Sistema de votación online mediante certificado electrónico y blockchain

Votings system based on electronic signature and blockchain

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Septiembre de 2022

Autor

Manuel Galán García

Tutor/a

Rodrigo Santamaría Vicente

D. Rodrigo Santamaría Vicente, profesores/as del Departamento de Informática y Automatización de la Universidad de Salamanca

CERTIFICA:

Que el trabajo titulado “Sistema de votación online mediante certificado electrónico y blockchain” ha sido realizado por D. Manuel Galán García, con DNI 71038300V y constituye la memoria del trabajo realizado para la superación de la asignatura Trabajo de Fin de Grado de la Titulación Grado en Ingeniería Informática de esta Universidad.

Y para que así conste a todos los efectos oportunos.

En Salamanca, a 07 de Septiembre de 2022

Firmado por SANTAMARIA VICENTE RODRIGO - 70879303L el día 07/09/2022 con un certificado emitido por AC FNMT Usuarios

D. Rodrigo Santamaría Vicente

Dpto. De Informática y Automatización

Universidad de Salamanca

Lista de cambios

Versión	Fecha de revisión	Descripción
1.0	05/09/2022	Primera versión final del documento
1.1	06/09/2022	Aplicación de cambios sugeridos por el tutor
1.2	07/09/2022	Revisión bibliográfica y revisión gramatical

Tabla de contenidos

Lista de figuras

Introducción

Es innegable que, a día de hoy, la web, la informática, y la tecnología en general ha revolucionado nuestra manera de ver el mundo. Gracias a las tecnologías actuales, tenemos sistemas que nos permiten comunicarnos con quien, donde y como sea de manera instantánea o usamos dispositivos que caben en la palma de la mano para cubrir necesidades básicas como comer u orientarnos.

Curiosamente, y a pesar de que es un aspecto que, en mayor o menor medida, permea todos los - ámbitos de nuestras vidas, las cuestiones políticas nunca se han llegado a implementar de una manera que el usuario pueda interaccionar de manera directa con ellas.

Es cierto que cada vez es mayor la posibilidad de realizar trámites *online* con tecnologías como Clave PIN o el DNI electrónico, e incluso hay países que se atreven a ir más allá, como Estonia, permitiendo obtener la nacionalidad, sin necesidad de tener una residencia o cualquier tipo de domicilio fiscal en el país, y que esta quede vinculada a una entidad digital (República de Estonia, n.d.).

Pero cuando hablamos de una implementación de las cuestiones políticas de una manera tecnológica no nos referimos a poder realizar trámites administrativos desde nuestra casa o que nuestra entidad física quede ligada a una entidad electrónica que nos sirva para identificarnos como residentes de un país, nos referimos a si, de alguna manera, podríamos lograr una participación activa de la población en las decisiones, una democracia directa donde, en lugar de elegir a nuestros representantes, podamos ser los habitantes los que realmente votemos estas decisiones que, en ocasiones, pueden ser fundamentales.

Pero, ¿es realmente necesaria una solución tecnológica para esto? ¿No valdría con convocar un referéndum para tomar cada una de estas decisiones? Para contestar a estas preguntas fijémonos en Suiza ya que, por su modelo de gobierno, tiende a realizar bastantes referéndums. Si vemos los resultados de los últimos dos años (Consejo federal de la Confederación Suiza, 2022), podemos ver como en la mayoría de votaciones la participación apenas alcanza la mitad de la población y, en las más exitosas, no llegan a votar más de 3 de cada diez personas, por lo que, si hay una parte grande de la población que no está votando – en ocasiones mayoritaria - ¿podemos considerar el modelo suizo un modelo de democracia real?

A esto hay que hay que sumar el alto coste que tienen las votaciones (Portillo, 2019), ya que hay que movilizar medios, crear dispositivos policiales en las zonas donde puedan existir conflictos, pagar los salarios de los miembros de la mesa, … por lo que multiplicar ese coste de una vez cada 4 años – en el caso de las elecciones generales – a varias veces al año, tampoco sería conveniente y una solución tecnológica podría ser una opción para no aumentar, e incluso reducir, el coste.

Sin embargo, y a pesar de las ventajas, el voto electrónico tiene una gran desventaja frente al tradicional y es que un fallo de un componente (Ross, n.d.) en las máquinas de votación o una falla de seguridad en el *software,* dejaría al sistema vulnerable de manera que podría ser atacado lo que desacredita nuestro objetivo de crear un sistema para implementar una democracia real (Scott, 2019).

Vistos todos estos puntos, tanto a favor como en contra del proyecto – y otros muchos que no se van a discutir aquí, pues este Trabajo de Fin de Grado es del Grado en Ingeniería Informática y no ha lugar a discutir estos puntos – solo nos queda preguntarnos, ¿podemos crear un sistema que sea seguro para votar de manera electrónica o las tecnologías no han avanzado lo suficiente para permitirnos alcanzar este objetivo?

Objetivos del proyecto

Comúnmente, la mayoría de los Trabajos de Fin de Grado del Grado en Ingeniería Informática suelen ser el desarrollo de una aplicación funcional siguiendo una metodología de Ingeniería de Software que nos permita llevar una documentación del proyecto, de manera que pueda existir una trazabilidad entre lo que se pide y lo que se ofrece. Este proyecto, por supuesto, también persigue ese objetivo como se plantea al final de la introducción, pero plantea algunos retos adicionales.

Por un lado trataremos de explorar el desarrollo de aplicaciones web. Este es un tipo de aplicaciones dentro del sector de la informática que, sin lugar a dudas, ha crecido a gran ritmo en los últimos tiempos, principalmente por una mayor penetración de Internet en los hogares y la aparición de dispositivos como los *smartphones,* que nos permiten una conexión constante. Es por estos factores que las tecnologías de desarrollo web están en auge en lo que a requisitos para el mundo laboral se refiere (Logan dev, 2022) y es por ello que resulta interesante explorar este tipo de desarrollos, ya que están muy solicitados en el mercado de trabajo.

Por otro lado trataremos de crear un producto utilizando de manera casi exclusiva tecnologías de código abierto. Cada vez es más y más habitual ver cómo grandes empresas controlan proyectos que se posicionan como la médula espinal de la industria – React pertenece a Facebook, Angular a Google, Java a Oracle, … - y por supuesto están dispuesto a luchar contra el uso ilegítimo de sus productos cuando así lo consideren (Reuters, 2010); además, en los últimos meses ha habido una gran cantidad de proyectos de código abierto que, ante esta situación y a modo de rebeldía, han cerrado (Fadilpašić, 2022) y es una cuestión que no parece que vaya a terminar. Es por ello que nos podemos preguntar, ¿hasta qué punto podemos continuar realizando proyectos que se sostengan, mayoritariamente, sobre código abierto y que sean más comunitarios que empresariales? Este Trabajo tratará de ser un ejemplo de que aún, es posible.

Conceptos teóricos

En este apartado trataremos los conceptos sobre los que tendremos que tener un conocimiento previo – y que por sus características no se considera que puedan ser de conocimiento general en el campo de la informática – para comprender el desarrollo de la aplicación.

Blockchain

Como hemos comentado en la Introducción el principal desafío que presenta el proyecto es que vamos a desarrollar una aplicación que maneja información altamente sensible que, en caso de ser corrompida, podría afectar ya no solo nuestro sistema de una manera aislada, si no a estructuras mucho más grandes, como gobiernos de países, que pueden no representar fielmente la opinión mayoritaria.

Por este motivo no podemos confiar en un sistema de datos centralizado, ya que estos pueden ser objetivo de ataques o sufrir fallos que provoquen una pérdida total e irremediable de los datos (Rosemain & Satter, 2021) Para combatir estas desventajas se suelen implementar políticas de copias de seguridad, almacenamiento de datos redundante (RAID), … pero este tipo de políticas y tecnologías suelen llevar asociado un aumento del coste, por lo que tampoco se pueden escalar *ad infinitum*, volviendo al problema de que, si se pierden suficientes nodos, de nada nos serviría.

No parece que exista una buena solución para almacenar los datos, o al menos las partes más sensibles (como son los votos en la votación) pero, ¿podríamos utilizar el principio de descentralización de la *blockchain* para almacenar datos?

Cuando hablamos de *blockchain*, lo primero que se nos viene a la cabeza son las criptomonedas, lo cual es justo pues sin lo uno no existiría lo otro, pero todo ese tema está muy relacionado con fraudes, esquemas piramidales, … (Hetler, 2022) por lo que no es una tecnología que goce de confianza. Sin embargo, si exploramos lo que es la tecnología de *blockchain*, podremos ver que puede llegar a ser muy útil, especialmente para nuestro caso, y que se basa en unos principios muy básicos.

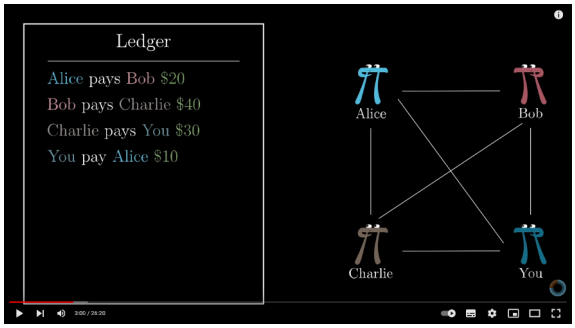


Figura 1: Creación de un libro de contabilidad (3Blue1Brown, 2017)

Para entender cómo se llega a lo que se llama la *blockchain*, empecemos suponiendo que 4 amigos tienen un libro de contabilidad común y en este libro apuntan lo que se van debiendo unos a otros, es decir, apuntan sus transacciones.

El primer problema que nos encontramos es que nada evita que uno de los amigos ponga varias veces que otro de los amigos le paga 3000 euros, por ejemplo. Para evitar esto lo que se hace es que cada uno de los amigos tenga un par de claves público/privada de manera que pueden firmar un mensaje con la clave privada, que debe mantenerse en privado, lo que producirá una firma para un mensaje; y se puede verificar que la firma pertenece a ese mensaje – y exclusivamente a ese mensaje – con la clave pública, que puede ser vista por todos.

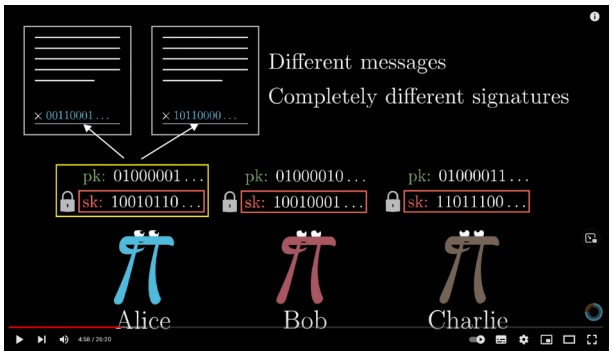


Figura 2: Firma con claves público/privadas (3Blue1Brown, 2017)

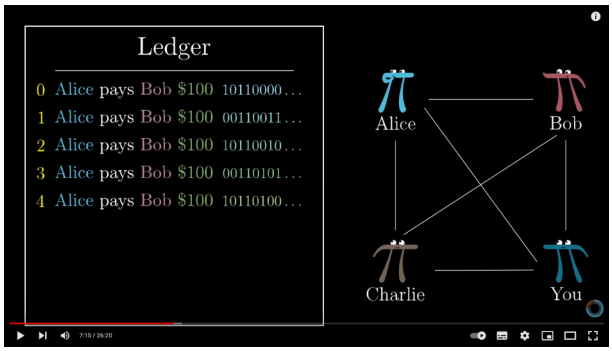


Figura 3: Numeración de transacciones (3Blue1Brown, 2017)

Como cada transacción crea una firma distinta, no podremos cambiar una transacción para que si, originalmente, nos debían 100 euros, ahora nos deban 3000 pues debería cambiar la firma. Sin embargo, si podríamos copiar la misma transacción varias veces pero, para evitar esto, podemos sencillamente numerar las transacciones y de esa manera todas las transacciones serían únicas y, por lo tanto, deberían producir una firma única.

Si suponemos que estos amigos lo único que hacen es deberse dinero unos a otros no necesitaríamos hablar en euros, ya que cada uno podría poner una cantidad en un bote al comenzar el libro de contabilidad y, con asegurarnos de que nadie se queda sin dinero dentro de nuestro sistema, todo iría bien. En este punto hemos conseguido una desconexión entre el dinero *fiat* o fiduciario y nuestro sistema de contabilidad, ya que solo nos intercambiamos dinero entre nosotros en este sistema.

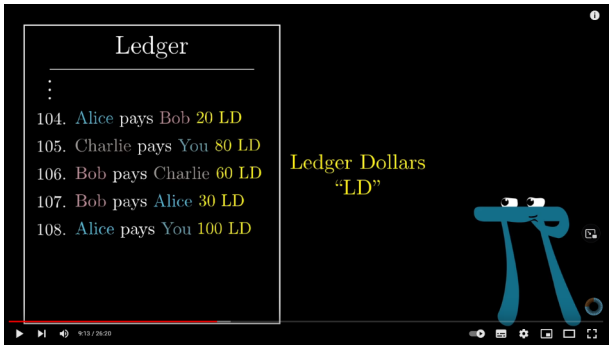


Figura 4: Creación de una criptomoneda (3Blue1Brown, 2017)

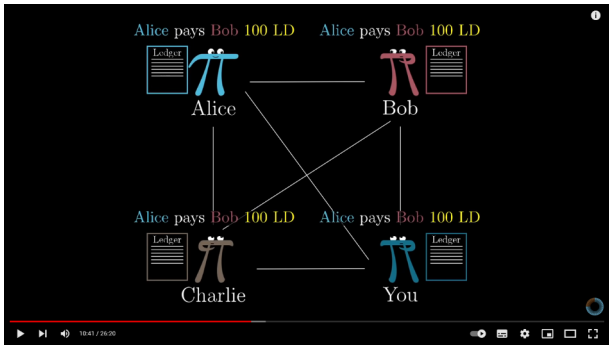


Figura 5: Descentralización de la contabilidad (3Blue1Brown, 2017)

Aunque ya no usamos dinero respaldado por una entidad central, como un banco, seguimos dependiendo de un libro de contabilidad por lo que, para evitar que este se pueda perder o destruir, hacemos una copia para cada uno de nuestros amigos y, cada vez que uno realice una transacción, el resto la deberán apuntar.

Pero, ¿qué pasa si uno de mis amigos no decide apuntar una transacción de que le debe dinero a alguien? ¿cómo sabemos de que libro de contabilidad fiarnos? Este es el principal punto del *paper* de Bitcoin (la primera criptomoneda) y lo resuelve estableciendo que siempre hay que fiarse de la *blockchain* que tenga mayor trabajo computacional (Nakamoto, n.d.)

Para saber esto lo que hacemos es que cada cierto tiempo todas las transacciones se juntarán en un bloque, que además estará unido al anterior para no perder el histórico (de ahí el nombre de *blockchain* o cadena de bloques), y se calculará un número especial llamado *nonce* el cual, cuando se añade al resto de bits que forman el bloque, al pasarlo por una función *hash* – que transforma un mensaje en un resumen de bits y que es único para cada menaje – devuelve un resumen que tiene un número determinado de ceros al principio. Como la única manera de saber qué *nonce* debe llevar cada bloque para que, al pasarlo por la función *hash,* nos devuelva el resumen con un determinado número de ceros es mediante prueba y error, se considera esta la prueba del trabajo computacional del bloque, pues es más fácil obtener este número si hay 3 ordenadores calculándolo de manera paralela, que uno solo que quiere introducir transacciones no aceptadas por la *blockchain*.

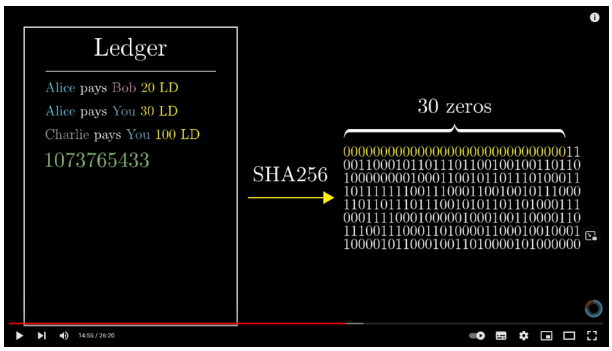


Figura 6: Prueba de trabajo computacional (3Blue1Brown, 2017)

Como hemos visto en el ejemplo (3Blue1Brown, 2017), cada vez que el libro de contabilidad se hacía más publico (cualquiera puede escribir, todos tienen una copia, …) hemos ido perdiendo confianza y hemos sustituido esa confianza por algún tipo de mecanismo tecnológico (claves de firma público/privada, prueba de trabajo computacional, …). Para nuestro proyecto esta es una solución ideal ya que no podemos confiar en un sistema central, así que confiaremos en la *blockchain*.

Contratos inteligentes

Sin embargo, es evidente que aún queda un pequeño problema porque la *blockchain* sirve para realizar transacciones, pero en ella no podemos realizar un manejo del estado de una aplicación y esto era cierto en el pasado. Ahora ciertas *blockchain* han pasado de ser un libro de contabilidad distribuido a una máquina de estado distribuida.

Las primeras *blockchain*, como Bitcoin funcionaban siguiendo el ejemplo que se ha visto: un gran libro de contabilidad donde los únicos mensajes que tenían cabida es que X le daba una cantidad Y de Bitcoins a Z.

Con el paso del tiempo se vio que la *blockchain* podía ser muy útil para construir aplicaciones que funcionarán de manera descentralizada. Existía una demanda por una tecnología que permitiera realizar este tipo de proyectos y las *blockchain* se han establecido como una manera de ganar dinero de manera fácil, por lo que había una gran cantidad de usuarios dispuestos a realizar el trabajo computacional a cambio de llevarse parte de la tasa que se cobra por realizar transacciones, era situación de *win-win*.

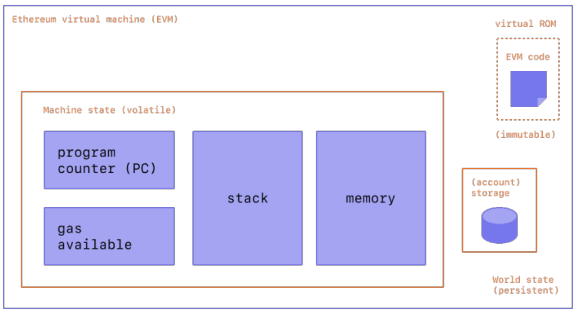


Figura 7: Ethereum Virtual Machine o EVM (minimalsm, 2022)

Así, nacieron las *blockchain* de nivel 2, como Ethereum o Polygon. Estas abandonaban las convenciones sobre que una *blockchain* no era más que un gran libro y en su lugar tenían una máquina de estado. En lugar ser una lista de transacciones de X le debe a Y, la *blockchain* recibía llamadas (entre las que se encuentra la función X le debe a Y) a funciones dentro del código de la máquina de estado de manera que, cuando se creaba el bloque, lo que se hacía era ejecutar ese mensaje en lugar de, simplemente, confirmar la transacción.

Esto dió cabida a la creación de los *smart contracts* o contratos inteligentes. Estos, como su propio nombre indica, son pequeños contratos, que es código que se ejecuta de manera automática cuando se cumplen ciertas condiciones.

Por ejemplo, supongamos que queremos comprar una casa, en tal caso tendríamos que tener un contrato que define que intercambiamos X dinero a cambio de una casa a la persona Y. Con un contrato inteligente esto sería mucho más sencillo ya que ahora, es posible que Y no reciba el pago, que la casa no esté en la condición ofertada o que Y nos pida parte del dinero en negro pero, con un contrato inteligente, la propiedad de la casa – con todas sus características – forman parte de la *blockchain* y, en cuanto Y reciba X dinero, la propiedad de la casa se actualiza dentro de la máquina para que pase a ser nuestra.

Esta solución es perfecta para nuestro caso ya que, además de que contamos con un soporte seguro, como ya hemos visto, para almacenar transacciones, ahora podemos almacenar cualquier tipo de información que, en nuestro caso, será los votos que tenga un candidato en una votación, que se representará como una instancia del contrato.

Meta transacciones

Hasta ahora vivimos en un mundo perfecto pero, en el entorno de la *blockchain*, cualquier mensaje que se emita a esta lleva asociado un coste. Este sobrecoste – llamado gas (minimalsm et al., 2022) – se plantea como un pago a la persona que se encarga de crear el bloque que contiene los mensajes o transacciones por el coste asociado al trabajo computacional realizado (coste de electricidad, componentes informáticos, …) para obtener el *nonce* del bloque.

El problema con esta cuestión es que dificulta la adopción por parte de los usuarios del uso DApps (o aplicaciones descentralizadas, como se llaman a las aplicaciones que hacen uso de la *blockchain*), ya que no todos tus usuarios puede que tengan una cartera con criptomoneda o estar dispuestos a pagar a cambio del uso de ciertos servicios.

En nuestro caso la adopción del sistema, que ya de por si es complicada, pues presentamos un sistema que puede generar cierta desconfianza, se complicaría más por lo que no nos podemos permitir que esa adopción sea menor por el hecho de exigir un pago para votar (además de romper con el principio de elecciones libres (art. 68 CE).

Es por esto que nuestro sistema hará uso de meta transacciones que, como resumen, podemos decir que funcionan de manera similar a lo mostrado en la figura de abajo a la derecha, aunque la figura y la explicación que hay a continuación no es técnica, pero si nos da una idea de las diferencias con las transacciones habituales.

Normalmente, cuando un usuario desea realizar una transacción, este debe utilizar algún tipo de cripto-billetera (como MetaMask), que le exigirá el pago de la transacción más el gas de esta, el cual es inevitable. Por último, si esta transacción devuelve algún valor, ese valor es devuelto al usuario.

En el caso de las meta transacciones, cuando un usuario desea realizar una transacción, no es este el que realiza el pago, si no que se delega a una cuenta relé, que será la que realicé la transacción en nombre del usuario, sin suponer ningún gasto para este. Por último, de igual manera que en el caso anterior, si se devuelve cualquier valor, ese valor es devuelto al usuario.

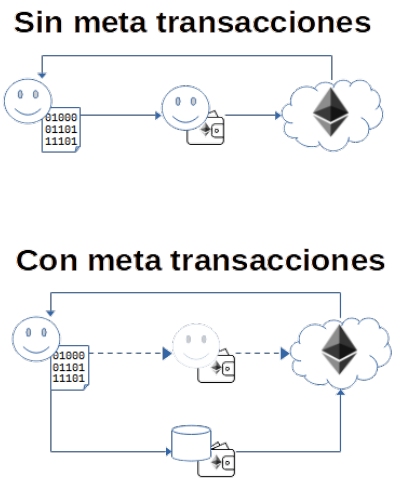


Figura 8: Esquema de metatransacciones

De esta manera conseguimos que el usuario pueda interactuar con el contrato inteligente, que simboliza nuestras votaciones en la *blockchain,* sin que necesite realizar ninguna inversión, permitiendo aumentar considerablemente la tasa de adopción del sistema.

Técnicas y herramientas

En este apartado se discutirán las distintas técnicas y herramientas que se han usado durante el desarrollo, dando un información básica sobre estas y explicando por qué se han elegido en lugar de otras que podemos ver como su competencia.

OpenUP

OpenUP es una metodología de desarrollo de software que, de manera similar a RUP (*Rational Unified Process*), describe un ciclo de vida iterativo. Al contrario que RUP, del cual deriva, trata de potenciar desarrollos siguiendo una filosofía ágil enfocándose principalmente en la naturaleza colaborativa del desarrollo de software, en el que tienen lugar múltiples disciplinas (Eclipse Software Foundation, 2012).

OpenUP sigue un ciclo de vida iterativo que divide en cuatro fases:

- una primera fase de conocimiento del proyecto, donde podremos definir cuál será su alcance, los objetivos de este o los integrantes del equipo de desarrollo;
- una segunda de elaboración, donde pasamos a definir de manera formal los requisitos y generamos el diseño de la aplicación;

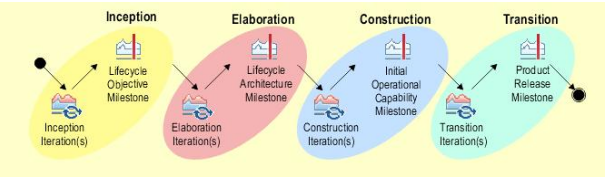


Figura 9: Ciclo de vida de OpenUP (Eclipse Software Foundation (2012)

- una tercera de implementación, donde, partiendo del diseño y los requisitos ya disponibles, implementaremos el código que dará funcionalidad a la aplicación;
- y, por último, una cuarta fase de transición donde dejaremos desplegado lo conseguido hasta el momento y empezaremos a plantear líneas de trabajo futuras.

Otro de los puntos claves de OpenUP es que, en lugar de dar una estructura rígida de fases, entregables y roles, permite que los usuarios adapten – y animan a ello – el proceso para que se amolde a las necesidades del proyecto.

Como ya se ha comentado este proceso deriva de RUP, desarrollado por Rational bajo el mando de IBM, y que es un estándar en la industria. Al principio, OpenUP se conocía como BUP (*Basic Unified Process*) y, tras ser donado por IBM a la Eclipse Foundation, pasó a considerarse como un proyecto de código abierto («OpenUP», 2020).

Este último punto es el que le da la mayor ventaja para ser la metodología de desarrollo que hemos escogido para nuestro proyecto. Al contrario que otros procesos, como RUP, que pertenecen a empresas privadas, toda su documentación es pública. Esto también lo hace especialmente fuerte frente a otras metodologías de desarrollo, como *eXtreme Programming* (XP) o *Agile*, ya que estos procesos no tienen una documentación formal y su aplicación sobre un proyecto se deja a elección del usuario, no definiendo en ningún momento realmente que supone un proceso *Agile* o XP.

Aunque para un usuario novato, por su gran cantidad de información es una metodología atractiva, también cuenta con sus desventajas, principalmente la falta de comunidad que hay entorno al proceso. Tenemos que tener en cuenta que ya es difícil encontrar documentación real utilizando otras metodologías de desarrollo pero para OpenUP, el cual es un jugador menor dentro de este grupo, será aún más difícil.

Django

Django es un *framework* de trabajo basado en Python que se enfoca principalmente en permitir diseñar aplicaciones de manera rápida pero sin sacrificar otras métricas como rendimiento, seguridad o escalabilidad (Django Software Foundation, n.d.).

La ventaja de Django como *framework* para desarrollar aplicaciones se encuentra principalemente en su arquitectura, la cual veremos más adelante, pues obliga a que las aplicaciones sigan un cierto modelo que resulta sencillo de comprender, eliminando por el camino decisiones complicadas de tomar, simplificando el proceso de desarrollo.

Además de esto, otro punto a favor de Django es que goza de una gran popularidad, basándonos en métricas como los proyectos de GitHub o las preguntas en StackOverflow relativos a este *framework*, por lo que no debería ser difícil encontrar ayuda por parte de la comunidad en caso de duda.

Respecto a su fiabilidad como *framework*, uno de los proyectos más populares creados con Django es Instagram (Jindal, 2021), llegando sus fundadores a afirmar a veces que, cuando comenzaron con la aplicación, tardaron un par de semanas de pasar del concepto a una demo real y que, al serles tan fácil la adopción del sistema, decidieron continuar con él.

Si a todo lo comentado hasta ahora le sumamos que Django es un proyecto de código abierto que es propiedad de la Django Software Foundation, la cual está inscrita como una organización sin ánimo lucro (Django Software Foundation, n.d.), esto hace de Django una de las mejores opciones sobre la que construir nuestra aplicación teniendo en cuenta nuestros objetivos.

Por supuesto, a pesar de todas estas ventajas, cuenta con sus desventajas, como que, a pesar de contar con una buena comunidad en base a su popularidad, esta no llega a ser tan grande como las de React o Angular, para las que existe todo tipo de ayuda y *plugins* para resolver cualquier necesidad, cuestión que en Django, en algunas ocasiones, puede no ser así.

Además tenemos que tener en cuenta que Django está escrito en Python y, aunque no se incluye como punto de estudio para este Trabajo, si que podemos tener en cuenta que Python es uno de los lenguajes menos amigables con el medio ambiente (Zetterlund, 2021) por lo tanto, aunque sigue siendo la mejor opción para nuestro proyecto, se podrían plantear otras opciones que sean igualmente libres pero más amigables con el medioambiente.

Django Rest Framework

Django, como ya hemos comentado, cuenta con una gran popularidad y esto hace que se desarrollen *plugins* de todo tipo para aumentar sus funcionalidades. Dentro de la variedad de *plugins* con los que cuenta Django, nos encontramos con uno que será especialmente útil para nuestra aplicación que es Django Rest Framework o DRF para abreviar.

Como veremos más adelante la aplicación se dividirá en dos componentes. Por un lado, uno de esos componentes será una aplicación al uso; pero, por otro, parte de la funcionalidad se delegará a una API que gestionará los datos sobre nuestras votaciones para así separar la lógica de datos, de la que se ocupará esta API, de la lógica de presentación.

Su principal ventaja es una que vemos en los *frameworks* basados en JavaScript y es que, si ya contamos con un lenguaje para desarrollar la lógica *front*, porqué no aprovechar ese lenguaje para desarrollar la lógica *back*. De igual manera aquí, si ya contamos ya no con un lenguaje, si no con un *framework* completo para desarrollar una aplicación, porqué no aprovecharlo también para desarrollar una API si así se desea.

Evidentemente, y como hemos comentado con el resto de tecnologías, esta sigue los principios del código abierto y, aunque es cierto que se pública bajo licencia con *copyright* permite su uso libre de en aplicaciones de todo tipo.

En cuanto a desventajas contra sus competidores no podemos encontrar ninguna que sea destacable ya que no hemos encontrado soluciones que sean tan específicas a la hora de crear una API.

Docker

Docker es la plataforma que utilizaremos para contenerizar nuestra aplicación. El hecho de contenerizar una aplicación permite que la aplicación puedan ser desplegadas de manera rápida y fiable, pudiendo mover la implementación de un entorno a otro (Docker, n.d.).

Estas características se derivan de que Docker, lo que hace realmente, es crear una imagen de lo que podríamos tener en un entorno de desarrollo, es decir, comprime el entorno de desarrollo, incluyendo todas sus dependencias, a una sola imagen que después puede ser desplegada, de manera aislada evitando así conflictos de dependencias, en cualquier lugar que tenga soporte para ejecutar el demonio de Docker.

Con respecto a sus ventajas y desventajas no merece la pena entrar a valorarlas pues su competencia se reduce a Kubernetes – que es propiedad de Google por lo que para nuestro proyecto no es una opción válida – y poco más ya que, aunque es una tecnología bastante práctica no ha conseguido tener relevancia más allá del mundo de los DevOps.

Aspectos relevantes del desarrollo del proyecto

A continuación, se discuten los aspectos más relevantes del desarrollo. Esta parte del proyecto se divide intentando seguir la misma estructura que sigue el ciclo de vida del proyecto. Así, nos encontraremos aspectos relevantes durante el inicio del proyecto, que compone la mayor parte de las tareas de gestión de este, la elaboración, en la cual definiremos como construir la aplicación, la construcción, que nos dará como resultado final el código ejecutable de nuestra aplicación, y, por último, el despliegue, que lanzará nuestra aplicación al mundo real.

Aspectos relevantes del inicio del proyecto

Durante este primer punto incidiremos en aquellos aspectos que hayan supuesto un reto por su novedad o complejidad para nosotros al comenzar el proyecto, siendo estos puntos principalmente relacionados con tareas de gestión y la definición de objetivos y que se encuentran estudiados en el Anexo I o que permean toda la aplicación al sentar las bases para continuar el desarrollo.

Adaptación de OpenUP a la práctica

Aunque no está recogido en el Anexo I pues se considera que se sale del alcance de dicho documento, OpenUP propone a sus usuarios que, partiendo de las bases de su metodología, adaptarlo de manera que se amolde a las características – como número de recursos, experiencia del equipo o tiempos de desarrollo – del proyecto, teniendo incluso una tarea con guías sobre como tratar este tema.

El primer cambio que realizaremos y, quizá, el más fundamental de todos será el de pasar de un ciclo de vida 4 fases a un ciclo de vida de 3 fases – aunque en los documentos sigan figurando cuatro fases – (inicio del proyecto, elaboración / construcción del proyecto y despliegue del proyecto).

Este primer cambio viene principalmente motivado por los objetivos del proyecto: desarrollar una aplicación web con herramientas de código abierto y que sigan filosofías libres. Aunque ya hemos desarrollado aplicaciones web, estas siempre se han basado en *frameworks* como Next.js – basado en React, desarrollado por Facebook –, Angular – desarrollado por Google – u otros *frameworks* propietarios y esta será la primera que desarrollaremos una aplicación con Django; una decisión que, como veremos más adelante, influye bastante en el diseño de nuestra aplicación; por lo que no nos podemos cerrar a que nuestro diseño no evolucione con el desarrollo.

Revisión de las prácticas propuestas

OpenUP define una serie de prácticas que los usuarios pueden optar por seguir o no, dependiendo de qué se adapte mejor a cada proyecto. Estas prácticas se dividen en prácticas de gestión, técnicas y de despliegue de las cuales veremos a continuación cuáles eliminaremos, cuáles mantendremos y cuáles modificaremos.

De las prácticas de gestión eliminaremos las siguientes:

- *Risk-Value Lifecycle* pues, como se ve en el Anexo I, al ser el equipo de desarrollo tan pequeño y al no tener que defender frente a un cliente unos gastos de desarrollo, podemos obviar la necesidad de enfocar nuestro ciclo de vida a unas entregas que aporten valor.
- *Team Change Management* se elimina por la finalidad del proyecto, pues este no deja de ser un Trabajo de Fin de Grado, y no corremos el riesgo de necesitar realizar cambios en el equipo

Aunque pudiéramos pensar en eliminar la práctica de *Whole Team* – ya que todo el equipo esta compuesto por una persona – la mantendremos por proponer tareas importantes para todo tipo de equipos como mantener una reunión diaria de seguimiento, una auto-organización de las tareas, …

De las prácticas técnicas mantendremos todas ellas ya que estas componen uno de los principales motivos por los que OpenUP resulta una metodología de desarrollo tan interesante, pues incluye prácticas como *Test-Driven Development* que veremos más adelante.

Por último, de las prácticas de despliegue, solo mantenedremos la denominada *Production Release*, eliminando *Documentation and Training* pues este Trabajo tiene un enfoque más experimental y de exploración de tecnologías y herramientas y no se ve necesaria crear documentación especifica para soporte, entrenamiento, usuario final, …

Revisión de los entregables propuestos

OpenUP define una serie de entregables o, como ellos denominan, productos de trabajo que no dejan de ser los documentos que se generan por el uso de la metodología. De estos, al igual que con las prácticas, eliminaremos, mantendremos y modificaremos los entregables que sean necesarios para que la metodología se adapte mejor a nuestro proyecto. Distinguimos entre:

- entregables de arquitectura, que solo cuenta con el Cuaderno de Arquitectura, que mantendremos pues recoge toda la arquitectura y, si lo eliminamos, nos quedaríamos sin arquitectura para el proyecto.
- entregables de despliegue de los que eliminaremos todos excepto el Plan de Despliegue pues no consideramos necesario crear documentación específica como ya hemos visto en las prácticas.
- entregables de desarrollo de los que mantendremos todos excepto el *Build* pues, al tratarse de una aplicación web en Django, no necesita ir compilada previamente

- entregables de entorno de los que mantendremos todos pues, aunque no aparezcan como si aparecen otros en los anexos, conforman buena parte de esta memoria
- entregables de gestión de los que mantendremos todos – incluyendo la Lista de Tareas que se puede ver aquí – excepto la Lista de Riesgos pues, como ya se ha comentado, al no ser expertos en las herramientas que se usan, no se puede hacer una buena estimación de estos
- entregables de requisitos de los que mantendremos todos excepto el Glosario pues, en este mismo documento, se intenta recoger y explicar los conceptos que pueden resultar más complicados.
- por último, de los entregables de testeo, se mantienen solo los Casos de Prueba, pues el resto de entregables son exclusivos si automatizamos las pruebas, cosa que no vamos a realizar en este proyecto.

Revisión de los roles propuestos

OpenUP propone una serie de roles que, en teoría, se usan para definir a los miembros del equipo con suficiente conocimiento como para realizar ciertas tareas asociadas a un conjunto de responsabilidades.

En el caso de nuestro proyecto no tiene sentido discutir los roles pues, al estar nuestro equipo conformado únicamente por una persona, esta asumirá todos los roles.

Definición de alcance

Uno de los puntos que han sido más problemáticos a la hora del desarrollo, y sobre el que quizá se tendría que haber puesto un mayor enfoque, pues ha provocado que finalmente no se pueda tener una implementación completa, es la definición de alcance.

El principal problema que se ha tenido con este punto es que, para poder definir bien el alcance de un proyecto, se tiene que tener claro qué se quiere implementar, es decir, tenemos que saber qué objetivos perseguimos, qué requisitos tenemos, … lo cual en nuestro caso, por lo que se puede ver en el Anexo I y II, creemos que tenemos bastante claro; pero, tan importante es conocer qué se va implementar como cómo se va implementar y es aquí donde pecamos de novatos.

Como ya hemos explicado, este Trabajo tiene el objetivo de implementar un sistema de votación seguro pero tiene la ambición también de explorar territorio desconocido como metodologías de diseño que no se conocían antes, lenguajes de programación que tampoco o tecnologías que son raras hasta verlas en desarrollos reales. Esto, aunque hace más atractivo el Trabajo para el desarrollador, ya que no se ve atado a ciertas herramientas, también le perjudica por el hecho de que no sabe con que se va a encontrar.

Aún así consideramos que, en parte, se ha hecho una buena definición de alcance pues se han priorizado correctamente los requisitos de manera que, aunque no tengamos una implementación completa, si que hemos conseguido implementar dos de los tres objetivos que aparecen en el Anexo I de manera funcional, dejando el segundo objetivo listo para comenzar su desarrollo por lo que, aunque la definición de alcance se ha quedado grande, finalmente ha sido cuestión de que, especialmente en las tareas de documentación, no se conocía las herramientas que se iban a usar.

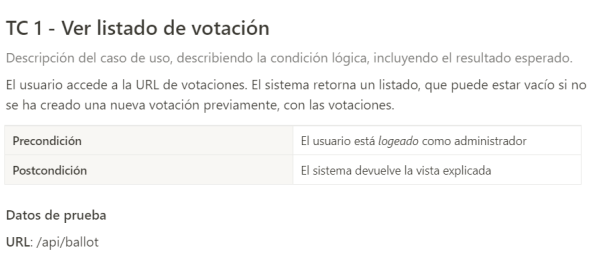
Aspectos relevantes de la elaboración del proyecto

Durante este segundo punto incidiremos en aquellos aspectos que hayan supuesto un reto por su novedad o complejidad para nosotros a la hora de definir los requisitos y crear una arquitectura para la aplicación, siendo estos puntos principalmente relacionados con tareas de diseño y análisis y que se encuentran especificados en el Anexo III.

Casos de prueba

Uno de los puntos que más se suelen obviar dentro de los Trabajos son las pruebas de software pues, aunque si que es cierto que se suelen probar las aplicaciones, estas pruebas no suelen estar estructuradas ni cuenta con una definición por lo que, al final, acaban quedando reducidas a lo que a los desarrolladores les parezca más oportuno probar.

En nuestro Trabajo, motivados en parte por las prácticas de OpenUP, sí incluiremos estas pruebas en el Anexo II que no hay que confundir con las pruebas de desarrollo, ya que estos casos de prueba pueden tratar diversos temas como funcionalidad, que es el tipo de pruebas que tenemos diseñadas en nuestro caso, pero también de seguridad o diseño.


TC 1 - Ver listado de votación

Descripción del caso de uso, describiendo la condición lógica, incluyendo el resultado esperado.

El usuario accede a la URL de votaciones. El sistema retorna un listado, que puede estar vacío si no se ha creado una nueva votación previamente, con las votaciones.

Precondición	El usuario está *logeado* como administrador
Postcondición	El sistema devuelve la vista explicada

Datos de prueba

URL: /api/ballot

Figura 10: Ejemplo de un caso de prueba diseñado para la aplicación

La diferencia clave entre estas pruebas de funcionalidad y las pruebas unitarias que se suelen incluir durante el desarrollo es que las primeras prueban directamente sobre la aplicación desplegada, mientras que las segundas prueban que la implementación sea correcta. Así, las pruebas de funcionalidad pueden descubrir fallos que no se habían visto antes bien porque no se habían implementado las pruebas unitarias adecuadas o bien porque estas no llegaban a cubrir todo el código.

En concreto, las pruebas de funcionalidad cuentan con tres partes:

- una primera parte donde se define la funcionalidad para la que se crea ese caso de prueba, especificando qué se va a probar en concreto incluyendo el resultado que tiene que dar el caso;
- una segunda parte donde se establece las precondiciones y postcondiciones que deben cumplirse para que se pueda probar el caso
- y una tercera parte donde se establecen los datos que el equipo de pruebas, que suele ser distinto que el equipo de desarrollo como se comenta a continuación, necesita para realizar las pruebas

Las pruebas de funcionalidad y de los otros tipos antes comentados suelen ser implementadas y ejecutadas por un equipo de aseguramiento de calidad, mientras que las unitarias suelen ser implementadas por los propios desarrolladores, que pueden tener una visión menos técnica y más centrada en la funcionalidad de la aplicación. En el caso de nuestro trabajo, al ser un único encargado de todas las tareas, no hemos podido cumplir con que existan estos dos roles (equipo de aseguramiento de calidad y equipo de desarrollo), aunque si se ha preguntado a personas expertas en el tema si los casos de prueba estaban correctamente diseñados.

Django y el modelo arquitectónico MTV

El patrón MTV es el patrón arquitectural que usa Django y que debe ser tenido en cuenta a la hora de desarrollar la aplicación.

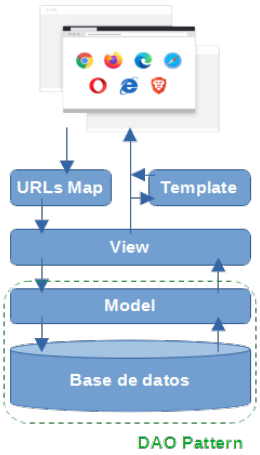


Figura 11: Funcionamiento interno de Django

Como se puede ver en la figura de la izquierda el funcionamiento es el siguiente:

- cuando llega una nueva petición al servidor, esta se envía al mapa de URLs, que recupera la vista y el método que corresponde a la URL a la que se accede
- la vista solicita los datos necesarios al modelo, el cual los recupera desde la base de datos
- una vez el modelo ha recuperado los datos, se los devuelve a la vista, el cual los inyecta en la plantilla
- por último, la plantilla es retornada como respuesta a la petición, con los datos que la vista ha recuperado desde el modelo.

El modelo arquitectónico que presenta Django nos puede recordar al patrón Modelo-Vista-Controlador pero, si nos fijamos, no contamos con un controlador pero parece que la vista toma parte de sus tareas entonces, ¿qué patrón arquitectónico usa Django, si es que usa alguno?

En su propia documentación (Django Software Foundation, n.d.) los desarrolladores de Django nos dicen que siguen el patrón MVC pero especifican que, en Django, la vista define *qué* datos se van a mostrar, delegando después en la plantilla para saber *cómo* se mostrarán esos datos; por otro lado, defienden que el controlador sería el propio *framework*, que realiza el *routing* de las URLs a las vistas específicas, mientras que el modelo queda igual que en MVC. Esto es, lo que ellos llaman, el modelo MTV.

También, como vemos en la figura, nos gustaría destacar que Django implementa el patrón DAO a la hora de comunicarse con la base de datos, de esta manera la implementación de la lógica de datos es independiente de la base de datos que se utilice y cuenta con mecanismos, como las migraciones, para poder mover los datos de una base a otra, independientemente del tipo que sean.

Compartición del proyecto

Una de las decisiones que más afectan a cómo se construirá el proyecto se toma durante la elaboración de la arquitectura, en el Anexo III, y es que pasaremos de tener una aplicación monolítica a tener una aplicación dividida en componentes.

La realidad es que la informática es un ámbito que avanza rápidamente y aquello que era puntero ayer, hoy ya ha perdido el efecto de novedad frente otra nueva tecnología, y mañana ya estará obsoleto. Esto hace que sea casi imposible mantener una aplicación utilizando siempre tecnología puntera.

Este es un problema que nos parece interesante tratar de solventar, pero no solamente eso si no que, también, en aplicaciones monolíticas corremos el riesgo de que, si cae alguno de los componentes críticos, caiga el resto de la aplicación. Utilizando el caso de nuestra aplicación como ejemplo, si cayera la base datos en la que se almacenan los datos de las votaciones no podríamos entrar de ninguna manera a la aplicación, ya que una de las primeras comprobaciones que hace Django es comprobar que la conexión a la base de datos es válida.

La manera que tiene nuestro proyecto de solventar estos problemas es dividir la aplicación en dos componentes. De esta manera contaremos con un componente que se encargue de llevar la lógica de presentación (Interfaz de votación) y otro que se encargue de llevar la lógica de datos (Repositorio de votación o API). Así, si no pudiéramos acceder a la información de las votaciones, podemos recuperarnos de ese error de una manera más grácil.

Esta solución, como decimos, también nos permite desarrollar estos dos componentes de manera independiente. Así si, por ejemplo, debiéramos dar cabida a un nuevo campo para las votaciones, podríamos actualizar la API para que retorne dicho campo y lo solicite durante la creación de dichos objetos, sin importar que la vista haya añadido o no el soporte para mostrar dicho campo. O si, por ejemplo, quisiéramos migrar la parte de la Interfaz a un *framework* basado en componentes montado sobre NodeJS, también podríamos sin necesidad de modificar el código que controla la gestión de las votaciones.

Así pues la estructura de nuestra aplicación queda como se muestra en la figura, con dos componentes donde cada uno gestiona su propia base de datos. Estos se comunican entre sí mediante el punto de acceso que hace público la API. Por otro lado, se comunican también de manera independiente con la *blockchain* para distintas tareas: la Interfaz para emitir los votos y la API para instanciar los contratos que representan las votaciones y recoger los resultados de los candidatos cuando finalice la votación.

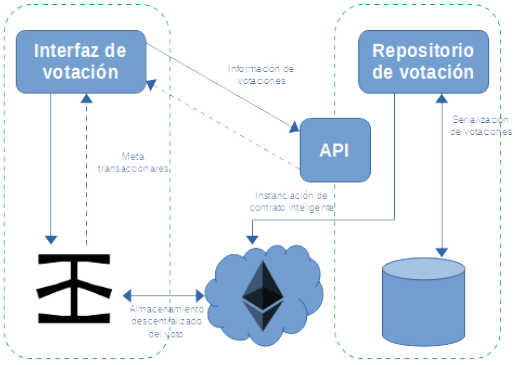


Figura 12: Estructura de la aplicación por componentes

Aspectos relevantes de la construcción del proyecto

Durante este tercer punto incidiremos en aquellos aspectos que hayan supuesto un reto por su novedad o complejidad para nosotros a la hora de implementar la aplicación, siendo estos puntos principalmente relacionados con tareas de codificación y que se tratan con mayor profundidad en el Anexo IV.

Test-Driven Development (TDD)

A la hora de desarrollar código, normalmente, los desarrolladores buscan desarrollarlo con la mayor brevedad posible, muchas veces debido a que hay que cumplir con fechas de entregas muy ajustadas y, a la hora de planificar el proyecto, no se suele atender a la necesidad de crear pruebas unitarias.

Las pruebas unitarias son parte de código cuya función no es aportar funcionalidad a la aplicación, si no que tratan de asegurar que el código realmente funciona de la manera esperada, siguiendo normalmente las definiciones de los requisitos.

Para que un producto siga el desarrollo basado en pruebas, deberá seguir los siguientes pasos:

1. Escribir el test: basándonos en los requisitos, deberemos escribir una o más pruebas unitarias que cubran las situaciones que definen los requisitos.

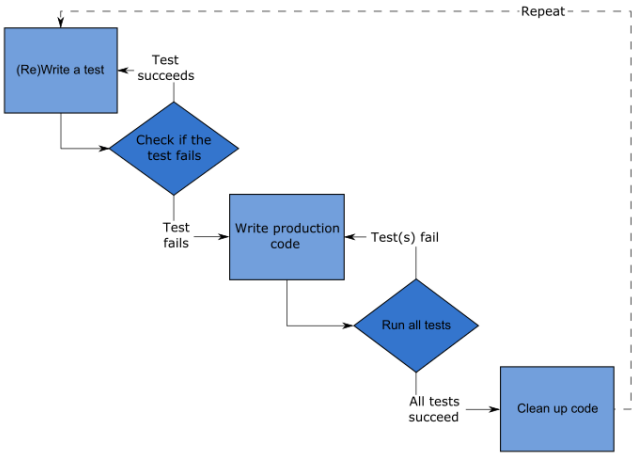


Figura 13: Esquema del desarrollo basado en tests (Excirial, 2016)

2. Ejecutar el test: una vez hemos escrito el test, lo ejecutaremos y, como es lógico, en una primera pasada, el test debería fallar, pues no hay ninguna funcionalidad implementada que las pruebas unitarias puedan realmente probar. Si el test no falla se debe volver al paso 1, pues no está bien creado y por lo tanto hay que reescribirlo.
3. Implementar el código: el desarrollador debe implementar el código de manera que, cuando se ejecute el test, este consiga superarse.
4. Ejecutar todas las pruebas unitarias: ejecutaremos todas las pruebas unitarias que, tras haber realizado la implementación, deberían pasar. Si no pasan, deberemos volver al paso 3, pues el código que se ha implementado no es correcto.
5. Limpiar código: por último, podemos refactorizar el código de manera que quede más limpio y legible para otros desarrolladores, dejándolo listo por si fuera necesario volver al paso 1 para implementar nueva funcionalidad.

Como podemos ver, el uso de esta técnica nos permite tener un código muy robusto ya que, si las pruebas unitarias están basadas en los requisitos, esto nos da una trazabilidad completa desde los objetivos del proyecto, a los requisitos, pasando por las pruebas unitarias, y llegando hasta el código final, por lo que si este falla, habría que replantear la aplicación desde cero pues no se trataría de errores de desarrollo, si no de errores de alcance.

Pruebas unitarias dentro de DRF

La parte de la API, implementada usando DRF, se ha implementado usando TDD siguiendo las recomendaciones de OpenUP.

```
class BallotViewTest(APITestCase):
    def test_get_inexistent_ballot(self): ···

    def test_get_ballot(self): ···

    def test_get_ballot_with_candidates(self): ···

    def test_get_finished_ballot_with_candidates(self): ···
```

Figura 14: Ejemplo de una clase de prueba unitaria

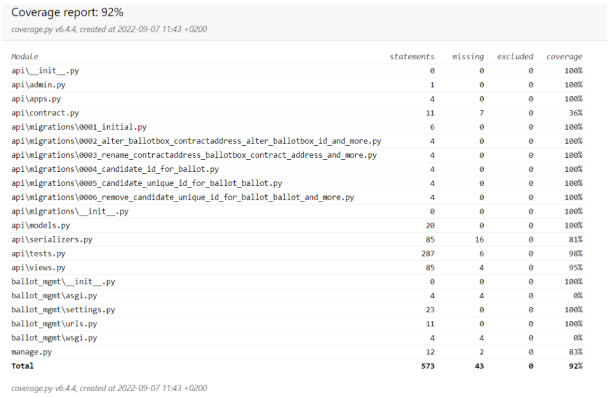
En nuestro proyecto las clases de pruebas unitarias se han generado atendiendo a dos cosas: la parte que se iba a probar y la funcionalidad que se iba a probar. Así, tenemos clases como *BallotViewTest* que contiene todas las pruebas unitarias relacionadas con ver el detalle de una votación. En estas clases valoramos también casos de excepción, como que la votación no exista, o casos secundarios, como que la votación haya terminado, por lo que debe mostrar los resultados de sus candidatos.

```
def test_get_ballot(self):
    start_datetime = timezone.now() + timedelta(hours=1)
    end_datetime = timezone.now() + timedelta(hours=10)

    ballot = BallotBox(
        name = 'Test',
        start_datetime = start_datetime,
        end_datetime = end_datetime,
    )

    ballot.save()

    response = self.client.get('/api/ballot/' + str(ballot.id))

    self.assertEqual(response.status_code, status.HTTP_200_OK)
```

Figura 15: Ejemplo de un test o prueba unitaria

Estas pruebas unitarias lo que harán será crear los datos que sean necesarios en una base de datos que Django crea específicamente para hacer este tipo de pruebas. Con ellos creados, al tratarse de una prueba para una API invocamos al cliente que viene incluido en la clase de la que heredan estos casos de prueba y le pedimos que haga una petición a la URL que estemos probando. Por último, cuando obtengamos el resultado, comprobaremos si el código HTTP devuelto por la API es el que se corresponde para el caso que se está probando (200 en caso de éxito, 400 en caso de error, 404 en caso de que no se haya encontrado el recurso, ...).

Así, utilizando esta metodología, conseguimos una cobertura de código muy alta y que es difícil de conseguir en caso de no seguir este proceso de escribir primero las pruebas unitarias. En nuestro caso, como vemos, no se alcanza el 100% aunque si se ha llegado a alcanzar pero, tras la integración con la *blockchain* no se han conseguido *mockear* los servicios que hacen uso de esta para que las pruebas unitarias que la usen puedan ejecutarse correctamente.



Coverage report: 92%

coverage.py v6.4.4, created at 2022-09-07 11:43 +0200

Module	statements	missing	excluded	coverage
api__init__.py	0	0	0	100%
api\admin.py	1	0	0	100%
api\apps.py	4	0	0	100%
api\contract.py	11	7	0	36%
api\migrations\0001_initial.py	6	0	0	100%
api\migrations\0002_alter_ballotbox_contractaddress_alter_ballotbox_id_and_more.py	4	0	0	100%
api\migrations\0003_rename_contractaddress_ballotbox_contract_address_and_more.py	4	0	0	100%
api\migrations\0004_candidate_id_for_ballot.py	4	0	0	100%
api\migrations\0005_candidate_unique_id_for_ballot_ballot.py	4	0	0	100%
api\migrations\0006_remove_candidate_unique_id_for_ballot_ballot_and_more.py	4	0	0	100%
api\migrations__init__.py	0	0	0	100%
api\models.py	20	0	0	100%
api\serializers.py	85	16	0	81%
api\tests.py	287	6	0	98%
api\views.py	85	4	0	95%
ballot_mgmt__init__.py	0	0	0	100%
ballot_mgmt\asgi.py	4	4	0	0%
ballot_mgmt\settings.py	23	0	0	100%
ballot_mgmt\urls.py	11	0	0	100%
ballot_mgmt\wsgi.py	4	4	0	0%
manage.py	12	2	0	83%
Total	**573**	**43**	**0**	**92%**

coverage.py v6.4.4, created at 2022-09-07 11:43 +0200

Figura 16: Código cubierto por las pruebas unitarias

Creación de la API en DRF

A continuación vamos a ver como hemos creado la API utilizando DRF. En esta sección mostraremos fragmentos del código como ayuda para explicar cada una de las partes de las que compone DRF pero, para tener un conocimiento profundo de la implementación se recomienda revisar el código fuente pues se hará referencia a cosas que no podemos mostrar por ser código muy largo.

```
router = routers.DefaultRouter(trailing_slash=False)
router.register(r'ballot', views.BallotBoxView, basename='ballot')
router.register(r'candidates/(?P<bk>\d+)', views.CandidateView, basename='candidates')
router.register(r'contract', views.ContractView, basename='contract')
```

Figura 17: Enrutado de nuestra aplicación

A modo de introducción general lo primero que hará DRF cuando reciba una petición será revisar en el mapa de URLs a que vista tiene que llamar basándose en el patrón de la URL. A continuación, en nuestro caso por el hecho de usar la clase *DefaultRouter* para crear nuestro *router*, DRF llamará a la función que el *router* vincula por defecto al verbo HTTP recibido; así, por ejemplo, si recibe una llamada con el verbo DELETE llamará al método *destroy* de la vista, pero si recibe una llamada con el método PATCH llamará al método *update*.

Dependiendo del tipo de llamada que haga es posible que sea necesario que devuelva unos datos para lo cuál la vista recogerá del modelo los datos que necesite y los pasará por el serializador de manera que sean manejables por DRF. Por otro lado, si la llamada es para crear nuevos datos en el modelo, la vista llamará al serializador con los nuevos datos que le llegan en la petición HTTP, siendo el serializador el encargado de crear dichos datos y validar que se pueden insertar en la base de datos.

Modelos

Los modelos son abstracciones de las tablas que se encuentran en la base de datos y tampoco hay mucho que contar sobre ellos. Para crear un nuevo modelo, basándonos en el Diseño de datos del Anexo III, definiremos una nueva clase que herede de la clase *Model* de Django y, dentro de esta, iremos especificando cada uno de los campos de los que se compone la tabla que irá en base de datos.

Figura 18: Ejemplo de un modelo en DRF

Para definir estos campos usaremos las clases que nos proporciona Django que son abstracciones de los tipos nativos que suelen manejar las bases de datos. Así si, por ejemplo, quisiéramos guardar una cadena de texto, tendríamos que crear un campo de tipo *CharField*.

Adicionalmente, contamos con la clase *Meta* dentro de la propia clase de modelo que nos permite definir otras propiedades que no son campos de las tablas, como la condición de unicidad que se ve en el ejemplo.

Una vez tengamos definido nuestros modelos deberemos llamar a

```
python manage.py makemigrations
```

que creará, dentro de la carpeta de migraciones, un archivo Python con la información sobre cómo ha cambiado la definición de la base de datos con respecto a su estado previo. Una vez tengamos el *script* de migración, lo aplicaremos llamando a

```
python manage.py migrate
```

para que la migración se haga efectiva en la base de datos.

Serializadores

Los serializadores, como se explica en el Anexo III, son clases que se encargan del *data mangling* de los datos, esto es, la traducción de los datos entre distintos tipos de representación como, por ejemplo, de un formato JSON a una instancia de una clase de modelo para qué pueda ser después manipulada por la vista.

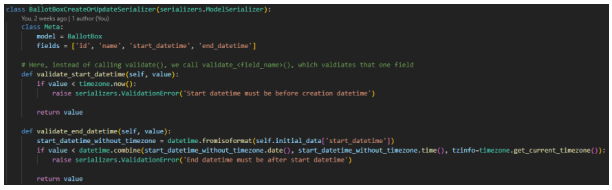


Figura 19: Ejemplo de un serializador

La manera más sencilla de definir un serializador es creando una clase que herede de *ModelSerializer* y, dentro de esta, definir la clase *Meta*. Dentro de esta clase tendremos que definir dos atributos: el modelo, que será una referencia a la clase del modelo que use el serializador, y una lista con los campos de dicho modelo que puede recibir el serializador bien para guardar o para devolver.

Es posible, también, que queramos recuperar datos que, en ocasiones, no estén definidos en el modelo (como, por ejemplo, el resultado de los candidatos) para lo cual deberemos crear un nuevo campo propio del serializador y deberemos especificar cuál es el origen de sus datos definiendo la función *get* seguido de barra baja y el nombre del campo para el que necesitamos definir la fuente de datos como se ve el ejemplo de la derecha.

Figura 20: Ejemplo de un serializador con un campo para el que hay que definir su fuente de datos

Como ya hemos comentado, el serializador también hace las veces de validador. Normalmente, a la hora de validar, solo comprobará que los datos recibidos son de los tipos adecuados para guardarlos en base de datos. Si queremos modificar este comportamiento tenemos dos opciones:

- Por un lado, podemos usar las funciones de tipo *validate* seguido de barra baja y el nombre del campo para el que se quiere añadir validaciones adicionales como vemos en el primer ejemplo.
- Por otro lado podemos reescribir la función *validate* por completo lo cual incluso nos permite introducir nuevos campos que puede que no nos lleguen durante la creación del serializador, si no usando el contexto

Además, como también se ha comentado, los serializadores son los encargados de crear nuevos datos para introducir en la base de datos. Este comportamiento puede ser también sobrescrito – sobrescribiendo el método *create* – para añadir pasos previos a la creación de la información en base de datos como, por ejemplo, para desplegar un nuevo contrato cuando se crea un nuevo candidato como en nuestra aplicación.

Vistas

El último punto que trataremos serán vistas que, como ya se ha explicado, serán las encargados de decidir qué datos se muestran, delegando después cómo se verán esos datos a las plantillas.

Para crear una vista deberemos heredar de la clase *GenericViewSet* que proporciona toda la base para generar una vista con un conjunto de funciones que respondan a los distintos verbos HTTP. Es recomendable, sin embargo, heredar también de *mixins*, que nos dan una implementación básica de la función que llamará el mapa de URLs según el *mixin* del que heredemos; o bien, heredar directamente de *ModelViewSet*, que nos da una implementación básica de todas las funciones que puede llamar el mapa de URLs.

```python
class BallotBoxView(viewsets.ModelViewSet):
    queryset = BallotBox.objects.all()
    permission_classes = [permissions.IsAdminUser|permissions.IsAuthenticatedOrReadOnly]
    authentication_classes = [authentication.SessionAuthentication, authentication.TokenAuthentication]

    def get_serializer_class(self):
        if(self.action == 'list'):
            return BallotBoxListSerializer
        if(self.action == 'retrieve'):
            return BallotBoxRetrieveSerializer
        if(self.action == 'create' or self.action == 'update' or self.action == 'partial_update'):
            return BallotBoxCreateOrUpdateSerializer

    def list(self, request, *args, **kwargs):
        return super().list(request, *args, **kwargs)

    def retrieve(self, request, *args, **kwargs):
        return super().retrieve(request, *args, **kwargs)

    def create(self, request, *args, **kwargs):
        return super().create(request, *args, **kwargs)

    def update(self, request, *args, **kwargs):
        ballot = shortcuts.get_object_or_404(BallotBox, id=kwargs['pk'])
        if ballot.start_datetime < datetime.now(timezone.utc):
            return response.Response("Votation " + str(ballot.id) + " has started and can't be changed", status.HTTP_409_CONFLICT)

        return super().update(request, *args, **kwargs)

    def partial_update(self, request, *args, **kwargs):
        return super().partial_update(request, *args, **kwargs)

    def destroy(self, request, *args, **kwargs):
        ballot = shortcuts.get_object_or_404(BallotBox, id=kwargs['pk'])
        if ballot.start_datetime < datetime.now(timezone.utc):
            return response.Response("Votation " + str(ballot.id) + " has started and can't be changed", status.HTTP_409_CONFLICT)

        return super().destroy(request, *args, **kwargs)
```

Figura 21: Ejemplo de una vista en DRF

Además de esto, al crear la vista, deberemos definir varios atributos como hacemos en el ejemplo:

- *serializer_class* especifica que serializador usa la vista. Si queremos usar distintos serializadores dependiendo de la acción que se realiza, podemos usar la función *get_serializer_class* y, dependiendo de la acción que esté siendo llamada, devolver un serializador u otro
    - además de *get_serializer_class,* disponemos de la función *get_serializer_context* por si, dependiendo de la acción, queremos pasar información adicional al serializador en el contexto
- *queryset* especifica el conjunto de datos que usará la vista. Si queremos hacer comprobaciones adicionales para poder definir el *queryset* para páginas que puedan llevar más de un parámetro – como la vista de candidatos en nuestra aplicación – podemos utilizar la función *get_queryset* y devolver un conjunto de objetos que decidamos dinámicamente
    - además de *get_queryset*, disponemos de la función *get_object*, que sigue el mismo principio que la anterior pero cuando solo tenemos que recuperar un objeto como en las vistas de detalle
- por último, los atributos *premission_classes* y *authentication_classes* definen los permisos necesarios para interactuar con la API (si se tiene que ser administrador, si se pueden ver las vistas de solo lectura aunque no se haya iniciado sesión, …) y que métodos de autenticación se pueden usar (sesión, *token*, autenticación básica HTTP, …), respectivamente

Una vez establecidos estos atributos podremos sobre escribir las acciones que se corresponden a los verbos HTTP cuyo comportamiento base queramos modificar de alguna manera o podemos dejar el comportamiento base.

Aspectos relevantes del despliegue del proyecto

Durante este cuarto y último punto incidiremos en aquellos aspectos que hayan supuesto un reto por su novedad o complejidad para nosotros a la hora de definir el despliegue, siendo estos puntos principalmente relacionados con tareas de diseño y ejecución del despliegue y que se encuentran especificados en el Anexo III en el apartado de Plan de desarrollo e implementación.

Docker y contenedores

Cuando se presentó Java se presentó bajo una premisa muy atractiva, la conocida como *compile once, run everywhere* (compila una vez, ejecuta en todas partes) ya que permitía compilar el código una sola vez y ejecutarlo en cualquier parte. Esto lo conseguía con el uso JVM; que permitía ejecutar el código que había sido compilado a *bytecode*, en lugar de a código máquina, en cualquier dispositivo que tuviera soporte para una JVM.

Con el paso del tiempo y el enfoque en los desarrollos web, aunque las dificultades no son las mismas que con Java, si que hay un cierto parecido ya que se suelen encontrar problemas con las dependencias de los distintos paquetes, especialmente si hay más de una aplicación siendo ejecutada en un servidor, por lo que los despliegues siempre suelen ser problemáticos por choques de dependencias, similar a como el código antes de Java necesitaba distintas librerías dependiendo de para que entorno se compilaba.

Docker lo que nos permite es tener un *host,* que contiene una o varias aplicaciones almacenadas en contenedores – similar a los contenedores de mercancías, que contienen varios envíos pero con dependencias – de manera que las dependencias y código de una no influyen sobre las dependencias y código de otra.

De esta manera, similar al ejemplo que se ha puesto con Java, podremos crear una imagen de nuestra aplicación y llevarla a cuantos servidores queramos sin molestarnos por tener que revisar dependencias, evita choque de puertos con otras aplicaciones desplegadas, …

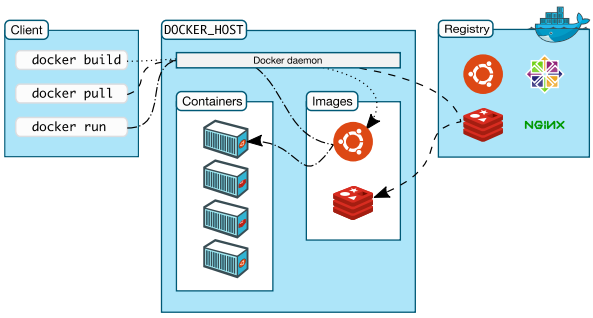


Figura 22: Funcionamiento de Docker (Docker, n.d.)

La manera en la que funciona Docker es mediante imágenes, las cuales podemos crear o recuperar de un repositorio, estas se descargan y se cachean para usarlas posteriormente, almacenándolas en contenedores; con los que interactuaremos a través del demonio Docker y que nos permitirá levantar contenedores, eliminarlos, ...

Creación de una imagen Docker

Para crear una imagen, lo más cómodo es comenzar por una imagen prefabricada como una imagen de Ubuntu o Alpine, si nuestro contenedor contendrá una aplicación bastante genérica, o una imagen creada específicamente para desarrollos basados en ciertos *frameworks* o tecnologías, como imágenes de Python – como para nuestra aplicación – o de PostgreSQL para crear una base de datos.

```
# syntax=docker/dockerfile:1
FROM python:3
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1
WORKDIR /code
COPY requirements.txt /code
RUN pip install -r requirements.txt
# Copy needed files
COPY ballot_interface /code/ballot_interface
COPY manage.py /code/
```

Figura 23: Ejemplo de creación de una imagen Docker

Una vez hayamos decidido de que imagen partir comenzaremos a crear la nuestra propia, copiando los archivos que son necesarios para que el sistema funcione, instalando las dependencias necesarias, definiendo variables de entorno, ejecutando comandos, …

Además, una vez creada la imagen, como es nuestra aplicación, podremos usar utilidades de Docker, como Docker Compose, que nos permite crear contenedores con más de un servicio desde distintas imágenes y definir relaciones de dependencia entre ellos pudiendo especificar, además, para cada uno de ellos, propiedades como variables de entorno, puertos que expone o *scripts* de inicio.

Conclusiones y líneas de trabajo futuras

Parte de los objetivos del sistema finalmente no han sido cumplidos, ya que hemos preferido tener una documentación lo más sólida posible para las partes que si vamos a entregar a tener un código funcional pero sin una buena documentación. Sin embargo, y como se comentó al principio, este Trabajo no tiene como objetivo único el desarrollar una aplicación, si no que también pretende explorar ciertos temas como nuevas metodologías de ingeniería, como OpenUP, lenguajes nuevos para el desarrollador, como Python, o nuevas tecnologías para este, como *blockchain*, en parte motivado por el objetivo de utilizar únicamente software de código abierto desarrollado principalmente por la comunidad.

Por una parte, si bien hacer un esfuerzo para tener una buena ingeniería en el proyecto es lo que, en parte, nos ha llevado a no poder completar la implementación, tenemos que admitir qué, durante el desarrollo de este proyecto, hemos podido comprobar la gran ayuda que es tener una buena ingeniería previa. Un aspecto muy destacable de esto es la facilidad con la que se puede hacer una trazabilidad completa entre código y requisito, evitando tener que preguntarnos si algo no debiera haberse implementado o si se tendría que haber implementado de otra manera, lo que nos ayuda a tener confianza en que el código que entregamos es código de calidad.

Por otra parte podemos hablar de Python como lenguaje de programación, ya que es el primer proyecto en el que usamos este lenguaje más allá de pruebas para testar el lenguaje y hay que decir que la curva de dificultad de este es excepcionalmente plana y permite crear código que es fácilmente legible por personas que ni siquiera se dedican a la informática, por lo que es natural que sea un lenguaje que se haya trasladado a otros ámbitos, como la investigación científica. Esta facilidad de programación, junto con un soporte de la comunidad muy grande, hace que sea muy sencillo resolver cualquier duda que se tenga con el lenguaje lo que ayuda a aplanar, aún más, su curva de dificultad.

Por último, hablemos de la *blockchain* como tecnología que hemos usado durante este desarrollo. Su principal problema es la causa de su éxito y es que toda la información que se encuentra sobre esta tecnología está muy relacionada con inversión en criptomonedas, arte digital, … que no es útil a la hora de plantear un desarrollo de una aplicación descentralizada y que incluso nos puede echar para atrás por dar mala imagen a la tecnología.

Sin embargo, si conseguimos buscar más allá, nos encontraremos con que quizá este tipo de aplicaciones si tengan cabida en un futuro, en la llamada Web 3.0, una web donde la información esté completamente descentralizada y que, con la incorporación de los dispositivos IoT, no es descabellado pensar que coja fuerza en los próximos años pues contamos con una capacidad de procesamiento distribuido nunca antes vista y que es capaz de superar a los mejores superordenadores del mundo.

Es evidente que aún tenemos que depender de bases de datos centralizadas para almacenar grandes cantidades de datos, pues estos no pueden vivir en un sistema totalmente distribuido aún por la sobrecarga que supondría en la red si, por ejemplo, todo el mundo tuviera una copia de la base de datos de Google, pero desde luego que el futuro de esta tecnología propone una visión similar a la visión de una web basada en las ya casi obsoletas redes P2P.

Pero, ¿hemos logrado cumplir los objetivos que nos propusimos? ¿Hemos logrado crear una aplicación de voto seguro usando tecnologías de código abiertos y potenciadas por la comunidad?

Desde luego, y a pesar de lo que pueda parecer, el código abierto sigue gozando de una salud de hierro. Puede parecer contraproducente que, en un mercado donde se mueven millones al día, haya gente dispuesta a invertir su tiempo en construir algo por lo que no va a recibir ningún beneficio económico. Sin embargo, esto se puede entender porque la informática siempre ha tenido un punto anárquico; al fin y al cabo tenemos en nuestro poder dispositivos más potentes que los que se emplearon para llegar a la Luna en la palma de nuestra mano, por lo que solo es cuestión de tiempo que alguien llegue para desafiar a los grandes de la industria con un ordenador que quizá tenga que pagar a plazos, aunque solo sea por el reconocimiento que ello merece.

Respecto a la seguridad de la aplicación no podemos estar seguros de ello por el simple hecho de que no se ha llegado a implementar por completo. Lo que si tenemos claro es que con este Trabajo dejamos sentados los precedentes sobre los que se puede terminar de construir la aplicación, asegurando ya un voto descentralizado con el uso de *blockchain*, para poder, si se hace una apuesta firme por un sistema de estas características, llegar a la democracia directa.

Líneas de trabajo futuras

Como punto final a este Trabajo de Fin de Grado, se proponen una serie de ideas para trabajos relacionados con este que puede servir como ampliación.

Una de las líneas más claras sería terminar la aplicación, incluyendo la interfaz de votación de la que hemos hablado. Tenemos ya realizados los requisitos y diseños por lo que lo único que quedaría sería realizar la implementación de esta parte del sistema siguiendo las técnicas de TDD descriptas e implementar un plan de despliegue que permita mantener la aplicación en un servidor.

Otra de las líneas es que, al utilizar una API para almacenar los datos, esto hace que la información esté completamente desvinculada de como se muestra y de las herramientas que se usan para mostrarla. Así, en un futuro, podríamos proponer reescribir la interfaz usando Vue, un *framework* NodeJS que sigue los principios de libertad por los que apuesta este Trabajo; reescribir la API como microservicios; o plantear una interfaz usando otro tipo de *framework* distinto a Bootstrap. Las posibilidades aquí son infinitas.

Por último, una línea de trabajo muy importante sería, como ya se ha comentado en las conclusiones, validar si la implementación de la solución final es lo suficientemente segura como para establecerse como alternativa real al voto, quedando este como una opción para un trabajo de investigación.

Bibliografía

1. República de Estonia. (n.d.). *The new digital nation.* Recuperado el 5 de septiembre de 2022 de https://www.e-resident.gov.ee/, en inglés.

2. Consejo federal de la Confederación Suiza (2022, 25 de agosto). *Indice cronologico (per scadenza) 2021-2022*. Recuperado el 5 de septiembre de https://www.bk.admin.ch/ch/i/pore/va/vab_2_2_4_1_2021_2030.html, en italiano.

3. Portillo, J. (2019, 18 de septiembre). El coste de las elecciones generales del 10 de noviembre será de 140 millones de euros para el contribuyente. *El País*. Recuperado el 5 de septiembre de https://cincodias.elpais.com/cincodias/2019/09/17/economia/1568751943_434386.html

4. Ross, B. (n.d.). *Touch-Screen Trouble.* ABC News. Recuperado el 5 de septiembre de https://abcnews.go.com/Politics/Vote2004/story?id=203866&page=1, en inglés

5. Scott T. (2019, 9 de diciembre). *Why Electronic Voting Is Still A Bad Idea* [Video]. Youtube. Recuperado el 5 de septiembre de https://www.youtube.com/watch?v=LkH2r-sNjQs, en inglés.

6. Logan dev (2022, 29 de junio). Top 8 Most Demanded Programming Languages in 2022. *devjobsscanner*. Recuperado el 5 de septiembre de https://www.devjobsscanner.com/blog/top-8-most-demanded-languages-in-2022/, en inglés.

7. Reuters (2010, 13 de agosto). Oracle sues Google over Android. *Reuters*. Recuperado el 5 de septiembre de https://www.reuters.com/article/us-google-oracle-android-lawsuit-idUSTRE67B5G720100813, en inglés.

8. Fadilpašić, S. (2022, 10 de enero). Thousands of open-source projects taken down by disgruntled developer. *TechRadar*. Recuperado el 5 de septiembre de https://www.techradar.com/news/thousands-of-open-source-projects-taken-down-by-disgruntled-developer, en inglés

9. Rosemain M. & Satter R. (2021, 10 de marzo). Millions of websites offline after fire at French cloud services firm. *Reuters*. Recuperado el 5 de septiembre de https://www.reuters.com/article/us-france-ovh-fire-idUSKBN2B20NU, en inglés

10. Hetler, A. (2022, 13 de junio). 9 common cryptocurrency scams in 2022. *TechTarget*. Recuperado el 5 de septiembre de 9 common cryptocurrency scams in 2022 (techtarget.com) , en inglés

11. Nakamoto S. (n.d.) Bitcoin: A Peer-to-Peer Electronic Cash System. Recuperado el 5 de septiembre de https://bitcoin.org/bitcoin.pdf, en inglés.

12. 3Blue1Brown (2017, 7 de julio). *¿Alguna vez te has preguntado cómo Bitcoin(y otras criptomonedas) realmente funcionan?* [Video]. Youtube. Recuperado el 5 de septiembre de https://www.youtube.com/watch?v=bBC-nXj3Ng4, en inglés.

13. minimalsm et al. (2022, 22 de julio). Gas y tarifas. *ethereum.org*. Recuperado el 5 de septiembre de https://ethereum.org/es/developers/docs/gas/

14. Constitución Española. Boletín Oficial del Estado, 29 de diciembre de 1978, núm. 311. Recuperado el 5 de septiembre de https://app.congreso.es/consti/constitucion/indice/sinopsis/sinopsis.jsp?art=68&tipo=2

15. Eclipse Software Foundation (2012, 1 de junio). *What is OpenUP?*. Recuperado el 5 de septiembre de https://download.eclipse.org/technology/epf/OpenUP/published/openup_published_1.5.1.5_20121212/openup/index.htm, en inglés

16. OpenUP (2020, 24 de noviembre). En *Wikipedia*. Recuperado el 5 de septiembre de https://en.wikipedia.org/w/index.php?title=OpenUP, en inglés

17. Django Software Foundation (n.d.). *Why Django?*. Django Project. Recuperado el 5 de septiembre de https://www.djangoproject.com/start/overview/, en inglés

18. Jindal, R. (2021, 12 de abril). *Respuesta de Rahul Jindal a "How does Instagram use Django?"*. Quora. Recuperado el 5 de septiembre de https://www.quora.com/How-does-Instagram-use-Django, en inglés

19. Django Software Foundation (n.d.). *About the Django Software Foundation*. Django Project. Recuperado el 5 de septiembre de https://www.djangoproject.com/foundation/, en inglés

20. Zetterlund T. (2021, 20 de noviembre). How Environmental Friendly is Programming Languages. Torbjorn Zetterlund. Recuperado el 5 de septiembre de https://torbjornzetterlund.com/how-environmental-friendly-is-programming-languages/, en inglés

21. Docker (n.d.). *What is a Container?*. Recuperado el 5 de septiembre de https://www.docker.com/resources/what-container/, en inglés

22. Django Software Foundation (n.d.). *Django appears to be a MVC framework, but you call the Controller the "view", and the View the "template". How come you don't use the standard names?*. Django Project. Recuperado el 5 de septiembre de https://docs.djangoproject.com/en/4.1/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names, en inglés

23. Excirial (2016, 27 de junio). *A quick overview of the Test-driven development lifecycle* [Imagen]. Wikimedia Commons. Recuperado el 5 de septiembre de https://commons.wikimedia.org/wiki/File:Test-driven_development.svg

24. Docker (n.d.). *Docker Architecture* [Imagen]. Docker. Recuperado el 5 de septiembre de https://docs.docker.com/get-started/overview/#docker-architecture, en inglés