrollo gira al rededor de buenas prácticas en proyectos software como testing, integración continua y control de versiones, así como mantenibilidad, legibilidad y calidad de código.

Palabras Clave: Democracia Directa, Participación Ciudadana, Propuestas Ciudadanas, Consul, Decide Madrid, Transparencia, Desarrollo Colaborativo, Ruby on Rails, API, GraphQL.

Abstract

During the last years we have witnessed a growth in the number and quality of initiatives that promote citizen participation and involvement in the decision making processes that had traditionally been completely delegated to elected political representatives.

Technology is playing a key role in creating tools aimed to facilitate and open new paths to citizen participation through the Internet: collaborative filtering, deliberation and discussion, decision making, participatory processes and strategies, collaborative drafting of legislation, etc.

In Spain, Madrid has been one of the pioneer cities to carry out this type of initiatives.

Consul is one of the projects that has had the most impact. This is the platform on which Decide Madrid is based, the citizen participation website of the Madrid City Council. Consul brings together mechanisms of citizen proposals, debates, and participatory budgets, among others. Consul is also enjoying great popularity, since it has been and is being adopted by many cities in Spain, Europe and Latin America. Getafe, Oviedo, Valencia and Paris, among others.

The objective of this project is to implement an API for Consul, that will give to this platform an effective method to release information of public interest to anyone who is interested in making use of it. During the development of this project I will work collaboratively with the Madrid City Council, the developers of Consul and other relevant actors.

The goal of this project is to develop an API that can be merged into the main repositories of Consul and Decide Madrid, and deployed to production at Decide Madrid. For this reason a very important part of the development process revolves around good practices in software projects such as testing, continuous integration and version control, as well as maintainability, readability and code quality.

Keywords: Direct Democracy, Citizen Participation, Citizen Proposals, Consul, Decide Madrid, Transparency, Collaborative Development, Ruby on Rails, API, GraphQL.

1. Introducción

La Democracia Directa es una forma de democracia en la que, a diferencia de en la democracia representativa, el poder no es ejercido por unos representantes elegidos por el pueblo sino por el pueblo directamente. Con ella, la ciudadanía obtiene poderes como la capacidad de aprobar o derogar leyes, elegir a los funcionarios públicos, etc.

Muchos países que poseen democracias representativas, permiten formas limitadas de democracia directa, como son *la iniciativa popular* - que permite a los ciudadanos presentar peticiones al Estado para que un determinado asunto público sea tomado en consideración - , el *referéndum* - que puede emplearse para aprobar o rechazar una determinada ley - y la *revocatoria* - que permite al pueblo destituir de su cargo a un funcionario electo antes de finalizar su mandato.

Los primeros registros que se conservan de democracia directa provienen de la antigua democracia ateniense, donde durante dos siglos el poder recayó en una asamblea en la que estaban todos los ciudadanos varones que no eran esclavos ni extranjeros.

Las nuevas tecnologías están favoreciendo la creación de herramientas que nos ayuden a caminar hacia una democracia directa y deliberativa utilizando las posibilidades que nos brindan. El Ayuntamiento de Madrid, a través del Área de Gobierno de Participación Ciudadana, Transparencia y Gobierno Abierto, se ha marcado como objetivo poner a la ciudad de Madrid al frente de las ciudades más avanzadas en participación ciudadana y transparencia. Para conseguirlo se ha trazado un plan ambicioso y vanguardista que incluye la adopción e implementación de una batería de medidas en estas materias.

Consul, la plataforma subyacente al portal Decide Madrid (*http://decide.madrid.es*), está teniendo un papel clave en este aspecto. Desde que Decide Madrid entró en funcionamiento en septiembre de 2015 la actividad de sus usuarios - ya sea publicando y apoyando propuestas ciudadanas, iniciando o participando en debates o interactuando mediante comentarios - ha generado un volumen enorme de información. De abrir al público esta información, se daría pie a la realización de estudios, análisis de datos, visualizaciones o herramientas informáticas capaces de procesar y presentar resultados que ayuden en la toma de decisiones relativas a las propuestas y preocupaciones de los habitantes de una población.

De acuerdo con la tendencia cada vez más frecuente en las ciudades españolas de publicar en portales web la información de interés público, se planea que esta información esté disponible en el portal de datos abiertos de Madrid *http://datos.madrid.es/portal/site/egob*. No obstante, formatos habituales en los portales de datos abiertos como CSV o XML, carecen de la expresividad de otros formatos más avanzados como JSON. Además, la publicación de datos en este tipo de portales todavía suele incorporar algún paso de carácter manual, pues no está automatizada completamente y en muchos casos no contiene información actualizada ni todo lo completa que podría desearse.

1.1. Objetivos

El objetivo de este Trabajo de Fin de Grado es el desarrollo de una API para la aplicación Consul, que sea integrable en Decide Madrid pero también reutilizable desde cualquiera de las instalaciones personalizadas. Esta API será de sólo lectura y deberá prestar especial atención a la información que expone acerca de los usuarios de la aplicación, teniendo una suite de tests exhaustiva que compruebe que ningún dato confidencial es expuesto a través de ella.

Gracias a esta API se proporcionará a Consul un método rápido, moderno, automático y completo de liberar la información de interés público almacenada en la plataforma a todo aquel que esté interesado en hacer uso de ella.

Otros objetivos secundarios de este proyecto son:

- Familiarizarse con las herramientas de control de versiones utilizadas en la actualidad y con los flujos de trabajo colaborativos, como GitHub, así como con los sistemas de integración continua y revisiones de código.

- Familiarizarse con los procesos de toma de requisitos y reuniones con los potenciales clientes.

1.2. Estructura del documento

La estructura que se ha decidido utilizar para detallar este Trabajo de Fin de Grado está basada en capítulos y es la siguiente:

- **Capítulo 2 - Estado del Arte:** se hace una pequeña introducción sobre las ideas detrás de los movimientos de Democracia Directa, participación ciudadana, transparencia y algunas de las herramientas informáticas ligadas a esta temática existentes actualmente en España, prestando especial atención a Decide Madrid. A continuación se presentan ideas generales sobre el desarrollo de aplicaciones web, APIs web y testing, que han estado muy presentes en el desarrollo del proyecto.

- **Capítulo 3 - Tecnologías del Proyecto:** en esta sección se exponen los lenguajes, librerías, herramientas y tecnologías utilizadas en este proyecto.

- **Capítulo 4 - Metodología:** primero se presenta a los actores externos relevantes durante el desarrollo del proyecto, así como el papel que ha desempeñado cada uno de ellos. Luego se hace una introducción del flujo de trabajo típico de proyectos alojados en GitHub.

- **Capítulo 5 - Plan de Trabajo y Desarrollo:** describe las acciones concretas que se han llevado a cabo para implementar la API desde un punto de vista técnico. Está estructurada en una serie de fases, cada una de las cuales ha requerido utilizar unas habilidades diferentes para resolver los retos que surgieron.

- **Capítulo 6 - Casos de Uso:** se habla de potenciales usos que podrían hacerse de la información proporcionada por esta API.

- **Capítulo 7 - Trabajo Futuro:** en esta sección se presentan varias ideas y funcionalidades que sería interesante añadir a la API, pero que se han decidido dejar fuera del alcance de este proyecto.

- **Capítulo 8 - Trabajo Relacionado:** en este apartado se exponen una serie de actividades que he tenido la oportunidad de realizar gracias a la temática de este proyecto y que, aunque no ligadas estrictamente a los objetivos del trabajo de este TFG, han supuesto experiencias complementarias muy interesantes y enriquecedoras.

1. Introduction

Direct Democracy is a form of democracy in which, unlike representative democracy, power is not exercised by representatives elected by the citizens but by these citizens directly. With this kind of democracy citizenship obtains powers like the capacity to approve or to repeal laws, to choose the civil servants, etc.

Many countries that have representative democracies allow limited forms of direct democracy, such as the *popular initiative* - which allows citizens to ask the state to take into consideration a particular public issue - the *referendum* - that can be used to approve or to reject a certain law - and the *revocation* - that allows the citizenship to remove an elected official from office before the end of his term.

The earliest surviving records of direct democracy come from the ancient Athenian democracy, where for two centuries the power fell to an assembly composed by all male citizens who were neither slaves nor foreigners.

Technology is favoring the creation of tools that help us moving towards a direct and deliberative democracy using the possibilities that they offer. The Madrid City Council, through the *Government Area of Citizen Participation, Transparency and Open Government*, has set itself the goal of putting the city of Madrid at the forefront of the most advanced cities in citizen participation and transparency. To achieve this, an ambitious and avant-garde plan has been drawn up which includes the adoption and implementation of a battery of measures in these matters.

Consul, the platform on which Decide Madrid (*http://decide.madrid.es*) is based, is playing a key role in this regard. Since Decide Madrid came into operation in September 2015 the activity of its users - whether publishing and supporting citizen proposals, starting or participating in debates or interacting through comments - has generated a huge amount of information. Opening this information to the public would ease the development of studies, data analysis, visualizations or software tools capable of processing and presenting results that help in making decisions regarding the proposals and concerns of the citizenship.

According to the growing tendency in Spanish cities about publishing information of public interest in websites, it is planned that this information will be available in the open data website of Madrid *http://datos.madrid.es/portal/site/egob*. However, common formats in open data websites such as CSV or XML, lack the expressiveness of other more advanced formats such as JSON. In addition, the publication of data in this type of websites usually incorporates some step of manual nature, as it is not fully automated and in many cases does not contain updated information, or as complete as could be desired.

1.1. Objectives

The goal of this project is the development of an API for the Consul application, which will be mergeable in Decide Madrid but also reusable from any of the custom installations. This API will be read-only and should pay special attention to the information that it exposes about the application users, and have a comprehensive test suite that verifies that no confidential data is exposed.

Thanks to this API, Consul will be provided with a fast, modern, automatic and complete method of releasing information of public interest stored on the platform to anyone interested in making use of it.

Other secondary goals of this project are:

- To become familiar with version control tools used nowadays and with collaborative workflows, such as the GitHub one, as well as continuous integration systems and code reviews.

- To become familiar with the process of requirements capture and meetings with potential clients.

1.2. Document Structure

This project report is split in the following chapters:

- **Chapter 2 - State of the Art:** first, a small introduction is made about the ideas under the Direct Democracy, citizen participation and transparency movements, as well as some of the related technological tools existing in Spain at the moment. Special focus is made on Decide Madrid. Later, I expose some general ideas about web applications development, web APIs and testing, which have been very present during the development of this project.

- **Chapter 3 - Technologies:** this section presents the languages, libraries, tools and techniques used in this project.

- **Chapter 4 - Methodology:** firstly I present relevant external actors during the development of the project, as well as the role played by each one of them. Then I make an introduction about the typical workflow for software projects hosted at GitHub.

- **Chapter 5 - Project Plan and Development:** describes specific actions I've taken to implement the API, from a technical point of view. This section is structured in a series of phases, each of them requiring of different skills to solve the challenges that arose.

- **Chapter 6 - Use Cases:** I mention potential uses of the information provided by this API.

- **Chapter 7 - Future Work:** this section presents several ideas and functionalities that would be interesting to add to the API, but which I've decided to leave out of the scope of this project.

- **Chapter 8 - Related Work:** this section presents a series of activities that I have carried out thanks to the theme of this project and, although not strictly linked to the objectives of this project, have provided complementary experiences very interesting and enriching.

2. Estado del arte

En esta sección se tratan los siguientes temas:

- Primero se hace un pequeño repaso de las iniciativas más importantes existentes en España en lo que a herramientas relacionadas con la participación ciudadana y transparencia se refiere. Se hace especial hincapié en el estado de Decide Madrid y Consul, pues es la aplicación con la que se va a trabajar durante el desarrollo de este proyecto.

  Aunque Consul es principalmente una plataforma de participación ciudadana, se ha decidido tratar también el tema de los portales de transparencia, pues son estos los que proveen al ciudadano de la información necesaria para luego poder tomar decisiones de forma informada en los portales de participación.

- Luego se hace un resumen del estado actual del desarrollo de APIs web. Primero se explica la arquitectura REST, predominante en este momento, y a continuación se hace una introducción de GraphQL, tecnología emergente pero muy prometedora.

2.1. Plataformas de Participación Ciudadana

2.1.1. Decide Madrid

Decide Madrid es el portal de participación ciudadana del Ayuntamiento de Madrid. Su desarrollo empezó el 15 de julio de 2015, y fue subido a producción por primera vez el 7 de septiembre de 2015.

El objetivo principal de este portal es aumentar de forma significativa la relevancia que tienen las opiniones de los madrileños en la toma de decisiones que afectan a la propia ciudad.

Las secciones principales de este portal son:

- **Debates**

  En esta sección cualquier persona puede abrir un hilo de discusión sobre cualquier tema, creando un espacio independiente donde la gente podrá debatir sobre el tema propuesto.

- **Propuestas**

  En esta sección cualquier persona puede proponer una iniciativa con la intención de recabar los suficientes apoyos como para que la idea pase a ser consultada a toda la ciudadanía con carácter vinculante.

Si bien cualquier persona registrada en la plataforma puede crear una propuesta, sólo los ciudadanos empadronados en Madrid que hayan verificado su cuenta en la web tienen la posibilidad de apoyarlas.

Cuando una propuesta alcanza una cantidad de apoyos equivalente al 1% del censo de Madrid (27.064 apoyos exactamente), pasa a ser estudiada por un grupo de trabajo del Ayuntamiento, y pasará a a la siguiente fase de consulta popular, en la que la ciudadanía de Madrid podrá votar si decide llevarla a cabo o no.

- **Presupuestos Participativos**

  Los presupuestos participativos son un proceso democrático en los que la ciudadanía decide de manera directa a qué se destina una parte del presupuesto municipal. Los ciudadanos proponen proyectos de gasto que son preseleccionados en una fase de apoyos ciudadana, se evalúan y finalmente llegan a una votación final. A partir del 1 de enero del año siguiente el Ayuntamiento lleva a cabo los proyectos seleccionados en la votación final.

  Aunque las secciones de Propuestas y Presupuestos Participativos pueden parecer idénticas, se rigen por unas restricciones diferentes.

  Para obtener más información acerca del funcionamiento de los presupuestos participativos de Decide Madrid lo mejor es consultar la sección de preguntas frecuentes de la propia web en: *https://decide.madrid.es/presupuestos/faq*.

- **Legislación Colaborativa** (en desarrollo)

  Esta sección va a proporcionar una herramienta para que los ciudadanos puedan participar en el proceso de redacción de leyes. Los legisladores irán publicando sucesivas versiones del borrador de la ley, sobre el cual cualquier persona podrá hacer observaciones y proponer modificaciones.

Una de las primeras iniciativas llevadas a cabo por el actual ayuntamiento de Madrid fue el empezar a utilizar software libre dentro de la propia institución con el objetivo de reducir el sobrecoste derivado de externalizar la realización de los proyectos informáticos de la administración a entidades externas. Por este motivo el código de Decide Madrid es software libre y está disponible en GitHub bajo la licencia AGPL v3.

El hecho de que el código de Decide Madrid esté disponible bajo este tipo de licencia ha sido uno de los responsables de que muchos otros ayuntamientos se hayan sumado a utilizar este mismo software.

Decide Madrid está desplegado en la siguiente dirección: *https://decide.madrid.es*, y el código está en este repositorio: *https://github.com/AyuntamientoMadrid/consul*.

2.1.2. Consul

Cuando se vio el potencial que tenía la idea de detrás de *decide.madrid.es* después de que más ayuntamientos, instituciones y organizaciones mostrasen su interés por el proyecto, se decidió extraer una versión "genérica" del código de Decide Madrid con el objetivo de que este fuese reaprovechable por cualquier persona o colectivo.

Esta versión se encuentra alojada en la siguiente dirección: *https://github.com/consul/consul*.

La siguiente tabla muestra una relación con los ayuntamientos, instituciones y organizaciones que ya han desplegado o que están investigando con Consul:

Entidad	Web	Estado
Ayuntamiento de Madrid	*https://decide.madrid.es*	En funcionamiento
Ayuntamiento de Barcelona	*https://decidim.barcelona*	Originalmente basada en Consul
Ayuntamiento de Oviedo	*www.consultaoviedo.es*	En funcionamiento
Ayuntamiento de Getafe	*https://participa.getafe.es*	En funcionamiento
Ayuntamiento de Valencia	*https://decidimvlc.valencia.es*	En funcionamiento
Diputación de Valencia	*https://go.dival.es/*	En desarrollo
Ayto. Calvià de Mallorca	*https://www.participacalvia.es*	En funcionamiento
Univ. Complutense de Madrid	*https://github.com/imartinezortiz/consul*	En investigación
París	*https://budget-participatif.rivp.fr*	En investigación

2.1.3. Decidim Barcelona

Antes de que se crease la versión genérica de Decide Madrid, Consul, el Ayuntamiento de Barcelona empezó el desarrollo de Decidim Barcelona, una plataforma que en su origen era similar a Decide Madrid pero que ha ido evolucionando de forma independiente para satisfacer las necesidades específicas de la ciudad de Barcelona en lo que a participación ciudadana se refiere.

Aunque en los inicios hubo numerosos intentos de coordinar el desarrollo de estas dos plataformas, finalmente fue imposible y el desarrollo de Decidim Barcelona acabó derivando en una serie de repositorios totalmente independientes de Consul.

El código original de Decidim Barcelona está alojado en el siguiente repositorio:

https://github.com/AjuntamentdeBarcelona/decidim.barcelona-legacy

Y el actual en el siguiente:

https://github.com/AjuntamentdeBarcelona/decidim-barcelona

La web está publicada en la siguiente dirección:

https://decidim.barcelona

2.2. Portales de Transparencia

Otra área de acción dentro de las aplicaciones informáticas, y que comparte algunos objetivos con iniciativas como la democracia directa / participación ciudadana o el *open data* son los portales de transparencia.

La *Ley 19/2013, de 9 de diciembre, de transparencia, acceso a la información pública y buen gobierno* tenía como objetivo ampliar y reforzar la transparencia de la actividad pública, regular y garantizar el derecho de acceso a la información relativa a esta actividad, y establecer las obligaciones que deben cumplir los responsables públicos.

Es posible consultar el texto de la ley en el BOE correspondiente:

https://www.boe.es/buscar/doc.php?id=BOE-A-2013-12887

Como consecuencia de su aprobación, las distintas instituciones y organismos afectados pusieron en marcha medidas con el objetivo de su cumplimiento, resultando en la aparición de los portales de transparencia, aplicaciones web donde es posible consultar toda esta información.

Cumplir la Ley de Transparencia es un gran avance en cuanto a la cantidad de información que se proporciona a nuestros vecinos. Pero muchas veces por falta de recursos se ofrecen documentos oficiales, como los presupuestos, que pueden ser muy áridos de consultar para un ciudadano. Y el efecto neto es que cumplimos la ley, pero no mejoramos la comunicación con nuestros vecinos.

Cómo presentar toda esta información al ciudadano de un modo entendible y fácil de consultar es el gran reto pendiente de estos portales.

2.2.1. Portal de Transparencia del Ayuntamiento de Madrid

El desarrollo del portal de transparencia del Ayuntamiento de Madrid comenzó en octubre de 2015, también es software libre, y está alojado en el siguiente repositorio:

https://github.com/AyuntamientoMadrid/transparencia

y está desplegado en la siguiente URL:

http://transparencia.madrid.es/portal/site/transparencia

2.2.2. Gobierto

Gobierto es una plataforma de gobierno abierto open source desarrollada por la empresa Populate Tools. Ofrece herramientas de transparencia, participación y rendición de cuentas para administraciones públicas y otras organizaciones que puedan estar interesadas.

Actualmente ofrece las siguientes funcionalidades:

- Visualizaciones relativas a los presupuestos de un municipio.
- Personas y agendas: muestra el organigrama de la organización y los datos clave de sus integrantes, como la biografía, el curriculum, las declaraciones de bienes y actividades, etc.
- Indicadores estadísticos: visualizaciones de los principales indicadores estadísticos de tu municipio - población, empleo, economía, criminalidad...
- Consultas sobre presupuestos: permite realizar consultas a la ciudadanía y obtener una base de opiniones de los temas que más preocupan para poder tomar las decisiones presupuestarias más adecuadas.
- Comparador de presupuestos: permite comparar, contextualizar y analizar el presupuesto de entidades equivalentes (un conjunto de municipios, por ejemplo).

El código se encuentra en el siguiente repositorio:

https://github.com/PopulateTools/gobierto

2.3. APIs web

Aunque el concepto de API y su utilización es anterior a la popularización de Internet, con la aparición de éste se vio la posibilidad de que distintas aplicaciones que residían en sistemas remotos interactuasen entre ellas a través de una API. Estos servicios de interacción entre sistemas a través de internet recibieron el nombre de ‘servicios web’. Posteriormente y, a raíz de la popularidad del protocolo HTTP, surgieron los servicios web denominados REST y de allí las API REST.

Que un determinado software ofrezca una API posibilita el desarrollo de otros productos y servicios que se apoyen en el software original, extendiendo su funcionalidad y muchas veces llevando sus servicios a plataformas y sistemas a los que los desarrolladores del software original no tienen la capacidad de llegar.

En este apartado se entra en detalle en las características de las APIs basadas en REST y GraphQL, temáticas alrededor de las cuales se ha desarrollado este TFG.

2.3.1. REST

REST, acrónimo de *Representational State Transfer*, es un estilo arquitectónico para aplicaciones web.

En una aplicación que sigue el estilo REST, los modelos de la aplicación están organizados en una abstracción denominada *recursos*. Cada recurso tiene una URL asociada, también llamada *endpoint*. Mediante los distintos verbos del protocolo HTTP es posible hacer peticiones a estos endpoint, resultando en la obtención o modificación de estos recursos.

El resultado concreto de estas peticiones está estandarizado, y depende fundamentalmente del verbo HTTP utilizado:

- `HTTP GET` - destinado a obtener recursos existentes.
- `HTTP POST` - destinado a crear nuevos recursos.
- `HTTP PUT` - destinado a reemplazar un recurso existente.
- `HTTP PATCH` - es similar a PUT en cuanto a que modifica un recurso existente, pero a diferencia de PUT, actualiza únicamente un subconjunto de las propiedades del recurso en vez de la totalidad de éste. No obstante es habitual utilizar PATCH o PUT indistintamente.
- `HTTP DELETE` - destinado a eliminar recursos existentes.

Los servicios web que siguen el estándar REST a menudo reciben el nombre de servicios RESTful. Un cliente y un servidor que realizan una comunicación de tipo RESTful no tienen por qué estar escritos en el mismo lenguaje o estar instalados en el mismo sistema operativo.

El estilo arquitectónico de REST define seis restricciones que deben cumplirse para que un sistema pueda ser considerado REST y aprovechar las características que éste ofrece: rendimiento, escalabilidad, simplicidad, modificabilidad, visibilidad, portabilidad y confiabilidad.

Las restricciones son las siguientes:

- **Interfaz uniforme**

  Esta restricción define la forma que debe tomar la interfaz que debe mantenerse entre cliente y servidor. Su uso simplifica y desacopla la arquitectura, lo que permite a cada una de las partes evolucionar de forma independiente. Los cuatro principios que marcan el diseño de esta interfaz son:

- *Basada en recursos* - Para distinguir cada uno de los recursos disponibles en una petición se hace uso de URIs. Un recurso no está ligado a una representación en particular, sino que el cliente tiene el poder de especificar que formato desea (HTML, JSON, XML, etc.).

- *Manipulación de los recursos a través de las representaciones* - Un cliente que posee una representación de un recurso, incluyendo los metadatos asociados, posee la información necesaria para modificar o eliminar dicho recurso del servidor, siempre y cuando tenga los permisos para hacerlo.

- *Mensajes auto descriptivos* - Cada mensaje incluye información suficiente acerca de cómo procesar dicho mensaje. Por ejemplo, el *parser* a utilizar puede ser indicado mediante un *MIME type*. Las respuestas también indican explícitamente si pueden ser cacheadas o no.

- *Hipermedia como motor del estado de la aplicación (HATEOAS)* - Un cliente envía estado como parte del *body*, de los parámetros del *query string*, los *headers* y la URI. Un servidor envía estado a un cliente mediante el *body*, los códigos de respuesta y los *headers*. Además es posible que parte de la información devuelta por el servidor venga en forma de identificadores únicos en forma de hipervínculos a otros recursos asociados.

  A estas diferentes alternativas de enviar la información se las conoce como hipermedia, y permiten a un cliente REST interactuar con cualquier aplicación o servidor sin requerir ningún tipo de conocimiento más allá de cómo trabajar con las distintas formas de hipermedia.

- **Sin estado**

  Esta restricción define que todo el estado necesario para responder a una petición debería estar autocontenido en la propia petición, ya sea como parte de la URI, en los parámetros del *query-string*, en el *body* o en las cabeceras.

  El uso de un protocolo sin estado permite una mayor escalabilidad, pues el servidor ya no tiene que mantener, actualizar o comunicar el estado de la sesión actual.

- **Cacheable**

  Un cliente puede cachear respuestas. Las respuestas tienen la responsabilidad de, implícita o explícitamente, definirse como cacheables o no con el objetivo de evitar que un cliente reutilice datos que han quedado obsoletos y que no deberían volver a ser utilizados en peticiones sucesivas.

- **Contrato cliente-servidor**

  La *interfaz uniforme* que separa a clientes de servidores permite, por ejemplo, que un cliente no tenga que preocuparse por el almacenamiento de datos o que un servidor quede liberado de aspectos como la representación de datos en una interfaz de usuario. Esta práctica mejora la portabilidad del código de ambas partes, de modo que pueden ser reemplazados o desarrollados de forma independiente siempre y cuando la interfaz no sea alterada.

- **Sistema basado en capas**

  Un cliente no tiene la capacidad de discernir si se encuentra conectado al servidor final o a uno situado en un punto intermedio del camino. Es frecuente hacer uso de servidores intermedios que actúan como balanceadores de carga o como cachés con el objetivo de mejorar la escalabilidad del sistema.

2.3.2. GraphQL

GraphQL (*http://graphql.org*) es un lenguaje de consultas para APIs creado por Facebook. Un servidor que implemente una API en GraphQL debe mantener un *esquema GraphQL*, de ahora en adelante ***GraphQL schema***. Este esquema, escrito normalmente de forma declarativa, especifica que información un cliente es capaz de obtener del servidor.

GraphQL como tal es una *especificación*, por lo que puede ser usado desde cualquier plataforma o lenguaje. Existe una implementación de referencia escrita en JavaScript y mantenida por Facebook, pero existen otras muchas implementaciones en otros lenguajes mantenidas por la comunidad, como es el caso de la implementación en Ruby (*https://rmosolgo.github.io/graphql-ruby*), que es la que se ha usado en este proyecto.

Inicialmente, la forma en que REST planteaba el diseño de una API, con su concepción CRUD basada en resources, verbos y códigos de respuesta HTTP era efectiva. No obstante, la necesidad de avanzar más rápido en productos cada vez más complejos, más allá de un simple CRUD, ha empujado un cambio en la forma en que interactuamos con las APIs que ha expuesto las limitaciones y la falta de flexibilidad de las APIs basadas en REST.

2.3.3. Los problemas de REST

El motivo por el que en Facebook se empezó a desarrollar GraphQL tiene que ver con los problemas de rendimiento que presentaban sus aplicaciones móviles de iOS y Android a la hora de cargar el *feed* de actividad de un usuario. Otras compañías como GitHub, Pinterest y Shopify ya han empezado a adoptar esta tecnología.

Las debilidades de REST que GraphQL soluciona son las siguientes:

- En RESTful utilizamos una URI para leer o escribir un único recurso, ya sea, por ejemplo, un producto, una persona, un post o un comentario. No obstante es habitual que en la práctica tengamos que trabajar con múltiples recursos de forma *simultánea*, teniendo que hacer peticiones a varios *endpoints* y encadenar distintas llamadas, para que posteriormente nuestra aplicación combine toda la información obtenida de la forma adecuada.

- Cuando hacemos una petición HTTP a una API REST, normalmente no recibimos únicamente la información que necesitamos, sino todo el conjunto de datos relativos al recurso solicitado que está alojado en la URI de nuestra petición. Por ejemplo, si solo queremos obtener el nombre de una persona al hacer la consulta recibiremos datos como, por ejemplo, su fecha nacimiento o su profesión. El recibir y tener que gestionar información innecesaria desde el lado del cliente realentiza, complica y entorpece el proceso de obtención de datos.

- El versionado de una API REST no es trivial. En muchas ocasiones necesitamos añadir nuevos campos o modificar el tipo de alguno de ellos pero para poder dar soporte de retrocompatibilidad los incluimos en el payload que los clientes viejos, lo cual hace crecer el payload de respuesta innecesariamente. O bajo otra estrategia, desde el lado del servidor gestionamos "vistas distintas" según la versión del cliente que consume el recurso. A veces eso nos lleva a innumerables if, código espagueti que poluciona el código original de respuesta.

Por este motivo GraphQL se está convirtiendo en una tecnología especialmente apetecible para el desarrollo de APIs consumidas desde dispositivos móviles, que están especialmente expuestos a menores anchos de banda (si no hay cobertura, wifi, etc.) y menor capacidad de procesamiento como para sobrecargar el dispositivo con información que no es necesaria.

Este apartado está fuertemente inspirado en el contenido de este blogpost: *https://www.genbetadev.com/desarrollo-aplicaciones-moviles/por-que-deberiamos-abandonar-rest-y-empezar-a-usar-graphql-en-nuestras-apis*

2.3.4. APIs GraphQL

Unas de las principales características de GraphQL es que el lenguaje y sintaxis usado en la request es el mismo que el de la respuesta. Analizando el JSON podemos comprender claramente el diccionario de key-value.

Otro de sus principales rasgos es que un único endpoint puede manejar sin problemas todas las peticiones de los clientes. Todas ellas puede preguntar sobre múltiples recursos y campos, tan sólo indicando qué necesitan y recibiendo exactamente la información solicitada. El diseño GraphQL permite obtener jerarquías de objetos para componer la vista que hace uso de ellos realizando una única petición HTTP, eliminando multitud de viajes de datos entre cliente y servidor.

Un ejemplo de *query* en GraphQL, en la que estamos solicitando información que en una API REST implicaría normalmente hacer peticiones a los endpoints de distintos recursos (propuestas, usuarios, geozonas y comentarios), es esta:

```
{
  proposal(id: 20) {
    id,
    title,
    public_author {
      username
    }
    geozone {
      name
    },
    comments {
      body
    }
  }
}
```

La información devuelta por el servidor para la *query* anterior podría tener este aspecto:

```
{
  "data": {
    "proposal": {
      "id": 20,
      "title": "Carril bici en San Fco. Sales",
      "public_author": {
        "username": "Alberto Miedes"
      },
      "geozone": {
        "name": "Chamberí"
      },
      "comments":[
        {
          "body": "¡Qué buena idea!",
        },
        {
          "body": "Si, la verdad es que hace falta!”,
        }
      ]
    }
  }
}
```

Este apartado está fuertemente inspirado en el contenido de este blogpost: *https://www.genbetadev.com/desarrollo-aplicaciones-moviles/por-que-deberiamos-abandonar-rest-y-empezar-a-usar-graphql-en-nuestras-apis*

2.4. Testing

2.4.1. Introducción

Uno de los retos dentro de los proyectos de software es conseguir automatizar las tareas mecánicas y repetitivas para que, en lugar de tener que ser ejecutadas por una persona, puedan ser ejecutadas por una máquina (el ordenador en este caso).

Tradicionalmente han existido equipos de *testers* o de *QA* encargados de testear el comportamiento de un programa informático. Los miembros de estos equipos recibían una serie de documentos con los casos de uso que debía soportar la aplicación y, uno a uno, iban siguiendo las instrucciones de estos documentos para comprobar que el resultado era el esperado. Si en algún detectaban un fallo, lo comunicaban al equipo de desarrollo para que lo corrigiese. Una vez solucionado, volvían a ejecutar los *tests* de nuevo.

El objetivo que se persigue con el testing, o mejor dicho, con el testing automático, es liberar a las personas de ejecutar esta serie de tareas de comprobación que de otro modo serían lentas y aburridas. Con el testing automático es posible delegar esta labor en una máquina y que las personas se dediquen a las labores más creativas, como puede ser el desarrollo de nuevas funcionalidades dentro de la aplicación.

Los tests automáticos también son código, y debe ser escrito por los desarrolladores. Concretamente los tests son trozos de código que no tienen tests. En los test se escriben una serie instrucciones que exponen a la aplicación a una serie de entradas y comparan las salidas obtenidas con las esperadas, para asegurarse de que el comportamiento de la aplicación es el esperado.

2.4.2. La Pirámide de Testing

La *Pirámide de Testing* es un modelo utilizado para describir los distintos tipos de tests automáticos que es frecuente tener en un proyecto de software. Dentro de esta pirámide, los tests están agrupados en una serie de niveles. Cada nivel utiliza técnicas propias y persigue objetivos diferentes.

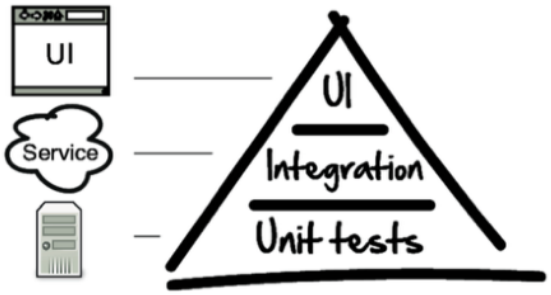
El motivo por el que están divididos en estos niveles está relacionado con una de las arquitecturas más típicas de las aplicaciones web. En esta arquitectura el código se organiza en tres capas distintas:

- *Capa de interfaz de usuario* - Es la capa que contiene los botones y controles que los usuarios utilizan para interactuar con la aplicación.

- *Capa de servicios* - Es la capa encargada de proporcionar datos a la interfaz de usuario, y traducir las acciones llevadas a cabo en la interfaz de usuario en acciones concretas a nivel de código.

- *Capa de lógica de negocio* - Contiene los cálculos, reglas etc. de nuestra aplicación.

El término *Pirámide de Testing* fue utilizado por primera vez por Mike Cohn en el libro *Succeeding with Agile*, y divide los distintos tipos de tests automáticos en tres niveles fundamentales:



Fuente: The Way of the Web Tester, Jonathan Rasmusson

Tests de Interfaces de Usuario

Los test de interfaces de usuario se sitúan en el nivel más alto de la pirámide y se caracterizan porque cuando se lanzan, provocan la ejecución de líneas de código pertenecientes a todos los niveles de la aplicación: desde código destinado únicamente a manipular la apariencia de la interfaz de usuario, pasando por los controladores, la lógica de negocio y la base de datos de la aplicación.

Son capaces de testear las interacciones entre multitud de componentes de la aplicación. Su principal desventaja es que son *muy lentos*.

Tests de Integración

Son similares a los tests de interfaces de usuario en cuanto a que atraviesan varios niveles de la aplicación, pero no suelen tocar componentes exclusivos de la interfaz de usuario. En el caso concreto de las aplicaciones web, es habitual que el tipo de *entradas* y de *salidas* que se comparan en este tipo de tests se sitúe en el nivel de las llamadas de red (llamadas HTTP, normalmente).

En este tipo de test se suele comprobar cómo interactúan distintos componentes de la aplicación entre sí.

Tests Unitarios

Por último y en el nivel más bajo, se encuentran los tests unitarios. Están destinados a testear pequeños componentes de la aplicación de forma aislada. Las principales ventajas de este tipo de tests es que, además de ser cortos en extensión, son *muy rápidos* a la hora de ejecutarse.

Conclusión

- No todos los tipos de test automáticos son iguales. Todos tienen sus puntos fuertes y débiles, y debemos saber en qué situaciones es mejor utilizar cada uno de ellos.

- Una de las principales características de una buena suite de tests es la velocidad con la que es capaz de ejecutarse. Encontrar el equilibrio entre tests de interfaz de usuario y tests unitarios resulta muy importante para mantener el tiempo de ejecución lo más bajo posible.

3. Tecnologías del Proyecto

Para el desarrollar este proyecto he tenido que aprender a manejar multitud de herramientas y librerías que eran totalmente nuevas para mí, así como profundizar en otras que ya conocía.

La mayoría de las tecnologías que he tenido que utilizar venían determinadas por el propio proyecto de Consul. La gran excepción serían dos de las librerías utilizadas en Ruby para hacer APIs REST o GraphQL: `active-model-serializers` y `graphql-ruby`. En este caso tuve que probar ambas por mi cuenta para evaluar cuál de las dos alternativas resultaba más apropiada para el proyecto.

3.1. Control de versiones: Git y GitHub

Git es un sistema de control de versiones distribuido. Su rapidez es un factor diferenciador sobre los sistemas de control de versiones anteriores como Subversion, CVS o Perforce.

La rapidez con la que se crean ramas *(branches)* en git favorece su uso para mantener varios *workflows* simultáneos en un mismo proyecto. Este sistema de creación de ramas a menudo recibe el nombre de *cheap branching*.

GitHub es una plataforma web que permite alojar proyectos de software utilizando Git como sistema de control de versiones. Aparte de alojamiento de proyectos en repositorios tanto públicos como privados, GitHub incorpora una serie de herramientas adicionales que facilitan enormemente el desarrollo de proyectos de software de forma colaborativa, como Wikis, un sistema de tickets (issues), estadísticas, espacios de discusión, tablones de gestión de tareas y proyectos, mecanismos de revisión de código y provisión de feedback, e integración con herramientas de análisis estático de código y sistemas de integración continua.

Debido a que se espera que la API resulte de utilidad para cualquiera de los *forks* de Consul, el desarrollo se ha llevado a cabo sobre la versión genérica de Consul, no la específica del Ayuntamiento de Madrid.

El repositorio con el código de Consul es accesible desde la siguiente dirección:

https://github.com/consul/consul

3.2. Ruby

Ruby (*https://www.ruby-lang.org/en*) es un lenguaje de programación interpretado, reflexivo y orientado a objetos. Es de código abierto y está enfocado en la simplicidad y productividad. Destaca por su elegante sintaxis y su facilidad de lectura.

Fue creado por Yukihiro Matsumoto, más conocido como "*Matz*". Su primera versión data de 1995.

3.3. Ruby on Rails

Ruby on Rails (*http://rubyonrails.org*) es un framework para aplicaciones web MVC. Su principal creador es David Heinemeier Hansson y tiene su origen en sus trabajos realizados dentro de la empresa Basecamp, anteriormente conocida como 37signals.

La primera versión de Ruby on Rails fue publicada en 2005. Ruby on Rails también es el principal responsable del aumento de popularidad de Ruby, prácticamente desconocido antes de esta fecha.

Es de código abierto y su filosofía está íntimamente relacionada con la del lenguaje de programación en el que está escrito: Ruby. Ruby on Rails está fuertemente inspirado por una serie de principios bajo los que subyacen a su vez una serie de buenas prácticas (opinable). Algunos de los principios más conocidos son:

Optimize for programmer happiness

Tanto Ruby como Ruby on Rails destacan por haber sido desarrollados con el objetivo de facilitar la vida al programador de un modo que otros lenguajes no hacen. Esto se ve reflejado en una sintaxis muy legible, y en una disminución *aparente* de la complejidad del código. Existe una enorme cantidad de azúcar sintáctico destinado a hacer el código más legible.

Un ejemplo típico son los métodos que añade Rails a la clase `Array` de la librería estándar de Ruby para poder acceder a sus elementos:

```
wadus = ['a', 'b', 'c']
wadus.first  # => 'a'
wadus.second # => 'b'
wadus.third  # => 'c'
```

Otro ejemplo es la forma en la que se crean los *getters* y los *setters* de un objeto en Ruby: mediante una llamada al método `attr_accessor` y pasando como parámetros los nombres de los atributos objetivo, se añaden nuevos métodos a la clase en tiempo de ejecución destinados a leer y modificar dichos atributos:

```
class Wadus
  attr_accessor(:foo)
end

wadus = Wadus.new
wadus.foo = 'bar'
wadus.foo # => 'bar'
```

Todas estas estrategias utilizadas para hacer el código más comprensible presentan una importante contrapartida: se añade una importante sobrecarga que provoca que la ejecución del código sea mucho más lenta que en otros lenguajes que no aplican esta serie de técnicas.

Convention over Configuration

Este principio se basa en que, estableciendo una serie de convenios de antemano, es posible liberar a los desarrolladores de tener que tomar una serie de decisiones que no tienen excesiva relevancia en la calidad del producto pero que el simple hecho de tener que tomarlas supone un gasto importante de tiempo.

Un ejemplo muy representativo es el funcionamiento de la librería de ORM *(Object Relational Mapping)* de Rails, `ActiveRecord`.

Supongamos que queremos crear una clase *Persona* con los atributos *nombre* y *dni*. En ORMs como Hibernate sería necesario especificar a qué tabla y columna de la base de datos se mapean cada uno de los campos de nuestra clase, y el tipo de cada uno de ellos. En Ruby on Rails basta con hacer:

```
class Person < ActiveRecord::Base
end
```

Y ya podríamos ejecutar líneas del estilo:

```
Person.create(dni: '42', nombre: 'Bob')
Person.first.id  # => 1
Person.first.dni # => '42'
```

Realmente lo que ha sucedido por debajo para que este procedimiento resulte tan sencillo es lo siguiente:

- Por convenio, la tabla de la base de datos de una clase se nombre igual que la clase, pero en minúsculas y pluralizada. En este caso, un objeto de la clase `Person` se guardaría en la tabla `people`. Rails dispone de mecanismos para averiguar la versión pluralizada de una palabra.

- Los atributos de la clase se leen del fichero `schema.rb`, que contiene el esquema de la base de datos. Automáticamente se crean *getters* y *setters* para cada una de las columnas.

- Por defecto, Rails crea un campo `id` en todas las tablas de la base de datos, y que actúa como clave primaria.

Don't Repeat Yourself (DRY)

Este principio está relacionado con la reducción de la duplicación en el código, y promueve que toda "pieza de conocimiento" de un programa debería estar expresada en un único lugar. Cuando determinada componente de un programa aparece más de una vez, se incrementa el nivel de dificultad para realizar una posterior evolución o adaptación del código, que perjudica la claridad y que aumenta la posibilidad de crear inconsistencias entre cada una de estas "piezas repetidas".

3.4. PostgreSQL

PostgreSQL es un sistema de gestión de base de datos relacional. Es el fruto de más de 15 años de desarrollo y su arquitectura se ha ganado una buena reputación por su estabilidad e integridad de datos. Es *ACID compliant* y soporta completamente claves foráneas, joins, vistas, triggers y procedimientos almacenados. También proporciona soporte para el almacenamiento de grandes datos binarios, como imágenes, sonido y video.

Consul trabaja con versiones de PostgreSQL iguales o superiores a la 9.4. No obstante, para el desarrollo de este proyecto no han sido necesarios conocimientos específicos de PostgreSQL pues las operaciones que ha habido que realizar con la base de datos han sido agnósticas en cuanto al sistema de bases de datos concreto.

3.5. RSpec

Minitest y *RSpec* son las dos librería de testing más populares dentro del ecosistema de Ruby. Aunque Minitest forma parte de la Librería Estándar de Ruby, RSpec se ha vuelto la más popular de todas.

La sintaxis utilizada en RSpec para escribir tests hace especial énfasis en que estos especifiquen comportamiento, no implementación. Esta sintaxis especial hace que la curva de aprendizaje de RSpec sea más elevada que la de Minitest, que utiliza una sintaxis más directa.

Ejemplo:

```
class Adder
  def self.add(first, second)
    first + second
  end
end
```

Un test en RSpec para la clase `Adder` tendría el siguiente aspecto:

```
describe Adder do

  describe '::add' do
    it 'adds two numbers' do
      result = Adder.add(3, 5)
      expect(result).to eq(8)
    end
  end

end
```

Y en Minitest:

```
class AdderTest < ActiveSupport::TestCase

  def test_add
    result = Adder.add(3, 5)
    assert_equal 8, result
  end

end
```

En Consul la librería utilizada es RSpec. Además es un proyecto en el que se ha prestado especial atención a la cobertura del código proporcionada por los test, situada habitualmente en torno al 95% de las líneas de código.

Por este motivo ha sido imprescindible que todo el código escrito durante el desarrollo de este proyecto estuviese debidamente cubierto por sus respectivos tests.

Testing de aplicaciones Ruby on Rails

Ruby on Rails, por ser un framework *opinado*, establece una estructura de directorios predeterminada y que, a efectos de esta sección, puede resumirse en:

```
/app
  /controllers/*
  /models/*
  /views/*
  ...
/spec
  /controllers/*
  /features/*
  /models/*
  ...
```

Dentro del directorio `app` se encuentra el código principal de la aplicación. Dentro de él existen directorios para cada uno de los componentes del MVC: dentro del directorio `controllers` irán los controladores de nuestra aplicación, dentro de `models` los modelos, y así sucesivamente.

Al mismo nivel que `app` existe otro directorio llamado `test` o `spec`, que sigue una estructura *aproximadamente* similar a la de `app` y que contiene los test de la aplicación. La idea es que dentro de cada uno de los subdirectorios de `spec` se encuentren los test de los correspondientes ficheros de los subdirectorios de `app`, pero en la práctica es habitual encontrar desviaciones de esta regla:

- `/spec/models` contiene los test de los modelos de la aplicación. La mayoría de ellos podrían clasificarse como tests unitarios.

- `/spec/controllers` contiene los test de los controladores de la aplicación. Los tests que se suelen encontrar en `/spec/controller` están cerca de lo que podría ser un test unitario, no obstante es habitual testear el comportamiento de los controladores directamente en los tests de integración o de interfaces de usuario, en cuyo caso podrían estar alojados en directorios como `/spec/features` o `/spec/requests`.

- Las vistas de `/app/views` hacen uso de tests de interfaces de usuario, que suelen encontrarse en `/spec/features`.

3.6. Atom

Atom (*https://atom.io*) es un editor de texto desarrollado por GitHub. Es open source y destaca por su gran número de paquetes, temas y su alta capacidad de personalización.

3.7. REST y *ActiveModelSerializers*

REST es un estilo arquitectónico para aplicaciones web. Para más información sobre el funcionamiento de una aplicación que sigue un modelo REST, consultar la sección dentro de *Estado del Arte* dedicada a esta misma temática.

Los datos devueltos por una API REST a menudo hacen uso del formato JSON. Con el objetivo de facilitar la serialización de los modelos de una aplicación escrita en Ruby a formato JSON se creó la librería `active_model_serializers`. Dos de los componentes que esta librería pone a nuestra disposición son:

- *Serializers* (serializadores) - Describen qué atributos y relaciones de los modelos deben ser serializados.

- *Adapters* (adaptadores) - Describen cómo deberían serializarse los atributos y relaciones especificados por los serializadores.

Para más información sobre `active_model_serializers`, consultar el siguiente repositorio en GitHub:

https://github.com/rails-api/active_model_serializers

3.8. JSON API

JSON API es una especificación para construir APIs JSON. El estándar JSON API propone una serie de convenciones a seguir a la hora de serializar datos en formato JSON, liberándonos de tener debatir explícitamente sobre el formato concreto de los datos y que podamos centrarnos en el desarrollo de nuestra aplicación, aumentando la productividad.

La librería `active_model_serializers` dispone de un *adapter* capaz de serializar los datos de nuestra aplicación de acuerdo a este estándar.

Para más información sobre el estándar JSON API, consultar la web *http://jsonapi.org*

3.9. GraphQL y *graphql-ruby*

GraphQL (*http://graphql.org*) como tal es únicamente una especificación de un lenguaje para hacer consultas a APIs. Con el objetivo de convertir esta especificación en algo tangible, han aparecido una serie de librerías en distintos lenguajes (*http://graphql.org/code*) que dan cuerpo a este estándar en forma de implementación.

En el caso de Ruby, la librería es `graphql-ruby`. Los detalles sobre la utilización de esta librería se darán dentro de la sección de *Desarrollo*.

Para más información sobre la librería `graphql-ruby` consultar la web:

http://graphql-ruby.org

O el repositorio de GitHub:

https://github.com/rmosolgo/graphql-ruby

3.10. Librerías HTTP

Con el objetivo de probar el correcto funcionamiento de la API he tenido que ejecutar peticiones HTTP contra la aplicación para comprobar su correcto funcionamiento.

La librería estándar de Ruby ya proporciona una librería para realizar peticiones HTTP, `Net::HTTP`. No obstante la sintaxis de esta librería es muy verbosa y la documentación no es especialmente buena. Por este motivo me he decantado por la alternativa `http`.

Para más información sobre la librería `http` acudir al repositorio:

https://github.com/httprb/http

3.11. GraphiQL

GraphiQL es una aplicación interactiva para realizar consultas contra una API GraphQL. Una de sus funcionalidades más útiles es la capacidad de ir proporcionando retroalimentación sobre la corrección de un consulta GraphQL a medida que la estamos escribiendo. La otra es su capacidad de generar documentación sobre la API directamente a través de la información extraída del *GraphQL Schema*.

Para más información sobre GraphiQL consultar el repositorio:

https://github.com/graphql/graphiql

Además de la aplicación de escritorio, existe una versión que puede integrarse directamente dentro de una aplicación Ruby on Rails:

https://github.com/rmosolgo/graphiql-rails

4. Metodología

En esta sección primero se presenta a los actores externos relevantes durante el desarrollo del proyecto, el papel que ha desempeñado cada uno, las repercusiones que han tenido en la evolución del desarrollo y mis interacciones con cada uno de ellos.

Luego se hace una introducción del flujo de trabajo típico de proyectos alojados en GitHub, así como de algunas buenas prácticas asociadas a ellos.

4.1. Actores

Una de las características más importantes de este Trabajo de Fin de Grado es que el desarrollo no se ha efectuado de manera aislada, sino que ha sido necesario coordinarse con varias personas y organizaciones externas para obtener los resultados más óptimos posibles, tanto en lo que a la calidad del código se refiere como a requisitos a satisfacer.

Los principales actores que han intervenido en el desarrollo de este proyecto son:

4.1.1 Desarrolladores de Consul

El equipo de desarrollo de consul, que forma parte de la plantilla externa del ayuntamiento, está compuesto por cuatro personas:

- Enrique García Cota - desarrollador - *https://github.com/kikito*
- Raimond García - desarrollador - *https://github.com/voodoorai2000*
- Juanjo Bazán - desarrollador - *https://github.com/xuanxu*
- Alberto García Cabeza - diseñador - *https://github.com/decabeza*

Las aportaciones del equipo de desarrollo a este proyecto se centran fundamentalmente en estos dos puntos:

- Orientarme en la parte de definición y priorización de requisitos y funcionalidades de la API.
- Resolver las dudas de carácter técnico que me han surgido a lo largo de todo el proceso.

Mi interacción con ellos ha sido a través del correo electrónico y en las reuniones mensuales que se realizaban en el Medialab Prado.

4.1.2. Ayuntamiento de Madrid

Ha habido también varias interacciones con el personal del Ayuntamiento de Madrid responsable del desarrollo de Consul, en concreto con Miguel Arana (*https://transparenciapersonas.madrid.es/people/miguel-arana-catania*), uno de los responsables del *Área de Participación Ciudadana, Transparencia y Gobierto Abierto del Ayuntamiento de Madrid*.

En las interacciones con Miguel se ha hablado fundamentalmente sobre la definición de la información de la base de datos que debería ser accesible a través de la API.

Un tema frecuente e importante en estas reuniones ha sido acerca de la privacidad. La base de datos de Decide Madrid alberga información confidencial acerca de sus usuarios, como contraseña, fecha de nacimiento, género y distrito de residencia entre otros, así como toda la información que generan estos usuarios durante sus interacciones con la web: comentarios escritos, propuestas y debates creados, apoyos a propuestas, debates y comentarios, etc.

Por este motivo ha sido crítico debatir acerca de qué información acerca de estos usuarios era lícito poner a disposición del público. Para ello han participado en el proceso varias personas del mundo legal para dar asesoramiento en relación con la Ley de Protección de Datos.

4.1.3. Medialab Prado

De acuerdo con la definición que figura en su web (*http://medialab-prado.es*), el Medialab Prado es un laboratorio ciudadano de producción, investigación y difusión de proyectos culturales que explora las formas de experimentación y aprendizaje colaborativo que han surgido como consecuencia de la aparición de las redes digitales.

Una característica muy representativa del Medialab Prado es que los proyectos que se realizan en él mezclan las nuevas tecnologías con otras disciplinas totalmente diferentes. Por este motivo los equipos que participan en estos proyectos acostumbran a ser fuertemente multidisciplinares.

Medialab Prado pertenece a la empresa *Madrid Destino Cultura, Turismo y Negocio*, que depende a su vez del propio Ayuntamiento de Madrid.

Dentro del Medialab Prado existen varias líneas de acción, agrupadas bajo el nombre de "laboratorios". Uno de ellos es el *ParticipaLAB*, *"Laboratorio de Inteligencia Colectiva para la Participación Democrática"*, orientado al estudio, desarrollo y práctica de procesos de participación que puedan impulsar una democracia directa, deliberativa y distribuida.

Dentro de las actividades organizadas por el ParticipaLAB se encuentra el *Coding Madrid*. Durante este evento, organizado con frecuencia mensual, los desarrolladores de Consul se reúnen con los contribuyentes al código de Consul para resolverles las dudas que puedan tener. Durante el desarrollo de este proyecto he acudido a todos los Coding Madrid para que los desarrolladores me echasen una mano con las dudas relativas al desarrollo de la API. A parte de los desarrolladores también estaban presentes dos de los coordinadores del ParticipaLAB (*http://medialab-prado.es/article/equipo*): Pedro Álvarez Gonzalez y Yago Bermejo Abati, que han participado en el proceso de definición de requisitos de la API.

Otra de las líneas de trabajo que existen dentro del Medialab Prado es *Visualizar*, especializada en técnicas de visualización de datos, y que busca explotar el poder de la comunicación visual con el objetivo de explicar, de un modo comprensible, las relaciones, significado, causa y dependencias que es posible encontrar entre la enorme masa de información generada por los procesos tecnológicos, científicos y sociales.

El equipo de *Visualizar* está especialmente interesado en la información que puedan obtener a través de la API de Decide Madrid, pues podrían realizarse visualizaciones muy interesantes con estos datos.

4.2. Flujo de trabajo en GitHub

4.2.1. Sistema de branching

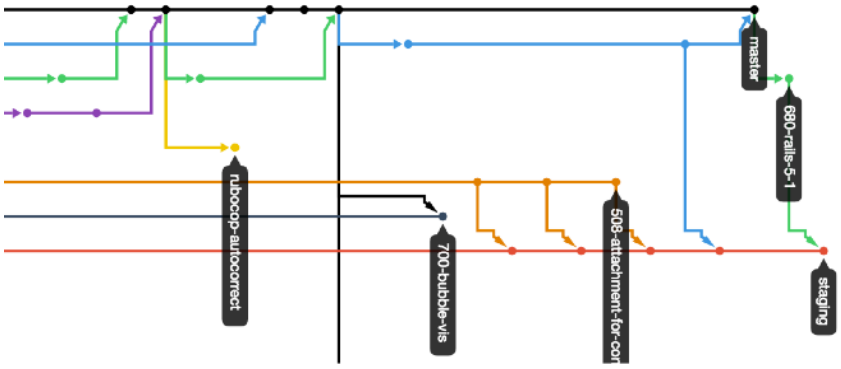
El sistema de *cheap branching* disponible en Git favorece que en los proyectos que utilizan este sistema de control de versiones se utilicen multitud de ramas (flujos de trabajo) con una jerarquía entre ellas en lo que a estabilidad del programa se refiere. De este modo, la versión desplegada del producto *("en producción")* debería estar en una rama totalmente estable y probada, mientras que los desarrollos de funcionalidades todavía incompletas o que no han pasado las pruebas pertinentes deberían permanecer en ramas distintas.

La siguiente imagen representa un diagrama con las distintas ramas presentes en un proyecto alojado en GitHub. En esta imagen puede verse:

- Una rama `master` - Contiene el código desplegado en producción.

- Una rama `staging` - Es la rama de *integración*. Antes de incorporar nuevo código a la rama de producción, es habitual hacer un simulacro de como se comportaría este nuevo código al interactuar con el resto de componentes de la aplicación. Esto se lleva a cabo en una rama auxiliar y, en el caso de las aplicaciones web, es habitual que este código también esté desplegado en un servidor y tenga datos de prueba dentro de la base de datos para poder probar el funcionamiento desde el navegador.

- Las ramas `700-bubble-vis`, `680-rails-5-1` y `508-attachment-for-content-block`, dedicadas al desarrollo funcionalidades concretas y que todavía tienen trabajo pendiente. A este tipo de ramas se las conoce bajo el nombre de *topic branches*.



Aunque es muy frecuente que en los proyectos de software existan los conceptos de *rama de producción*, *rama de integración* y *topic branches*, la forma de nombrar cada una de ellas puede variar. Por ejemplo, la rama de integración o staging también recibe el nombre de *preproducción*.

También es frecuente que el nombre de cada una de las topic branches contenga el número del *ticket / issue* relativo al problema que se está intentando resolver en ella.

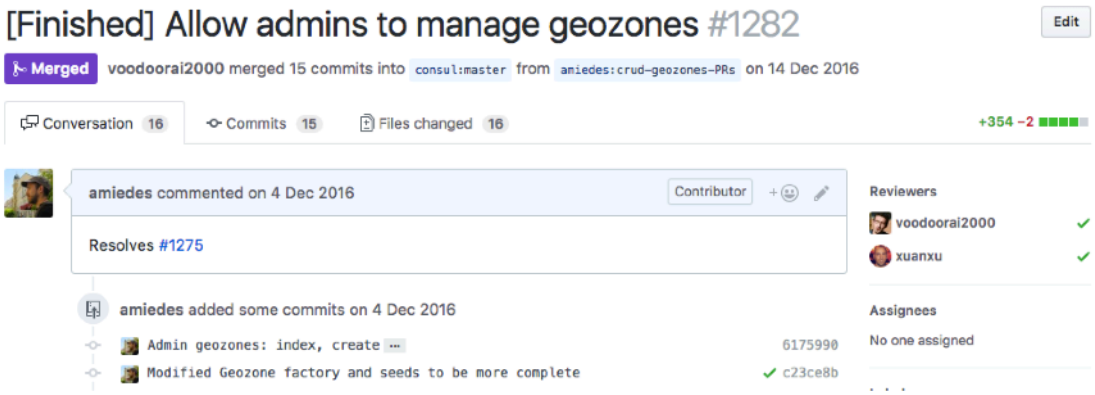
4.2.2. Sistema de Pull Requests

Una funcionalidad aportada por GitHub que se ha vuelto muy popular son las denominadas *Pull Requests* (o PRs).

Cuando un desarrollador considera oportuno incorporar el código de una de estas ramas más experimentales a otra de un nivel superior de la jerarquía, puede crear una pull request. Una pull request es una solicitud para incorporar este nuevo código a una rama distinta. Durante este proceso, otros desarrolladores del proyecto tienen la oportunidad de evaluar este código y las consecuencias de incorporarlo a la rama destino.

Normalmente se aprovechan las pull requests para ejecutar la batería de tests del proyecto, pudiendo detectar con antelación fallos derivados de la incorporación de este nuevo código.

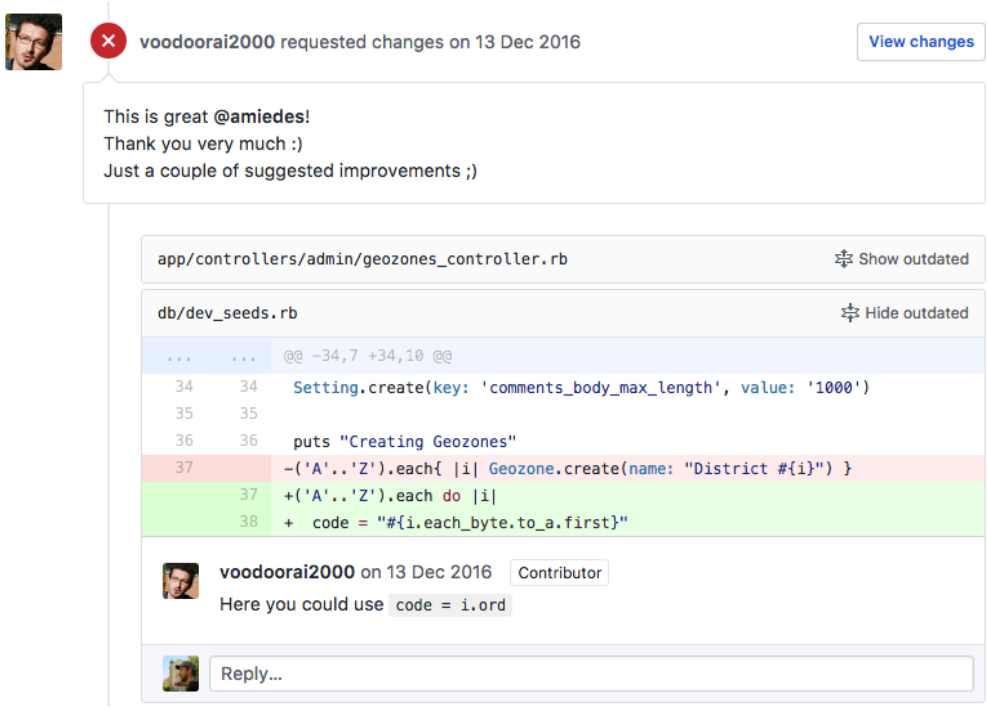
La siguiente imagen muestra una captura de una de mis primeras pull requests creadas en el repositorio de Consul. Dentro de esta ventana es posible ver una lista con los commits que componen la pull requests, los archivos y líneas de código que han cambiado, la conversación con otros desarrolladores en relación con esta pull request y los desarrolladores que han sido asignados la tarea de revisar este código.



4.2.3. Revisiones de Código

Dentro de una pull request otros desarrolladores pueden darte retroalimentación sobre el código escrito, sugiriendo modificaciones si fuese necesario.

También es posible añadir comentarios y empezar conversaciones sobre líneas concretas de código:



4.2.4. Integración Continua

La integración continua es una práctica de desarrollo de software mediante la cual los desarrolladores combinan los cambios en el código en un repositorio central de forma automática, periódica y frecuente, tras lo cual se ejecutan una serie de pruebas automáticas sobre dicho código.

Es habitual que estas pruebas se ejecuten varias veces al día, siendo frecuente hacerlo cada vez que alguien sube código nuevo al repositorio del proyecto.

A menudo la integración continua está asociada con las metodologías de programación extrema y desarrollo ágil.

Los principales objetivos que se persiguen con la integración continua son los siguientes:

- **Encontrar y arreglar errores con mayor rapidez.** El hecho de ejecutar las pruebas de forma frecuente y periódica permite a los desarrolladores descubrir y arreglar errores en el código de forma temprana, evitando que estos aparezcan a última hora.

- **Reducir el tiempo que se tarda en validar y publicar nuevas actualizaciones de software.** La integración continua le permite a su equipo entregar actualizaciones a los clientes con mayor rapidez y frecuencia.

- **Mejorar la productividad de desarrollo**, al liberar a los desarrolladores de determinadas tareas manuales y fomentar comportamientos que ayudan a reducir la cantidad de errores y bugs del producto.

Algunas aplicaciones destinadas a facilitar las prácticas de integración continua son Jenkins, CircleCi y Travis. Travis es la utilizada en Consul y una de las más populares para proyectos alojados en GitHub debido a su buena integración con el sistema de pull requests.

Las pruebas más frecuentes que se ejecutan durante la integración continua son los tests unitarios y de integración. No obstante no son las únicas y existen otras de gran utilidad como:

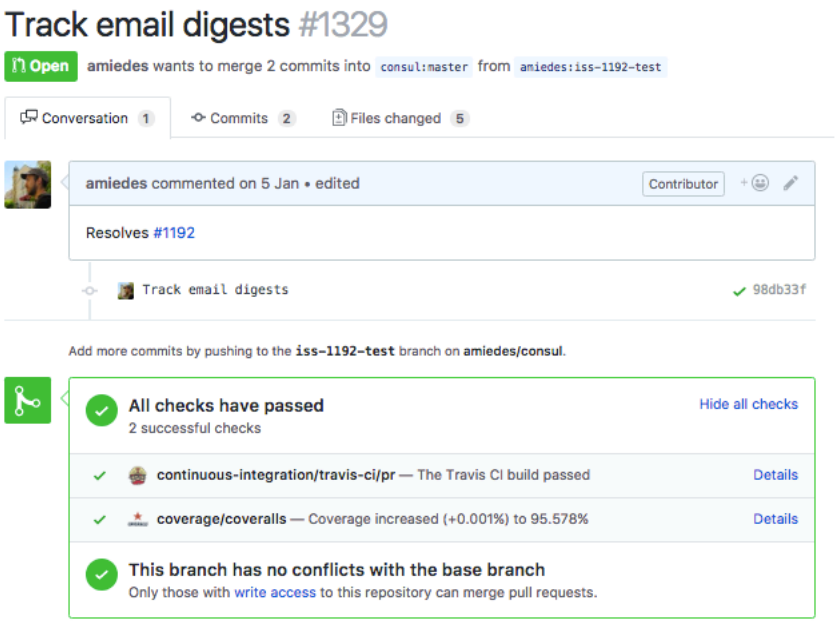
- Analizadores de la **cobertura del código** o *code coverage*. La cobertura del código de un programa se define como el porcentaje de líneas de código que son ejecutadas una o más veces durante la ejecución de la batería de tests.

  La situación ideal sería alcanzar el 100% de cobertura, pero una cifra más realista sería situarla en torno al 95%.

- Herramientas de **análisis estático del código**. El análisis estático es un tipo de prueba que se efectúa sin tener que llegar a ejecutar el código de la aplicación.

  En concreto dentro de Consul se utiliza *Code Climate*, que permite generar informes sobre la calidad del código de nuestra aplicación, teniendo en cuenta para ello distintas métricas relacionadas con la complejidad, seguridad, duplicación y estilo del código escrito, entre otros.

La siguiente imagen muestra la retroalimentación proporcionada por varias de las herramientas de integración continua utilizadas en Consul acerca de una de mis pull requests:



En concreto las pruebas que se han ejecutado en este caso han evaluado que la batería de tests se ejecutase correctamente y que la cobertura del código de la aplicación no disminuyese de forma excesiva.

5. Plan de Trabajo y Desarrollo

5.1. Plan de Trabajo

En cuanto al desarrollo ha sido difícil prever con antelación los desafíos que iban a surgir, pues era un área totalmente nueva para mí, además de que se ha hecho uso de librerías muy recientes, acerca de las cuales existía muy poca documentación al respecto en el momento de inicio de este proyecto. El resultado final han sido una serie de fases en las que se han planteado diferentes objetivos y requisitos a cumplir, se ha hecho el desarrollo, se ha recibido feedback y se han hecho las modificaciones pertinentes en el código, para posteriormente pasar a la fase siguiente.

Ha habido siete fases bien diferenciadas:

- **Preparación y toma de contacto**

  A pesar de que Consul es un proyecto Open Source, por lo que cualquier persona puede contribuir, se consideró adecuado reunirnos con personal del ayuntamiento para valorar la viabilidad de esta colaboración. En esta sección detallo las reuniones y encuentros que hubo previos al comienzo del desarrollo.

- **Aprendizaje previo**

  Aunque el desarrollo de la API propiamente dicho empezó en septiembre de 2016, los meses anteriores aproveché para ir familiarizándome con las tecnologías utilizadas en Consul. En esta sección nombro algunos de los recursos de aprendizaje en los que me apoyé.

- **Primer prototipo: REST vs. GraphQL**

  Evalué dos tecnologías distintas con las que era posible realizar la API: REST y GraphQL. Hice un pequeño prototipo en cada una de ellas, recopilé información e hice un balance para ver cuál resultaba más ventajosa. Cuando GraphQL resultó el candidato ganador, empecé a trabajar en una versión más extensa de la API utilizando esta tecnología.

- **Mejoras en la mantenibilidad del código**

  En la segunda etapa tuve que reescribir la mayor parte del código con el objetivo de hacerlo menos extenso y más mantenible. Para ello tuve que utilizar algunas técnicas de metaprogramación.

- **Testing**

  Durante esta fase escribí los tests automáticos encargados de probar el correcto funcionamiento de la API. Fue probablemente la más dura de todas.

- **Privacidad y seguridad**

  A medida que los requisitos de la API se fueron concretando, sobre todo en lo que respecta a la información que se iba a exponer al exterior, empezaron a surgir una serie de tareas relacionadas con la seguridad, privacidad y mantenibilidad del código que fue necesario tratar.

- **Pruebas en preproducción y despliegue a producción.**

  Durante esta fase los desarrolladores de Consul probaron la API en el entorno de preproducción de Decide Madrid y me dieron un pequeño feedback que tuve que incorporar a la aplicación.

En esta sección se presentan multitud de fragmentos de código con el objetivo de dar una idea más tangible de las acciones concretas que se han llevado a cabo. No obstante es importante tener en cuenta los siguientes puntos:

- No todo el código escrito durante el desarrollo de este proyecto aparece reflejado en esta memoria. Para obtener una copia del código real lo mejor es hacerlo consultando la pull request abierta en Consul respecto a la API: *https://github.com/consul/consul/pull/1257*

- No todo el código escrito fue comiteado a GitHub. A menudo antes de escribir funcionalidades he tenido que practicar en pequeños proyectos personales de forma paralela, con el único objetivo de aprender.

- Los fragmentos presentes a menudo han sido simplificados con el objetivo de facilitar su comprensión. Por este motivo pueden ser erróneos o contener comportamiento no deseable en el sistema real.

5.2. Desarrollo

5.2.1. Preparación y toma de contacto

El 4 de febrero nos reunimos en las oficinas del *Área de Participación Ciudadana, Transparencia y Gobierno Abierto* del Ayuntamiento de Madrid con Miguel Arana para presentarle mi intención de colaborar en su desarrollo como parte de mi Trabajo de Fin de Grado. Él nos hizo una pequeña introducción sobre el proyecto para luego comentar las distintas áreas que estaban en desarrollo en ese momento, o presentes en el *roadmap* para ver en cuáles de ellas me apetecía y me veía capaz de colaborar.

También nos nombró que una vez al mes, se organizaban en el Medialab Prado unas reuniones conocidas con el nombre de *Coding Madrid* en la que los desarrolladores de Consul se reunían con los contribuyentes para resolver dudas acerca del proyecto y ayudar a la gente a participar en el desarrollo. Sabido esto, el 10 de marzo acudí al Medialab Prado para conocer a los desarrolladores, presentarme y comentarles mi intención en participar en el desarrollo.

El 14 de abril volví a acudir al *Coding Madrid*, pero esta vez acompañado de Sara y Antonio, directores del TFG, para plantear más seriamente esta colaboración. Tras charlar un rato con ellos quedamos en que yo iría ampliando mis conocimientos de Ruby y Rails hasta el comienzo del proyecto, y que ellos pensarían en un área del proyecto adecuada para mí.

A finales de junio de 2016 Enrique se puso en contacto conmigo para proponerme realizar la API de Consul. Fue una propuesta muy acertada, pues además de ser una temática muy interesante para mí, era una parte de la aplicación que estaba bastante desacoplada de otras secciones de la aplicación en las que el desarrollo era más frenético, así que iba a ser más sencillo de cara a que no apareciesen conflictos en el código y al poder trabajar de un modo más independiente.

Aceptada esta proposición, detuve el desarrollo del TFG hasta septiembre, que fue cuando retomé el proyecto.

5.2.2. Aprendizaje previo

Debido a que iba a tener que enfrentarme a una cantidad considerable de herramientas nuevas, decidí empezar a investigar sobre Ruby y Ruby on Rails con mucha antelación (Navidades de 2015/2016), de modo que para cuando el Trabajo de Fin de Grado comenzase de forma oficial (septiembre de 2016), ya tuviese una base de conocimientos para poder empezar a trabajar directamente en el objetivo del proyecto.

Algunos de los recursos de los que hice uso durante este tiempo de preparación para aprender Ruby y Ruby on Rails fueron:

- **Programming Ruby 1.9 & 2.0** - Dave Thomas, Chad Fowler, Andy Hunt - *https://pragprog.com/book/ruby4/programming-ruby-1-9-2-0*

- **Agile Development with Rails 4** - Sam Ruby, Dave Thomas, David Heinemeier Hansson - *https://pragprog.com/book/rails4/agile-web-development-with-rails-4*

- **Rails, Angular, Postgres and Bootstrap** - David Bryant Copeland- *https://pragprog.com/book/dcbang2/rails-angular-postgres-and-bootstrap-second-edition*

- **Rails 4 Test Prescriptions** - Noel Rappin - *https://pragprog.com/book/nrtest2/rails-4-test-prescriptions*

- **Agile Development Using Ruby on Rails - Basics** - Curso online de edX por la U.C. Berkeley - *https://www.edx.org/course/agile-development-using-ruby-rails-uc-berkeleyx-cs169-1x-0*

- **Agile Development Using Ruby on Rails - Advanced** - Curso online de edX por la U.C. Berkeley - *https://www.edx.org/course/agile-development-using-ruby-rails-uc-berkeleyx-cs169-2x-0*

- Cursos online sobre Ruby y Rails en CodeSchool - *https://www.codeschool.com/learn/ruby*

También tuve que mejorar mis conocimientos de *git*, especialmente en lo que a trabajar con repositorios remotos se refiere. Para ello acudí al siguiente libro:

- **Pro Git** - Scott Chacon, Ben Straub - *https://git-scm.com/book/en/v2*

5.2.3. Primer prototipo: REST vs. GraphQL

Retomado el desarrollo del proyecto a principios de septiembre, una de las primeras tareas que me encargaron los desarrolladores de Consul fue realizar dos prototipos de la API utilizando dos tecnologías que estaban barajando: REST y GraphQL, a modo de análisis exploratorio para poder hacer una comparativa entre ambas.

A priori, las ventajas y desventajas de cada una eran las siguientes:

- Las APIs REST eran un terreno muy explorado, con librerías que facilitaban su implementación muy maduras y con mucho contenido disponible en Internet acerca de sus ventajas, desventajas y recomendaciones a la hora de implementarlas.

- GraphQL era una tecnología muy novedosa. Cuando se me propuso investigar acerca de ella, el documento con la especificación tenía aproximadamente un año de antigüedad y la primera y única librería para trabajar con GraphQL en Ruby tenía unos pocos meses de vida, y la documentación era escasa. No obstante Facebook llevaba utilizando esta tecnología desde 2011 y GitHub había liberado recientemente una versión *early access* de su nueva API GraphQL.

Para evaluar cada una de las alternativas, a parte de buscar información en Internet sobre experiencias que habían tenido otras personas trabajando con cada uno de estos dos paradigmas (fundamentalmente en blogpost escritos por trabajadores de empresas de software que habían estado experimentando con GraphQL), implementé una versión muy sencilla de una API para Consul que permitía extraer un subconjunto muy limitado de la información presente en la base de datos.

Para ver una comparativa más exhaustiva entre REST y GraphQL, consultar la sección dedicada a ello en *Estado del Arte*.

Prototipo REST

Del 9 al 26 de septiembre estuve trabajando en un prototipo de API REST. Esta versión permitía obtener propuestas y comentarios de la aplicación.

Constaba de dos endpoints:

- `decide.madrid.es/api/v1/proposals/:id`
- `decide.madrid.es/api/v1/comments/:id`

El *id* en ambos casos era opcional y permitía recuperar un único elemento del endpoint en lugar de la colección completa.

En esta versión hice uso de la librería `active-model-serializers` (*https://github.com/rails-api/active_model_serializers*), que facilita enormemente la tarea de serializar los modelos de la aplicación a formato JSON.

Una de las tareas que tuve que realizar fue crear un controlador adicional para cada uno de los recursos expuestos en la API (`Proposal` y `Comment` en este caso), con un método para cada una de las "variantes" de las peticiones que es posible efectuar contra ese endpoint: `index`, que devuelve la colección entera y `show`, que devuelve un objeto en particular.

Esta es una versión simplificada del controlador de propuestas:

```ruby
# app/controllers/api/v1/proposals_controller.rb
class Api::V1::ProposalsController < ApplicationController
  before_action :authenticate_user!, except: [:index, :show]

  authorize_resource

  def index
    proposals = Proposal.all.page(params[:page])
    render(
      json: proposals,
      each_serializer: Api::V1::Proposals::IndexSerializer
    )
  end

  def show
    proposal = Proposal.find(params[:id])
    render(
      json: proposal,
      serializer: Api::V1::Proposals::ShowSerializer
    )
  end
end
```

También hay que crear un `Serializer` para cada una de las acciones de los controladores. En ellos se especifican los atributos y relaciones a incluir en la representación JSON del modelo. Es habitual hacer uso de la herencia para evitar repetir la lista de atributos en varios *serializers*.

```ruby
# app/serializers/api/v1/proposal_serializer.rb
class Api::V1::ProposalSerializer < ActiveModel::Serializer
  attributes :id, :title, :description, :external_url
end
```

```ruby
# app/serializers/api/v1/proposals/index_serializer.rb
class Api::V1::Proposals::IndexSerializer < Api::V1::ProposalSerializer
end
```

El siguiente *serializer* corresponde al endpoint `api/proposals/:id`, que devuelve una única propuesta. En este caso, además de los atributos especificados en `Api::V1::ProposalSerializer` se incluye un enlace a los comentarios de esa propuesta. En REST es habitual que los endpoints destinados a devolver un único objeto devuelvan información más completa que el equivalente para obtener la colección completa de objetos:

```
# app/serializers/api/v1/proposals/show_serializer.rb
class Api::V1::Proposals::ShowSerializer < Api::V1::ProposalSerializer
  has_many :comments, serializer: Api::V1::CommentSerializer do
    link(:related) do
      api_comments_path(
        commentable_id: object.id,
        commentable_type: 'Proposal'
      )
    end
  end
end
```

Resultado

Una vez terminado, abrí una *pull request*. El código completo de este prototipo puede consultarse aquí:

https://github.com/consul/consul/pull/1238/files

O haciendo un git diff:

```
git diff 9144c7da05deab8b60d7f8997eff58e243e68ad6 \
49f99d2fb24af8f4d0b6e184a6e3a0d2fb6e5ede
```

Prototipo GraphQL

Del 26 de septiembre al 12 de octubre estuve trabajando en el prototipo de GraphQL. Esta versión permitía obtener comentarios, debates, propuestas, geozonas y usuarios.

Para montar GraphQL dentro de consul los pasos a seguir fueron los siguientes:

Primero añadir una ruta (endpoint) a la que llegarán todas las *queries* de GraphQL:

```
# config/routes.rb
get '/queries', to: 'graphql#query'
```

Luego crear un controlador con una única acción, al que llegarán todas las peticiones realizadas a la ruta anterior:

```
class GraphqlController < ApplicationController
  skip_authorization_check

  def query
    render json: ConsulSchema.execute(
      params[:query],
      variables: params[:variables] || {}
    )
  end
end
```

Crear el *GraphQL schema*, que aglutina todos los *GraphQL types* de nuestra aplicación:

```
ConsulSchema = GraphQL::Schema.define do
  query QueryRoot

  resolve_type -> (object, ctx) {
    type_name = object.class.name
    ConsulSchema.types[type_name]
  }
end
```

Crear un *GraphQL type* para cada uno de los modelos de nuestra aplicación, con un *field* para cada uno de los atributos que queremos exponer en la API. Por ejemplo, para una propuesta:

```
ProposalType = GraphQL::ObjectType.define do
  name 'Proposal'
  description 'A single proposal entry'

  field :id, !types.ID, 'The id of this proposal'
  field :title, types.String, 'The title of this proposal'
  field :author, UserType, 'Author of this proposal'
end
```

Finalmente hay que crear un *GraphQL type* denominado *QueryType* o *QueryRoot* que actúa como punto de entrada para todas las *queries*, con un único *field* (en este caso) que permite recuperar una propuesta:

```
QueryRoot = GraphQL::ObjectType.define do
  name 'Query'
  description 'The query root for this schema'

  field :proposal do
    type ProposalType
    description 'Find a Proposal by id'
    argument :id, !types.ID
    resolve -> (object, arguments, context) {
      Proposal.find(arguments['id'])
    }
  end
end
```

Resultado

El código completo de este prototipo puede consultarse en esta pull request:

https://github.com/consul/consul/pull/1239/files

O con el siguiente git diff:

```
git diff df5caea56f18da6e6f7b698933b562aadfff74c9 \
757e68969df8fd5443b5c6800ffa263e7efa2f2f
```

Presentación en el Coding Madrid

En el Coding Madrid del 13 de octubre presenté a los desarrolladores de Consul el código de ambos prototipos, así como las ventajas e inconvenientes que había visto en cada uno de ellos. Tras deliberar, decidieron decantarse por la versión en GraphQL.

Aunque el feedback sobre el prototipo en GraphQL fue bueno, salieron varios puntos importantes a la luz que iban a implicar cambios significativos en el código actual, y que fueron los temas a tratar en las fases posteriores.

Estos puntos fueron:

- **La ausencia de tests**

  Como ya he comentado en otros apartados, resulta imprescindible que todo código incorporado al repositorio venga con sus tests correspondientes. Hasta el momento no había escrito ningún test, y apenas tenía experiencia con RSpec, así que era un reto importante.

- **La duplicación de código**

  En esta memoria sólo se ha incluido un fragmento del código del *GraphQL type* para el modelo `Proposal`, pero si se consulta el código para los modelos `Debate`, `Comment` y `Geozone` en el enlace a la pull request puede verse que todos presentan una estructura muy similar. El tener toda esta información repartida en multitud de pequeños ficheros podía suponer un problema de cara a la mantenibilidad de la API, así que se decidió utilizar técnicas de metaprogramación de Ruby para leer directamente del esquema de la base de datos la información relativa a los atributos de un modelo que iban a hacerse públicos en la API.

- **Aspectos relacionados con la privacidad**

  En ese momento todavía no se había elaborado una especificación detallada con la información que se deseaba y era posible exponer en la API, de modo que para ir familiarizándome con lo que podría ser el desarrollo me basé en mi intuición a la hora de decidir que campos mostraba en la API y cuáles no.

  Por este motivo, y por desconocimiento del significado de algunos de los campos de la base de datos, estaba exponiendo información que era confidencial. Si bien esto no tenía ninguna repercusión, pues este código era un prototipo y no estaba conectado con la base de datos real, quedó claro que era necesario elaborar un listado detallado con la información que era prudente mostrar.

5.2.4. Mejoras en la mantenibilidad del código

El objetivo de esta fase era automatizar el proceso de creación de los *GraphQL types* a partir de los propios modelos de la aplicación Rails. Para ello había que escribir un fragmento de código que se ejecutase en el momento en el que la aplicación arrancase, leyese todos los modelos existentes, y crease un *GraphQL type* para cada uno de ellos.

Como no se querían mostrar ni todos los modelos ni todos los atributos, se utilizó una técnica de *whitelist* para indicar qué modelos y que campos iban a verse afectados. La primera versión de este *whitelist* venía dada en forma de hash. Posteriormente se leería de un fichero `yaml`.

El código encargado de la generación automática de tipos es posiblemente el más complejo de todo el proyecto, además de tener una sintaxis muy poco legible debido a las limitaciones del DSL de la librería `graphql-ruby`. Por este motivo aquí se presenta directamente un fragmento de la versión final de este código (que fue modificado multitud de veces a lo largo de las fases posteriores).

Este método se invocaría una vez por cada uno de los modelos de la aplicación para los que se desea crear un *GraphQL type,* y almacenaría el tipo creado en la variable `created_types` para su posterior utilización:

```ruby
def self.create_type(model, fields, created_types)

  created_types[model] = GraphQL::ObjectType.define do

    name        model.graphql_type_name
    description model.graphql_type_description

    # Make a field for each column, association or method
    fields.each do |field_name, field_type|
      case ApiTypesCreator.type_kind(field_type)
      when :scalar
        field(
          field_name,
          SCALAR_TYPES[field_type],
          model.human_attribute_name(field_name)
        )
      when :singular_association
        field(field_name, -> { created_types[field_type] }) do
          resolve -> (object, arguments, context) do
            association_target = object.send(field_name)
            if association_target.nil?
              nil
            else
              field_type.public_for_api.find_by(
                id: association_target.id
              )
            end
          end
        end
      when :multiple_association
        field_type = field_type.first
        connection(
          field_name,
          -> { created_types[field_type].connection_type }
        ) do
          resolve -> (object, arguments, context) do
            field_type.public_for_api & object.send(field_name)
          end
        end
      end
    end

  end
end
```

Resultado

Este mecanismo permitió no tener que crear un fichero específico para cada uno de los tipos GraphQL de la aplicación, sino hacerlo de forma automática a partir de la lista de modelos y campos de la API.

El código de esta fase no dispone de una *pull request* propia, pero corresponde con los commits comprendidos entre el `e7de5c3a7f0c6e7bcc7b4caea1ada23e2bef02cb` y el `195d1c5f690525b59c77a27ce10eb222ed788fe9` de la siguiente pull request:

https://github.com/consul/consul/pull/1239/files

También es posible ver los cambios con los siguientes `git diff`:

```
git diff 757e68969df8fd5443b5c6800ffa263e7efa2f2f \
59a355df1b17df6a8ca05776a961b2bd14514884
git diff dd4b29898cb9e557fb568701833f95343be9524f \
195d1c5f690525b59c77a27ce10eb222ed788fe9
```

5.2.5. Testing

Para abordar el testing de la API empecé por leer dos libros relacionados con este tema, y que me vinieron muy bien:

- **The Way of the Web Tester**, Jonathan Rasmusson - *https://pragprog.com/book/jrtest/the-way-of-the-web-tester*

- **Rails 4 Test Prescriptions**, Noel Rappin - *https://pragprog.com/book/nrtest2/rails-4-test-prescriptions*

Paralelamente a la lectura de estos libros, empecé varios pequeños proyectos con el objetivo de ir practicando los conceptos que iba aprendiendo, pues aunque los libros vienen con multitud de casos prácticos, el enfrentarte a problemas relacionados pero diferentes y a los que no tienes la solución escrita en la página siguiente viene bien.

La curva de aprendizaje de RSpec me resultó un poco violenta sobretodo en lo relacionado con el "evitar el estado compartido entre los tests".

Por otro lado, estos libros presentaban ejemplos bastante sencillos y acotados de lo que es el testing de una aplicación real. Esto se juntó con que no era fácil aplicar los conceptos de estos libros a los tests de GraphQL, pues debido a la naturaleza recursiva de sus consultas, y al hecho de que cada una de ellas puede acceder a información acerca de multitud de modelos de la aplicación, era difícil determinar con claridad la responsabilidad que debía asumir cada uno de los tests escritos. Aunque en la propia documentación de `graphql-ruby` hay una sección dedicada al testing, para gente sin experiencia como yo en aquel momento se quedaba un poco corta.

Es posible consultar dicha guía en la siguiente dirección:

https://rmosolgo.github.io/graphql-ruby/schema/testing

Organización de los tests de la API

Siguiendo los principios descritos descritos en la sección de *Estado del Arte* respecto a testing de aplicaciones web, los tests de la API se encuentran en dos de los tres niveles definidos por la *Pirámide de Testing*:

- Tests unitarios
- Tests de integración

Por tratarse de una API, no existen tests de interfaz de usuario. No obstante, dentro de los tests de integración es posible definir dos grupos distintos en base al nivel de abstracción de dichos tests:

- A nivel de peticiones HTTP
- A nivel de *GraphQL schema*

Tests unitarios

Se encuentran en estos dos ficheros:

- `spec/lib/graph_ql/query_type_creator_spec.rb`
- `spec/lib/graph_ql/api_types_creator_spec.rb`

Los tests que encontramos en ellos están destinados a comprobar el correcto funcionamiento del `QueryTypeCreator` y del `ApiTypesCreator`.

Un ejemplo sencillo de test que se puede encontrar en `api_types_creator.rb`:

```
describe GraphQL::ApiTypesCreator do
  describe "::create_type" do
    it "creates fields for Int attributes" do
      debate_type = GraphQL::ApiTypesCreator.create_type(
        Debate, { id: :integer }, {}
      )

      created_field = debate_type.fields['id']

      expect(created_field).to be_a(GraphQL::Field)
      expect(created_field.type).to be_a(GraphQL::ScalarType)
      expect(created_field.type.name).to eq('Int')
    end
  end
end
```

En este test concreto el objetivo es comprobar que el `ApiTypesCreator` es capaz de crear un tipo de GraphQL para un determinado modelo de la aplicación, y que dicho tipo contiene a su vez un campo (field) que expone uno de los atributos de tipo entero del model objetivo.

El método `create_type` recibe los siguientes parámetros:

1. El modelo de la aplicación que va a estar asociado al *GraphQL type*, `Debate` en este caso.
2. Un hash que contiene como claves el nombre de los *GraphQL fields* que va a tener el tipo creado, y cuyo valor representa el tipo del valor retornado por dichos campos. En este caso, el *field* creado va a exponer el *id* de un debate, y el tipo es `Integer`.
3. Un objeto en donde se van a ir almacenando los tipos que crea el `ApiTypesCreator`. Para este test esto es irrelevante, pero en la aplicación algunos de los tipos creados dependen de otros y así sucesivamente, y es necesario mantenerlos en alguna variable para el correcto funcionamiento del `ApiTypesCreator`.

En las siguientes líneas se realizan varias aserciones sobre las características de dicho *field*. Lo que hace cada una de ellas puede deducirse fácilmente del propio nombre de los métodos utilizados.

Tests a nivel de *GraphQL schema*

Es el nivel de abstracción inmediatamente inferior a las llamadas de red efectuadas contra el controlador de la API. En estos tests se ejecutan *queries* contra el schema de GraphQL y se compraran los resultados obtenidos. Se encuentran dentro del fichero `spec/lib/graphql_spec.rb`.

En este fichero se crea un *GraphQL schema* idéntico al que usaría la aplicación cuando esté desplegada, y sobre dicho *schema* se ejecutan distintas *queries* con el objetivo de comprobar que:

- Los campos que deberían ser visibles son, efectivamente, visibles.
- Los campos que deberían estar ocultos permanecen ocultos.

En las primeras líneas del fichero se crean los *GraphQL types*, el *QueryType* y el *GraphQL schema*:

```
api_types  = GraphQL::ApiTypesCreator.create(API_TYPE_DEFINITIONS)
query_type = GraphQL::QueryTypeCreator.create(api_types)
ConsulSchema = GraphQL::Schema.define do
  query query_type
end
```

A continuación se definen una serie de métodos de tipo *helper* destinados a facilitar la lectura de los tests posteriores:

- `execute` define un alias para realizar queries sobre el *GraphQL schema*
- `dig` proporciona una forma más amigable de leer los campos de los datos JSON devueltos por la query.
- `hidden_field?` comprueba que un campo que debería estar oculto está oculto.

```
def execute(query_string, context = {}, variables = {})
  ConsulSchema.execute(
    query_string,
    context: context,
    variables: variables
  )
end

def dig(response, path)
  response.dig(*path.split('.'))
end

def hidden_field?(response, field_name)
  data_is_empty = response['data'].nil?
  error_is_present = (
    (
      response['errors'].first['message'] =~ Regexp.new(
        "Field '#{field_name}' doesn't exist on type '[[:alnum:]]*'"
      )
    ) == 0
  )
  data_is_empty && error_is_present
end
```

A continuación vienen los tests. En las primeras versiones de este conjunto de tests sólo se comprobaba la disponibilidad y ocultación de de un caso de ejemplo para cada uno de los distintos tipos de GraphQL que era posible encontrar en la aplicación: un tipo básico numérico (`Int`), uno de texto (`String`) y asociaciones `has_one`, `belongs_to` y `has_many`, puesto que se supone que el comportamiento del creador de tipos GraphQL era exactamente el mismo para todos los campos de tipo Int, todos los campos de tipo String, etc. No obstante, en una de las fases finales del proyecto decidí hacer un testeo muchísimo más exhaustivo del *Schema* por motivos que detallaré más adelante.

Un ejemplo de test que puede encontrarse en este fichero es el siguiente:

```ruby
describe 'ConsulSchema' do
  let(:user) { create(:user) }
  let(:proposal) { create(:proposal, author: user) }

  it 'returns fields of String type' do
    response = execute(
      "{ proposal(id: #{proposal.id}) { title } }"
    )
    expect(
      dig(response, 'data.proposal.title')
    ).to eq(proposal.title)
  end

  it 'hides confidential fields of String type' do
    response = execute(
      "{ user(id: #{user.id}) { encrypted_password } }"
    )
    expect(
      hidden_field?(response, 'encrypted_password')
    ).to be_truthy
  end
end
```

Tests a nivel de peticiones HTTP

En mi caso, es el nivel de abstracción más alto de todos. Manda peticiones HTTP ficticias a la aplicación y compara tanto la cabecera como el contenido de las respuestas obtenidas.

Estos tests se encuentran en el fichero `spec/controllers/graphql_controller_spec.rb`. Una versión *muy simplificada* de este fichero, pero bastante representativa de los tests que es posible encontrar en este fichero es la siguiente:

```
describe GraphqlController, type: :request do
  let(:proposal) { create(:proposal) }

  describe "handles POST requests" do
    specify "with json-encoded query string inside body" do
      post(
        '/graphql',
        {query: "{proposal(id: #{proposal.id}){title}}"}.to_json,
        { "CONTENT_TYPE" => "application/json" }
      )

      expect(response).to have_http_status(:ok)
      expect(
        JSON.parse(response.body)['data']['proposal']['title']
      ).to eq(proposal.title)
    end
  end

end
```

El fichero completo contiene más tests dedicados a comprobar los siguientes casos:

- Peticiones GET correctas, con el *query string* como parámetro de la URL.
- Peticiones POST correctas, con el *query string* dentro del *body*, en formato raw y JSON.
- Peticiones GET incorrectas por no tener *query string* o porque está mal formado.
- Peticiones POST incorrectas por no tener *query string* o porque está mal formado.
- Comprobar el correcto parseo de las variables del *query string*, si las hubiese.

Resultados

Al final del proyecto se consiguió una cobertura de tests de prácticamente el 100%. Además, el propio proceso de escritura de los tests me sirvió para detectar bugs que de otro modo habría sido muy difícil detectar.

El código escrito durante esta fase es (aproximadamente) el escrito en estos dos intervalos de commits:

```
git diff b10b7013193ed409e0d60632dcee63b938d8c5fd \
860704d908c54cf711581b4c33ec2386a78822d7
git diff 8b30aaa6d4de28c47baf8b22dc41d7935f2359af \
a0f1976c1adba3de67c6204dec2df1324d5fa34a
```

5.2.6. Mejoras en la *whitelist* y soporte para campos calculados

Soporte para campos calculados

La primera tarea que llevé a cabo en esta fase fue el permitir exponer en la API los llamados *campos calculados,* es decir, campos de un modelo que no tienen una correspondencia directa con una columna de la base de datos sino que se obtienen a través de un método del objeto que no es un *getter*. El problema principal que presentaba esto es que para crear un *GraphQL field* es necesario conocer el tipo de retorno de dicho campo. Para los *getters* era posible sacar esta información del propio esquema de la base de datos, pero para el resto de métodos no había forma de saber esto anticipadamente. Por este motivo hubo que modificar la *whitelist* para que a partir de ahora incluyese información sobre el tipo de dato de cada *field* de forma explícita.

Los cambios hechos se ven muy bien en el siguiente commit:

https://github.com/consul/consul/pull/1257/commits/7dc4d80e7bbab49a45d818ccc321a2abfe291be5

O en el siguiente git diff:

```
git diff 0f663603d0af0876e481d5c5b8a071335b346009 \
7dc4d80e7bbab49a45d818ccc321a2abfe291be5
```

Mejoras en la *whitelist*

Declarar los modelos y campos expuestos en la API dentro de un hash de una clase Ruby era un enfoque un poco *sucio*, así que, tras consultarlo con los desarrolladores de Consul decidí modificar el modo en que se declaraba la *whitelist* de modelos y atributos de la API para que en vez de ser un hash, se leyese de un fichero .yml, que posee una sintaxis mucho más legible.

Un ejemplo de *whitelist* en las primeras versiones de la API tenía este aspecto:

```
API_TYPE_DEFINITIONS = {
  User => % I[id username proposals],
  Debate => % I[id title description author_id author created_at],
  Proposal => % I[id title description author created_at],
  Comment => % I[id body author_id author commentable]
}
```

Aunque correcto, el ir añadiendo más modelos y campos a la API, y especialmente cuando hubo que añadir a cada uno de los campos el tipo de retorno, provocaron que este enfoque se volviese demasiado torpe de manejar.

Este problema puede verse muy bien aquí:

https://github.com/amiedes/consul/blob/7dc4d80e7bbab49a45d818ccc321a2abfe291be5/config/initializers/graphql.rb

Un ejemplo de *whitelist* en el fichero `.yml` tras la refactorización del código es la siguiente:

```
User:
  fields:
    id:                integer
    username:          string
    proposals:         [Proposal]
    comments:          [Comment]
Proposal:
  fields:
    id:                integer
    title:             string
    geozone:           Geozone
    comments:          [Comment]
    public_author:     User
Comment:
  fields:
    id:                integer
    commentable_id:    integer
    commentable_type:  string
    body:              string
    public_author:     User
Geozone:
  fields:
    id:                integer
    name:              string
```

Esta refactorización se llevo a cabo en el siguiente commit:

https://github.com/consul/consul/pull/1257/commits/b36deb03829d8e42932c821a05c8e41a3daf6311

También puede consultarse con el siguiente git diff:

```
git diff 9cfb3eb52e246f0b62cb147ab89d2d78cea400b5 \
b36deb03829d8e42932c821a05c8e41a3daf6311
```

5.2.7. Privacidad y seguridad

Privacidad

A medida que se acercaba el final del desarrollo, el saber con exactitud qué modelos y campos quería exponer en la API el Ayuntamiento de Madrid se volvió crucial, pues algunos de los posibles requisitos podían implicar cambios importantes en el código.

Tras obtener un listado con los requisitos, procedí a realizar las modificaciones pertinentes. Estos requisitos podían dividirse en dos tipos:

- Qué modelos y campos mostrar. Esto era fácil conseguirlo, pues simplemente había que añadir el nombre del modelo o el campo al fichero `yml`.

- Bajo qué condiciones un campo podía ser mostrado. Esto no era tan sencillo, pues ya no era cuestión de si mostrarlo o no sino de *cuándo*. Para lograr esto añadí un método llamado `public_for_api` a cada uno de los modelos de la base de datos, y que devolvía en cada caso los registros de la tabla correspondiente que cumplían las restricciones estipuladas.

  No obstante no todos los modelos tenían restricciones respecto a qué registros se podían mostrar, así que modifiqué una de las clases del núcleo de Rails (`ActiveRecord::Base`) para que todos los modelos incluyesen automáticamente un método `public_for_api` que, en caso de no ser sobrescrito posteriormente, por defecto devuelve todos los registros de la tabla del modelo.

Por ejemplo, en el caso del modelo `ProposalNotification`, se decidió que un registro sólo sería público si las propuesta asociada no había sido ocultada por un administrador. El método quedaría así:

```
class ProposalNotification < ActiveRecord::Base

  # …

  def self.public_for_api
    joins(:proposal).where("proposals.hidden_at IS NULL")
  end
end
```

Este caso es sencillo, pero otros como este son mucho más enrevesados:

https://github.com/amiedes/consul/blob/3c7f60d62547d36ccdb50574bda4da48336ae032/config/initializers/acts_as_taggable_on.rb#L47-L62

Seguridad

Hay una serie de mecanismos disponibles para GraphQL relacionados con la seguridad.

Profundidad máxima de las consultas GraphQL

Con las APIs hay que tener especial cuidado con las consultas que permitimos a un cliente ejecutar en el servidor, pues en el caso de ser demasiado pesadas podrían provocar una denegación de servicio. En el caso de GraphQL esto es todavía más crítico, pues con la posibilidad de enlazar varios recursos en una sola consulta aparece la posibilidad de crear consultas circulares o excesivamente complejas.

Afortunadamente es posible establecer una profundidad máxima para las consultas de GraphQL. Esto se hace en el momento de la creación del *GraphQL schema*:

```
GraphQL::Schema.define do
  query query_type
  max_depth 12
end
```

En este caso el valor lo asigné yo arbitrariamente, pero es trivial de modificar una vez que el código esté desplegado en función de la carga que sea capaz de soportar el servidor.

Paginación

Otro de los mecanismos que permiten aligerar la cantidad de información que el servidor tiene que procesar en cada petición es la paginación. La paginación permite que, en consultas que devuelve colecciones de registros, no se devuelvan todos en una única petición sino que haya que realizar varias peticiones consecutivas para obtener la totalidad de los datos.

En GraphQL la paginación se realiza mediante los *cursores* y las *conexiones*. Para más información sobre estos dos términos, consultar *http://graphql-ruby.org/relay/connections.html*

Todas las colecciones que aparecen en la API están paginadas. No obstante, no fue hasta la última fase cuando se forzó a que fuese obligatoria en vez de opcional.

Autenticación

Al ser una API de sólo lectura se decidió que no iba a haber ningún tipo de autenticación, pues iba a ser imposible modificar datos maliciosamente y se esperaba que los servidores del ayuntamiento fuesen capaces de soportar las peticiones a la API siempre y cuando el tiempo necesario para resolver estas consultas no fuese demasiado elevado.

Capturar todas las excepciones en producción

Aunque durante un proceso de depuración ver una descripción detallada de las excepciones que han ocurrido durante la ejecución de la aplicación puede ser de gran ayuda, no es prudente realizar esto en un entorno de producción, pues podría exponer posibles vulnerabilidades de la aplicación. Por este motivo realicé un control mucho más exhaustivo de todas las posibles excepciones que podrían ocurrir durante una petición a la API:

```
class GraphqlController < ApplicationController

  # …

  def query
    begin
      if query_string.nil?
        raise GraphqlController::QueryStringError
      end

      response = consul_schema.execute(
        query_string,
        variables: query_variables
      )
      render json: response, status: :ok
    rescue GraphqlController::QueryStringError
      render(
        json: { message: 'Query string not present' },
        status: :bad_request
      )
    rescue JSON::ParserError
      render(
        json: { message: 'Error parsing JSON' },
        status: :bad_request
      )
    rescue GraphQL::ParseError
      render(
        json: { message: 'Query string is not valid JSON' },
        status: :bad_request
      )
    rescue
      unless Rails.env.production? then raise end
    end
  end

  # ...
end
```

5.2.8. Pruebas en preproducción y despliegue a producción.

En el Coding Madrid del 11 de mayo estuve examinando con Raimond el código de la API en vistas de un posible despliegue a producción. El feedback fue bueno así que sólo tuve que hacer unos pequeños cambios:

- La rama de la PR llevaba bastante tiempo sin ser actualizada con los contenidos de *master*, así que tuve que resolver los conflictos.

- Actualicé las gemas de *graphql-ruby* y *graphiql-rails*, puesto que al ser tan recientes era posible que hubiese aparecido algún agujero de seguridad durante los últimos meses.

- Raimond había estado trabajando en realizar una exportación a CSV de los datos de Decide Madrid para subirlos al portal de datos abiertos del ayuntamiento. Muchos de los criterios de privacidad para la API eran comunes con esta funcionalidad, así que decidí adaptar unos tests que había hecho para poder utilizarlos en la API y así asegurarnos de que todas las condiciones que se estaban cumpliendo en la exportación a CSV se cumplían también en la API.

En la semana del 15 al 19 de mayo estuvieron probando la API en preproducción. Se vio que para algunas consultas complejas el tiempo que tardaba el servidor en ejecutarlas era excesivo, así que decidimos hacer los siguientes cambios:

- Hacer la paginación obligatoria. Establecí un límite de 50 resultados por colección.

- Disminuir la profundidad máxima de las consultas a 8.

- Establecer un máximo de 2 colecciones presentes en cada consulta.

Lamentablemente estas mejoras no fueron suficientes, pues en pruebas posteriores se vio que el tiempo de ejecución de las consultas seguía siendo demasiado elevado. Tras una pequeña investigación descubrí que algunas de las operaciones que estaba haciendo con la base de datos eran excesivamente costosas de realizar tal y como estaban escritas. No obstante, una vez identificado el problema fue sencillo de solucionar con este commit: *https://github.com/consul/consul/pull/1653/commits/9ec8b166d775894*

Durante la segunda semana de junio Enrique estuvo realizando una serie de pruebas exhaustivas dedicadas a asegurarse de que al desplegar la API no se liberaba información confidencial. Estas pruebas consistían en comparar que las piezas de información expuestas a través de la API coincidían exactamente con las expuestas a través del exportador a CSV que ya tenían, y cuyo correcto funcionamiento ya había sido verificado con anterioridad.

A fecha de 15 junio la API fue mergeada en Consul y desplegada en Decide Madrid.

6. Resultados y Trabajo Futuro

6.1. Resultados del desarrollo

A modo de resumen, las funcionalidades aportadas por la API en el momento de finalización de este Trabajo de Fin de Grado son las siguientes:

- Una API de lectura para una gran cantidad de los modelos y atributos de Consul, muy fácilmente extensible para cualquier otro modelo, atributo o relación que quiera añadirse en el futuro.

- Una suite exhaustiva de tests que comprueba el correcto funcionamiento de la API y que no exista ninguna fuga de información. La cobertura de código está situada cerca del 100%.

- Mecanismos de seguridad para evitar posibles ataques de denegación de servicio provocados por consultas que requieran excesivo tiempo de procesamiento por parte del servidor para ser atendidas. Entre ellos están la paginación, la limitación de la profundidad máxima de las consultas y la limitación en el número máximo de campos que es posible pedir en una consulta.

- Soporte tanto para realizar consultas a través de cualquier librería HTTP como a través de GraphiQL.

- Documentación de la API autogenerada a partir de metainformación presente en los propios modelos de la aplicación.

6.1.1. Listado completo de modelos

Esta sección contiene una réplica del archivo `api.yml`, que contiene la lista completa de los modelos de Consul soportados por la API y sus respectivos atributos en el estado final de la pull request:

```
User:
  fields:
    id:                     integer
    username:               string
    public_debates:         [Debate]
    public_proposals:       [Proposal]
    public_comments:        [Comment]
    organization:           Organization
```

```
Debate:
  fields:
    id:                     integer
    title:                  string
    description:            string
    public_created_at:      string
    cached_votes_total:     integer
    cached_votes_up:        integer
    cached_votes_down:      integer
    comments_count:         integer
    hot_score:              integer
    confidence_score:       integer
    comments:               [Comment]
    public_author:          User
    votes_for:              [Vote]
    tags:                   ["ActsAsTaggableOn::Tag"]
Proposal:
  fields:
    id:                     integer
    title:                  string
    description:            string
    external_url:           string
    cached_votes_up:        integer
    comments_count:         integer
    hot_score:              integer
    confidence_score:       integer
    public_created_at:      string
    summary:                string
    video_url:              string
    geozone_id:             integer
    retired_at:             string
    retired_reason:         string
    retired_explanation:    string
    geozone:                Geozone
    comments:               [Comment]
    proposal_notifications: [ProposalNotification]
    public_author:          User
    votes_for:              [Vote]
    tags:                   ["ActsAsTaggableOn::Tag"]
```

```
Comment:
  fields:
    id:                      integer
    commentable_id:          integer
    commentable_type:        string
    body:                    string
    public_created_at:       string
    cached_votes_total:      integer
    cached_votes_up:         integer
    cached_votes_down:       integer
    ancestry:                string
    confidence_score:        integer
    public_author:           User
    votes_for:               [Vote]
Geozone:
  fields:
    id:                      integer
    name:                    string
ProposalNotification:
  fields:
    title:                  string
    body:                   string
    proposal_id:            integer
    public_created_at:      string
    proposal:               Proposal
ActsAsTaggableOn::Tag:
  fields:
    id:                     integer
    name:                   string
    taggings_count:         integer
    kind:                   string
Vote:
  fields:
    votable_id:             integer
    votable_type:           string
    public_created_at:      string
    vote_flag:              boolean
Organization:
  fields:
    id:                     integer
    user_id:                integer
    name:                   string
```

6.1.2. Ejemplos de consultas soportadas

En esta sección se muestran pequeños ejemplos de consultas que es posible realizar a la API en su estado actual. Estas consultas están realizadas contra la instalación de Consul que tengo en mi máquina, así que los datos obtenidos proceden de una base de datos rellenada con datos semilla.

Consulta simple

Aquí se muestra un ejemplo de una consulta sencilla: se pide la propuesta con `id` = 3 y varios de sus atributos. También se pide información relacionada pero que pertenece a modelos distintos: el distrito y el autor.

```
{
  proposal(id: 3) {
    id,
    title,
    external_url,
    hot_score,
    geozone {
      id,
      name
    },
    public_author {
      id,
      username
    }
  }
}
```

Este es el resultado obtenido:


```
{
  "data": {
    "proposal": {
      "id": 3,
      "title": "Nulla at voluptatum quidem facilis quibusdam et.",
      "external_url": "http://funk.org/madisen.witting",
      "hot_score": 7107226112,
      "geozone": {
        "id": 11,
        "name": "District I"
      },
      "public_author": {
        "id": 108,
        "username": "Francisco Cervántez Pelayo"
      }
    }
  }
}
```

Consulta a una colección

Este es un ejemplo de una consulta realizada contra una colección, propuestas en este caso. Por brevedad sólo se piden las dos primeras, pero si no se especificase el parámetro `first: 2` se devolverían las 50 primeras, que es el máximo establecido actualmente.

```
{
  proposals(first: 2) {
    edges {
      node {
        id,
        title,
        external_url
      }
    }
  }
}
```

Estos son los datos devueltos:

```
{
  "data": {
    "proposals": {
      "edges": [
        {
          "node": {
            "id": 1084,
            "title": "Laboriosam aliquam qui optio et ipsa est.",
            "external_url": "http://walker.biz/christ"
          }
        },
        {
          "node": {
            "id": 782,
            "title": "Consectetur ratione blanditiis aliquid.",
            "external_url": "http://ankundingolson.org/myrtle"
          }
        }
      ]
    }
  }
}
```

Consulta a una colección (paginación)

Esta consulta es similar a la anterior, pero en este caso se solicita información relativa a la paginación:

```
{
  proposals(first: 2) {
    pageInfo {
      startCursor,
      endCursor,
      hasNextPage
   }
    edges {
      cursor,
      node {
        id,
        title
      }
    }
  }
}
```

Esta es la respuesta:

```
{
  "data": {
    "proposals": {
      "pageInfo": {
        "startCursor": "MQ==",
        "endCursor": "Mg==",
        "hasNextPage": true
     },
     "edges": [
       {
         "cursor": "MQ==",
         "node": { "title": "Laboriosam aliquam..." }
       },
       {
         "cursor": "Mg==",
         "node": { "title": "Consectetur rationae..." }
       }
     ]
    }
  }
}
```

Para navegar a través de las distintas páginas de la colección, hay que incluir en las peticiones sucesivas el parámetro `after` con el valor del `endCursor` de la petición inmediatamente anterior. En este caso sería tal que así:

```
{
  proposals(first: 2, after:"Mg==") {
    # etc.
}
```

Protección contra consultas excesivamente costosas

Permitir que un cliente personalice la información que desea obtener de la API tiene muchas ventajas, pero también presenta importantes problemas de seguridad si no se implementan mecanismos para limitar la cantidad de procesamiento que el servidor debe llevar a cabo para satisfacer las consultas. En el caso de Consul se ha establecido un máximo de 8 niveles a la profundidad de las consultas, y un máximo de dos campos de tipo "colección" por consulta. El siguiente ejemplo muestra una consulta que supera tanto la profundidad como la complejidad máxima permitidas:

```
{
  users {
    edges {
      node {
        id,
        public_debates {
          edges {
            node {
              id,
              comments {
                edges {
                  node {
                    id
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Esta es la respuesta dada por el servidor:

```
{
  "errors": [
    {
      "message": "Query has depth of 10, which exceeds max depth of 8"
    },
    {
      "message": "Query has complexity of 3009, which exceeds max
complexity of 2500"
    }
  ]
}
```

6.2. Resultados de la metodología

Una buena práctica extendida y aceptada en el desarrollo de software, ligada a las metodologías ágiles y a la integración continua, es fragmentar las funcionalidades pendientes en tareas lo suficientemente pequeñas como para que puedan ser realizados en un período corto de tiempo. Particionar las tareas grandes en otras más sencillas facilita mucho la comprensión del código que se está escribiendo y da la confianza necesaria al desarrollador para poder hacer despliegues a producción de manera regular.

Si bien el equipo de desarrolladores de Consul si que hace uso de este procedimiento en su día a día, una serie de factores han contribuido a que mi forma de trabajar no fuese tan ágil.

No obstante en mi caso hacer despliegues de forma regular implicaba que alguien dentro del equipo de desarrolladores, con mucha más experiencia, revisase mi código antes de cada despliegue. El equipo de desarrolladores de Consul tenía sus propios *deadlines* impuestos por el Ayuntamiento de Madrid, y aunque la API era una funcionalidad deseada, no era la más prioritaria dentro del *roadmap*. Revisar todo el código de la API suponía un trabajo considerable por su parte, así que en vez de estar haciendo esta revisión de forma frecuente y subiendo los cambios a producción, se optó por hacer la revisión al final.

No formar parte del equipo *core* de Consul, y por lo tanto ver limitado el contacto presencial con ellos a algo esporádico también ha sido uno de los factores que ha contribuido a realentizar el desarrollo.

Mi falta de experiencia en las tecnologías utilizadas también ha sido otro factor a tener en cuenta en el *alejarme del agilismo*. A menudo he tenido que acudir a libros de programación y empezar pequeños proyectos paralelos para practicar con determinadas técnicas antes de ponerlas en prácticas en el desarrollo de la API.

Respecto a la forma de organizar las ramas: todo el Trabajo de Fin de Grado correspondía a una única *topic branch*. No obstante, debido a la envergadura de esta *feature*, he hecho uso de ramas auxiliares por motivos de organización a la hora de trabajar.

Respecto a las comunicaciones, la mayoría no se han efectuado a través de las herramientas que facilita GitHub para ello sino a través del correo electrónico. Por este motivo apenas existe diálogo dentro del espacio habilitado para ello en la *pull request* de la API: *https://github.com/consul/consul/pull/1257*

6.3. Trabajo Futuro

El conjunto de funcionalidades que podrían añadirse a la API de Consul es muy extenso. Algunas características que serían especialmente interesantes añadir son:

Exponer información demográfica a través de la API

Aunque con la información expuesta actualmente a través de la API ya es posible hacer análisis de datos muy interesantes como medir el nivel de participación de cada distrito, ver la evolución de los apoyos y comentarios de una propuesta a lo largo del tiempo, o ver las temáticas más habituales en los debates y propuestas de cada distrito, hay otro conjunto de datos que podrían exponerse para realizar análisis mucho más potentes.

Liberar información relacionada con la edad, género, y lugar de residencia de los usuarios permitiría además hacer análisis como:

- Ver la edad, género y lugar de residencia asociados a los votantes de una determinada propuesta.
- Ver el grado de participación en la plataforma en función de la edad, género y distrito de residencia de los usuarios.
- Ver las temáticas que resultan más interesantes para determinados colectivos de personas.

Exponer esta clase de información presenta un reto importante relacionado con la privacidad. Ahora mismo este tipo de información de cada usuario es totalmente confidencial, y en el caso de que fuese a liberarse habría que garantizar que es imposible trazar un voto hasta un usuario concreto a partir de la información demográfica del votante.

Desarrollar una API de escritura

Ahora mismo la API es de sólo lectura, lo que implica que puede utilizarse para obtener datos de la base de datos de Consul, pero no es posible añadir información a ésta ni modificar los datos existentes.

Desarrollar una API de escritura lleva asociado otro reto importante: la autenticación. Si se va a permitir a determinados clientes modificar la base de datos de la aplicación es imprescindible que estos estén autenticados para evitar un uso malicioso de esta funcionalidad.

La forma más habitual de autenticar las peticiones a una API es a través de mecanismos de *tokens*.

Proveer a la API actual de mecanismos de escritura sería muy interesante en el caso de querer aprovecharla para hacer una aplicación móvil.

Desarrollar un mecanismo de *Rate Limit*

Ahora mismo no hay establecido ningún mecanismo de control relacionado con el número de peticiones que un cliente puede hacer a la API.

Una técnica habitual en el desarrollo de APIs para evitar un posible uso malintencionado de la API, como efectuar ataques de denegación de servicio, es limitar el número de peticiones que un determinado usuario puede efectuar en un período acotado de tiempo.

Establecer un mecanismo de *Rate Limit* también implicaría que las peticiones a la API deberían estar autenticadas, de forma que sea posible trazar cada una de ellas hasta el cliente que las ha efectuado, de modo que cuando uno de ellos exceda el cupo de peticiones que tiene permitido hacer, descartar las que lleguen a continuación.

7. Otros

En este apartado se exponen una serie de actividades complementarias que he realizado durante el desarrollo de este TFG.

7.1. Jornadas de Inteligencia Colectiva para la Democracia

Durante un período de 15 días, del 21 de noviembre al 2 de diciembre de 2016, el Medialab Prado organizó un taller intensivo y que tenía a la Democracia Directa como temática.

Se formaron un total de ocho equipos multidisciplinares para desarrollar una serie de prototipos de aplicaciones destinadas a mejorar la democracia y generar nuevos mecanismos de participación ciudadana.

Estos equipos estaban formados de unas diez personas y eran fuertemente multidisciplinares e internacionales. Algunos de los perfiles más habituales eran periodistas, arquitectos, ingenieros y desarrolladores. Además de los participantes, había una serie de mentores encargados de ayudar y guiar a los distintos grupos durante en desarrollo de sus proyectos.

La convocatoria de proyectos original puede consultarse aquí:

http://medialab-prado.es/article/herramientas-para-una-democracia-real-2016

Uno de los proyectos seleccionados fue la **Integración de la localización geográfica en la participación ciudadana mediante la conectividad entre Consul y Emapic**, que tenía como objetivo proveer a Consul de determinadas funcionalidades de geolocalización aportadas por la herramienta Emapic (*https://emapic.es*), desarrollada por la Universidad de La Coruña.

Durante estas dos semanas estuve ayudando en el desarrollo de esta integración, pues mis conocimientos de Ruby y de Consul resultaban de gran ayuda. En concreto realicé varias modificaciones en el código de Consul con el objetivo de poder hacer varias visualizaciones en la plataforma de mapas de Emapic.

La parte principal del trabajo consistió en modificar el código de Consul para que cada vez que se crease una propuesta en la plataforma, o se emitiese un voto, se enviase a la API de Emapic información acerca del título de la propuesta, geolocalización de ésta (distrito al que afecta) e información demográfica del votante (en el caso de los votos). Es importante matizar que, por restricciones legales y de privacidad, este prototipo se ejecutó con datos de prueba, no con la base de datos real de Decide Madrid.

Los resultados del trabajo realizado se encuentran repartidos entre:

- Los commits realizados entre del 26 de noviembre y 1 de diciembre de esta rama: *https://github.com/jorgelf/consul/commits/jlopez*

- Esta librería: *https://github.com/amiedes/emapic_consul*

- Este documento - *https://docs.google.com/document/d/1rHHp5UVhesCG9TF4EFpN-p5CSLCEmDHDgpyK2W8RUDQ/edit*

- Este blogpost - *http://www.unigis.es/mapas-y-participacion-ciudadana*

7.2. Charla en la Semana de la Informática

Durante la III Semana de la Informática de la Faculta de Informática de la UCM di la charla *No Time to Open Source*.

En esta charla conté mi experiencia trabajando en un proyecto open source y colaborando con los desarrolladores de Decide Madrid, haciendo énfasis en todo lo que aprendí durante los meses previos y situaciones interesantes que tuve que afrontar y a las que es difícil estar expuesto dentro de la Universidad. El propósito de esta charla era animar a otros estudiantes de la facultad a participar en el desarrollo de otros proyectos open source durante su estancia en la Universidad, ya sea como parte del Trabajo de Fin de Grado o como pasatiempos.

Es posible encontrar el programa completo de toda la semana, junto con el título y descripción de mi charla en la siguiente dirección:

http://informatica.ucm.es/iii-semana-de-la-informatica-2017

Conclusiones

Desarrollar la API de Consul ha sido para mi uno de los retos más importantes a los que me he enfrentado, pero también uno de los más satisfactorios. Cuando a finales de 2015 me enteré de la existencia de este proyecto y más tarde vi la posibilidad de colaborar en su desarrollo como parte de mi Trabajo de Fin de Grado fue una enorme alegría.

Una de mis principales motivaciones para realizar este proyecto era la posibilidad de realizar un desarrollo capaz de aportar un valor real a la comunidad que iba a hacer uso de él, pues a lo largo de la carrera he echado en falta oportunidades de realizar trabajos capaces de tener un impacto real.

Desde el punto de vista técnico ha sido un reto enorme y la formación autodidacta ha ocupado un tiempo considerable. He tenido que aprender a utilizar nuevas tecnologías y herramientas, mejorar mi conocimiento en otras, realizar numerosas investigaciones y pruebas con herramientas que todavía estaban en un estado muy temprano de desarrollo. También he aprendido a interactuar y coordinarme con otras personas y prestar atención a aspectos del desarrollo del software con los que apenas había tenido contacto hasta el momento como son los tests automáticos, la mantenibilidad y la legibilidad del código y algunos de los problemas de rendimiento que pueden aparecer en aplicaciones.

Aprender a leer, entender y modificar el código escrito por otras personas ha sido otro de las experiencias nuevas para mí, pero también ha sido una de las mejores formas que he visto hasta la fecha de aprender a programar.

Desarrollar este proyecto también me ha servido como un acercamiento al mundo laboral, pero me ha permitido explorar y profundizar en las distintas áreas a mi ritmo, sin la presión de un trabajo real.

Quedan muchas cosas que mejorar en la API de Consul, pero espero que esta primera versión permita a otras personas aprovechar la información de esta plataforma para desarrollar herramientas capaces de darnos un mejor entendimiento de las cuestiones importantes para las personas que nos rodean y ayudarnos a tomar decisiones en la dirección adecuada.

Conclusions

Developing the Consul API has been for me one of the hardest challenges I've faced, but also very satisfying. When I found out about this project in late 2015 and after I noticed the possibility of collaborating in its development as part of my degree final project, it was a great joy.

One of my main motivations for carrying out this project was the possibility of building something capable of bringing real value to the community that was going to make use of it, because throughout the career I have missed opportunities to do things capable of having a real impact.

From the technical point of view it has been a huge challenge and the self-taught training has taken a considerable time. I have learned to use new technologies and tools, improved my knowledge in others, done extensive research and testing with tools that were still in a very early stage of development. I have also learned how to interact and coordinate with other people and pay attention to aspects of software development that I have barely had contact with so far such as automatic tests, maintainability and legibility of the code and some of the performance problems that can appear in applications.

Learning to read, understand and modify code written by other people has been another new experience for me, but it has also been one of the best ways to improve my programming skills up to date.

Developing this project has also served to me as an approach to the work world, but has allowed me to explore and deepen the different areas at my pace, without the pressure of a real job.

There are still many things to improve in the Consul API, but I hope this first version will allow people to take advantage of the information on this platform to develop tools capable of giving us a better understanding of the issues important to the people around us and helping us to take decisions in the right directions.

Bibliografía

Intro

- *https://es.wikipedia.org/wiki/Democracia_directa*
- *http://medialab-prado.es/article/herramientas-para-una-democracia-real-2016*

Plataformas de Participación Ciudadana y Portales de Transparencia

- Decide Madrid - *https://decide.madrid.es*
- Decide Madrid (repositorio) - *https://github.com/AyuntamientoMadrid/consul*
- Consul (repositorio) - *https://github.com/consul/consul*
- Decidim Barcelona - *https://www.decidim.barcelona*
- Decidim Barcelona (repositorio) - *https://github.com/AjuntamentdeBarcelona/decidim-barcelona*
- Decidim Barcelona (repositorio, obsoleto) - *https://github.com/AjuntamentdeBarcelona/decidim.barcelona-legacy*
- Portal de Transparencia del Ayuntamiento de Madrid - *http://transparencia.madrid.es/portal/site/transparencia*
- Propuesta de cambios en la Arquitectura de Consul - *https://www.gitbook.com/book/alabs/propuesta-de-cambios-en-la-arquitectura-de-consul/details*
- Populate Tools - *http://populate.tools*
- Gobierto - *http://gobierto.es*
- Gobierto (repositorio) - *https://github.com/PopulateTools/gobierto*
- Gobierto (blog) - *http://gobierto.es/acerca-de*
- *Ley 19/2013, de 9 de diciembre, de transparencia, acceso a la información pública y buen gobierno - https://www.boe.es/buscar/doc.php?id=BOE-A-2013-12887*

APIs, REST y GraphQL

- *http://www.ttandem.com/blog/la-economia-de-las-apis-ya-esta-aqui*
- *http://graphql.org/*

- *https://facebook.github.io/react/blog/2015/05/01/graphql-introduction.html*
- *https://www.genbetadev.com/desarrollo-aplicaciones-moviles/por-que-deberiamos-abandonar-rest-y-empezar-a-usar-graphql-en-nuestras-apis*
- *https://www.lynda.com/Software-Development-tutorials/What-REST/126131/145957-4.html*
- *http://www.restapitutorial.com/lessons/whatisrest.html*
- *https://en.wikipedia.org/wiki/HATEOAS*
- *https://carlosvillu.com/introduccion-practica-a-graphql-i*
- *https://learngraphql.com/basics/introduction*
- *https://www.quora.com/What-is-GraphQL*
- *https://dev-blog.apollodata.com/graphql-explained-5844742f195e*
- *https://kadira.io/blog/graphql/initial-impression-on-relay-and-graphql*
- *https://www.meteor.com/articles/what-is-graphql*
- *https://medium.com/the-graphqlhub/your-first-graphql-server-3c766ab4f0a2*
- *https://thoughtbot.com/upcase/videos/rest*

Integración Continua

- *https://aws.amazon.com/es/devops/continuous-integration*
- *https://es.wikipedia.org/wiki/Integraci%C3%B3n_continua*
- Continuous Integration with Ruby (charla) - *https://vimeo.com/157427268*

Testing

- The Way of the Web Tester, Jonathan Rasmusson - *https://pragprog.com/book/jrtest/the-way-of-the-web-tester*
- Rails 4 Test Prescriptions, Noel Rappin - *https://pragprog.com/book/nrtest2/rails-4-test-prescriptions*
- GraphQL Ruby implementation testing guide - *https://rmosolgo.github.io/graphql-ruby/schema/testing*

Otros

- Convocatoria de proyectos Inteligencia Colectiva Para la Democracia - *http://medialab-prado.es/article/herramientas-para-una-democracia-real-2016*
- Emapic - *https://emapic.es*
- Emapic (repositorio) - *https://github.com/Emapic/emapic*
- *https://github.com/jorgelf/consul/commits/jlopez*
- *https://github.com/amiedes/emapic_consul*
- *https://docs.google.com/document/d/1rHHp5UVhesCG9TF4EFpN-p5CSLCEmDHDgpyK2W8RUDQ/edit*
- *http://www.unigis.es/mapas-y-participacion-ciudadana*
- *http://informatica.ucm.es/iii-semana-de-la-informatica-2017*