Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo multiplataforma de tipo "Full Stack": creación de un Back

Autor: Daniel Yebra Acero

Tutor: Salvador Otón Tortosa
Co-Tutor: José María Gutiérrez Martínez

2017

Escuela Politécnica Superior
Graduado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo multiplataforma de tipo "Full Stack": creación de un Back-End

Autor: Daniel Yebra Acero
Tutor: Salvador Otón Tortosa
Co-Tutor: José María Gutiérrez Martínez

TRIBUNAL:

Presidente:

Vocal 1o:

Vocal 2o:

FECHA:

Resumen

Con el presente trabajo se pretende dar una visión completa del proceso de creación de una herramienta software multiplataforma, a través de una metodología ágil enfocada a equipos multidisciplinares con un perfil Full Stack. El trabajo tiene varios niveles de profundidad, comienza en la capa de negocios, para luego bajar a capas de definición más técnicas y centrarse en el trabajo Back-End.

Summary

The present work intends to give a complete view of the process of creating a multi-platform software tool, through an agile methodology focused on multidisciplinary teams with a Full Stack profile. The job has several levels of depth, starting in the business layer, then moving down to more technical definition layers and to focus on the Back-End work.

Palabras clave

FullStack, StartUp, Back-End, Django, Agile Method.

Resumen extendido

Con el presente trabajo se pretende dar una visión completa del proceso de creación de una herramienta software multiplataforma, así como los pasos a seguir en cuanto a iniciar desde cero un grupo de trabajo a nivel empresarial.

Se explicará una visión de negocio con los sistemas de información como producto. Explicaremos los pasos a seguir de las empresas que deciden emprender y cómo lograr sus objetivos. En este punto se tratarán todas las partes decisivas del proceso de negocio y operaciones, desde las primeras etapas donde se definen los ideales y motivaciones de la empresa, hasta la etapa final de cierre, con sus posibles desviaciones. Se mostrarán las técnicas y las herramientas que se han usado en el proyecto que ha servido de base de estudio y las conclusiones en cada etapa.

También se explicará la parte técnica del diseño y su arquitectura, enfocado principalmente al Back-End, donde se expondrán y compararán las alternativas más comunes.

A lo largo de este documento, ya sea durante la explicación de la parte de negocio o durante la exposición del desarrollo del aspecto técnico, se ha añadido el punto de vista del autor, adquirido mediante la experiencia de realizar un proyecto de estas características.

1. INTRODUCCIÓN

Desde 2013 en el sector de las nuevas tecnologías se está tendiendo a usar cada vez más el término FullStack.

Con el siguiente estudio se pretende dar a conocer el concepto FullStack, las diferentes ramas que lo conforman, tanto en empresa como en el rol de desarrollador, así como explicar la importancia de estos perfiles a día de hoy.

Para ello profundizaremos primeramente en la terminología y aclararemos cómo ha ido evolucionando el concepto a lo largo de años y cómo se ha ido extendiendo. Con esto, lo que se pretende es dar una visión clara de lo que conlleva un desarrollo FullStack. Se comenzará por el enfoque empresarial y se terminará con el perfil de desarrollador; este incluye el enfoque técnico, en el cual se repasarán cada una de las especializaciones y se mostrará el conjunto de tecnologías más usadas.

Asimismo, se seleccionarán unas tecnologías representativas con el propósito de dar una visión aún más técnica. En esta lectura nos centraremos principalmente en una de las partes del desarrollo, el BackEnd. En esta parte, y con una línea más técnica, se hará énfasis en FrameWorks, plataformas de servicios de despliegue, bases de datos, creación de API REST y conectores con los clientes, entre otros. También se tratarán algunas tecnologías FrontEnd con el fin de mantener siempre la visión de conjunto que corresponde con el rol de FullStack.

2. OBJETIVO

El objetivo principal es el de facilitar una guía al lector interesado en desarrollar un proyecto con un enfoque completo de BackEnd-FrontEnd. Cuando finalice la lectura tendrá un conocimiento completo y actual de las funciones de un desarrollador FullStack, además de tener una visión general de las herramientas más usadas en cada uno de sus roles. Asimismo, conocerá las fases por las que tiene que pasar un proyecto en una empresa FS (FullStack).

Con la elección de este tema se ha pretendido ocupar los siguientes requisitos: encontrar un tema que fuese original, de actualidad y que no estuviera muy tratado; que se pudiera desarrollar con tecnologías punteras y en alza; que se pudieran aplicar los máximos conocimientos aprendidos en los años de carrera. En referencia a este último requisito se va a aplicar lo aprendido tanto en las asignaturas de empresa, gestión y planificación, como en las asignaturas relacionadas con el ciclo completo de la vida del software y con la realización de un desarrollo completo de un proyecto.

Paralelamente a la elaboración del TFG se ha desarrollado un proyecto real completo, que cumple tanto los requisitos que necesita una empresa para considerarse FullStack como los que necesitan los desarrolladores que pretendan tener un perfil FSD (FullStack Developer). Por lo tanto, el grueso del trabajo consiste en dejar plasmada la experiencia, tanto en la explicación de las herramientas que se han considerado necesarias para llevarlo a cabo como en la orientación en todos los ámbitos del desarrollo.

Como objetivo secundario se ha tenido en cuenta la formación paralela y complementaria que será necesaria aprender para abarcar un proyecto tan extenso, con lo que se ganaría experiencia real sobre problemas que ocurren en el mundo laboral, utilizando las herramientas más demandadas actualmente en este mercado.

Además, llevar a término un proyecto con estas características en el que el desarrollador tiene que ocupar roles muy distintos ya que tratará con: la visión del producto, la toma de requisitos, el plan de negocios, diseño, ingeniería del software, venta … hace que se obtenga una visión general de cómo se debe realizar un producto software.

3. ESTADO DEL ARTE

Origen del FullStack

FullStack Startup

Podemos definir una empresa como FullStack cuando se encarga de desarrollar un producto gestionando toda la cadena de actividades, desde el diseño primigenio hasta la venta al consumidor final del producto completo.

El origen de esta filosofía empresarial llega con las nuevas tecnologías, las cuales permiten tener al alcance de empresas pequeñas soluciones de marketing, contratación, venta… que antes sólo estaban reservadas a grandes empresas por sus recursos.

Como introdujo Chris Dixon (2014), la finalidad de una empresa con esta filosofía es la de eliminar la mayor parte de intermediarios comerciales y logísticos, para así capturar la mayor parte de beneficios económicos. De igual modo se controlaría completamente la experiencia del cliente desde el comienzo hasta el final y no se perdería el enfoque inicial. Dixon puntualiza que este enfoque ha ayudado a muchas empresas de reconocido nombre a tener el éxito que tienen en la actualidad, como es el caso de Tesla, Uber y Netflix, y augura que estamos al principio del movimiento y que grandes empresas cambiarán este enfoque siguiendo a las STUP (Startups).

Se conoce como Startup, a una compañía emergente fuertemente ligada con el mundo de las tecnologías, la cual comienza con una idea de negocio novedosa o creativa y que destaca sobre todo porque trabaja bajo condiciones de incertidumbre muy altas. Por norma general estas empresas carecen de una gran inversión inicial, al menos en sus orígenes, por lo que se ven forzadas a suplir esta carencia de muchas formas, entre ellas: ahorrando en el lugar donde se desarrolla el producto (trabajando en oficinas pequeñas e incluso en su propio hogar), buscando incubadoras (empresas que acogen un proyecto en sus primeras fases), externalizando lo menos posible (intentando que la mayoría de las fases del negocio recaigan sobre la propia empresa) y teniendo el personal imprescindible y lo más polivalente posible. Es en este último punto donde se aúnan el concepto de FSStartUp y FSDeveloper, ya que son los empleados más buscados por este tipo de empresas.

Son varios los puntos importantes a la hora de comenzar una STUP. El primero y en palabras de Luis Ángel Fernández: “Tener una idea con alto potencial de crecimiento, escalable y promovida por un buen equipo emprendedor”. Detallaremos esto al comienzo de la vida del software, también es necesario conocer el modelo organizativo y saber la forma de financiarse:

Organización

La estructura organizativa de este tipo de empresas no mantiene necesariamente una configuración estricta, y esta puede variar según las necesidades de la STUP (Startup) y su ciclo de vida, añadiendo nuevos puestos o roles de trabajo o aunando en una sola persona varios de ellos, dándose esta situación de manera muy frecuente en los primeros años de crecimiento. Además, el éxito depende en gran medida de una buena alineación asignando cada uno de los roles a las personas adecuadas y cambiándolos cuando fueran necesarios, por lo que todos ellos tienen que mostrar una buena capacidad de adaptación y cambio.

Actualmente los más usados en una STUP media son los siguientes:

CEO (Chief Executive Officer):
Es lo que tradicionalmente se conoce como Consejero Delegado, Director Ejecutivo, Director General, etc. Normalmente es la cara visible y el puesto suele estar ocupado por uno de los fundadores de la empresa. Entre sus funciones más comunes está la de asumir el liderazgo y el rumbo de la empresa, creando una estrategia que parte de la visión y de la misión actuales.

COO (Chief Operating Officer):
Es el siguiente en la escala de mandos, equivaldría al Director de Operaciones. Tiene la responsabilidad de llevar a término las estrategias globales marcadas por el CEO, teniendo un control más directo sobre estas. Tradicionalmente suele ser el sucesor del actual CEO.

CMO (Chief Marketing Officer):
Como Director de Marketing está a cargo de la imagen de marca, de la publicidad, de los equipos comerciales, de las ventas, de los análisis de mercado, etc. Entre estas funciones también está la de relacionarse con todos los departamentos para mantener una cohesión del producto de cara a mejorar la experiencia del cliente.

CFO (Chief Financial Officer):
Es junto con el CEO el encargado de tomar las decisiones financieras de la empresa, analizando las inversiones, los riesgos y los resultados económicos. Es el asesor que da la visión más analítica a los proyectos.

CIO (Chief Information Officer):
En el grado de responsabilidad técnica operativa el CIO es el de mayor cargo. Su traducción suele ser Jefe del departamento de TI (Tecnologías de la información). Por lo tanto, sus responsabilidades van, desde hacer que todas las tecnologías de la información funcionen como es debido, hasta la incorporación de nuevas tecnologías que ayuden al procesado de la información y a mejorar el trabajo de los empleados.

CTO (Chief Technology Officer):

Es el encargado directo del equipo de ingeniería y de los desarrolladores de la empresa. Es el que tiene la responsabilidad de diseñar lo que ha requerido el CEO e implementar su estrategia técnica para lograrlo. También suele ser el responsable del departamento de I+D. Es bastante común que el CTO también realice las funciones del CIO.

CCO (Chief Comunication Officer):

Tiene la responsabilidad de manejar las comunicaciones de la empresa, tanto las internas como las externas. Suele ser el encargado junto con el CMO del Branding de la empresa (imagen de marca). Tiene contactos con los medios de comunicación y las RRSS.

CSO (Chief Security Officer):

Como encargado de la ciberseguridad de la empresa el CSO, es un rol que tendrá que encargarse de toda la seguridad que conlleva la IT. Además, hace las funciones de consultor interno de seguridad tanto en el ámbito de la estrategia como en el desarrollo de productos. Otra función de este rol es la de formador de seguridad de los equipos técnicos y de los empleados, con el fin de ser más eficiente y atajar problemas derivados de los fallos humanos.

Además de estos roles estarían **los perfiles más técnicos**, propios del desarrollo del producto y mucho más cerca de la tecnología, que serían en quienes los anteriores cargos delegarían las funciones. Estos roles pueden dividirse en dos grandes grupos, el FrontEnd y el BackEnd.

El primero es el encargado de traducir el diseño de las interfaces que van a utilizar usuarios, y engloba muchas tecnologías y roles distintos; sus funciones principales son la realización de sitios y de aplicaciones web. Como más adelante veremos, dentro de este grupo se tiende a englobar también a los Mobile Developer, especializados en realizar aplicaciones para dispositivos en movilidad.

El segundo grupo es el encargado de la realización de la lógica que no ve el usuario y que no pertenece al ámbito del FrontEnd, como puede ser la creación de servidores de todo tipo, APIS… Este perfil, al igual que el anterior, puede especializarse en cada una de las tecnologías que se requieran, dando lugar a perfiles mucho más concretos.

En definitiva, el tema principal que nos ocupa es hallar un perfil que, entre otras cualidades, sea conocedor de la tecnología que engloba a estos dos grandes grupos, y que está siendo muy demandado por este tipo de empresas FS Startup.

Financiación

Uno de los primeros pasos necesarios para afrontar un proyecto es encontrar la financiación necesaria para llevarlo a término.

Se han estudiado las diferentes posibilidades de financiación, todas ligadas a la etapa en la que se encuentre el proyecto. Estas se han dividido en tres fases, que corresponden con las etapas de nacimiento, crecimiento y madurez/mantenimiento. Aunque no es necesario que una STUP pase por todas las etapas de inversión, sí es lo más natural.

Las tres fases son:

1ª Fase. Inversión inicial

Es la fase donde se forja la entidad. Tiene como finalidad conseguir fondos para comenzar el proyecto e ir trazando una hoja de ruta profesional.

Dependiendo de la complejidad y de la ambición del proyecto es posible que no se necesite una gran cantidad de capital externo, por lo que esta primera inversión podría no producirse, aunque en la mayoría de los casos la inversión suele partir de los 5000 €. Para estos casos hay típicamente tres medios para obtener esta primera etapa de financiación.

a) El primer recurso es acudir al entorno de los artífices de la idea, que son los familiares y amigos. Con mucha frecuencia se tiende a usar el término *Family, Friends and Fools*[1], (FFF´s). Este método de financiación tiene varias ventajas para los emprendedores, como que al haber un gran vínculo personal con los inversores es más sencillo venderles la idea. Además, al no tratarse de inversores profesionales la contractualización de las aportaciones será menos exigente y exigirán menos contrapartidas. Como inconvenientes notables están que, al igual que es fácil hacerles empatizar con el proyecto, las implicaciones afectivas pueden dar lugar a enfrentamientos y/o enemistades, ya que es usual que quieran ser partícipes de una manera más activa del negocio. Aparte de esto se pueden crear presiones personales al intentar cumplir las expectativas de los inversores y por miedo a defraudar.

   Esta forma de financiación no sería muy plausible si se necesitara de una gran inversión inicial, debido a la cantidad a solicitar a cada persona del entorno, y siempre considerando el número de personas a las que se podría llegar.

b) Otra forma común de obtener este primer impulso es recurrir a las ayudas públicas para nuevos emprendedores. En Europa el año pasado se destinaron partidas de financiación de 90.000 millones de euros para estos conceptos

   Una de las más conocidas son los préstamos participativos de Enisa[2] (Empresa Nacional de Innovación). Estos tienen bastantes variantes, sirvan como ejemplo las líneas de crédito para

[1] Expresión inglesa muy extendida en este ámbito. Viene a decir como Familiares amigos y "confiados". Aquí.

[2] Información actual sobre esta financiera. Aquí.

jóvenes o las ayudas para impulsar a empresas ya constituidas. Tienen varios requisitos para optar a la financiación; el más llamativo es que las aportaciones mínimas de socios, vía capital/ fondos propios, tiene que ser de al menos el 50% de la cantidad del préstamo concedido. Suele ser habitual que se combine con financiaciones FFF's para conseguir estas aportaciones mínimas. La cantidad mínima oscila entre 25.000 € y 1.500.000 € dependiendo de la línea de crédito ofertada.
Otra partida de financiación muy conocida es la de Instrumento PYME y Horizonte PYME[3], únicamente enfocada a las STUP, que en su fase uno está destinada a impulsar estas empresas. Para poder optar hay que pasar unas fases de concurso que se suelen celebrar cada tres meses.

c) Al igual que los fondos públicos[4], existen fondos privados con líneas de crédito a empresas emprendedoras. Las condiciones suelen ser sensiblemente menos atractivas que las de fondos públicos. Muchos bancos y fondos de inversión privados ofrecen estás líneas de créditos.

2º Fase. Potenciación.

Con el proyecto en marcha y la STUP ya arrancada, el siguiente objetivo es encontrar una financiación para acelerar el proceso de creación de esta. Las opciones más comunes son las siguientes:

a) Las aceleradoras e incubadoras son empresas que se dedican a potenciar y a ayudar en el crecimiento de las STUP. La característica principal de las incubadoras es que dan **cobijo a la empresa**, prestando **instalaciones** y **servicios básicos**, mientras que las aceleradoras se implican en aportar ideas y mejorar el negocio o bien con contactos para obtener financiación, dar salida al producto, mejorar la visibilidad… No obstante, suelen existir empresas mixtas que brindan todos los servicios.

	Infraestructura	*Mentoring*	*Inversión de capital*	*Aceleración, contactos, …*
Aceleradora	no	si	si	si
Incubadora	si	no	sin obligación	no

b) El *crowdfunding* es una pasarela de inversión para obtener capital de una manera rápida y completa. Básicamente es una plataforma de inversión colectiva y micromecenazgo en la cual la STAP envía su proyecto a la plataforma, indicando su funcionamiento base, con la descripción de su producto, vídeos promocionales, la cantidad mínima necesaria para continuar con el proyecto y un plazo de obtención del capital solicitado. La plataforma hace una primera valoración de la propuesta y si la acepta, la publicita. Pasado el plazo, si se ha

[3] Acceso a la información de las rondas actuales. Aquí.
[4] Ampliación de la información de las líneas de financiación. Artículo de la revista emprendedores más información aquí.

conseguido la financiación se abona a la empresa, si no lo ha conseguido, se devuelve lo aportado a los inversores. Es habitual que las aportaciones tengan un capital mínimo.

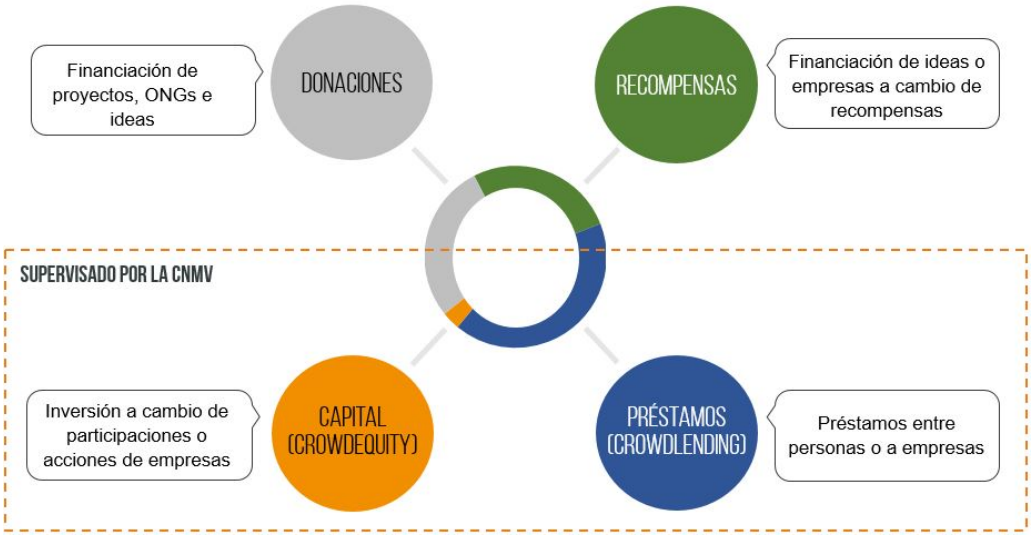


Ilustración 1 Tipos de crowdfunding. Diagrama de crowdCube

Existen 4 modalidades básicas:

- **Equite crowdfunding:** Permite que los inversores obtengan un retorno económico en función del crecimiento de la empresa.

- **Crowdfunding de recompensas:** La inversión se hace a cambio del producto final que se está apoyando y suele tener algún extra añadido o edición especial como contraprestación

- **Crowdfunding de donaciones**: No se espera ningún beneficio de la aportación, tiene carácter puramente altruista, se suele dar en productos para un bien común, gasto libre, ONGs, etc.

- **Préstamos mediante crowdfunding (*crowdlending)*:** Previo un acuerdo de intereses, plazos de devoluciones y garantías, se realizan préstamos entre particulares P2P(*peer to peer*) o para dotar de liquidez a una empresa P2B (*peer to business*).

La cantidad invertida mediante estas plataformas supera los 34.000 millones de dólares en todo el mundo mediante más de 1200 pasarelas de crowdfunding.

3º Fase. Continuación y mantenimiento

En el punto en el que el proyecto está en una fase de funcionamiento y se trata de un producto atractivo, innovador, ambicioso y escalable, suele llamar la atención a inversores más tradicionales, que ven en el producto una rentabilidad con menos riesgos. Estas rondas de financiación suelen

oscilar desde los 10.000 a los 1.000.000 euros. Podemos tratar principalmente con tres tipos de inversores:

a) Inversores privados. Pueden ser de cualquier tipo, pero se suelen caracterizar en invertir en negocios que no dominan. Sus decisiones de inversión se basan en métricas propias, experiencias similares… El dinero aportado suele provenir de terceros, como en las entidades de Capital Riesgo.

b) Por otro lado tenemos a los inversores llamados *Business Angel*, son inversores que no aportan únicamente capital sino que aportan conocimientos y aconsejan en las decisiones operativas y estratégicas. Son inversores que únicamente invierten en mercados y sectores que dominan. En la gran mayoría de los casos invierten su propio dinero y su motivación no tiene que ir ser únicamente sacar la máxima rentabilidad.

c) Absorción de la STUP por otra empresa. Para algunos esta es la meta principal y punto culmen donde, o bien comenzar empezar otro proyecto usando los conocimientos adquiridos al desarrollar este o continuar trabajando como división de la empresa compradora.

FullStack Developer

Una primera definición de FullStack Developer es la de un perfil de desarrollador capaz de desenvolverse con soltura en todas las capas y manejar las herramientas desde la parte del servidor hasta la parte del cliente.

La demanda de este perfil surge de forma natural dada las necesidades del sector. En la última década, gracias a las facilidades del emprendimiento en el sector del software, ha proliferado la creación de muchas empresas emergentes. Estas empresas de nueva creación suelen tener un modelo de negocio escalable, por lo que en sus orígenes tienen que ajustar los costes de desarrollo. Esto provoca la necesidad de tener una plantilla reducida. Es aquí donde se desarrolla la definición de FSD, cuando se empiezan a demandar perfiles multidisciplinares. Por lo tanto, los perfiles más demandados en el departamento de desarrollo serían aquellos que supieran manejarse en la parte FrontEnd y BackEnd. Ya cuando el producto y la empresa evolucionen y escalen se podrán contratar perfiles más especializados o redistribuir a los actuales desarrolladores a un solo ámbito.

Dejando a un lado las Pymes y las Startups, una de las primeras empresas grandes del sector en generalizar este perfil, dándolo a conocer, fue Facebook ya que, sobre el 2010 buscaba, indistintamente del puesto que fuera a ocupar el desarrollador, que su perfil fuera FSD. Esta demanda creció aún más tras popularizarse varios Frameworks que ayudaban a aunar el Front y el BackEnd facilitando y automatizando parte del trabajo.

Laurence Gellert (2012) analiza las razones por las que una empresa como esta, con amplio recorrido y consolidada en el mercado, puede precisar un perfil FSD. En su análisis informa que “Los buenos desarrolladores que están familiarizados con la pila entera saben cómo hacer la vida más fácil para los que les rodean”, por lo que, si un desarrollador está trabajando en una parte de la capa, sabe lo que precisa su compañero de la capa superior y qué esperar de los de la capa inferior; así podrá hacer su trabajo con visión de conjunto y no tratará al resto de las capas como cajas negras. Además, la comunicación técnica entre los responsables de otras capas será mucho más ágil. Todo esto lo que consigue es que, en conjunto, el desarrollo del proyecto sea más eficiente y colaborativo.

Adicionalmente, según se consolida el desarrollador FullStack, se le van añadiendo nuevas características y requisitos al perfil, por lo que aparte de conocer las dos caras del desarrollo FrontEnd y BackEnd se les requiere: conocer cómo realizar el despliegue de una app, diseñar una buena UX (experiencia de usuario), dominar pruebas de código, hacer despliegues, etc.; lo que convierte así al FullStack en un perfil altamente polivalente y demandado.

Full-Stack Developers en la actualidad

En origen, se consideraba un perfil FullStack aquel que podía diseñar una web desde cero, pasando desde el FrontEnd al BackEnd usando las herramientas que tenía al alcance.

En esa época, las herramientas de desarrollo eran escasas y más sencillas. Con el tiempo, estas tecnologías han ido evolucionando, aumentando y variando el paradigma (1). El perfil de FullStack ha ido necesariamente cambiando. Además de los conocimientos se busca que el desarrollador tenga una serie de cualidades como: ser una persona con gran capacidad resolutiva, tener buena capacidad de adaptación a los cambios y facilidad en el aprendizaje, ser políglota, etc. (2). Estas capacidades y cualidades son necesarias para adaptarse a la situación actual del software que tiene que manejar un FSD.

Especializaciones

La evolución de la tecnología ha traído varias novedades que afectan directamente a este sector. Los cambios más importantes han sido, por un lado, la incorporación de un nuevo hardware para el que diseñar, y por otro las crecientes alternativas y especializaciones en las herramientas de desarrollo software, tanto en el lado del cliente como en el del servidor.

Hardware

A mediados del 2008 Apple presentaba su tienda de aplicaciones AppStore junto a su kit de desarrollo de aplicaciones SDK y su programa de desarrolladores (3). A finales de este mismo año, Google sacaba su tienda de aplicaciones AndroidMarket (renombrada en 2012 GooglePlayStore) y su AndroidSDK.

Ambas plataformas han conseguido grandes cifras tanto en número de descargas anuales, (en torno a 300 millones en 2015), como de aplicaciones subidas (2,7 millones también en 2015) dejando unos ingresos conjuntos superiores a los 300 millones de dólares (4).

Estas cifras, que no se habían visto hasta la fecha, han logrado que gran parte del desarrollo FrontEnd se vea forzado a diversificarse por la necesidad de tener trabajadores expertos en este ámbito, creando perfiles especializados en tecnología móvil. Estos nuevos roles pueden estar dedicados en exclusiva a una de las plataformas utilizando las herramientas nativas que proveen los fabricantes, o bien especializarse en ambas, conociendo la programación nativa de las dos o utilizando herramientas de desarrollo multiplataforma como Xamarin.

Software

Cuando comenzó a acuñarse el termino full-stack, a principios del 2000, las herramientas de desarrollo front y back estaban muy definidas. Existían pocos servidores y con funcionalidades muy

concretas, como generar interfaces de usuarios o mantener el estado de la aplicación. Entre ellos su programación, aunque no igual, era muy parecida. Además, sólo existían bases de datos relacionales, utilizaban un JS básico, las resoluciones de pantallas estaban más limitadas, solían seguir un standard, etc.

Actualmente todo se ha ido incrementando tanto en variedad como en complejidad: Javascript se está volviendo más sofisticado llegando a tener un nivel parecido a C# y Java y siendo igual de complejos que el backend. La maquetación de aplicaciones web ahora es mucho más potente, teniendo que adaptarse a diferentes tamaños de pantalla, densidades e interfaces. Hay más variedad de BBDD, incorporando las bases de datos no relacionales. Se han credo aplicaciones REST para nutrir a las aplicaciones. Y otros muchos cambios que hacen que la dificultad crezca según avanza la tecnología (1).

Es práctica habitual dividir y agrupar las tecnologías en stacks, con el conjunto de herramientas más usado. Por ejemplo, LAMP + WEB fue de los más usados, que es un acrónimo formado por el comienzo de las palabras que conforman el conjunto de herramientas que trabajan en todas las capas, con Linux como sistema operativo; Apache, como servidor web; MySql, para la base de datos y PHP-Python como lenguajes de CGI-scripting (Interfaz de entrada común). Además de LAMP hay más stack para cada conjunto de tecnologías, como el conjunto de Google, MEAN (MongoDB, Express.js, AngularJS y Note.js). No obstante, aunque sea habitual su uso conjunto, cada tecnología puede ser substituida por otra distinta, y no todas las tecnologías tienden a usarse únicamente con ellas mismas, como es el caso del Framework Django, el cual nos da muchas facilidades y funciona de una manera muy sencilla con cualquier tecnología con el que se quiera complementar.

Otros conocimientos

Además del aumento de software y la inclusión de nuevos elementos del frontEnd, Damian Wajser (5) asegura que el perfil de un verdadero desarrollador FullStack no termina en conocer las herramientas de desarrollo FullStack, sino que tiene que ser experto también en las que lo rodean, como son: gran manejo del modelado de datos, práctica con sistema de control de versiones, despliegue en PAAS y en sistemas en la nube, soltura con herramientas de testing y deployment, posicionamiento WEB y nociones sobre seguridad informática e incluso sobre experiencia de usuario, además de comprender las necesidades del cliente y del ámbito de negocio.

Conclusión

Esta evolución que se ha descrito ha provocado que sea mucho más difícil definir el perfil de un desarrollador FullStack, dando lugar a especializaciones dentro del mismo perfil. Por un lado, tenemos a desarrolladores que se especializan en un Stack concreto de las capas del desarrollo y desconocen el resto, también están los desarrolladores que incorporan a sus conocimientos de FrontEnd el desarrollo de aplicaciones móviles que pueden estar especializados en el desarrollo con multiplataforma o en nativo, otros que obvian la parte móvil… además, hay que incluir los conocimientos externos a la capa, que cada vez se exigen más.

Lo que se puede sacar en claro es que lo que convierte a un desarrollador en un desarrollador FullStack es la experiencia y la extensión de conocimientos, por lo que no podemos encontrarlo en un perfil Junior. En los últimos años, con el crecimiento de las tecnologías y las exigencias, se ha creado un nuevo perfil para aquel desarrollador que no solo es capaz de desenvolverse por todas las capas web, sino que además es capaz de trabajar con varias stack y beneficiarse de las ventajas de una o de otra según el proyecto. A este nuevo perfil altamente experto se le está empezando a denominar de manera coloquial como Ninja Developer.

Futuro del FSD

Algunos expertos del sector predicen que, junto con el crecimiento y la diversificación que están sufriendo las tecnologías, añadido a que actualmente es cada vez más complejo encontrar un desarrollador FS, en un futuro la única opción será la especialización en Stacks (1), con lo que la esencia de un FSD no se perderá, ya que las Stacks engloban todas las partes del desarrollo, aunque sí se frenará la tendencia a conocer todas las herramientas técnicas.

4. EXPERIENCIAS DEL DESARROLLO

Proceso de ingeniería, clarificación de las partes

Como se ha podido deducir, el proceso de vida de un Startup va fuertemente ligado al ciclo de vida del software, compartiendo sus etapas. Esto es evidente ya que las STUP tecnológicas solo apuestan por un producto, por lo que, en la etapa más temprana de la empresa se estará empezando a formalizar la visión y misión del producto, empezando a tener claro los primeros requisitos. A su vez, las últimas etapas del ciclo de vida suelen coincidir con el final de la STUP, bien porque sea adquirida o porque cambie a una empresa más tradicional o amplíe el abanico con nuevos productos basados en el actual.

Se va a estructurar esta parte del ciclo de vida del software de la siguiente manera:

Se recorre cada una de las **etapas que conforman el ciclo habitual de creación de software** en consonancia con la situación de la STUP y en cada una de las partes se tratará lo siguiente:

- **Explicación del proceso**, desde un enfoque de FullStack, viendo las posibilidades de actuación que se tiene y la viabilidad de las alternativas.
- Explicación del proceso llevadas a nuestro **caso práctico**, explicando nuestras decisiones y el por qué.
- Explicación de las **herramientas y las tecnologías** usadas, qué finalidad tienen y una explicación somera de su uso. Se desarrollará con distintos niveles de profundidad dependiendo de la fase a analizar.

Aunque la estructura explicativa parece seguir un proceso en cascada tradicional, únicamente se refiere a ella como parte del proceso del desarrollo Ágil[5], en las que la mayoría de las partes tendrán que iterar varias veces hasta la finalización del proyecto

[5] Se detalla en el segundo punto del desarrollo.

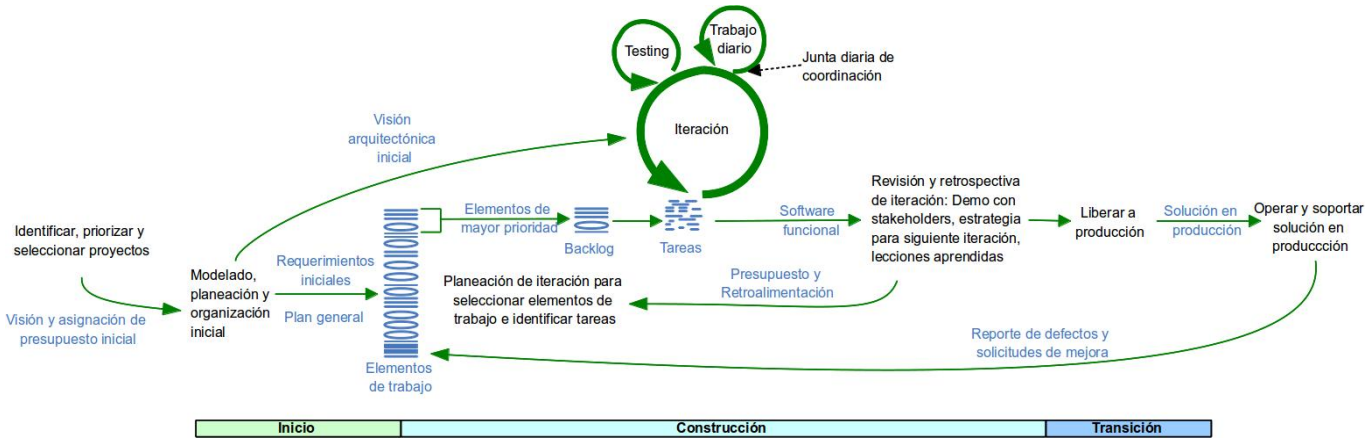


Ilustración 2 diagrama ciclo de vida de metodología ágil. SGBUZZ

Visión y misión

Explicación del proceso

Es la primera parte y la fundamental en todo proceso de creación de un STUP. Son los cimientos que dan pie a todo lo demás y con lo que se inicia el proceso de ingeniera del software. Además, es la etapa en la cual se va a construir el enfoque de la empresa, la imagen que se mostrará al mundo y lo único tangible que podrá usarse para conseguir financiación y captar colaboradores. Cualquier persona o entidad interesada en nuestro producto querrá saber:

- Nuestra realidad a día de hoy, dónde estamos: Que implica el panorama actual de nuestra idea.
- El compromiso de dónde vamos a estar mañana: Mostrar el estado en el que estará nuestro producto en un corto espacio de tiempo.
- Explicación de la visión global: Hacer ver la meta hacia la que se aspira y hacia dónde se está apuntando y se dirige la Startup.

Teniendo claro lo importante de la visión y misión vamos a definir y diferenciar ambas partes:

Visión:

Es la meta idealizada, lo que se pretende conseguir a lo largo de su existencia. Es, de manera abstracta, la forma que tiene la STUP de ver el futuro del mundo tras su paso exitoso por él, de una forma ambiciosa pero realista.

Además, la visión siempre es el punto de guía y de inspiración al que acudir en caso de incertidumbre. Como expresó Pedro Serrahima[6] en una entrevista sobre el caso de éxito de PepePhone : " Si sabes cómo eres, siempre sabrás lo que tienes que hacer ".

Típicamente se han establecido una serie de preguntas que todo emprendedor tendría que hacerse para ayudarse a obtener su visión; las más significativas son:

- *¿Por qué hacemos lo que hacemos?*
- *¿Qué nos gustaría que cambiara?*
- *¿Cómo vemos el futuro?*

Qué básicamente se resume en buscar los "problemas, necesidades, recursos" que nos han inspirado a emprender intentando acotar los límites de actuación, definiendo el qué nos gustaría cambiar y definiendo cuál es nuestra causa. Las respuestas a estas preguntas nos centran en el "*por qué*" más que en el "*qué*", de esta manera se consigue una mejor alineación con los clientes, ya que no nos centramos únicamente en el producto sino en la idea y en el "problema".

[6] Entrevista realizada. Aquí.

Misión:

Misión y visión son dos conceptos que, aunque distintos entre sí, se tienen que trabajar juntos. Por lo que se puede definir la misión como el rol que va a desempeñar la STUP para el logro de su visión, es la razón de ser de la empresa. Es la forma en la que se aspira a resolver el "problema" que se ha planteado en la visión y de cumplir así el propósito de cambiar el mundo.

La misión debe de ir ligada siempre a los medios y a los recursos de los que dispone la empresa, así como del entorno en el que es partícipe. Lo que implica que la misión tiene que ser realizable es el foco donde apuntar en el tiempo presente. Se podría interpretar como que la misión es el enfoque que se tiene en el presente y la visión la proyección del enfoque en el futuro.

Las preguntas que tendrían que hacerse para definir bien una misión son muchas más concretas que en el caso de la visión:

- *¿A qué se dedica la startup?*
- *¿Cuál es el público objetivo del producto?*
- *¿Qué quieren hacer?*
- *¿De qué forma ayudará a alcanzar la visión?*

La misión tiende a encajar en la intersección de los conjuntos de la Visión y de la realidad actual. Así que se presenta como una mezcla de ellos teniendo la visión presente y llevándola a nuestra realidad actual.

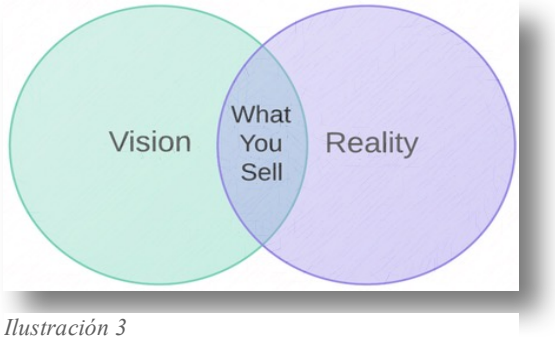


Ilustración 3

IDEA:

Cuando nos planteamos la idea de negocio, una vez que se han enfocado la misión y la visión que queremos hacer tangible mediante un STUP, hay que plantearse una serie de premisas:

- Se tiene que tratar de una **idea novedosa**: En este punto hay que hacer un análisis del mercado actual para ver si tenemos competencia y, en el caso de tenerla, comprobar si se comparte visión y misión y cómo está el mercado, viendo si hay espacio suficiente para todos. Hay que tener claro el punto diferenciador con la competencia, así como conocer cuáles son nuestras ventajas y desventajas frente a los productos existentes.

- **Identificar el sector** (*target)* al cual va dirigido: Muy importante a lo largo de la creación del producto software es tener claro a qué tipo de cliente vamos a dirigirnos y buscar la forma de empatizar con él para lograr captar mejor las posibles necesidades y sobre todo las más destacas

- **Encontrar el máximo número de problemas:** a los que se enfrenta el cliente y en los que podamos ayudarles: Tener claro todos los problemas nos hace tener una visión de conjunto y podremos crear una estrategia basándose en las prioridades que nos marquemos, teniendo como objetivo solucionar las más importantes pero sin perder la visión del conjunto de ellas.

- Realizar lienzo de **propuestas de valor y early adopters**: El uso de herramientas para empezar a encontrar las soluciones a los problemas anteriormente planteados es un camino muy útil y práctico para descartar ideas, aportar nuevas y hallar nuevo problemas.

- Asentar las ideas y elección del camino con una estrategia: Usando herramientas de elección como modelos *canvas*, en los que se puede apreciar una visión general de la idea de negocio, cómo sacar la rentabilidad y cómo atajar los puntos clave. Esta herramienta se ha convertido desde 2008 en la herramienta estrella para la gestión estratégica de un negocio. Se basa en nueve puntos clave divididos en segmentos que se pueden moldear en un solo folio. Además, tenemos que recordar que, al estar enfocados como una empresa Fullstack, tendremos que cubrir cada uno de los sectores que típicamente se delegan a otras empresas. Los puntos clave a tratar son: Segmento de clientes, propuesta de valor, relación con el usuario potencial, canal de distribución y comunicación, ingresos, actividades clave, recursos clave, socios y estructuras de coste.

Poniendo algunos ejemplos de casos de éxito, encontramos a:

Kickstarter.
Misión: Ayudar a las personas a lograr sus sueños creativos.
Visión: Redefinir la manera en la que los proyectos se financian.
Idea: Crear una web de micromecenazgo enfocada en proyectos creativos, en la que poner en contacto mediante un portal web, a modo de escaparate, a los impulsores de los proyectos y al público interesado en invertir, quedándose con un porcentaje del dinero recaudado.

Foursquare.
Misión: Ayudar a que tú y tus amigos descubran la ciudad.
Visión: Aumentar la facilidad de exploración del mundo.
Idea: Aplicación móvil geolocalizada en la cual cada usuario puede marcar lugares específicos que ha visitado e ir ganando puntos según va descubriendo nuevos lugares.

Caso práctico

En el caso particular que tratamos a continuación se ha seguido el esquema propuesto y se ha creado la idea final de negocio a través del análisis de la misión y de la visión.

La idea principal abarca varias fases. El primer acercamiento surge de forma natural al notar una necesidad propia, que es la de gestionar las clases de niños entre 10 y 18 años en las actividades extraescolares de carácter deportivo. Las siguientes aproximaciones van a ir surgiendo en las etapas de diseño y están en constante modificación. A raíz de la idea principal empezamos a analizar en busca de definir la misión y la visión.

Se llegó a la siguiente conclusión de:

Visión: Redefinir la educación mediante las nuevas tecnologías.

Misión: Mejorar la educación deportiva, creando una relación entre la institución, los deportistas y sus familiares en la que, sin esfuerzo, pueda fluir la información de forma útil y segura.

Idea: Crear un conjunto de aplicaciones que gestionen las comunicaciones y los datos capturados en las instituciones deportivas, con el fin de usar esos datos para mejorar los entrenamientos y poniendo especial atención al flujo de información entre padres y club para así agilizar las comunicaciones.

Por lo que nos pondremos a desarrollar una suite multiplataforma con unas características que nosotros consideramos mínimas; luego se podrán ir incorporando más a medida que las soliciten los clientes.

En la parte de estrategia analizamos la idea y se concluye que la mejor manera es realizar un proyecto siguiendo la filosofía FS. En el caso elegido, validamos la idea utilizando las herramientas anteriores y partimos de las siguientes premisas positivas y negativas:

- Una idea relativamente novedosa.
- Distanciados de la poca competencia que existe.
- Enfocado al sector deportivo.
- Tecnológicamente viable y que compete varias tecnologías.
- Rentable.
- Grupo reducido de integrantes.
- Carencia de financiación y de necesidad de tenerla al comienzo.
- Experiencia limitada en las tecnologías necesarias.

Uso de herramientas y tecnologías

Se van a explicar de manera más detallada las herramientas utilizadas para determinar la idea de negocio y estrategia general. Es importante reseñar lo imprescindible de que una empresa definida como FullStack haga partícipe a todos los integrantes principales en las reuniones de estrategia, ya que la alineación en las ideas es un pilar fundamental en esta filosofía.

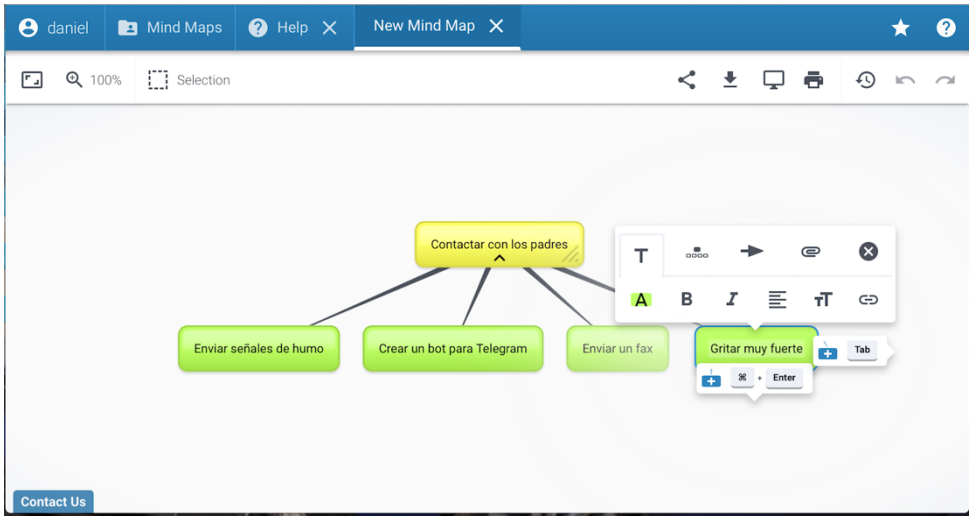
Brainstorming bubbl.us.

La tormenta de ideas es una técnica grupal para obtener nuevas ideas alrededor de un asunto. Su práctica consiste en postular un problema sobre el que trabajar, en ese momento los integrantes del grupo empiezan a aportar ideas, que en principio no se descarta ninguna sin antes analizarla, por muy alejada de la realidad que parezca, de esta manera se van creando mapas mentales de ideas conjuntas. Los preceptos de esta herramienta consisten en que, trabajando la creatividad en grupo se pueden obtener mejores ideas que de manera individual, ya que la creatividad aumenta enlazando las propuestas entre los participantes. Generalmente se suelen hacer rondas y en cada ronda se analizan las propuestas.

Se examinaron las opciones tecnológicas que existían para tratar esta técnica; se encontró una herramienta que tenía varias ventajas, tanto sobre trabajar en un medio físico (pizarra, papel) como sobre utilizar software de diagramas. La herramienta se llama **bubbl.us** y es un software online enfocado precisamente al brainstorming.

La ventaja de este software radica en: lo sencillo y rápido de usar, la posibilidad de trabajar en remoto compartiendo la misma pizarra online, varios formatos de exportación y la persistencia digital de los mapas creados.

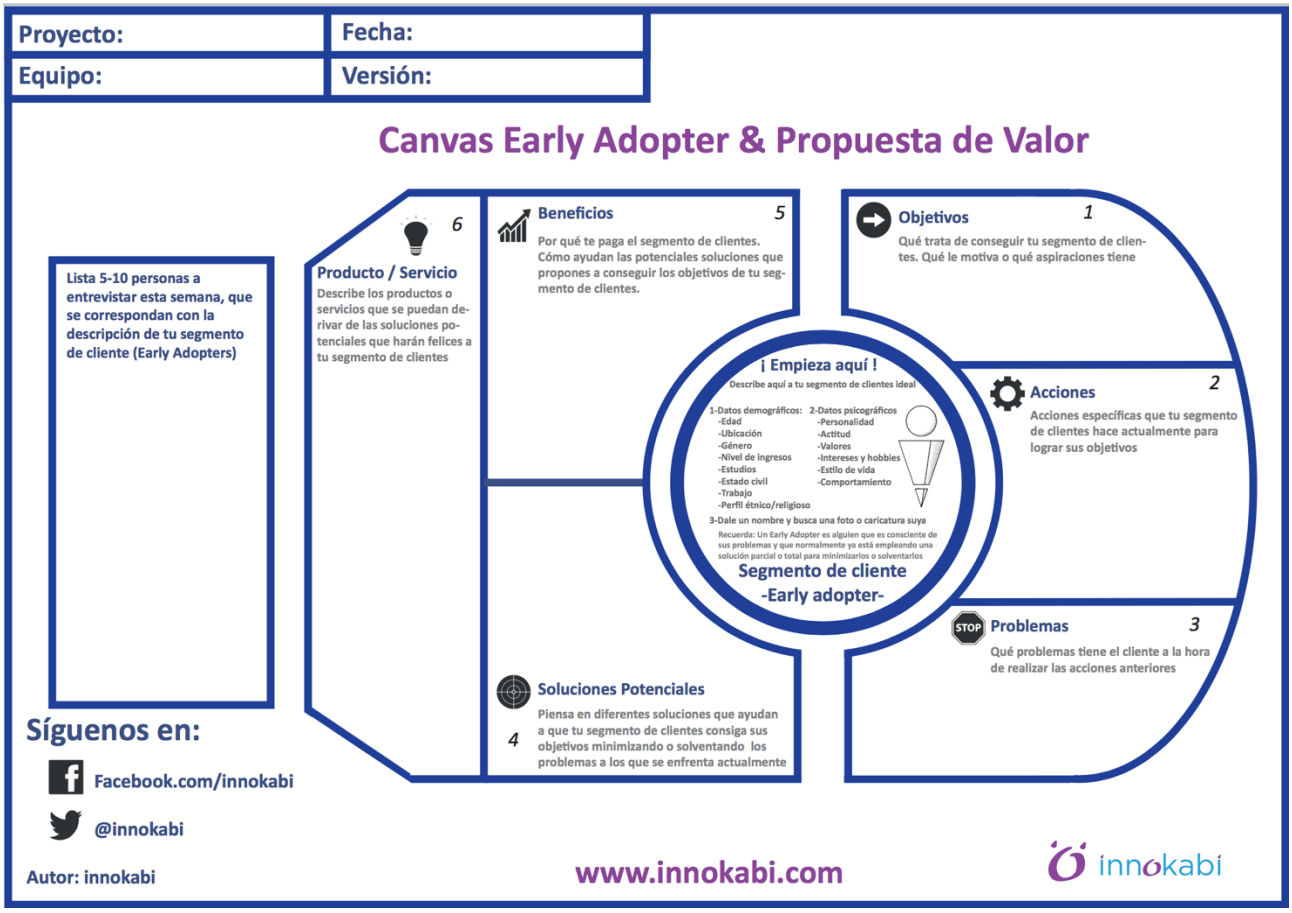
Su funcionamiento es sencillo, el programa nos facilita una pizarra vacía en la cual podemos generar mediante atajos de teclado y de forma muy rápida nuevas burbujas. Se parte de una principal, la cual llevará escrito el tema sobre el que dar ideas, después se puede crear una jerarquía de nuevas burbujas, que son las ideas aportadas. El programa permite una variedad de formatos como color, bold, tamaño…; permite organizar las burbujas de varias formas, árbol, parrilla y circular y añadir flechas y documentos adjuntos.



Canvas y propuesta de valor[7]

Consiste en un método de representación visual con el que obtendremos información muy valiosa sobre la estrategia de negocio, que se compone típicamente de al menos tres pilares que están presenten en un lienzo: el perfil del cliente, en el que se explican las características encontradas más importantes del *target* objetivo; el mapa de valor, que es lo que la empresa puede ofrecer al target elegido y el valor que puede tener para estos; y el tercer pilar es el encaje, que relaciona estas dos partes.

Como representación de este sistema se ha encontrado como una buena opción el *Canvas Early Adopter & Propuesta de valor* de Innokabi, siendo bastante completo y sencillo de usar.



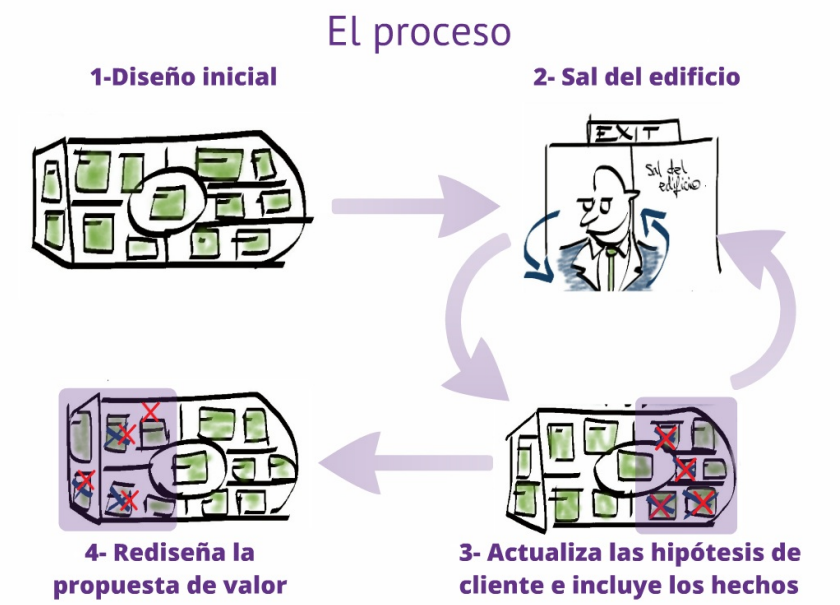
[7] Se puede ampliar esta información en los siguientes sites:
https://www.leadersummaries.com/ver-resumen/disenando-la-propuesta-de-valor
http://innokabi.com/early-adopters-5-claves-para-detectar-a-tus-primeros-clientes/
http://innokabi.com/lienzo-de-propuesta-de-valor-descubre-que-quieren-tus-clientes/

En este canvas se hace mención a los *Early Adopters*, esta es una denominación del cliente ideal para tu producto, suele ser el primer cliente en comprar el producto y el que está más receptivo a realizar un *feedback*, a asumir errores de las primeras versiones, etc.

Este modelo tiene una subdivisión en 7+1 bloques, siendo el último una confirmación a modo de entrevista con los earlyAdopters que has definido a modo de validación de las características obtenidas en el canvas. Y se llega a la estrategia de negocio a partir de las necesidades del cliente.

Su funcionamiento básico consiste en:

1) Mediante una reunión se define el cliente potencial ideal, tanto características demográficas como psicográficas. Se suelen sacar varias propuestas de clientes objetivos para quedarse con tres.
2) Detectar cómo piensa el cliente objetivo, plasmando sus objetivos, problemas/necesidades y acciones. Se realiza en el punto 1,2,3.
3) Diseñar la propuesta de valor a través de las necesidades que hemos captado del cliente, aportando soluciones viables emparejándolas con los problemas obtenidos. 4.5.6
4) Obtener los productos y servicios a partir de las soluciones planteadas.
5) Comprobar con una serie de clientes potenciales reales que los conceptos planteados son válidos; en caso contrario, realizar los ajustes sobre el punto 1 y repetir el ciclo.



Metodologías de trabajo y organización

Explicación del proceso

A lo largo del desarrollo de un producto y en el entorno laboral de una empresa se hace imprescindible tener fijados unos estándares de trabajo y de organización, como los que se explican a continuación:

Metodologías de desarrollo: En el desarrollo del software tienen como fin planear, estructurar y controlar el proceso de creación de un sistema de información, con el fin de mejorar el éxito y ser más eficiente.

Comunicación: Establecer los diferentes canales de comunicación entre los miembros del equipo, clientes …

Repositorio de documentos y control de versiones: Se tiene que establecer un espacio común donde persistir, comunicar y acceder a todo tipo de información. Además de fijar la herramienta para el control del software que se está desarrollando.

Metodologías de desarrollo

En sus inicios, las metodologías hacían mucho énfasis en el control del proyecto de manera que estuviera todo perfectamente definido, tanto los roles de los integrantes del equipo, los artefactos, entregables, fechas… así como hacer un fuerte seguimiento de las partes y dejar todo perfectamente documentado y acordado bajo contrato. A estas metodologías se las denominan a día de hoy como "tradicionales". Se ha contrastado su funcionamiento, sobre todo en proyectos de gran envergadura y con un gran número de trabajadores.

Algunos ejemplos basados en estas metodologías son:

RUP (Rational Unified Process): Es una metodología dividida en cuatro fases: Inicio, Elaboración, Desarrollo y Transición. En cada una de las partes se tienen que realizar una gran cantidad de artefactos de manera iterativa que conllevan la creación de multitud de diagramas, especificaciones y pruebas. Está basado en un modelo en cascada.

MSF (Microsoft Solutions Framework): Tiene un planteamiento de metodología espiral incremental bastante amplio, en la que cada vuelta sigue una variante del modelo tradicional de cascada en cinco fases. Al comienzo de cada una de ellas se tiene que validar el diseño y después de su cumplimiento hay un registro en forma de entregable que se exige antes de la siguiente iteración

Actualmente es cada vez es más común que en proyectos que no tengan tanta envergadura no usar este tipo de metodología, que requiere una gran cantidad de tiempo y esfuerzo en crear documentaciones y validaciones; además no están muy optimizadas para cambios constantes de requisitos teniendo un tiempo de respuesta muy elevado. Por lo que en la mayoría de las empresas actuales no suele ser eficiente recurrir a estas metodologías. A raíz de esto surge el manifiesto Ágil, que son los principios en los que se basan estas nuevas metodologías de trabajo. Sus pilares fundamentales son cuatro:

1) **Individuos e interacciones** sobre procesos y herramientas.
2) **Software funcionando** sobre documentación extensiva.
3) **Colaboración con el cliente** sobre negociación contractual.
4) **Respuesta ante el cambio** sobre seguir un plan.

Con estos pilares como base y de forma natural se han ido creando nuevos modelos más ligeros, más adaptables a cambios y preparados para una estructura organizativa más horizontal que los tradicionales y encajan perfectamente con las características que requieren las empresas de tipo **startup** y los desarrollos **fullstack**.

Las diferencias esenciales que presentan ambas tecnologías son las siguientes:

Ágil	*Tradicional*
Basadas en eucarísticas provenientes de prácticas de producción de código	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo
Especialmente preparados para cambios durante el proyecto	Cierta resistencia a los cambio y compleja
Impuestas internamente (por el equipo)	Impuestas externamente
Proceso menos controlado, con pocos principios	Procesos mucho más controlados con numerosas políticas y normas
No existe contrato tradicional o al menos es bastante flexible	Existe un contrato prefijado
El cliente es parte del equipo de desarrollo	El cliente interactúa solo con el jefe de equipo en reuniones concertadas
Grupos pequeños (<10 integrantes) y trabajando en el mismo sitio	Grupos grandes y posiblemente distribuidos
Pocos artefactos	Más artefactos
Pocos roles	Más roles
Menos énfasis en la arquitectura del software	La arquitectura del software es esencial y se expresa mediante modelos

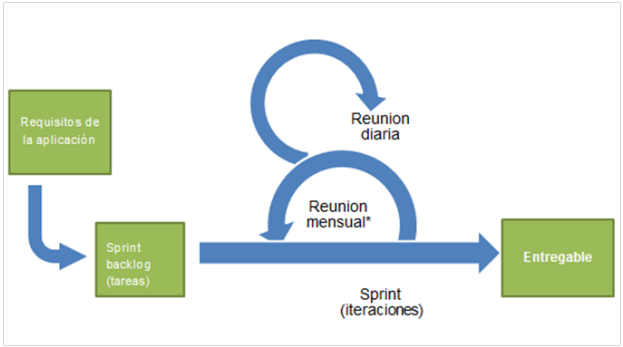
Las metodologías ágiles más conocidas y usadas son:

SCRUM : Esta metodología se caracteriza por:

- Tener un modelo de desarrollo incremental, el cual va a ir creando un producto por partes y de forma iterativa, creando entregables en periodos muy cortos. Cada uno de los entregables va a añadir alguna funcionalidad al proyecto. Cada una de estas iteraciones se llaman **Sprint** y pueden durar desde unos días hasta un par de meses, estos a su vez se dividen en tareas que de manera ideal no tendrían que superar 2 jornadas de trabajo.

  Un gran control sobre las actividades. Se realiza una reunión exhaustiva al comienzo de cada uno de los sprint. También de forma diaria se lleva un control de los procesos y del equipo en forma de reuniones al inicio de las jornadas de duración de no más de 15 minutos. En estas se refleja el estado de las tareas y se salvan los problemas que vayan surgiendo. Al finalizar el Sprint se produce otra reunión para revisarlo y comprobar que se haya completado, cada miembro aportará sus impresiones y así se conseguirá mejorar el proceso.

- Enfoque en los requisitos principales. Los sprint han de ser diseñados de dentro hacia afuera, empezando por los requisitos fundamentales que dotan de funcionalidad al producto software que se está trabajando. En siguientes iteraciones se van incluyendo los requisitos secundarios. Una vez terminado y probado un sprint, este no debe volver a abrirse.



Programación Extrema (XP)[8]: Como la gran mayoría de metodologías ágiles, pone más atención en la adaptabilidad que en la previsibilidad, siendo idónea para proyectos poco previsibles y cambiantes. Tiene 5 pilares:

- Simplicidad: Un diseño simple facilita un buen desarrollo. Esto se aplica desde la parte de codificación hasta la de documentación, atendiendo a un buen nombre de las variables y funciones y de una programación bien estructurada y legible. Este principio

[8] Referencia: XP : Métodologías Ágiles en el Desarrollo de Software /José H. Canós, Patricio Letelier y Ma Carmen Penadés DSIC -Universidad Politécnica de Valencia Camino de Vera s/n, 46022 Valencia
Enlace a :Enlace

tiene gran importancia en las interrelaciones de los miembros haciendo más sencillo el trabajo colaborativo.

- Comunicación: En todos los ámbitos, entre los compañeros programadores: ya que el código se presenta en esta metodología como una propiedad compartida, por lo que el código tiene que estar autodocumentado, tener documentación de ejemplo, usar estándares de codificación, etc. También colaborando con el cliente de una forma constante y sin mucha jerarquía de intermediarios. Usos de las tarjetas CRC, que son como una herramienta de brainstorming para crear soluciones de POO de forma colectiva.

- Retroalimentación: Se realizan iteraciones muy cortas, por lo que se encuentran antes errores en los requisitos o en la funcionalidad. Al estar en contacto constante con el cliente se hace una captura de cambios o errores de concepto de forma muy temprana.

- Coraje: Tiene que estar presente durante todo el desarrollo, y se muestra a la hora de saber desechar un código, a la hora de programar para el hoy y no para el mañana, tomar decisiones de manera ágil, etc.

- Respeto: Entre todo el conjunto del equipo. Dado el gran trabajo colaborativo que demanda esta metodología es un requisito fundamental tenerlo siempre presente, ya que respetar el trabajo del resto hace que no se perjudique la autoestima y mantiene un buen ritmo de producción.

Métodos Lean: Están basados en "LeanStartup" (que no deja de ser un conjunto de metodologías) diseñadas por Eric Ries[9], que se concibieron para ayudar a levantar una empresa desde cero. Centrándonos en la parte de desarrollo de software tiene como principal ideología buscar y eliminar las prácticas ineficientes y centra todo el esfuerzo en la producción durante las fases de desarrollo. Sus características son las siguientes:

- Búsqueda del Producto mínimo viable (PMV): Consiste en recoger con el mínimo esfuerzo la máxima cantidad de información posible. Hablando en código, sería el equivalente a crear un código muy simple pero que cumpliera con la necesidad fundamental que se cree que tiene el cliente y evaluar la demanda que suscita y evaluar sus impresiones. Se pone en práctica con los Early adopters.

- Producción continua: También llamada integración continua, consiste en introducir todo código testeado del desarrollo a producción de manera constante. Lo que ayuda a aligerar de una manera notable los tiempos de producción.

[9] Autor del libro de referencia *The lean Startup*

- *Split-test*: Se usa en varios ámbitos y consiste en dar dos versiones de un producto a un usuario y valorar sus reacciones. En el ámbito de la integración continua podría ser cambiar una funcionalidad de un tiempo y valorar si la nueva es mejor aceptada.

- Crear-Medir-Aprender: En toda la fase de desarrollo de un proyecto Lean el factor de aprender es constante y recurrente siendo el pilar fundamental del progreso del producto y de la empresa. En el proceso del software la equivalencia sería: crear-codificar- verificar/aprender

Existen multitud de metodologías que no se han comentado y muchas variaciones de las presentadas. No obstante, como se ha visto, todas comparten en mayor o menor medida las mismas filosofías y es más lo que las une que lo que las separa.

Entre las metodologías ágiles y las tradicionales puede parecer en un principio, que ambas tienden a ser contrarias; la verdad es que gracias a que las metodologías ágiles tienden a ser muy flexibles se pueden utilizar de forma conjuntas ambas o adaptarlas a las necesidades de cada proyecto y empresa creando su propia metodología.

Comunicación

Ya se ha visto anteriormente que en una startup la comunicación es un pilar fundamental. Establecer las vías de comunicación tiene que ser uno de los primeros pasos a determinar.

- Comunicación interna: Para la comunicación interna habitualmente se tiende a usar el correo electrónico, reuniones presenciales, comunicados en plataformas, etc. Pero debido a la particularidad de la forma de trabajar en una STUP se demanda una forma de comunicación más ágil. Por lo que se tiende a usar:

- Correo electrónico para comunicaciones formales y de difusión.
- Mensajería instantánea, para la comunicación rápida entre los miembros del equipo. Normalmente las específicas permiten crear diferentes grupos y contienen herramientas para adjuntar documentos.
- Programas de videoconferencias, ya que son habituales las reuniones en remoto.

-Comunicación con los clientes: Establecer los canales de comunicación con el cliente desde un principio para poder trabajar mano a mano desde el primer momento. La vía más habitual continúa siendo el correo electrónico y la comunicación telefónica, pero es importante designar los enlaces de comunicación y que sean estos responsables los que hagan la comunicación al resto del equipo, para evitar que se pueda perder o tergiversar información. Las reuniones presenciales son también muy frecuentes, por lo que es necesario disponer de un lugar donde poder realizarlas.

Repositorio de documentos y control de versiones

Un aspecto importante es el repositorio de documentos, que tiene que ser un espacio común, sincronizado y en línea, donde se va a almacenar la información y los documentos clave de la empresa y de los proyectos, para que así todos los integrantes tengan actualizada y disponible la información. Al comienzo de la STUP es factible utilizar herramientas online basadas en la nube para este propósito, existen muchas alternativas gratuitas y otras de pago que aumentan según el uso. Cuando el tamaño de la empresa crece lo habitual es tener un servidor propio y realizar conexiones por FTP.

El control de versiones es una herramienta software utilizada para la gestión de los cambios que se realizan sobre algún elemento software. En un proyecto es un elemento clave, ya que no sólo protege el trabajo realizado de algún posible problema, sino que permite una sincronización entre todos los desarrolladores que trabajan en el mismo proyecto, pudiendo acceder a los cambios producidos. Existen muchas herramientas de control de versiones, algunas de ellas especializadas en un ámbito concreto, pero todas ellas comparten esta estructura:

- Repositorio: Es el espacio donde se localizan los datos actualizados y todo el histórico de cambio. Suele tener una copia en local, que se actualiza con mucha frecuencia, y una copia en un servidor en remoto a la que se suben cada más tiempo.
- Módulo: Es el espacio de trabajo dentro de un repositorio.
- Revisiones o versiones: Es la forma de identificar de forma inequívoca cada una de las subidas de datos.
- Publicar: Sucede cuando realizamos una nueva incorporación a una copia local
- Rama: Es una copia del módulo principal que evoluciona de forma independiente.
- Mezclar: Realizar una fusión entre dos ramas.

Caso práctico

Vamos a detallar las elecciones de herramientas y tecnologías usadas en nuestro caso particular de STUP.

Elección de metodología

Desde un primer momento se tenía claro que la metodología no sería tradicional, por estos siguientes puntos:

1. Aunque teníamos muy definida la idea, en un primer acercamiento a los requisitos nos dimos cuenta que podían ir fluctuando mucho a lo largo del desarrollo; las metodologías ágiles facilitan esta tarea.
2. El proyecto va a estar formado por un equipo muy pequeño; en su desarrollo principal únicamente por dos desarrolladores y, aunque hay previsión de que se doble en un corto espacio de tiempo, en cualquier caso nunca sería suficiente para que compensara el gasto de esfuerzo del control de las tradicionales.
3. El tiempo disponible para la realización del proyecto es importante y el exceso de documentación en las tradicionales hace que compense su adopción.

Se quiso racionalizar más la decisión y acudimos a una encuesta creada por Ian Sommerville[10], en la que a través de una serie de preguntas se puede determinar qué metodología es la acertada. Si la respuesta predominante es el *Sí* tendríamos que inclinarnos por una metodología ágil, en caso contrario por una tradicional.

¿Es necesario una especificación?	No
¿Los clientes son inaccesibles?	No
¿Se trata de un proyecto de grandes dimensiones?	No
¿Es un sistema muy complejo?	Sí
¿Se trata de un producto al que se le prevé mucho tiempo de vida?	Sí
¿Las herramientas de desarrollo son limitadas?	No
¿El equipo de trabajo está distribuido?	No
¿El equipo está acostumbrado a documentar?	No
¿El equipo tiene conocimientos técnicos limitados?	No
¿El sistema está sometido a regulaciones legales?	No

Como se ve en el cuadro, las respuestas que predominan son negativas, por lo que coincide con nuestra predicción anterior, ágil. Además de para esta elección, también nos es útil para decidir el tipo de metodología ágil, al tener ya claro los conceptos. También queremos añadir los siguientes conceptos, que junto a las respuestas anteriores darán una mejor visión:

[10] Ampliación en el libro Software Engineering de Sommerville ISBN-13: 978-0137035151

- Equipo muy reducido (en un principio de 2 a 4 integrantes).
- Incertidumbre en tiempos (tecnología nueva).
- Equipo multidisciplinar (Fullstack).
- Proyecto dividido en módulos (multiplataforma).
- Cambios de roles (el equipo no está asignado a un módulo en concreto).

A raíz de estos requisitos, vemos que Scrum, al tener como base de la metodología plazos de entrega muy definidos (Sprint), no sirve, ya que existe una gran incertidumbre temporal provocada por no haber trabajado con muchas de las tecnologías que hay que implementar. Además, en Scrum, una vez que se cierra una tarea o un sprint, este se cierra y no se vuelve a modificar, y dado se está aprendiendo es bastante probable que tengamos que volver a incidir en algunas tareas para hacerlas más eficientes. Otro factor que hace que se descarte, es que la forma de trabajo del *Scrum Team* está enfocada de forma individual y con roles asignados y en nuestro proyecto, al estar formados por un equipo fullStack, es habitual el cambio de roles y de módulos durante el proyecto.

Tras este análisis, la elección de metodología que se va a usar de base va a ser la **XP**, por los siguientes motivos: Aplica el principio del código común y la buena comunicación, algo fundamental para nuestro tipo de proyecto, ya que se intercambiaran los roles e incluso se hará *pair-programing*. Está enfocadas para grupos muy pequeños y está basada en prueba y error.

Por lo tanto, nuestra metodología va a tener de base Extream Programing, al cual vamos a añadirles una serie de modificaciones para adaptarlo a nuestro proyecto y al desarrollo de un proyecto FS en una STUP:

- XP usa un modelo de **ciclo de vida** llamado ModeloXP, que no conlleva una distinción muy clara de las etapas del proyecto, ya que asume que las especificaciones irán llegando según las necesidades del cliente, aunque como es lógico prevé unas especificaciones iniciales obtenidas en las primeras reuniones. Debido a que nuestro proyecto nace de una idea propia y nosotros presentaremos un producto funcional, nos es necesario en esa primera etapa inicial tener un carácter más enfocado al modelo en cascada, ya que no se prevén cambios significativos en las primeras tandas de historias de usuario, no obstante, seguirá teniendo un carácter de ModeloXP e irá incorporando las funcionalidades que demanden los clientes finales, creando así una especie de modelo incremental.

- De forma usual, en esta metodología se escriben las pruebas antes de empezar a codificar, en nuestro caso, como se desconocen muchas de estas tecnologías, no vamos a exigirnos la implementación anterior a la codificación y podremos plantearlas antes, durante o después.

- Aunque la metodología contempla el aprendizaje continuo del equipo a manos del resto de los compañeros, queremos hacer especial mención a este punto, ya que debido a la falta de experiencia el aprendizaje va a ser una constante a lo largo del proyecto y se ha establecido que en cada historia de usuario y antes de cerrar una iteración se comparta lo aprendido con el resto del equipo, a través de cursos y tutoriales creados por los compañeros. Se ve como parte crucial del proceso, a pesar del tiempo que supone la realización y la enseñanza, pero se compensa a corto plazo, pudiendo solicitar ayuda a los compañeros en determinadas

situaciones, intercambiar puestos, potenciar el desarrollo en situaciones en las que sólo se trabaje sobre un artefacto, además de cumplir la misión de añadir valor a todos los miembros del equipo ampliando sus conocimientos.

Una vez elegida la metodología es recomendable usar herramientas software para llevar el control de las tareas y poder echar la vista atrás cuando se finalizan estas así como poder mejorar los procesos. En el mercado existen muchas alternativas, para este propósito algunas muy complejas y que llevan un control minucioso como Project de Microsoft, la cual te permite sacar graficas de los datos estimados, costes, tiempos, etc. Y otras especialmente simples como Trello, que únicamente muestra un **canvas** web muy personalizable. En nuestro caso se ha optado por el uso de **Trello** como herramienta principal ya que coincide muy bien con la metodología ágil elegida y por ser un equipo tan pequeño; una herramienta sencilla ahorra tiempo

Comunicación

Para la comunicación informal diaria se establece como programa de mensajería **Telegram**. Se ha elegido este frente a otras opciones como Slack, Hangouts o WhatsApp ya que tiene como características únicas el envío de archivo de todo tipo y de gran tamaño, sin la necesidad de aplicaciones de terceros; también tiene la posibilidad de crear canales de información (únicamente de lectura) y funciona mediante alias, por lo que no es necesario compartir el número de teléfono personal.

Skype ha sido la plataforma elegida frente a Hangouts, ya que tiene una aplicación nativa para todos los sistemas operativos, cosa que su competidor directo de google no tiene; además su utilización está muy extendida, lo que conlleva menor dificultad a la hora de reunirse.

Para el correo electrónico se han abierto cuentas gratuitas en los servidores de **google**, se ha optado por esta empresa dada su escalabilidad; google permite a posteriori agregar un dominio privado y pasar a cuentas empresariales de una forma sencilla.

Repositorio de documentos y control de versiones

Existen muchas alternativas para guardar repositorios de archivos en la nube, Dropbox, OneDrive, SugarSinc, iCloud, GoogleDrive... La única diferencia notable entre ellos que se ha encontrado ha sido la capacidad en su versión gratuita, siendo la que más capacidad tiene OnDrive de Microsoft con 1TB de almacenamiento. Aun así, se ha optado por usar Dropbox, ya que es la habitual de los integrantes del equipo, y gracias a un acuerdo del Dropbox con las universidades facilita 25G extra de almacenamiento, cantidad más que suficiente para el uso que se le va a dar.

Las decisiones sobre el control de versiones son tres:

Tipo de control de versiones.

Repositorio donde se almacenará.
Forma de uso: consola, interfaz, plugings.

En el control de versiones se ha tenido que decidir entre dos tipos GIT o Subversión. Sus diferencias más notables las podemos encontrar en la siguiente tabla:

	Subversion (SVN)	*GIT*
Control de versiones	Centralizada	Distribuida
Repositorio	Un repositorio central donde se generan copias de trabajo	Copias locales del repositorio en las que se trabaja directamente
Autorización de acceso	Dependiendo de la ruta de acceso	Para la totalidad del directorio
Seguimiento de cambios	Basado en archivos	Basado en contenido
Historial de cambios	Solo en el repositorio completo, las copias de trabajo incluyen únicamente la versión más reciente	Tanto el repositorio como las copias de trabajo individuales incluyen el historial completo
Conectividad de red	Con cada acceso	Solo necesario para la sincronización

Tabla comparativa obtenida de 1&1

Hemos observado que tiene más ventajas GIT, ya que permite trabajar en local y se dispone de todas las copias de cada modificación, además el mercado está tendiendo a GIT, por lo que tendremos previsiblemente mejor soporte en un futuro.

En cuanto a los repositorios, vuelve a existir mucha oferta sin que existan muchas variaciones entre la mayoría de ellos. En muchos casos su modelo de negocio parte de un modelo gratuito pero público, por lo tanto cualquier persona podrá ver y descargarse el código, y si se quiere optar por repositorios privados hay que abonar una mensualidad. Esto no crearía ningún problema si estuviéramos desarrollando software libre, en nuestro caso tendremos que mirar las diferencias entre unos y otros y valorar la oferta y las limitaciones de unos y otros. Los repositorios más usados más conocidos son Github y Bitbucket, por lo que vamos a limitar la comparativa a estos dos:

Github únicamente permite **repositorios públicos** en su versión gratuita, si se quiere optar por privados se tienen que abonar 7$ al mes; existe la posibilidad de acceder a una modalidad gratuita para estudiantes, esta modalidad nos permitiría crear 5 repositorios privados. Además, cuenta con otras versiones superiores, que incluyen más funcionalidades relacionadas con la gestión de equipos de trabajo sobre los repositorios.

Bitbucket, a diferencia del Github, permite en su versión gratuita disponer tanto de repositorios públicos como privados de manera ilimitada, pero como contra limita el número de usuarios por repositorios a 5. Si queremos quitar esta limitación se tendría que pagar una mensualidad de 10$.

Dado que la estimación de crecimiento del equipo de desarrollo es a 4 programadores la limitación de **Bitbucket** no es un problema (si aumentara el equipo se podría cambiar al plan premiun) y siendo imprescindible usar un repositorio privado, la mejor opción es, sin duda, Bitbucket.

Una vez elegido el sistema y el repositorio que vamos a utilizar, hay que elegir la herramienta principal con la que vamos a trabajar. En esta parte, aunque la empresa pueda dar unas pautas, suele ser una elección libre del desarrollador; en nuestro caso vamos a optar por hacer uso de la consola de comandos y de la herramienta **Sourcetree** para poder tener una representación más visual. Existe también la alternativa de utilizar añadidos para las IDEs pero se han descartado como uso principal, ya que en ocasiones no están bien depuradas y su forma de uso cambia con cada IDE, por lo tanto es más sencillo realizar las acciones siempre de la misma manera, independientemente de la parte del producto que se esté desarrollando; esto es importante en nuestro caso ya que, según la metodología escogida, cada desarrollador va a poder rotar por distintas partes del proyecto y así siempre estará familiarizado con el sistema..

Uso de herramientas y tecnología

Trello

El funcionamiento de esta herramienta web es muy sencillo. Tiene un uso jerárquico, que se compone de:

- Un **escritorio**, que contiene separados, por equipos de trabajo, un conjunto de tableros.
- Los **tableros**, que funcionan de manera independiente entre sí. Cada uno puede tener una configuración distinta (colores, sobrenombres, equipos de trabajo…), y además pueden ser compartidos con otros usuarios, teniendo así tableros para organización personal y para equipos.
- En cada tablero se organizan distintas **listas** en forma de tareas, y cada tarea contiene un conjunto de tarjetas.
- Cada **tarjeta** es personalizable, tanto con texto como con *stickers*; además admite adjuntar archivos (vinculándolos antes con Dropbox, GoogleDrive, OneDrive, etc).

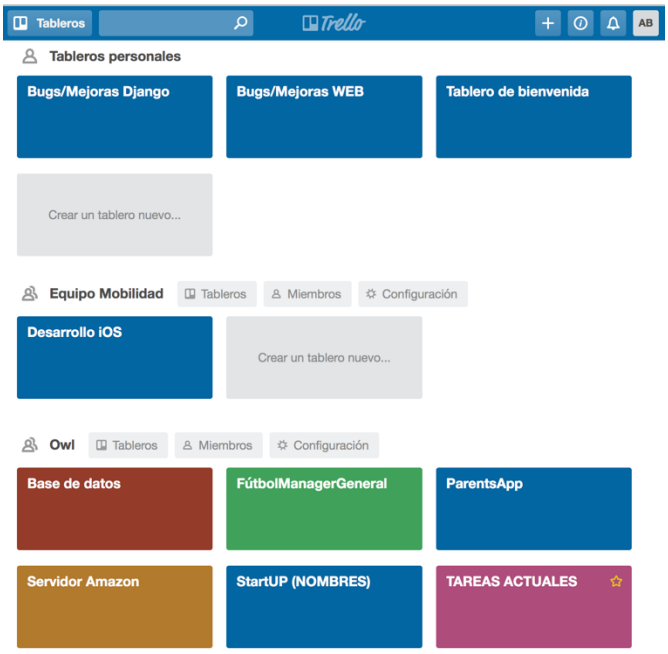


Ilustración 5

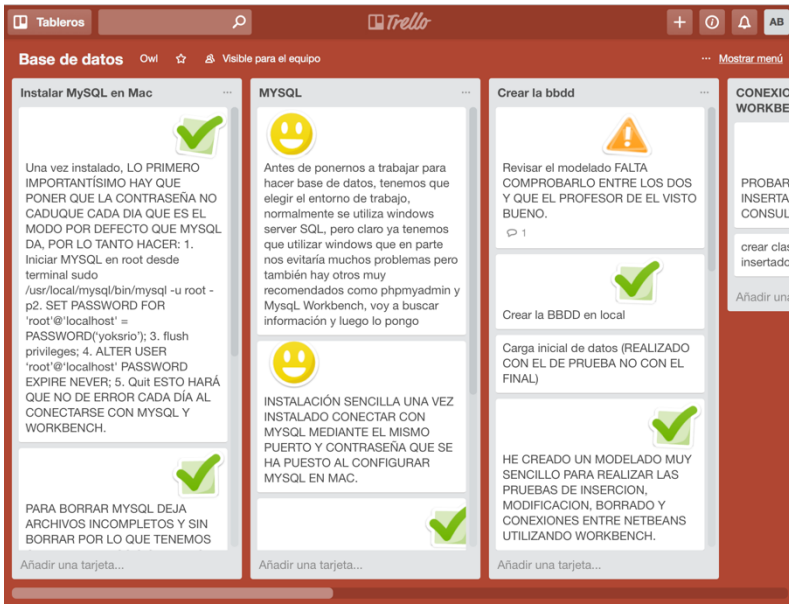


Ilustración 4

Mediante este sistema tan sencillo y nada pretencioso, se pueden organizar y hacer un seguimiento de cualquier metodología ágil y también es útil para la organización personal de tareas diarias o semanales. También dispone de aplicaciones móviles, que funcionan de la misma manera que la interfaz web.

GIT

Se va a explicar de forma muy resumida los pasos más esenciales para utilizar la herramienta git desde terminal con el fin de hacernos una idea de su funcionamiento:

1. Instalación y creación del repositorio

   Algunos sistemas operativos incluyen Git, en el caso contrario tendríamos que descargarnos la versión que corresponde con nuestro SO

   Repositorio local: únicamente hay que posicionarse en la carpeta que queramos tener la gestión de versiones y escribir *git init*. El sistema indicará que es todo correcto

   Repositorio remoto: Si ya tenemos un repositorio remoto y queremos agregarlos mediante el comanto *get remote add[nombre][url]* así podremos acceder a él y mediante el comando *get fetch [remote-name]* podemos recuperar los datos; de esta manera tendremos los datos del remoto referenciados con los locales. Para conocer los repositorios remotos que tenemos en el sistema hay que
   escribir el comando *git remote*.

2. Flujo de trabajo.

   Cuando queramos conocer el historial de cambios que se han producido, podremos llamar a *git log*. Con *git status* entre otras cosas, podremos ver si tenemos alguna acción pendiente.

3. Commit:
   Se realiza cuando queremos guardar de forma local el estatus actual de los archivos seleccionados del workspace. Al realizarlo tendremos en nuestro repositorio local una nueva copia de los datos. Para realizar esta acción hay que escribir *commit –a [comentario referente al commit]*

4. Push:
   Al realizarlo estamos subiendo los commit al repositorio remoto, es importante tener la última versión de los datos en remoto antes de poder subir los nuestro.

5. Pull:
   Descarga la última versión que esté alojada en el repositorio remoto al local. Es habitual que se tengan que tratar diferencias entre las versiones .

6. Branch:
   Git nos permite tener más de una rama, de un mismo proyecto apra cambiar entre ramas tendremos que utilizar el comando *git* por ejemplo para des logearnos *git checkout [rama]* y para logearnos en una nueva

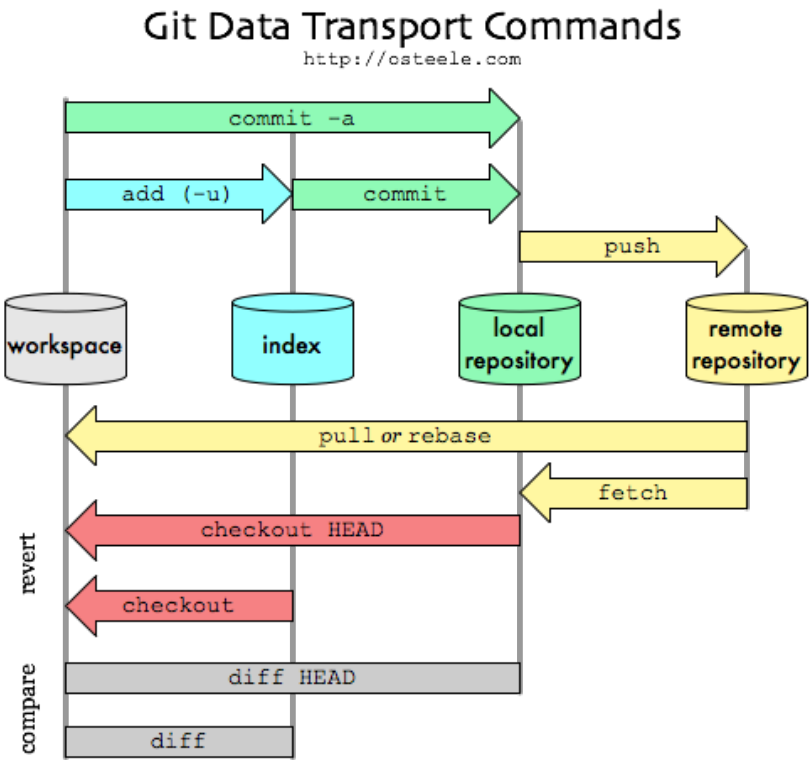


Ilustración 6 Representación gráfica de los comandos habituales

Requisitos

Explicación del proceso

Tradicionalmente la captura de requisitos[11] ha sido una tarea muy lenta, costosa y que a la larga traía bastantes problemas. Originalmente se intentaba hacer una captura de requisitos mediante una serie de reuniones con el cliente y se intentaba obtener la totalidad de ellos; cuando se tenían se empezaban las gestiones y el análisis, para así crear un modelo detallado a seguir en las siguientes fases de desarrollo. Por lo que era fundamental intentar que el cliente no cambiase los requisitos ya que significaba un coste importante en tiempo y en dinero.

Los problemas de cambios de requisitos suelen venir por:

- No conoce verdaderamente todas sus necesidades.
- Según avanza el proyecto, descubre necesidades nuevas.
- Entendimiento pobre del requisito.

Este problema era habitual solucionarlo mediante un contrato (aún se hace en proyectos extensos), lo que daba lugar a un cliente que no estaba satisfecho, por conformarse con algo que no quiere o por tener que desembolsar más dinero; además provocaba una pérdida de eficiencia por parte del equipo de desarrollo al tener que rehacer trabajo.

Ya que el enfoque es puramente STUP no va ser habitual encontrarse con proyectos con la envergadura suficiente para que usen una metodología con una métrica tan exigente, en la que sea necesario realizar una toma de requisitos cerrada; lo habitual es utilizar metodologías ágiles o tradicionales adaptadas.

La obtención de requisitos en una metodología ágil, tiene la ventaja de que no se hace un refinamiento de los requisitos hasta que esté cercana su implementación, lo que facilita replantearlos en el fututo teniendo en cuenta la situación de ese momento, ya que las necesidades o gustos del cliente han podido cambiar o la resolución de otro requisito ha evolucionado de una forma diferente, o bien ha revelado información que provoca el cambio del requisito a tratar, etc. Ya que cuanto más cercanos en el tiempo estén la especificación de requisitos y su desarrollo, menos posibilidades habrá de que este varíe y tengamos que repetir el trabajo.

No obstante, esta forma de tratar los requisitos no está exenta de problemas y puede provocar lo que se conoce como *ScoopCreep*. Esto sucede cuando se produce una avalancha de cambios no controlados que ponen en peligro el proyecto. Para evitarlo es importante definir ciertas condiciones y limitaciones para los cambios así como las nuevas inclusiones de requisitos. Una herramienta útil es el uso del *Product Backlog*.

[11] Requisito: *una condición o capacidad que debe estar presente en un producto, servicio o resultado para satisfacer un contrato u otra documentación formalmente impuesta. También conocido como Requerimiento. (PMBOK 5a edición)*.

El **Product Backlock**, es una lista viva de requisitos o conjuntos de requisitos, que sigue un concepto SMART[12], que va a indicar la prioridad a la hora de desarrollarlos. Esta lista se prioriza según las reuniones con el cliente y puede cambiarse y variarse según las necesidades. La toma de requisitos se hace mediante las **historias de usuarios**, estas se obtienen directamente del cliente y son unas plantillas en las que el cliente define las características que el sistema debe poseer (es indistinto el tipo de requisito que pida); este requisito se caracteriza por ser dinámico y flexible y se puede desplazar por el Product Backlock según su prioridad. El contenido de estas plantillas no sigue una norma y cada STUP tendrá que crear la suya propia dependiendo de su estilo y del proyecto en el que se trabaje. Kent Beck, cuando desarrolló este sistema, introdujo las siguientes: fecha, tipo de actividad (nueva, corrección o mejora), prueba funcional, número de historia, prioridad técnica y del cliente, referencia a otra historia previa, riesgo, estimación técnica, descripción, notas y el seguimiento, con comentarios, fechas de modificaciones.

Una de las **ventajas más notables** del desarrollo FullStack se ve claramente en la toma de requisitos. Un FS toma las funciones de un encargado de operaciones, pero con los conocimientos añadidos de la parte técnica de todas las capas de desarrollo. Algunas de las ventajas más importantes de este perfil, frente al tradicional de analista, es:

- Se eliminan intermediarios, lo que facilita el entendimiento con el cliente al haber un trato directo, evitando errores de comunicación al transmitir la información
- Visión técnica de todo el conjunto de tecnologías, lo que facilita una búsqueda de soluciones más efectivas y realistas junto al cliente.
- Experiencia real sobre la inclusión de esos requisitos, lo que permite hacer una valoración temporal más realista.
- Mejor abstracción de los requisitos técnicos a partir de los requisitos en bruto que facilita el cliente.

Hay varios factores a tener en cuenta a la hora de hacer la toma de requisitos, teniendo presente el tipo de metodología, el perfil de STUP y del desarrollador.

Es importante determinar qué tipo de captura de requisitos se va a realizar, si es con el usuario, con el cliente final, propia, con asesores… Además, las reuniones de tomas de requisitos hay que tratarlas de diferente manera, dependiendo en la fase en la que se encuentre el proyecto.

Primera reunión de requisitos o ***kick off***. Un problema intrínseco en las metodologías ágiles es la pregunta en la primera reunión de requisitos de "¿por dónde empezar?". Hay varias formas de enfrentarse al problema. Robert Cecil Martin[13] empezaba a tratar un proyecto como habitualmente se trataba en las metodologías tradicionales, para cuando se había aclarado borrar todo y empezar a componer desde cero, pero con ideas generales forzadas. También están metodologías como canvas, para empezar a obtener los primeros requisitos. Indistintamente de la forma que se use para elaborar los primeros requisitos hay que tener claro, que el fin de estas primeras reuniones es la de **esclarecer**

[12] SMART. Acrónimo inglés, en su traducción: Específicos, Mensurables, Alcanzables, Realistas y Tiempo determinado
[13] Robert Cecil Martin. Coautor del manifiesto Ágil.

el alcance del proyecto, y de acuerdo con las metodologías ágiles, el alcance obtenido hay que reducirlo hasta encontrar el proyecto básico sobre el que ir trabajando. Generalmente se empieza con un esbozo de la idea general con la que empezar a desarrollar.

Salvada la primera reunión y con idea principal desarrollada, las siguientes reuniones consistirán en ir incorporando las nuevas historias de usuarios y empezar a ir generando las nuevas ideas, que conformarán las siguientes iteraciones. En estas reuniones se tienen que mantener el enfoque global del proyecto y ver si las derivaciones de los requisitos actuales modifican los anteriores. Las conclusiones de estas reuniones se engloban en un ***scoop***[14]. En cada scoop puede haber más de una historia de usuario, pero tiene que estar definido un propósito, el cual al terminarse genere el entregable; se puede acordar ciertas características en el entregable, por ejemplo, se pueden definir que se entregue junto con una documentación, o un plazo de tiempo determinar o alguna característica específica, pero siempre tiene que guardar un estilo de alto nivel.

Sabiendo ya que esperar de las reuniones, ahora hay que tratar con la información obtenida en estas. Al contrario que con las metodologías tradicionales en las ágiles no existe el proceso de vida de un requisito de la forma clásica de **Obtención**, **Análisis**, **Especificación**, **Verificación y Aceptación**. La obtención y la aceptación se hacen en las reuniones comentadas, aunque la aceptación puede replantearse en sucesivas reuniones. El análisis y la especificación las hace directamente el desarrollador al enfrentarse a la historia de usuario, y la validación la plantea antes de codificar y la pone en práctica al terminar. Por lo que vemos que los requisitos no se analizan y se dividen en funcionales y no funcionales[15] en una fase posterior, sino que es el propio desarrollador quien los determina cuando está programando. Lo que si se hace en esta etapa es dividir entro los desarrolladores las historias a realizar, para ellos se organiza una reunión en el que se priorizan estas historias y se contabiliza el esfuerzo, que requieren. Los desarrolladores dependiendo de su fortalezas y debilidades escogen las historias en las que van a trabajar, teniendo presente la dificultad de estas, pudiendo designar a más de un desarrollador a una sola. Al usar una metodología ágil y tener reuniones diarias, se pueden reasignar esfuerzos si se captura que existe algún problema con alguna de ellas, con el fin de evitar que afecte al conjunto del scoop.

Siendo conocedores que en la STUP los grupos de desarrollo son muy pequeños, el trabajo en equipo es fundamental por lo que las siguientes características son clave:

- Fortaleza. Un punto muy importante es que en las reuniones de toma de requisitos tiene que existir una fortaleza conjunta para llegar a una conclusión que satisfaga a todos.

- Compromiso. El conjunto del equipo tiene que mostrar un compromiso para la tarea designada. Ya que es un esfuerzo conjunto, el compromiso de todos los integrantes es vital.

[14] Entrada al blog *It's a Delevery Thing* en el que Steven Thomas habla del *Agile Project Scope* aquí

[15] Los requerimientos funcionales son declaraciones de los servicios que proveerá el sistema, de la manera en que éste reaccionará a entradas particulares. Los no funcionales se refieren a todos los requisitos que no describen información a guardar, ni funciones a realizar, sino características de funcionamiento

- Comunicación. Si existe algún problema ser proactivo en la actuación para resolverlo, no esperar a que sea demasiado tarde, y comunicarlo al resto del grupo.

Caso práctico

En el desarrollo de nuestra aplicación partimos de un caso particular pero usual en las STU, en primera instancia nosotros somos los clientes.
Recordemos que el propósito es crear una aplicación con una estructura bien construida y funcional, la cual se presentará a distintos clientes finales que añadirán funcionalidades extra. Este sistema nos deja con la responsabilidad de crear nuestras propias historias de usuarios y dividir el proyecto en fases varias poniendo una meta a la que llegar y en la que ceder el control.

Al no tener un cliente final hemos tenido que hallar la forma de obtener las historias de usuario por nuestros propios medios, esto tiene la gran ventaja de que podemos dar las funcionalidades que queremos, que no tenemos que lidiar con el entendimiento de un cliente, tendremos las ideas más claras y más cerradas, al haber menos implicado será más sencillo llegar a un consenso, etc. No obstante, hay que tener en cuenta que mucho de lo que se puede considerar una ventaja al principio se puedo volver en contra, al no tener una línea marcada por un cliente que usar como referencia puede provocar el "síndrome del folio en blanco[16]"; desacuerdos entre los desarrolladores al tener ideas enfrentadas; intentar abarcar más de lo que se debe y terminar cogiendo manías de las metodologías tradicionales; falta de información sobre el campo en el que se está trabajando y suponer historias de usuarios que no son ciertas, etc. Para enfrentarnos a algunos de estos problemas hemos seguido los siguientes pasos:

1. Utilizar la herramienta canvas que ya se utilizó para obtener la idea principal, y repetir el procedimiento esta vez siendo más específico en los problemas y creando varios para cada uno de los posibles usuarios de la aplicación, cuerpo técnico, deportistas, familiares y aficionados. Una vez verificados las soluciones propuestas a los problemas se tomaron las necesidades conjuntas, lo que nos da un punto de partida y un esbozo de lo que pretende ser la aplicación.

2. Buscar ayuda. Aunque dentro de nuestro equipo teníamos un experto, se realizaron reuniones con gente experta en el sector, contamos con la ayuda de una institución deportiva, con un abogado especialista en protección de datos y de un conjunto de padres. De esta manera solventamos las carencias técnicas además de que nos sirvieron como parte de los early adopter en la realización del canvas.

3. Eliminar jerarquías. La opinión de todos los integrantes en las reuniones, tiene el mismo peso.

[16] Síndrome del folio en blanco, situación en la que un autor creativo, debido a un bloqueo, se siente incapaz de generar contenido por un determinado periodo de tiempo. Más información en *What is Writers Block*, aquí.

4. Siempre que la historia de usuario lo permita, estás historias se van plasmando usando herramientas de creación de interfaces gráficas. De esta manera se tiene un concepto tangible de la historia. El diseño va variando constantemente con cada cambio de historia de usuario. De forma paralela la ventaja que tiene este método es que la UI va avanzando a la vez que avanza el proyecto y que el cliente puede ver el resultado visual antes de codificarlo.

Estas soluciones, aunque se empezaron a implementar en la primera reunión kick off, ha servido para todas las reuniones posteriores que se han tratado historias de usuario y así crear uno scoops más realistas realistas.

Herramientas y las tecnologías

Se han vuelto a utilizar herramientas que se han tratado en anteriores apartados, como el modelo **Canvas** de innokavi, para obtener la primera abstracción; la herramienta online de tormenta de ideas **Bubble.us** para ir creando algunas historias de usuarios y **Trello** para crear e ir manteniendo el backlog. Además, para la creación de las interfaces de usuario se ha utilizado un programa de maquetación vectorial llamado Sketch que se explica a continuación.

Sketch.

Se trata de un programa de dibujo vectorial por capas, enfocado a las interfaces de usuario, el cual aporta muchas herramientas para facilitar las tareas relacionadas. Además, es un programa muy aceptado por la comunidad de diseñadores de UI, aún así hay que tener en cuenta que es exclusivo para sistemas operativos macOS y tiene un coste de 99$, no obstante, existen programas vectoriales que pueden suplir a este como Adobe Firework.

El programa viene incluido con una serie de *templates* creados exprofeso para el diseño UI de iOS, con lo que los elementos más comunes del diseño están ya creados y se puede llegar a construir una interfaz completa y realista en apenas unos minutos simplemente copiando y arrastrando los componentes.

Todos los templates son modificables y además el programa también te permite añadir otros distintos creados por terceros, como uno para el diseño web o para una aplicación Android, además de permitir crear unos propios.

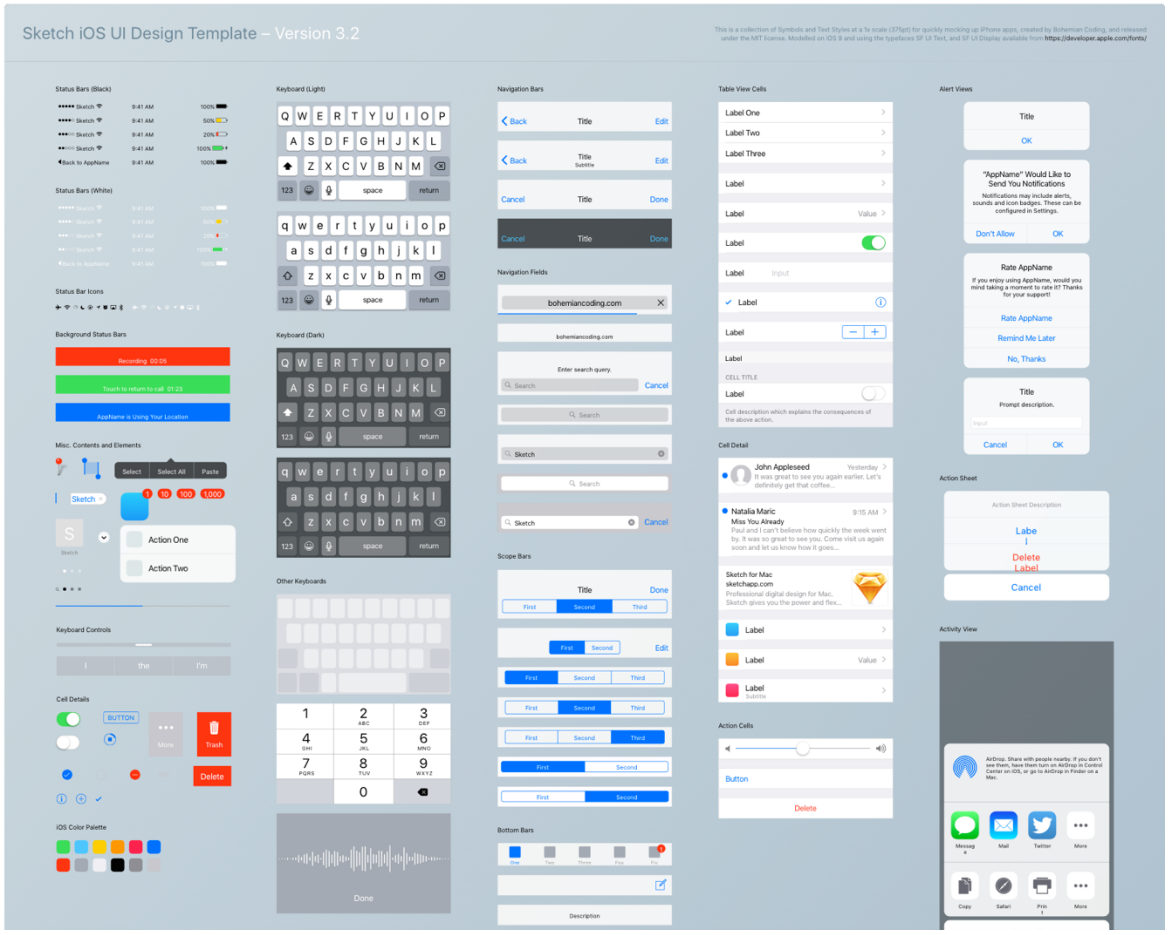
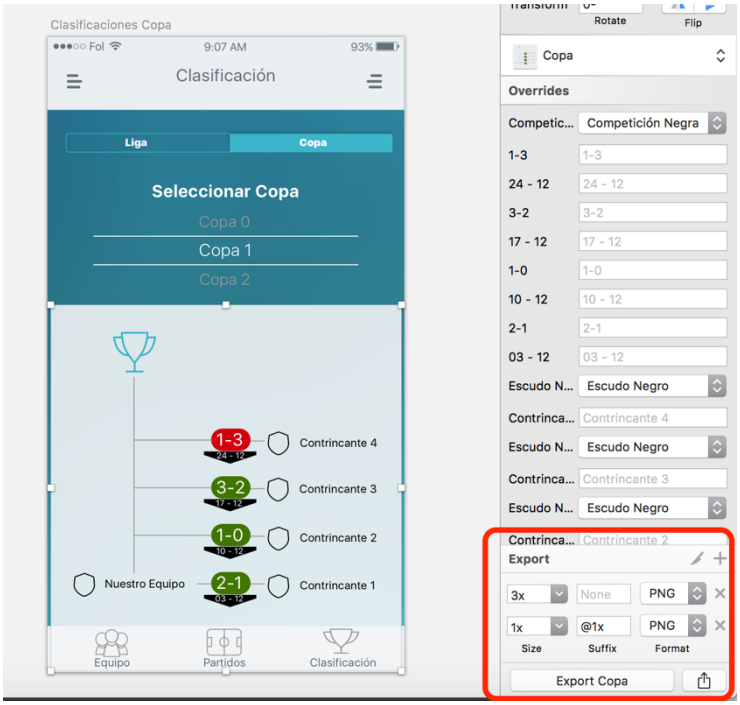


Ilustración x

Una funcionalidad clave y extremadamente útil es que todo lo diseñado con esta aplicación puede ser exportado en formato .png al tamaño que se pida. Por lo tanto, si diseñamos un botón que queremos implementar en nuestra aplicación real únicamente hay que exportarlo con los tamaños que se precisen e incluirlos a nuestro paquete de *resources*.



El programa provee la posibilidad de mostrar el aspecto visual de los diseñado en un dispositivo real, haciendo un *mirror* a escala mediante una aplicación que se descarga en los dispositivos, opción muy útil para mostrar avances con el cliente.

Historias de usuario.

Para la realización de las historias de usuarios se ha utilizado un template que se ha ido modificando dependiendo de la historia pero que tiene de base los siguientes elementos:

- Código de historia de usuario: en nuestro caso se creó un código por actor por lo que el código de historia empezaba por las siglas del actor principal.
  Iteración: En qué reunión se había definido la historia.
- Nombre: Breve pero completamente autodescriptivo.
- Usuarios: El actor si existía que realizaba la historia.
- La prioridad para el cliente: Se puede cuantificar de muchas maneras, nosotros elegimos una escala de cuatro valores, baja, normal, urgente, inmediata.
- Puntos de coste: la dificultad que se estima que puede tener su realización
- Descripción de la historia.
- Pruebas supuestas: Un esbozo de las pruebas que se podrían realizar para comprobar la historia.

Arquitectura y diseño

Es el apartado más extenso de esta memoria ya que aquí es donde entra el desarrollo de las tecnologías además de tratar el proceso de operaciones y organización. Se va a diferenciar desde un principio la parte de diseño enfocada al backend, donde entraremos en detalle comparando tecnologías y explicando su funcionamiento y partes más importantes y de forma somera en el frontend, donde solo explicaremos algunas decisiones.

Explicación del proceso

Procesos de operaciones

El modelado de diseño es una etapa en la que se gastaba una gran parte del tiempo en la forma tradicionalmente iterativa en cascada del ciclo de desarrollo. De hecho, se puede imaginar que el mayor ahorro de tiempo cuando usamos metodologías ágiles se produce precisamente en esta parte ya que en el proceso tradicional después de la captura de requisitos venía la especificación del modelado del diseño en el que se trabaja el modelado de toda la aplicación dividido en las siguientes partes: Arquitectura, patrones detectados y como se tienen que aplicar, modularidad de la arquitectura, el diseño de los objetos, el diseño de sistemas con la elección de lenguaje de programación, las especificaciones y modelado de las interfaces gráficas, elección de bases de datos, … además hay que pasar por varias fases antes de que terminen revisando y validando todas.

Cada una de estas partes estaban deducidas por la toma de requisitos anterior, y el problema principal radica en que, si se ha realizado mal la captura o el análisis de los requisitos, o estos han cambiado o evolucionado, en el mejor de los casos hay que repetir la parte que compete a ese requisitos pero en un caso desfavorable en el que ese requisito tenga una gran cohesión con el resto, hay que volver a replantear el apartado de modelado de diseño con la consiguiente pérdida de tiempo que implica Por este motivo se dice que las metodologías tradicionales tienen una gran resistencia a los cambios.

En nuestro caso al trabajar con una metodología ágil no vamos a sufrir estos problemas, ya que la forma de enfrentarse al modelado de diseño es completamente contraria a la tradicional. En programación ágil la etapa de modelado está presente de forma continua, cada vez que se crea una historia de usuario se modifica el modelo de alguna manera y sólo se hace el esfuerzo de definirlo en profundidad si así lo requiere el scope o si el desarrollador lo ve necesario, pero sólo se desarrolla el que se esté trabajando es esa tarea o iteración, para así si hay algún cambio en los requisitos lo único que hay que cambiar son las historias de usuario.

A pesar de esto, hay partes generales que hay que tener claras en el proyecto. Para ello las metodologías agiles dejan libertad de actuación. La metodología Scrum prevé una fase que la llaman "Pre-juego" En la que abarcan la primera parte del desarrollo engloban la visión y el análisis, aquí hacen una primera estimación de costes y de agenda y también definen el análisis del sistema creando el diseño de implementación básico. Al comienzo de cada spring se vuelve a hacer otra valoración por si hubiera ocurrido algún cambio que requiera la modificación en la arquitectura. Aún así, es labor del desarrollador idear en cada tarea que realiza el análisis del modelo de diseño. Sin embargo, no existe nada tan definido en XP, una aproximación podría ser que en la fase de **planteamiento** del lanzamiento de la iteración hacer actuar de forma similar a scrum, las implementación de XP enriquecidas que usan este sistema, la suelen encasillar en la fase de Planning Game.

Hay que tener presente en todo momento que las elecciones que se hagan sobre el modelo de arquitectura y de diseño, sea en la fase que sea, no tienen por qué mantenerse a lo largo del proyecto como una piedra angular. Por lo que hay que poner especial cuidado en trabajar con **baja cohesión** y tenerlos presente en las decisiones que tomemos ya que cualquier elección tomada es posible que haya que revertirla o que sufra cambios importantes.

Procesos técnicos

Como punto de partida en esta parte se trabaja constantemente con diagramas, así que las herramientas de creación de UMLs son un aliado continuo y necesario.

En esta fase ya se conoce el boceto de la aplicación, si va a ser un sistema empotrado, cliente-servidor, distribuido, … por lo que es necesario plasmarlo creando un **diagrama de despliegue** que nos sirva de referencia. La creación de este diagrama, como ya es habitual, es una tarea a desarrollar por todo el equipo, ya que cada uno va a estar implicado en cada una de las partes. El diagrama se compone por **nodos**, que son representaciones de objetos físicos que existen en tiempo de ejecución y las relaciones entre ellos. En nuestro caso el diagrama tiene que ser un objeto vivo que va a ir cambiando y mejorando si es necesarios.

Una vez que el diagrama de despliegue está definido, llega la hora de elegir los componentes que lo forman, de todas las opciones que nos da el mercado[17].

Generalmente en una aplicación distribuida multiplataforma se tienen los siguientes nodos:

- Clientes. Ya sean aplicaciones móviles, ordenadores de escritorio, tablets, …
- Servidores principales. Realiza las conexiones con los clientes.
- Servidores secundarios. Gestionan las bases de datos, recursos, etc
- Alojamiento de los servidores. El espacio físico donde residen.

Vamos a exponer las opciones que tenemos.

Tecnologías frontend, el cliente

Tradicionalmente lo componen las aplicaciones web junto con las aplicaciones móviles ya que ambas tecnologías corren en el lado del cliente.

El **desarrollo web** es la primera tecnología frontend, se basa en el desarrollo de páginas y aplicaciones basados en protocolos web, que tiene una gran capacidad de adaptabilidad ya que se puede visualizar en cualquier dispositivo que contenga un navegador, PCs, móviles, tablets, televisiones, … Existen una gran cantidad de tecnologías en el desarrollo web, las más comunes y eficientes son el conjunto de HTML, CSS y JavaScript, donde el control del desarrollador es completo; además existen multitud de frameworks web para la parte frontend que nos ayudan con la codificación y diseño como Bootstrap, Semantic-UI, Foundation, …

Hay que hacer mención especial al uso de **CMS** (Content, Management, System), es un framework que permite un desarrollo rápido y completo de una página web, mediante una interfaz que controla una base de datos que gestiona el contenido de forma transparente para el diseñador. La creación de portales bajo este sistema de contenido se basa en un conjunto de plantillas editables, a las que mediante añadidos, se les puede incorporar todo tipo de módulos, funcionalidades y diseños, la mayoría de ellos creados por una gran comunidad de desarrolladores que la respalda. Los CMS más conocidos son WordPress, Druppal, Joomla, … similares en su funcionamiento, pero con nichos de mercado distintos, algunos enfocados más a la creación de blogs y páginas de empresas; otros a tiendas online; portales educativos; aplicaciones web … Es sin duda la forma más rápida de realizar un proyecto web.

Los puntos clave a la hora de elegir el procedimiento a usar y en los que hay que poner especial atención son los siguientes:

- Tiempos de carga.
- Mantenibilidad de la web.
- Preparado para múltiples navegadores y escalado.
- Seo[18].

[17] Para abreviar solo nos vamos a referir a las opciones para una aplicación distribuida de tipo cliente-servidor como la que precisamos.

[18] Search Engine Optimization. Técnica que consiste en optimizar un sitio web para que mantenga un buen posicionamiento en los resultados de los motores de búsqueda de Internet.

La elección del procedimiento según estos valores tampoco es tarea fácil y depende mucho del contenido de desarrollo, siendo típicamente más configurable y óptimo el desarrollo usando HTML, CSS y JS, pero más mantenible y con optimización automática usando CMS

Al otro lado del frontend está el desarrollo para las **tecnologías móviles,** existen tres sistemas operativos sobre los que programar, iOS, Android y WindowsPhone; todos ellos con lenguajes de programación distintos e IDEs de desarrollo y guías de estilo UI diferentes. En primera instancia esto implicaría un desarrollo de las aplicaciones distinto para cada uno de los dispositivos, llevando tres proyectos de forma independiente, es lo que se conoce como **desarrollo nativo**. Es la forma más tradicional y robusta de desarrollo móvil y suele existir perfiles especializados en cada uno de los ecosistemas.

No obstante, existe la posibilidad de utilizar un software *cross-plataform*[19] y hacer sólo un desarrollo en un único lenguaje y con una única IDE que luego se adaptará a los tres entornos, actualmente el software más utilizado para ello es Xamarin de Microsoft.

Xamarin, permite la creación de ejecutables iOS y Android codificando únicamente en el lenguaje C#, el lenguaje nativo de WindowsPhone. Como cada una de estas plataformas tiene un conjunto de características diferente, Xamarin también da acceso completo a los SDK individuales de cada entorno, además de permitir embeber el código nativo, invocando directamente las librerías de [C], java, c y c++, que permite usar código de terceros o implementaciones propias. Con estas características y usando sus IDEs específicas para cada SO, se pueden construir aplicaciones multiplataformas con una sola codificación y compartiendo aproximadamente el 90% del código entre ellas.

Servidor Web HTTP

Un servidor web, es un programa que procesa las peticiones de las aplicaciones web mediante conexiones con el cliente y generando una respuesta a estas peticiones, para la transmisión se utiliza un protocolo HTTP que es un estándar de la capa de comunicaciones OSI[20]. En el servidor se van a alojar el código de las webs que se han creado en el apartado del frontend.

Existen multitud de servidores WEB y versiones de estos vamos a tratar los más importantes.

Apache: Es un servidor HTTP de código abierto para la creación de servicios web, es un servidor multiplataforma se puede instalar en entornos con Windows, Linux y macOS, tiene un gran rendimiento ya que es capaz de manejar más de un millón de visitas/día, Está muy extendido y tienen una gran comunidad que le da soporte, además es de código abierto y completamente gratuito.

Nginx: Es la alternativa principal a Apache, es también multiplataforma, es muy ligero, es compatible con la mayoría de los CSM. Es muy usado como balanceador de carga y además se puede usar integrar en Apache y usar ambos conjuntamente. Hoy en día existe una tendencia al uso de Nginx sobre el resto de servidores.

[19] Forma de llamar al software que implica codificar en un lenguaje e IDE particular y que porte el contenido diseñado a otras plataformas.
[20] El modelo de interconexión de sistemas abiertos (ISO/IEC 7498-1), más conocido como "modelo OSI", (en inglés, Open System Interconnection) es un modelo de referencia para los protocolos de la red de arquitectura en capas.

Tomcat - Apache Tomcat: Nació como un contenedor web de servlets[21] que se instalaba en un servidor web, pero actualmente funciona de forma autónoma como un servidor. Es multiplataforma ya que está basado en java y únicamente se necesita tener instalada la máquina virtual de java.

Trabajar con un framework

Un Framework es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar. En el mundo del desarrollo suele ser como una guía de buenas prácticas, que nos ayuda a enfrentarnos a un problema recurrente del que se existe un patrón de resolución aceptado por la comunidad que facilita su realización atendiendo a patrones de diseño. Generalmente un framework usa:

- Una estructura conceptual: patrón de diseño muy marcado que está necesariamente presente al usar el framework.
- Tecnología de soporte definida: define unos lenguajes de programación, generalmente el framework está creados con ellos, puede incluir más de uno.
- Módulos: que amplían el objetivo principal aumentando su funcionalidad.
- Incorpora recursos: programas de soporte, bibliotecas…

La utilización de un framework tiene muchas ventajas, la más importante es la reutilización de código y que promueve las buenas prácticas facilitando mucho el mantenimiento y la escalabilidad además un framework suele estar muy testado por lo que la fiabilidad es bastante alta. No obstante, hay que valorar que la curva de aprendizaje de un framework suele ser amplia por lo que hay que pararse a valorar el proyecto, para optar por elegir un framework u otro, casi siempre basándonos en si algún miembro del equipo está familiarizado con alguno. Un buen desarrollador FullStack tendría que conocer varios de los framework de desarrollo, tanto en la parte del frontend como en la del backend. Muchas veces la elección de uno va ligado a la elección del otro ya que forman parte del mismo stack y existen ventajas en su utilización conjunta.

Framework desarrollo web

Centrándonos en los frameworks web más comunes del backend tenemos tres opciones muy consolidadas que son: Django, Ruby on Rail y Larabel, todos ellos tienen una gran comunidad detrás que les apoya, librerías muy completas con todas las funcionalidades que podríamos necesitar, además todas ellas comparten casos de éxito al estar implantados en empresas de renombre. Por lo que la mayor diferencia que vamos a encontrar entre ellos es el lenguaje y la filosofía de desarrollo que implementan.

Se van a exponer las peculiaridades fundamentales de cada uno:

Django: Desarrollado en Python, lenguaje ampliamente reconocido y el quinto en la lista de lenguajes más populares de Tiobe[22], juntos están enfocados en hacer fácil la lectura del código y en mejorar el mantenimiento. Dispone de una interfaz de administrador CRUD[23], que se despliega en minutos y muy configurable, permitiendo también la administración de usuarios además de los datos de la BBDD. Permite tratar la entrada peticiones de URL como expresiones regulares. El despliegue puede ser costoso dependiendo el servidor que se elija.

---

[21] Clase de Java enfocada a extender los servicios de las aplicaciones alojadas en los servidores web.

[22] The Importance of Being Earnest) es un informe mensual que elabora y publica la empresa TIOBE Software BV.

[23] Las acciones que se pueden realizar sobre un dato. Create, Read, Update y Delete.

Ruby on Rails: Desarrollado en Ruby, lenguaje que gira en torno a hacer la escritura del código más sencilla y ágil permitiendo varias formas de realizar un mismo propósito, dejando así mucha libertad creativa al programador. Este conjunto suele trabajar de forma transparente, añadiendo funcionalidades que el programador no ha establecido de forma implícita, como que todas las expresiones devuelven algo. Ruby tiene integrado el uso de AJAX, por lo tanto, se pueden añadir efecto a las páginas sin tener que usar JS.

Larabel: Es el más reciente, está escrito y se basa en PHP. Usa MVC, pero no es necesaria una implementación estricta, por lo que deja más libertad en su desarrollo; se dice que tiene una filosofía más POO. Está muy orientado a la creación de pruebas automatizadas, que se ejecutan cada vez que se hace una incorporación al proyecto.

Framework de servicios

La **programación móvil,** a diferencia de la programación web, puede desarrollar el funcionamiento del backend en el mismo dispositivo, incorporando su propia base de datos y sus propias librerías de servicios, siendo así autosuficiente en su funcionamiento. No obstante, en determinadas aplicaciones es necesario un contacto con el exterior, ya sea por la propia naturaleza del funcionamiento como en aplicaciones de comunicaciones, aplicaciones distribuidas, servicios externos… o por necesidad de mejorar la computación.

Con la evolución de los servicios en la nube se crean los servicios Baas *(Backend as a service*) facilitando un conjunto de servicios backend como: almacenamiento de archivos y bases de datos, notificaciones push[24], RRSS, servicios propios… El proveedor de estos servicios facilita estos mediante una serie de APIs[25].

Este tipo de servicios aportan muchas ventajas al desarrollo de las aplicaciones multiplataforma. La principal es que optimiza el tiempo de trabajo, ya que se crea un único backend que, mediante una interfaz. van a poder utilizar los distintos dispositivos. De esta manera los desarrolladores móviles se podrán centrar únicamente en el desarrollo front de la aplicación e implementar, mediante llamadas http a la interfaz, todos los servicios backend que se necesiten. Como desventaja principal comentamos que el sistema queda anclado a trabajar, por lo que también es posible y habitual trabajar de una forma híbrida, no cediendo todos los servicios a las APIs y tener también servicios en local.

Como posibilidades para implementar servicios web tenemos SOAP, WSDL[26], REST y GraphQL. Aunque a día de hoy los servicios basados en SOAL y WSDL están completamente desfasados, ya que son complejos, tienen una gran cohesión, son difíciles de entender, ineficientes por que ocupan bastante ancho de banda y además no tiene un buen funcionamiento con servicios móviles. Por lo tanto, no tenemos que tenerlos en cuenta para ningún tipo de proyecto.

---

[24] *Push* es una forma de comunicación en la que una aplicación servidora envía un mensaje/alerta a un cliente-consumidor.

[25] *Aplication Programming Interface*. Capa de abstracción con un conjunto de métodos, que se ofrece para ser utilizado por otro software.

[26] Ampliación de la información sobre porqué no usar servicios SOAP y WSDL. Aquí.

REST

La opción más extendida actualmente es la **REST** (Representational State Transfer), se considera un estilo de arquitectura de software para sistemas distribuidos, que se apoya en el estándar HTTP y que no tiene estado, ya que en cada mensaje HTTP enviado está toda la información necesaria para realizar la petición. Una de las mejoras que implementó REST fue que no era necesario publicar servicios RPC[27], con un conjunto de métodos y operaciones, en su lugar publica **recursos** sobre los que interactuar. Las ventajas que tiene para los desarrolladores son muchas: Un desacoplamiento entre el cliente y el servidor, es siempre independiente de los lenguajes y las plataformas de los clientes y facilita la escalabilidad de los proyectos (al permitir cambiar de forma opaca al cliente la estructura de la base de datos).

Los recursos en los que se basa REST son considerados entidades, son la información a la que queremos acceder, modificar o borrar, independientemente del formato que tenga, sea xml, json, imágenes u otros. La implementación que tenga el recurso en el servidor decide qué información es visible o no. Para manipular estos recursos únicamente se pueden usar cuatro tipos de operaciones definidas que coinciden con el patrón CRUD:

CREATE / POST: Es la petición que le indica al servidor que quiere crear otro recurso y el servidor le responde con el identificador global que le ha asignado. **READ/GET**: Indicando el identificador de un recurso el servidor devuelve este; es importante determinar que las copias no se mantienen sincronizadas y si otro cliente modifica el recurso el resto no reciben la modificación hasta que no lancen otra petición READ, además de que se tiene que especificar en qué formato se requiere. **UPDATE/PUT**: Actualización de un recurso desde el cliente. **DELETE/DELETE**: Elimina el recurso del servidor. No hay que pensar que únicamente una API REST es un CRUD de una base de datos, ya que las acciones que se realizan en el servidor pueden tener mucho más alcance y realizar llamadas a otros servicios backend y no sólo limitarse a relacionarse con la BD.

Los servicios web RESTful son aquellos servicios web que implementan una arquitectura REST. Por lo tanto este servicio web tiene:

- **URI** (*Uniform Resource Identifier*) es la forma de identificar a cada recurso, los nombres nunca deben de implicar acción (ser verbos) y tienen que ser únicos, además en una URI tampoco tiene que aparecer el formato del recurso (.doc, .pdf…) y tiene que tener una jerarquía lógica. Ejemplo de URI: http://futbolssm.es/equipos/123/photos/

- **Verbos HTT** Get, Post, Put y Delete entre otros. Por ejemplo, GET/equipos/ nos devuelve el conjunto de equipos mientras que GET http://futbolssm.es/equipos/123/json nos devuelve el equipo identificado como 12 con el formato json.

[27] Llamada a procedimiento remoto.

Bases de datos

Una de las primeras cuestiones que aparecen al comenzar el modelo de un proyecto es el tipo de base de datos que se va a usar, como nuestro proyecto es escalable y va a ir variando continuamente es posible que también varíe la BD, por lo que debemos prepararnos tanto para hacer la mejor elección como para sustituirla en el caso que sea necesario.

Hay fundamentalmente dos tipos de bases de datos, las relacionales y las no relacionales.

Bases de datos relacionales: Se componen de un conjunto de tablas que están relacionadas entre sí mediante unas claves primarias y secundarias, cada tabla la forman un conjunto de filas (registros) y de columnas (campos, atributos). Uno de los campos de la tabla, llamado clave principal, lo contiene otra tabla como clave ajena, de esta manera se suceden las relaciones. No pueden existir dos tablas ni registros con el mismo nombre.

El uso de estas bases de datos se dan en los siguientes ámbitos[28]:

- Desarrollos web: para mantener jerarquía de datos, siempre y cuando la capacidad de concurrencia, almacenamiento y mantenimiento no sean de considerable dificultad y la información sea consistente.
- Negocios: inteligencia y análisis de negocios. Son temas que requieren el uso de SQL para facilitar el consumo de la información y la identificación de patrones en los datos.
- Empresarial: porque tanto el software a la medida y el software empresarial, poseen la característica de mantener información con estructura consistente.

Los gestores de bases de datos que siguen este esquema son principalmente **MySQL** y **PostgreSQL**. La diferencia entre los dos radica en que PostgreSQL puede soportar hasta el triple de carga en peticiones simultáneas que MySQL, pero consume mucha cantidad de recursos, lo que lo hace mucho más lento; como ventaja puede almacenar procedimientos en la propia base de datos. Por el contrario MySQL es uno de los gestores que ofrece mayor rendimiento dado su bajo consumo, lo que lo hace muy apto para incluirlos en sistemas con bajos recursos, pero no maneja integridad referencial ni rollback. Así que en este caso es bastante sencillo decantarse por una base de datos y sólo tendremos que priorizar las fortalezas de una sobre la otra.

Bases de datos no relacionales:

Las bases de datos no relacionales (NoSQL) surgen con las STUP y nacen debido a que los proyectos suelen tener una gran escalabilidad; como mantener un fuerte ritmo de crecimiento puede provocar que el mantenimiento base de datos relacional sea bastante tedioso, se crea una BD con una estructura mucho más versátil, a costa de perder funcionalidades como los *Join*[29]. Un punto muy

[28] Enumeración obtenida de facilcloud. Aquí.
[29] Producto cartesiano de dos tablas relacionales.

importante es que carece de *schema*[30], de ahí su escalabilidad al no tener que estar reajustando las tablas en cada nueva incorporación.

El uso de estas bases de datos se suele dar en los siguientes ámbitos[31]:

- Redes sociales.
- Desarrollo Web: debido a la poca uniformidad de la información que se encuentra en Internet; aun cuando también puede emplearse SQL.
- Desarrollo Móvil: debido a la tendencia – en crecimiento- de Bring Your Own Device.
- BigData: debido a la administración de grandísimas cantidades de información y su evidente heterogeneidad.

Existen un gran número de bases de datos NoSQL, cerca de 150 diferentes, aunque todas ellas se pueden englobar en tres grandes grupos: **orientadas a documento, orientadas a grafos y de clave-valor**.

Modelo de dato	*Formato*	*Características*	*Aplicaciones*
Documento	Similar a JSON (*JavaScript Object Notation*)	– Intuitivo – Manera natural de modelar datos cercana a la programación orientada a objetos – Flexibles, con esquemas dinámicos – Reducen la complejidad de acceso a los datos	Se pueden utilizar en diferentes tipos de aplicaciones debido a la flexibilidad que ofrecen
Grafo	Nodos con propiedades (atributos) y relaciones (aristas)	– Los datos se modelan como un conjunto de relaciones entre elementos específicos – Flexibles, atributos y longitud de registros variables – Permite consultas más amplias y jerárquicas	Redes sociales, software de recomendación, geolocalización, topologías de red …
Clave-valor /Columna	**Clave-valor:** una clave y su valor correspondiente **Columnas:** variante que permite más de un valor (columna) por clave	– Rendimiento muy alto – Alta curva de escalabilidad – Útil para representar datos no estructurados	Aplicaciones que sólo utilizan consulta de datos por un sólo valor de la clave[32]

Las más conocidas dentro de orientado a documentos son: **MongoDB** y **CouchDB** a grafos: **Neo4j** y **HypergrophDB**; y orientado a clave-valor/columna: **Riak Redis** y **Cassandra**. Aún, existiendo tanta variedad, a día de hoy MongoDB es la que se usa con mayor frecuencia y tiene más penetración de mercado.

[30] El esquema define las tablas de la base de datos, sus campos en cada tabla y las relaciones entre cada campo y cada tabla.
[31] Enumeración obtenida de facilcloud. Aquí.
[32] Cuadro comparativo obtenido de ondho. Aquí.

Hosting

Primero vamos a hablar del concepto de **Infraestructura como Servicio (laaS)**. Es uno de los tres modelos principales del cloud computing, que tiene como ventaja tener una gran **escalabilidad,m** ya que desaparecen los tiempos de espera. En el momento que se necesita más hardware se obtiene al momento, además de **no desperdiciar potencia** si no se va a usar. Dados los cambios constantes de una STUP, el no tener que hacer un gran desembolso en una inversión en hardware es un punto crucial para elegir este concepto, ya que se tarifica por los recursos que realmente se están utilizando. Estos servicios **no tienen punto de fallo únicos** por los que, si se produce algún fallo, el servicio global no se ve afecto, ya que existen multitudes de configuraciones redundantes, en el propio centro de datos o incluso en otros.

Siendo más específicos, un proveedor de laaS ofrece los siguientes servicios: aplicaciones informáticas, almacenamiento, bases de datos, análisis, redes, herramientas para desarrolladores, herramientas de administración, aplicaciones para IoT[33], seguridad en todos los ámbitos y herramientas empresariales.

Actualmente hay dos empresas que nos proveen de estos servicios.

Amazon Web Service:

El laaS de Amazon nos brinda todos los servicios y todos los productos globales basados en la nube que acabamos de mencionar. Nos da todo lo que podamos necesitar (y mucho más) de un hosting y nos ayuda con tareas que generalmente, si lo implantáramos en un servidor propio, serían muy costosas de hacer. De lo reseñado anteriormente vamos a explicar las tres más relevantes para nuestro propósito:

- Computación EC2: Elastic Compute Cloud. Sirve para crear entornos informáticos virtuales para conectarse a una AMI (Amazon Machine Image), creando así una máquina virtual con el sistema operativa que queramos. Amazon nos da un gran abanico de opciones para que escojamos entre diferentes instancias, la más básica es la **Instancias T2**, muy apta para servidores HTML o bases de datos, ya que ambos no hacen un uso excesivo de la CPU. El siguiente paso son las **instancias M3 y M4,** pensadas para entornos más exigentes. A partir de esta instancia existen 4 opciones que incrementan la capacidad de la máquina en CPU y GPU hasta las más altas exigencias, que quedan muy lejos de nuestro alcance de negocio.

  EC2 incluye además una serie de características adicionales, como el **AutoScaling,** que permite escalar automáticamente de una instancia hasta la siguiente; el **CloudWatch**, servicio que nos muestra en tiempo real los datos del uso real de las instancias; además, si tenemos más de una instancia, con la aplicación **Load Balancing** podemos distribuir de forma automática el tráfico que reciban los servidores.

  Todas estas opciones se pueden controlar mediante un panel de control online y se pueden modificar en cualquier momento. El sistema suele tener varios asistentes y automatizaciones, que nos ayudaran a configurar todas las opciones.

[33] Internet de las cosas. Conecta objetos y sensores entre sí para poder proporcionar soluciones beneficiosas para la vida conectada a través de internet.

- Almacenamiento S3: Simple Storage Service. Es el servicio que nos permite almacenar multitud de datos, no hay que confundirlo con su sistema de bases de datos relacionales. Las funcionalidades más típicas de este servicio pueden ser, el almacenamiento y la distribución de archivos, programas, fotos y videos de gran tamaño; también se usa frecuentemente para almacenar copias de seguridad y como pool de datos para análisis de datos en BigData. Como ventaja, esta diseñados para ofrecer un 99,99% de durabilidad, por lo que los datos estarán más seguros que en discos duros convencionales.

- Base de datos RDS: Relational Database Service. Es un servicio que crea, mantiene y gestiona bases de datos. Encontramos aquí varías ventajas, la **rapidez**, ya permite elegir entre los discos duros la tencnología SSD; **backup automático**s cada cierto tiempo, creando imágenes de la bases de datos en determinados momentos; gran **seguridad**, ya que se puede aislar la base de datos para sólo ser accesible desde una VPN con IPsec cifrada; **replicación** de los datos en una instancia paralela de manera síncrona. También, la creación y la administración son muy sencillas, ya que contiene una consola de administración para su gestión, además de poder conectarse con sencillas llamadas a su API.

Microsoft Azure:

De la misma manera que Amazon, Azure nos proporciona un servicio igual de fiable y robusto. Las aplicaciones similares a las de Amazon son las siguientes.

- Azure VM: Virtual Machine. Con características similares al EC2 de Amazon, en las instancias más sencillas no existen diferencias reales. Es ya cuando subimos por las instancias cuando vemos que AVM no dispone de aceleración por GPU, aunque realmente en un caso como el que estamos tratando jamás haría falta.

- Azure AS: Azure Storage. También se usa para el almacenamiento de documentos, videos, imágenes, copias de seguridad. Utilizando tres versiones: **Blob Storage**, para el almacenamiento estándar, facilita links de acceso mediante URL o interfaz REST; **File Storage** para el almacenamiento Premium, únicamente en ssd y con posibilidad de que los accesos incluyan un token de firma de acceso compartido (SAS); y **Queue Storage**, usado para almacenar colas de mensajes de datos, para almacenar listas de mensajes y para procesarlas de forma asíncrona.

- Microsft SQL Azure: Al igual que en Amazon, Azure SQL provee una gran cantidad de opciones en seguridad y funcionalidad. Incluye herramientas de *Machine Learnig* y de SQL data *Warehouse.*

Amazon y Microsoft. Tienen buenas ofertas para comenzar a trabajar rápido fácilmente. E incluso gratis para nuevas cuentas. Creando (han creado) su modelo de negocio basándose en la escalabilidad futura de nuestras necesidades. (de las necesidades de los usuarios)

Caso práctico

Como hemos podido, ver en la mayoría de los casos, cualquier decisión va a satisfacer nuestras necesidades, y la tendencia es que las diferencias presentes actualmente entre las herramientas se vayan acortando a medida que pase el tiempo, lo que hará más difícil las elecciones y se terminará basando en gusto personales y costumbres de uso.

Elección del Frontend

Para el desarrollo de las **aplicaciones móviles** se ha optado por realizar aplicaciones nativas. Se ha priorizado el que la totalidad del equipo tiene conocimientos sobre desarrollo en iOS, que las aplicaciones de forma nativa son más mantenibles y que preferimos mantener dos interfaces de usuario diferenciadas, ya que consideramos que la experiencia de uso en cada uno de los sistemas es muy diferente, por todo esto es por lo que vamos a respetar las guías de estilo marcadas por los sistemas. Esta elección tiene como consecuencias negativas que el tiempo de desarrollo va a aumentar considerablemente y que no se va a desarrollar para WindowsPhone (ya que su uso es marginal); si se hubiera elegido programar en Xamarin sí hubiera sido opción, ya que se desarrolla para los tres sistemas de forma automática.

El desarrollo de la **página web comercial** se va a desarrollar usando un el CMS Wordpress, porque consideramos que, como únicamente es una web de contacto, no merece la pena el coste de desarrollo de otro tipo de web, y en apenas unas horas tenemos una web funcional y vistosa. Y la elección de WP sobre otros CMS viene dada porque es la que más penetración tiene en el mercado.

Sin embargo, el desarrollo de la **aplicación web** va a ser enteramente desarrollada utilizando **HTML5, CSS**. Sé ha elegido esta tecnología porque, al ser un estándar, cuya forma de funcionamiento y desarrollo son ampliamente conocidos, los demás sistemas en su mayor medida van a utilizar este tipo de lenguaje. Por esos hemos dado, en este caso, preferencia a un sistema conocido, en detrimento del tiempo que habrá que emplear en el desarrollo de la aplicación web.

Elección del framework

Desde un primer momento se tomó la decisión de realizar el backend con un framework, ya que nos ayudan a las buenas prácticas, además de separar responsabilidades y la organización del código, de otra manera tendríamos que replantearnos nosotros los propios patrones, lo que supondría mucho tiempo y esfuerzo para finalmente llegar a una aproximación muy limitada de lo que sería un framewrok, por lo que carecería de sentido.

Sobre la elección del tipo de framework a usar, la decisión sí ha sido bastante difícil, ya que cualquier opción era completamente viable y todas tenían ventajas similares. Al final nos hemos decantado por **Django**, sobre todo por su lenguaje de implementación, **Python**; ya que analizando los lenguajes de los otros frameworks, que son Ruby y PHP, vemos que se alejan más de lo que está acostumbrado el equipo.

Python nos ha parecido muy cercano y la curva de aprendizaje viniendo de otros, como C y Swift, ha sido inexistente; además de por ser uno de los lenguajes más usado y con cantidad de referencias en la red. Analizando los lenguajes de los framework como Ruby y PHP se alegan más de lo que está acostumbrado el equipo.

Teniendo esta elección clara, la elección del **servidor REST** ha sido obviamente **Django REST**, ya que tiene su propio módulo para trabajar conjuntamente.

Elección del servidor

Viendo que la tendencia en el mercado es **Nginx** y que su velocidad, en nuestro contexto de uso actual, estaba por encima de alternativas como Apache y además de la facilidad para desplegar Django en él, frente al uso de mod_wsgi para hacerlo en Apache, se tomó la decisión de usar Nginx.

No obstante, para almacenar la web construida con un CMS se va a utilizar un servidor Apache mientras se hace el desarrollo en local. Se utilizará apache ya que hay herramientas como MAMP que configuran de forma automática y local una base de datos con servidor apache, en cuestión de un par de minutos y además está ya preparado para acoger un CMS en local de una manera muy sencillas.

Elección de tipo de BD

En un primer contacto con las bases de datos se barajó la opción de usar un BBDD NO-SQL orientada a documentos, basándonos en las ventajas que se le suponía atendiendo principalmente a que al carecer de esquema nos permitía ir añadiendo campos sin tener que controlar ni reestructurar el esquema. Dado el carácter incremental del proyecto, consideramos esto como una ventaja significativa.

Basándonos en esta primera decisión intentamos crear un modelo no relacional, pero por la lógica de negocio y los múltiples objetos que se necesitaban, la estructura se parecía más a un relacional. Además, nos dimos cuenta que necesitaríamos realizar más de una consulta para casos que solucionaríamos con un simple Join en el modelo relacional. Los creadores de MONGO (la BBDD NO-SQL pensada), tienen una solución a este problema a través de los framework "Aggregation Framework" y "Map Reduce". Valorando estas opciones decidimos que nuestra app encajaba más con un **modelo Relacional** tradicional por:

- Podemos deducir el alcance de la aplicación en el futuro y prever los campos agregados en nuevas funcionalidades con bastante exactitud. Por lo que los cambios no van a ser tan determinantes.

- Nuestra cantidad de datos no va a alcanzar el volumen necesario para qué al realizar una consulta, que implique cruzar las tablas con JOINS, tarde tanto que ralentice el sistema.

- La BBDD, por la lógica del sistema planteado, está muy estructurada y la gran mayoría de consultas van a necesitar el cruce de tablas.

No obstante, siempre se tiene presente que al usar la metodología ágil podemos portar a un modelo NO-SQL si suponemos que así la aplicación lo demanda.

Elección gestor BD

La elección en este punto es sencilla, ya que se han descartado los modelos no-relacionales, que prima la rapidez y que nuestro volumen de datos es muy escaso, **mySQL** es la elección más lógica.

Elección Host

Se eligió finalmente Amazon Web Service. Aunque tecnológicamente las dos opciones barajadas estaban muy equiparadas, la elección fue tomada basándonos en las ofertas de prueba que tenían ambos y ser más simple la de **Amazon**, la cual, a fecha de creación de este documento, tiene las siguientes características:

Capa gratuita que incluye 12 meses de:

- Amazon EC2 con un máximo de 750 horas al mes de cómputo. Aclarar que no está limitado a una máquina, ya que podemos levantar más máquinas, pero entre ellas las horas de funcionamiento en paralelo no podrán superar las horas limitadas al mes.

- Amazon RDS *db.t2.micro* con un máximo de 750 horas. Al igual que con EC2 cada instancia consume horas de actividad. La capa gratuita también incluye la posibilidad de combinar discos duros SSD y otras características como la replicación, siempre con un límite de capacidad.

- Amazon s3 con 5GB de almacenamiento estándar, AWS Lambda con un millón de solicitudes al mes, etc.

La opción de Azure la descartamos por su funcionamiento en la parte de pruebas. Este ofrece 170 € al mes de forma gratuita, que se podrá gastar en cualquiera de sus servicios. Como en una primera instancia no sabíamos bien qué servicios vamos a utilizar, ni donde destinar esa parte del dinero, nos pareció una opción más compleja ya que requería de un análisis inicial, lo que implicaba mucho tiempo que no hacía falta gastar si usábamos Amazon. No obstante, hay que considerar que que Azure tiene una ventaja muy notable frente al resto y es que no se hace ningún cargo en la cuenta asociada sin antes confirmarlo con el cliente. En Amazon, si sobrepasas las horas o las capacidades de la capa de prueba, automáticamente se empieza a tarificar el consumo. Esto es muy peligroso en las primeras fases, ya que la implementación puede lanzar resultados inesperados, además, al ser novatos con la plataforma podemos activar algo que luego nos repercutirá económicamente, sin obtener ningún beneficio. Otro punto que también tenía de Azure es que nos pareció que tanto la asistencia para la contratación como los manuales de uso estaban bastante más claros y accesibles.

Herramientas y las tecnologías

Python

Entes de entrar en el desarrollo del backend es necesario familiarizarse antes con el lenguaje de desarrollo con el que se ha construido. Por ello vamos a tratar algunos puntos esenciales de Python.

Versiones

Unas de las primeras dudas que surgen cuando nos adentramos con él (en él), es que existen dos versiones de Python que se usan actualmente y que reciben actualizaciones, **Python 2.X** y **Python 3.X** (actualmente 2.7.X y 3.6.X). No es un caso habitual, en el resto de lenguajes de programación las versiones más nuevas suelen ir desbancando a las anteriores, y no se continúan actualizando de manera paralela. El motivo de mantener las dos versiones es que se han producido cambios entre ellas tan significativos que hay proyectos que simplemente no se pueden actualizar de la versión 2 a la 3. Estos cambios no han sido tanto por la cantidad de cambios sino por la profundidad de estos. Concretamente, el cambio más determinante ha sido que en Python 3 las cadenas de texto de forma predeterminada tienen codificación **Unicode**[34]**,** mientras que en Python 2 la codificación por defecto es **ASCII**[35]. Por lo tanto para solventar este problema y usar añadir caracteres como "ñ" o "ç" había que especificar una codificación que los soportara, como por ejemplo # -*- coding: utf8 -*- .
Además del mencionado, los cambios más reseñables son:

- *Print*, es una función en Python 3, por lo que hay que añadirle paréntesis.
- División de enteros en Python 3 no siempre da un número entero y no es necesario forzar un número a *float* para conseguir un resultado con decimales.
- Cambio en el manejo de excepciones por Python 3 al eliminar la forma básica con la que se trataba en Python 2.
- Se cambió la funcionalidad en Python 3 de la función de lectura por teclado de *raw_input() a input()*.

El problema de **incompatibilidad** radica en que las versiones 2.X no son del todo compatibles con las nuevas y por lo tanto todas las librerías existentes, las cuales están escritas en Python 2, no van a funcionar en Python 3. Sabiendo esto es comprensible que muchos desarrolladores hayan preferido mantenerse en las versiones anteriores a pesar de que Python 3 lleve unos años en el mercado; y no por problemas de estabilidad típicas de las nuevas versiones, sino por compatibilidad con proyectos antiguos y librerías.

Por lo tanto, y para concluir, **es fundamental** conocer el alcance que va a tener nuestro proyecto y comprobar si las librerías que vamos a usar están creadas ya en Python 3. No obstante es recomendable, debido a la incertidumbre en los proyectos con una metodología ágil, usar a día de hoy Python 2.X .

---

34 Unicode es un estándar de codificación de caracteres diseñado para facilitar el tratamiento informático, transmisión y visualización de textos de múltiples lenguajes y disciplinas técnicas, además de textos clásicos de lenguas muertas. El término Unicode proviene de los tres objetivos perseguidos: universalidad, uniformidad y unicidad.

35 ASCII (acrónimo inglés de American Standard Code for Information Interchange — Código Estándar Estadounidense para el Intercambio de Información), es un código de caracteres basado en el alfabeto latino, tal como se usa en inglés moderno, tiene 127 símbolos, de los cuales del 33 al 126 son imprimibles (números y letras) y el resto caracteres de control, pensados originalmente para controlar dispositivos entre otras tareas.

Instalación

Antes de empezar a usar Python hay que tener en consideración qué versión se va a querer usar y el SO en el que se va a realizar el desarrollo del proyecto. En la mayoría de entornos Unix y likeUnix viene por defecto una instalación de Python 2, por lo tanto, si queremos desarrollar en Python 3 tenemos que realizar una nueva instalación, pero es muy importante ser conscientes de que el SO usa Python 2, por lo tanto nunca hay que forzar la eliminación de la versión del sistema. Lo más habitual, si interfiere con nuestro desarrollo, es el uso de entornos virtuales.

La función de un **entorno virtual** es la de aislar un entorno de desarrollo de otro; es como un espacio cerrado en el que puedes trabajar sin miedo a afectar al *mundo exterior* y verte afectado por éste. Por ejemplo, si en un entorno estamos creando un proyecto nuevo de Django usaremos probablemente la nueva versión del sistema y de Python. Pero para el mantenimiento de una aplicación anterior es probable que no queramos actualizar la versión y prefiramos mantenernos en la actual. Para esto y para otras cosas crearemos un entorno virtual, que mantiene en "una carpeta" las instalaciones de un proyecto manteniéndolas ajenas a las del sistema y a otros Entornos Virtuales. Por lo tanto, los cambios de uno no afectarán a los del otro. (En la sección de anexos se incluye un tutorial que se ha realizado para la enseñanza del equipo transparencia 16).

La instalación de Python 3 se puede realizar fácilmente por cualquier gestor de paquetes de la *Shell* o desde su web oficial[36].

Lenguaje

Python es un lenguaje moderno, sencillo y elegante. Las particularidades de este lenguaje son las que mencionamos a continuación. Tiene una **gestión de tipado dinámica**, lo que implica que infiere automáticamente los tipos durante el ciclo de vida de la variable, aunque permite forzar el tipo mediante un casting de la clase del tipo a cambiar. Es **de naturaleza interpretada**, por lo que es muy usado para *scripting*, además de para desarrollos rápidos. Funciona en cualquier de los SO más extendidos, ya que es **multiplataforma**. Es ***open source***, gestionado por Python Software Foundation, lo que facilita que tenga una gran comunidad trabajando constantemente para mejorarlo, además de tener un amplio abanico de librerías[37].

Cuando nos adentramos a programar en este lenguaje y repasamos la sintaxis básica, una de las primeras cosas que nos llaman la atención es la forma de ***indentación*** y la carencia de llaves para abrir y cerrar funciones. Esta indentación se puede realizar, o bien con cuatro espacios, o con tabulaciones, pero no es recomendable mezclarlos ya que, según la configuración del intérprete (ejecutado con la opción -tt) puede lanzar errores. La mayoría de IDEs y editores vienen con opciones para tratar estas separaciones, generalmente existe una opción que sustituye el tabulador por 4 espacios, ya que es la opción más aceptada.

También es una característica poco usual de Python que carezca del flujo de control ***Switch Case***. La solución para suplir la carencia no pasa por realizar un extenso anidado de estructuras if/else, la solución se encuentra en sus potentes estructuras de datos, concretamente en sus **diccionarios,** que hace las funciones de array asociativo[38]. Por lo tanto, se crean diccionarios con clave valor, siendo la

[36] Python software fundation, aquí.

[37] Las veinte librerías más útiles y conocidas de Python.

[38] Tipo de estructura de datos que contiene elementos indexados con valores únicos (no pueden existir dos elementos con la misma clave índice dentro del mismo array asociativo).

clave el elemento comparable en el case y el valor lo devuelto, que puede ser un tipo de dato o una función. Para controlar las entradas que no estén definidas en el diccionario como claves, la función típica del *default*, tendríamos que recurrir al uso del **try-except**. Un ejemplo sería:

Forma tradicional:

```
switch(nombre):
  case 'Yuri':
    edad = 30
    break
  case 'Dani':
    edad = 32
    break
  case 'Paco':
    edad = 30
    break

print edad
```

Y en Python:

```
edades = {'Yuri':30,'Dani':32,'Paco':30}
nombre = raw_input('Escribe un nombre')
try:
    edad = edades[nombre]
    print edad
except:
    print ('Nombre no registrado')
```

Aparte de estas peculiaridades, el resto de la sintaxis[39] tiene una curva de aprendizaje bastante corta si se conoce anteriormente cualquier otro lenguaje tipo, java, c, Swift, etc.

Django

Versiones e Instalación

La versión de Django que tendremos que instalar va ligada con la versión de Python que queramos utilizar. Desde la versión de Django 1.9[40] sólo admite las versiones de Python 2.7 y la 3.5 o superior; si por alguna razón queremos utilizar alguna anterior tendremos que instalar la versión de 1.8. No obstante, los cambios entre versiones han sido menores, cambios en el fichero de configuración, cambios menores en funciones y nuevas características de validaciones, mejoras del sistema de migraciones…

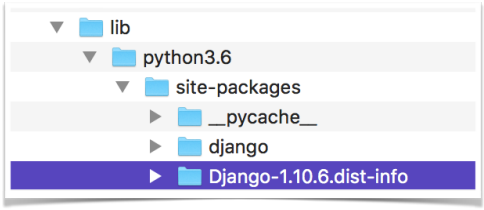
Django tiene un entorno de desarrollo y un entorno de producción, que es cuando se sube al servidor donde va a estar alojado. Mientras nos encontramos en desarrollo es recomendable tenerlo

---

[39] En la transparencia 23 del anexo "CursoDjango" hay una descripción concisa de la sintaxis que se realizó para impartir en el equipo de desarrollo, además de otro curso en ppt explicando más en profundidad los principios de la programación utilizando de lenguaje base Python.

[40] La versión actual de Django es la 1.10, aunque la mayoría de los manuales hacen referencias (creo que aquí falta algo)

instalado en la máquina a través de un entorno virtual[41]. Su instalación se puede hacer desde el Terminal a través del instalador de paquetes PIP. Instalar Django implica la creación de un nuevo directorio donde se almacenarán todas las librerías que necesita para su funcionamiento, se suele encontrar en el directorio de Python en **site-packages**.



El framework desde un primer momento se mantiene muy estructurado y nos intenta poner siempre las cosas fáciles. Por ejemplo, iniciar un proyecto es tan sencillo como ejecutar en la carpeta que queramos contenerlo el comando: django-admin.py startproject ”misitio”, al iniciar el nuevo proyecto, Django crea de forma automática una serie de archivos y directorios.

miSitio/	Contenedor del proyecto
manage.py	Utilidad para interactuar con el proyecto
miSitio/	
__init__.py	Requerido por python trata la carpeta como “un paquete”
settings.py	Opciones y configuración del proyecto
urls.py	Declaración de URL del proyecto.
wsgi.py	

Controlar dónde se van creando los archivos es muy útil, ya que la filosofía de Django es **DRY** (Don't Repeat Yourself), por lo que nos obligará a crear aplicaciones, al añadir funcionalidades, BBDD … que son pequeños programas reutilizables que se guardarán en la carpeta del proyecto.

[41] Instalación de entorno virtual en anexo “Curso Django” transparencia 16.

Modelo vista controlador y estructura básica

El diseño de Django se basa en el patrón, modelo vista controlador de forma que vamos a explicar su funcionamiento para entender el sentido de utilizarlo en el framework.

Modelo: Implementa la lógica. Esto significa que es responsable de la recuperación de datos convirtiéndolos en conceptos significativos para la aplicación, así como su procesamiento, validación, asociación y cualquier otra tarea relativa a la manipulación de dichos datos.

Vista: La vista hace una presentación de los datos del modelo estando separada de los objetos del modelo. Es responsable del uso de la información de la que dispone, para producir cualquier interfaz de presentación de cualquier petición que se presente. Por ejemplo, como la capa de modelo devuelve un conjunto de datos, la vista los usaría para hacer una página HTML que los contenga. O un resultado con formato XML, para que otras aplicaciones puedan consumir.

Controlador: Los controladores pueden ser vistos como administradores cuidando de que todos los recursos necesarios para completar una tarea se deleguen a los trabajadores más adecuados. La capa del controlador gestiona las peticiones de los usuarios. Es responsable de responder la información solicitada con la ayuda tanto del modelo como de la vista.

Por lo tanto, el objetivo de establecer un MVC en Django atiende a, principalmente, bajar la cohesión entre distintas partes del desarrollo.

El modelo que utiliza Django es una ligera variación llamada **MTV** (Model Template View).

Para explicar su funcionamiento vamos a ver tres niveles de abstracción: Uno **general** con el modelo MTV, otro con el **funcionamiento del modelo** en Django y el nivel más técnico en el que repasaremos **las partes internas** que programaremos para configurar el servidor, con la finalidad de familiarizarnos con ellas.

	MODELO	VISTA	CONTROLADOR
CLÁSICO	LÓGICA DE LA APLICACION (Clase control base de datos y tratamiento)	DISEÑO DE LA INTERFAZ (Interface builder)	"COMUNICADOR" ENTRE MODELO Y VISTA (Funcionalidad de una acción)
	MODEL	**TEMPLATE**	**VIEW**
DJANGO	CAPA DE ACCESO A BD **models.py** (Clases que simulan entidades con objetos)	Parte de estructura visual con su propia lógica de etiquetas **documentos.html + etiquetas django**	Lógica que accede al modelo y la delega a la plantilla apropiada. **views.py**

Tabla 1 MTV vs MVC general

Es importante no confundir el View de un modelo con el del otro, ya que en Django las **vistas** son la lógica que controla el acceso a los datos.

Bajando al siguiente nivel podemos ver cómo el primer contacto con el framework se hace a través de la url; la url invoca a una vista que se puede poner en contacto con el modelo o bien con un template; el template va a devolver código html que se cargará en el navegador. Cada una de las partes está bien delimitada en el modelo MTV.

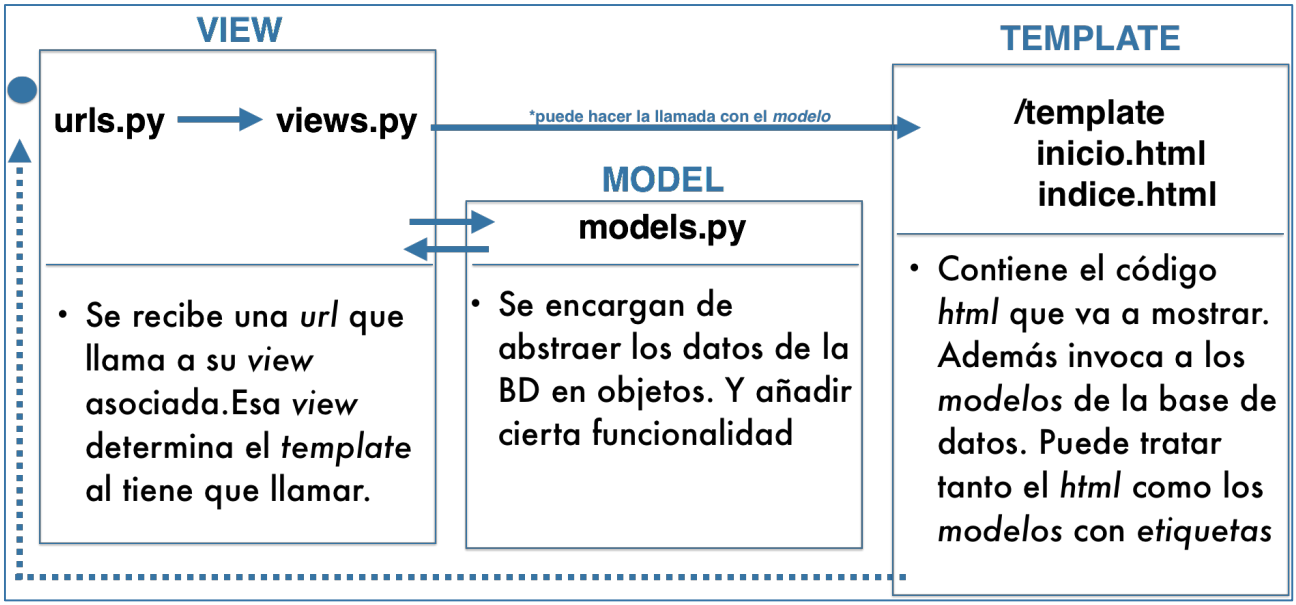


Tabla 2 Modelo MTV secuencia funcionamiento.

Vamos a descender al siguiente nivel. Teniendo presente la siguiente tabla vamos a hacer un recorrido por el proceso analizando las partes de Django que intervienen en él.

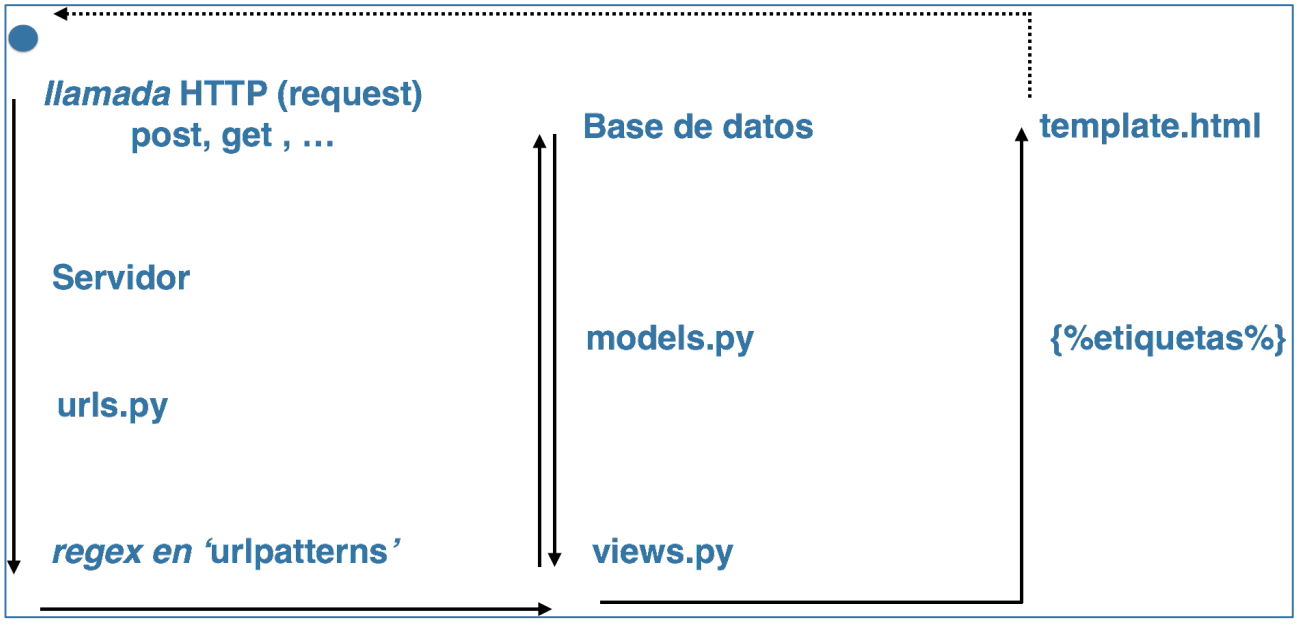


Tabla 3 Transicion del funcionamiento Django

1. El inicio del proceso ocurre cuando se produce una petición **HTTP** hacía la URL de un **servidor**.

2. El servidor (por ahora el de desarrollo alojado en nuestra máquina, más adelante puede ser apache, nginx u otro, alojado en otro lugar como AWS…) recibe la petición y envía la petición a **urls.py** mediante unos ***regex*** (expresiones regulares [42]), busca una coincidencia entre la url solicitada y las expresiones o url creadas en urls.py. Si la encuentra llama a la view asociada, adjuntando el objeto **request** que viene con la petición HTTP, si no encuentra coincidencia en urls.py llama por defecto a la página 404 predefinida.

3. Las **views** determina a qué *template* tiene que llamar. Puede depender del tipo de parámetro que se le ha pasado, de la llamada html, si es get o post…, también se comunica con la base de datos a través de los modelos.

4. Los **modelos** contienen una clase por cada **tabla de la BD de datos**. Usan sus propios tipos de datos y añaden cierta funcionalidad a las clases, pudiendo crear funciones. El acceso a las columnas de la base de datos se hace como si fueran atributos. Y cada instancia figura como un objeto. Además, Django autogestiona la BD con el modelo de forma casi automática. También acepta la mayoría de los motores de bases de datos y puede estar alojada en cualquier máquina. Para facilitar la creación también incorpora una base SQLite de datos por defecto, lista para usarse.

5. El **sistema de plantilla** después de renderizarse se convierte en un archivo de HTML. Este sistema permite heredar plantillas, incluir variables… Al igual que otros sistemas, permiten introducir código en el html mediante un sistema de **etiquetas**. Estas etiquetas pueden crear flujos de código como condicionales, bucles, etc. También podemos crearnos nuestras propias etiquetas. Gracias a esto crearemos código HTML más reusable y eficiente

[42] Las expresiones regulares son patrones utilizados para encontrar una determinada combinación de caracteres dentro de una cadena de texto.

Servidor de desarrollo e inicio

Django tiene un servidor web para usarlo en la fase de desarrollo y así emular un servidor HTTP. Además de ser muy ligero, se reincida (se reincide?) con cada cambio detectado en el código de manera automática, acogiendo los últimos cambios. El servidor se mantiene abierto en línea de comandos y además de como servidor, también es útil para saber las acciones que está realizando Django internamente, para capturar errores y ver las advertencias del sistema. Cuando se inicia el servidor, este nos dará su dirección IP y añadiendo una url detrás accederemos a las urls. Mostramos el ejemplo de una primera conexión al servidor sin añadir ningún patrón url a urls.py, siempre con el servidor arrancado, *'python manage.py runserver'*. Introducimos en el navegador XXX.X.X.X:8000/holaDani. Y nos devolvería:

Ilustración 7 Error 404 por defecto

Ilustración 8 Servidor desarrollo

Como estamos viendo, el servidor nos advierte que no encuentra una url que corresponda con */holaDani* y por otro lado devuelve un html lanzando un error típico 404.

Urls.py

Como hemos visto, los patrones url van a permitir conectar con nuestro servidor, por lo que el primer paso será configurar las direcciones que consideremos válidas. Para ello nos servimos del uso de **expresiones regulares**. Hay que tener en cuenta que la identificación del patrón la realiza en cascada, empieza por la primera línea de código a buscar coincidencias y si no encuentra una, lanza un error 404 como hemos visto antes.

Cada patrón está asociado a una **vista** (pueden asociarse la misma vista a más de un patrón), por lo que una vez que encuentra una coincidencia activa la función vista correspondiente, pasándole un objeto http.

Un ejemplo real de un urls.py sacado de nuestra aplicación es:

```
from django.conf.urls import url,include
from django.contrib import admin
from django.conf import settings
from django.conf.urls.static import static
from django.contrib.auth.views import login,logout
from DesarrolloFSM.views import *


urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^login/$',cargaLogin),
    url(r'^comprobarLogin/$',comprobarLogin),
```

```
url(r'^logout/$',logout),
url(r'^index/$',cargaInicio),
#Dossier
url(r'^noticia1/$',cargarNoticias1),
url(r'^crearNoticia/$',createNew),
...continua
```

Un proyecto Django puede contener más de un archivo urls.py, pero para que funcione tiene que estar referenciado en el urls.py principal. También hay fórmulas [43] para crear conjunto de patrones, URLS dinámicas, paso de una parte de url como parámetro de la función vista…

Views

Como hemos visto un (un ó una) *view* no es más que una función de Python que se utiliza una vez que se ha identificado un patrón. Estas se crean y almacenan en un archivo views.py; toman como argumento de entrada un objeto HttpRequest de forma transparente y devuelven un objeto HttpResponse (request[44]).

Dentro del alcance de las vistas está el poder pasar argumentos determinados por nosotros a las vistas. Un ejemplo podría ser una situación en la que el código de dos vistas sea casi igual y que sólo cambie, por ejemplo, la plantilla de destino. Para solucionar esto podemos pasar entre URL y View argumentos que determinen, en este caso, qué plantilla utilizar. La estructura podría quedar así.

En urlpatterns : url(r'^inicio/$', vista_con_atributo,{'plantilla':'inicio.html'}),
url(r'^indice/$', vista_con_atributo,{'plantilla':'indice.html'}),

En views: def vista_con_atributos(request,plantilla):
Return render (request, plantilla, {'autor': Daniel})

Además, una vista es capaz de determinar si una llamada http request es GET o POST accediendo al atributo *.method* del objeto request, pasado desde el urls.py. Esto sólo es un ejemplo del paso de parámetros del url patterns a las vistas; el tema es muy extenso y tiene muchas soluciones, en el tutorial del anexo se tratan algunas, más.

Templates

Los templates de Django es la forma de poder utilizar código Python en un documento html. Nosotros generamos el código html en un documento *.html* junto con **etiquetas** de Python. El uso que tienen las etiquetas es de lo más variado, desde introducir variables en el código, usar condicionales, bucles, crear listas, resaltar palabras html… además de que podemos crear nuestras propias etiquetas añadiendo funcionalidades.

Otra función muy útil en los templates es la de **reutilizar partes del código** mediante el uso de etiquetas, y hay varias formas de hacerlo. La más usada es el uso de la **herencia de templates**. El funcionamiento de esta utilidad es muy simple, primero hay que localizar las partes del código que se van a repetir en más de una página html, luego se crea una plantilla que contenga este código recurrente y marcamos, mediante el uso de unas **etiquetas de bloque,** las partes del código que sí van

[43] Para ver más opciones de configuración, mirar en la pág. 58 del "Tutorial de Django" en anexos.
[44] Un objeto request contiene información sobre la llamada http mediante atributos y funciones propias, como una lista de variables pasadas (¿basadas?) en formularios POST y GET (¿coma?) el path completo, metadatos …

a variar. Ahora tendremos que crear otros templates en los cuales tendremos que indicar, mediante un código al inicio del documento, quién es su padre, y en ellos incluiremos la información que queremos que se inserten en las etiquetas de bloque. Por lo tanto, podemos tener el grueso del código en una etiqueta padre y luego muchos html que serán extensiones de esta, con el código que difiere de un html a otro y manteniendo el del padre. Una de las muchas utilidades que tiene este sistema es, por ejemplo, en las barras de menús o laterales y elementos del pie de página, que son elementos que típicamente se repiten. Con este sistema nos ahorraremos escribir el mismo código una y otra vez, además de facilitar las modificaciones, ya que por ejemplo, si queremos añadir un nuevo botón a una barra de menús y no tuviéramos implementado este sistema nos tocaría recorrer todos los html que implementaran la barra e ir cambiándola una a una, con el consiguiente problema de tiempo y posibles errores. Usando el sistema de herencia se cambiaría solamente en las plantillas que contuvieran el código.

Un ejemplo de este sistema que hemos usado en nuestra aplicación sería el siguiente.

```
<--! base.html -->

{% load staticfiles %}
<!DOCTYPE html>
<html lang="es">
 <head>
  <meta charset="utf-8">

  <!-- Titulo -->
  {% block title %} <title> FSM : Herramienta de Gestión del Club Academia Albiceleste </title> {% endblock %}

  <link rel="stylesheet" href="{% static 'css/normalize.css' %}">
  <link rel="javascript" href="{% static 'js/modernizr-custom.css' %}">
  <link rel="stylesheet" href="{% static 'css/index.css' %}">
  <link rel="stylesheet" href="{% static 'css/icomoon.css' %}">

  <!-- Link especificos -->
  {% block link %} {% endblock %}

  <script src="http://code.jquery.com/jquery-latest.js"></script>
 </head>
 <body>

  . . . fragmento del código
```

Este fragmento de código corresponde al padre, el cual incluye dos bloques de código *block title* y *block link*, la plantilla que extienda de esta tendrá que definir estos dos bloques con el contenido que precise.

```
<--! convenio.html -->

{% extends "base.html" %}

<!-- Titulo -->
{% block title %} <title> Convenios : Visualización de las diferentes convenios  </title> {% endblock %}

<!-- Link especificos (si en el bloque se usa staticfiles hay que cargar con la etiqueta load)-->
{% load staticfiles %}
{% block link %} <link rel="stylesheet" href="{% static 'css/mutuas.css' %}"> {% endblock %}
  . . . fragmento del código
```

En este html podemos ver que en la primera línea extiende de la plantilla anterior y cómo define el código a insertar en la plantilla padre. De esta manera podremos tener multitud de variaciones sin necesidad de repetir el código.

Otra característica fundamental del sistema es el uso de variables. Es fundamental que los valores estén en constante cambio, para ello utilizamos otras etiquetas, que tienen las características de que pueden variar su valor e incluso su apariencia. Las etiquetas de las variables corresponden con dobles corchetes de apertura y de cierre {{variable}}. Cuando hablemos de los *renders* definiremos cómo llegan estas variables al documento html y desde dónde.

Una vez se tiene el documento creado, este va a contener tanto el código Python con sus etiquetas como el propio código HTML con su css y su JS[45]. Por lo tanto, si devolviéramos este documento al navegador web, su visualización estaría llena de errores. Para evitar esto los Templates hay que renderizarlos antes de devolverlos y para ello se utiliza una función que traduce las partes del código de etiquetas a código html, también es el render quien se encarga de pasar el valor de las variables contenidas en el documento. Esta función se llama en las funciones vistas, generalmente al devolver el valor.

Un ejemplo real de nuestro proyecto es el siguiente:

```
def buscarUsuario(request):
    errors = []
    if 'busqueda' in request.GET:
        busqueda = request.GET['busqueda']
        if not busqueda:
            errors.append('introduce un termino de búsqueda')
        elif len(busqueda)>20:
            errors.append('introduce un termino de búsqueda inferior a 20 caracteres')
        else:
            usuarios = Usuarios.objects.filter(usuario__icontains=busqueda)
            return render(request,'resultados.html',{'usuarios':usuarios,'query':busqueda})
    return render(request,'pruebas.html',{'errors':errors})
```

Si nos fijamos en las devoluciones de la función, vemos que se llama a una función render y que se le pasa por parámetros la plantilla html junto con las variables que usará, además del request obligatorio. De esta manera y de forma interna, Django sustituye las etiquetas por los valores y aplica la lógica impuesta en las etiquetas y en la herencia, para devolver finalmente un documento html común que podrá ser interpretado por el navegador.

Modelos (bases de datos)

Django es capaz de trabajar prácticamente cualquier motor de bases de datos y de cualquier tipo, ya sea relacional o no-relacional. Para ello únicamente hay que instalar al framework un conector con la base de datos elegida y realizar una instalación sencilla; no obstante, Django ya facilita una base de datos SQLite configurada automáticamente, para empezar a trabajar de forma inmediata.

Django utiliza un sistema de **mapeo objeto relacional**, **ORM** (Object-Relational mapping). Es la forma de conseguir un mapeo de los objetos usados en la aplicación web y (de) los almacenados en la base de datos relacional, por lo tanto, es una herramienta para persistir objetos con el formato que nos pide la base de datos, mientras nosotros los usamos como si de objetos del lenguaje se trataran, teniendo los atributos en los tipos de datos del lenguaje. La ventaja que tiene Django es que este mapeo o casteo lo hace de forma automática y transparente para el programador. Y las ventajas no acaban ahí, al usar el ORM la creación de las tablas y las relaciones se crean de forma automática, y si en algún momento tenemos que cambiar alguna tabla al tener que añadir un atributo, por ejemplo, el sistema facilita la labor permitiendo cambiar sólo el modelo del lenguaje y actualizando las tablas

[45] JavaScript lenguaje de programación interpretado, se utiliza de forma habitual en el lado del cliente, permitiendo mejorar la interfaz de usuario en páginas web dinámicas.

de forma automática y sus *INSERT*, *SELECT* y *UPDATE*. Por lo tanto y para concluir, un ORM es una abstracción completa de la base de datos al nivel del lenguaje de POO [46] usado.

Django llama a su sistema ORM como **Models** y las define como clases de Python que extienden de la clase *models.Model* que facilita el framework. El funcionamiento es muy simple, se crea una clase por cada tabla y se crea un atributo con un tipo de dato especial por cada columna del modelo. Las relaciones se definen como atributos y dependiendo de si son m2m[47], o2o[48] o foreingkey se utiliza un tipo u otro.

Veamos un ejemplo de la clase Empleado de nuestra aplicación:

```
class Employee(model.Model):

    GENDER_CHOICE = [('Male','Male'),('Female','Female'),('ND','ND')]

    timestamp = models.DateTimeField(auto_now_add=True)
    historic = models.BooleanField(default = False)
    historic_date = models.DateTimeField()

    id_employee = models.AutoField(primary_key=True)
    date_of_hire = models.DateTimeField(blank=True, null=True) #contratación
    dni = models.CharField(max_length=30, blank=True, null=True, unique=True)

   ...fragmento de código

    photos = models.ManyToManyField('Photo')
    rrss = models.OneToOneField('RRSS',blank=True, null=True)
    address = models.ForeignKey('Address',blank=True, null=True)

    def __str__(self):
        return '%s %s %s'%(self.name,self.first_surname,self.second_surname)
```

Podemos ver cómo se crean los atributos a partir del objeto models, este contiene varios tipos de datos habituales en la creación de la base de datos. También nos tenemos que fijar que admite parámetros y, como vemos, son opcionales y se utilizan para añadir distintas funcionalidades a los atributos, que van desde convertir el atributo en una *primaryKey* a limitar el tamaño del atributo, hacerlo *unique* y que no admita duplicados, etc. Como parte importante está el determinar las relaciones, en la parte final del código podemos ver cómo se tratan los tres tipos distintos. Al tratar photos como un m2m, cuando accedamos a ese atributo, podremos tener acceso a la tabla photo como si fuera un array con todas las fotografías asociadas; si accediéramos a la *foreingkey* tendríamos acceso a todos los datos de esa tabla como si fuera una extensión de esta misma, sólo que a partir del nombre del atributo, *addres.atributoDeAddres*.

También podemos ver que se pueden añadir funciones dentro de cada tabla, estas funciones pueden ser de lo más variadas, desde realizar un *toString* a limitar búsquedas…

[46] Sólo es posible esta abstracción de la base de datos en lenguajes que permitan las clases con atributos.
[47] Many to many.
[48] One to one.

Los pasos que hay que hacer para crear las tablas consisten en crear en (creo que sobra 'en') un archivo models.py con las clases que queramos que conformen nuestras tablas y sus relaciones. Luego, desde la terminal de Python sólo tendremos que ejecutar una serie de comandos[49] para convertir nuestro modelo ORM en una base de datos relacional completa.

Aplicar un modelo CRUD con este sistema es muy sencillo, un ejemplo de cada una de las partes es el siguiente:

Crear datos: Únicamente hay que importar el modelo que queramos usar, luego bastará con crear una variable, a la que a partir de la clase importada se instanciará. Por último, queda llamar a la función *save* que tiene la instancia.

```
from escuelaDeportiva.models import Escuela
e1 = Escuela(nombre='EscuelaRandom',cif=4201,'web=www.superescualarandom.com')
e1.save()
```

Recuperar datos y actualizarlos: Cada instancia tiene un manejador para realizar las búsquedas en los modelos, por defecto se llama *objects*[50]. Su forma de funcionar es sencilla, el manejador tiene una serie de funciones que ayudan a rescatar objetos de la BD, sustituyendo a las consultas sql. La consulta o devuelve un conjunto de resultados o un objeto único. Una vez tengamos el objeto instanciado, podremos modificarlo y guardarlo.

```
from escuelaPrueba.models import Escuela
eb = Escuela.objects.get(nombre ='SuperEscuela')
eb.nombre = 'MegaEscuela'
eb.save()
```

Eliminar un registro: Es tan sencillo como instanciar el registro que queramos borrar y llamar a la función delete().

```
from escuelaPrueba.models import Escuela
eb = Escuela.objects.get(nombre ='SuperEscuela')
eb.delete()
```

El uso de los modelos se da en las vistas y lo habitual es que se conecte con la base de datos para hacer la consulta, luego trate la información obtenida y la pase al render junto con la plantilla, para procesarla y obtener el html. En el siguiente ejemplo podemos observar este proceso:

```
def cargarConvenios(request):
    convenios = Agreement.objects.all().order_by('timestamp').reverse()
    return render(request,'convenio.html',{'convenios':convenios})
```

Lo mostrado en estas páginas sirve como referente para entender el funcionamiento de un framework web basado en MVC, ya que el funcionamiento entre ellos es parecido. Si se quisiera profundizar recordamos que se realizó un tutorial con los apartados más relevantes del desarrollo con Django, que se adjunta en el anexo.

[49] En el curso de anexo transparencia 83, se muestra cómo usar los comandos para la creación de las tablas.
[50] Pueden crearse manejadores de datos distintos, configurar su funcionamiento y añadirles funcionalidades nuevas.

Django REST

Django REST Framework (DRF) es una aplicación no oficial de Django aún así está entre las aplicaciones del Framework más usadas. Actualmente está en la versión 3 y está mantenida por Tom Chirtie.

DRF3 nos ofrece muchas ventajas como una API navegable desde el navegador, buena seguridad al tener integración con OAuth 1 y 2, una gran comunidad detrás y bastante documentación.

Instalación y primeros pasos

DRF se instala en el sistema desde cualquier instalador de paquetes de terminal como PIP o EasyInstall, cuando esté instalado en un nuestro ordenador hay que añadirlo en forma de aplicación a nuestro servidor Django, esto se hace en settings del proyecto en INSTALLED_APPS = () de la misma manera que se instala, por ejemplo la aplicación creada para la base de datos. El settings.py del proyecto también será el settings de DRF siempre que le añadamos el espacio correspondiente. Al igual que con otra aplicación tendremos que añadir el URLs.py de nuestra nueva aplicación al urlpatterns de entrada del servidor, para poder dirigir todas las peticiones REST.

Models

Si únicamente estamos creando una REST y no un mixto de servidor web y REST, tendremos que crear un archivo models en la nueva aplicación con todos los modelos de datos de nuestra BD por el contrario si vamos a usar la misma base de datos, solo tendremos que enlazarla y usar la misma BD para las dos app.

Serializers

Dado que la información que vamos a estar intercambiando mediante la API van a ser objetos json, tenemos que crear una herramienta que se encarga de serializar y des-serializar, desde y hacia nuestro modelo. Para ello se crea una clase Serializers y definar una clase por cada modelo de models.py que tengamos, esta clase tiene que heredar de **serializers.Serializer** y definir las funciones create y update. Aquí vasmoa indicar a la API que para el modelo establecido en esta clase devuelva los campos marcados

ViewSets

Al igual que con el framework Djando con View, esta clase ViewSet va a ser el punto de resolución una vez llegue la URI. Para ellos tendremos que importar los serializers que se han creado antes. Las clases creadas aquí heredan de **viewsets.ModelViewSet** y solamente hay que indicar el *queryset* que va a indicar lo que queremos devolver, por ejemplo: *queryset = Empleados.objects.all()* si queremos devolver todos los empleado o bien la devolución que queramos, además no hay porque usar objects podemos usar cualquier manejador que tengamos definido. También hay que indicarle a la clase a que serializer_class pertenece, en este caso: *serializer_class = EmpleadosSerializer*.

Después de agregar la nueva aplicación y los nuevos archivos la ruta nos tiene que quedar de una forma similar a esta, siempre que no tengamos otra aplicación más instalada en el proyecto.

```
proyecto
├── proyecto
├── __init__.py
├── settings.py
├── urls.py
├── wsgi.py
├── manage.py
└── api
    ├── migrations
    ├── __init__.py
    ├── admin.py
    ├── models.py
    ├── tests.py
    ├── serializers.py
    └── viewsSets.py
```

Routers

Por último hay que agregar las URIS al router de la aplicación. Esta es la herramienta que nos permite definir los verbos HTTP con el conjunto de URLs y además definimos qué métodos de una class se tiene que ejecutar cuando llegue la petición. Consiste en asignar al router creado la parte de la URI con el ViewSet asociado. Se Implementa en el URL de la aplicación REST o directamente en el URLS.py del proyecto.

En el punto en el que estamos si hacemos una petición HTTP por terminal a nuestro servidor, solicitando los empleados, vemos que nuestra API ya es funcional, aunque muy básica pero nos sirve para ver el alcance de este framework REST

```
http://127.0.0.1:8000/Empleados/

HTTP/1.0 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Date: 30 Apr 2017 14:11:50 GMT
Server: WSGIServer/0.1 Python/2.7.10
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "count": 2,
    "next": null,
    "previous": null,
    "results": [
        {
            "email": "daniel.yebra@gmail.com",
            "groups": [
                "http://127.0.0.1:8000/groups/1/"
            ],
            "url": "http://127.0.0.1:8000/users/2/",
            "username": "dani"
        },
        {
            "email": "paco@edu.uah.es",
            "groups": [],
            "url": "http://127.0.0.1:8000/users/1/",
            "username": "Fran"
        }
    ]
}
```

Host y servidores

MAMP

Es un paquete de desarrollo completo que trae las herramientas necesarias para montar un servidor html de desarrollo. Es donde vamos a realizar la construcción del CMS antes de subir todo al host de producción.

El conjunto de paquetes es multiplataforma y existen versiones tanto para Mac como para Windows. Existen dos versiones, una gratuita y otra de pago. La de pago sólo aumenta las posibilidades de configuración de las herramientas, pero para el alcance del proyecto no nos es necesario, por lo que nos centramos en la versión gratuita.

Las herramientas que forman este paquete son Apache, Mysql y PHP. Una vez tengamos instalados el paquete y arranquemos los servidores veremos que tanto Apache como Mysql corren como servicios del sistema, esperando una conexión.

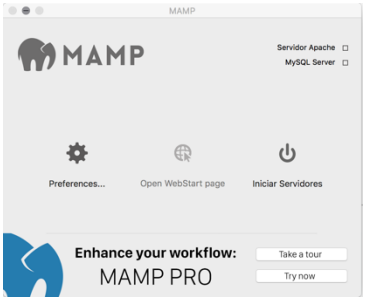


Ilustración 9 Panel de inicio MAMP

Como se puede ver en la ilustración 6, el panel es muy sencillo y sólo tendremos que dar a un botón para iniciar los servidores. Antes de hacer esto y, si nuestro propósito es trabajar con un CMS, tendremos que indicarle a MAMP en qué carpeta está el código del CMS. Para ello sólo hay que ir a preferencias, entrar en webserver y buscar la carpeta para que arranque desde ahí.

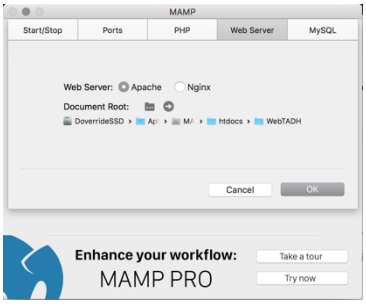


Ilustración 10Acceso de mamp al root

Como se ha comentado, también dispone de una base de datos Mysql; para acceder a ella el sistema viene incluido con PHPmyAdmin, que es una herramienta escrita en php para la administración web de bases de datos; nos permite crear, eliminar y actualizar tablas y campos, además de visualizar su contenido. Para acceder sólo tenemos que ir a nuestro navegador y teclear la dirección del servidor con la url *mamp* y se abrirá una ventana, en la que podremos lanzar phpMyAdmin, por defecto la dirección es : localhost:8888/MAMP.

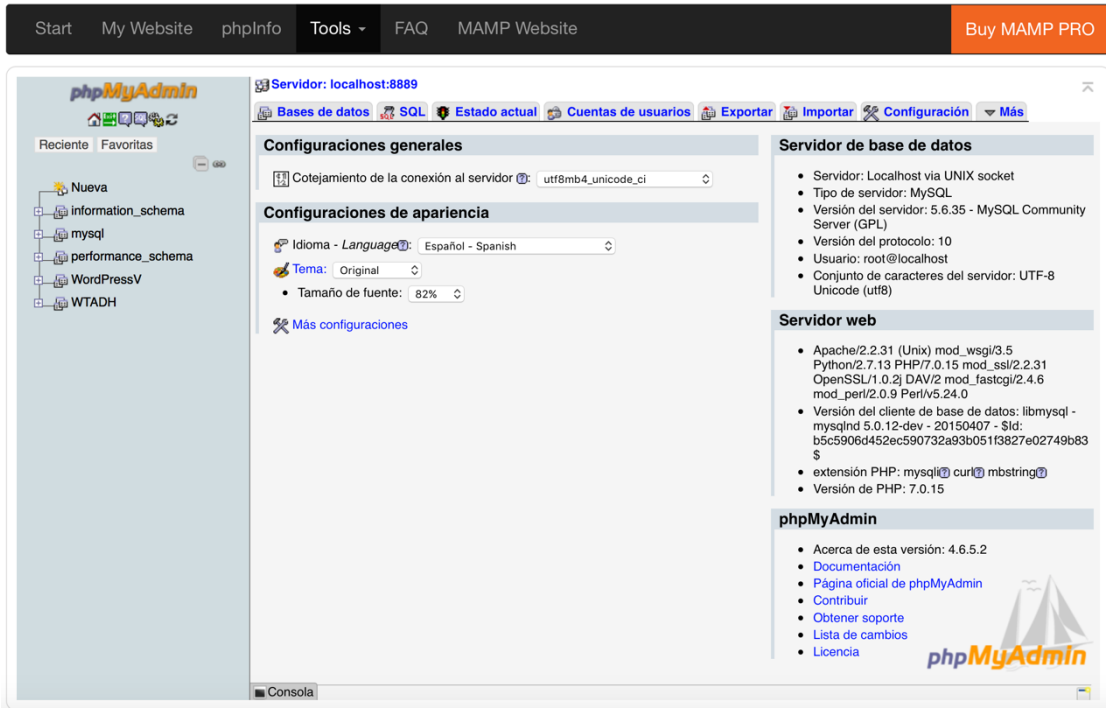


Ilustración 11Ventana inicio phpMyAdmin

Amazon web Service

IAM y seguridad

Para poder usar todos los servicios que nos facilita AWS, el primer paso es darse de alta en el sistema, para lo que necesitaremos inscribirnos con una tarjeta de crédito además de una confirmación en dos pasos vía teléfono. Una vez inscritos, antes de utilizar cualquier sistema, hay que crearse un superusuario **IAM** (Identity and Access Management). Al activar IAM es necesario disponer de un dispositivo de MFA[51]; en nuestro caso hemos utilizado una emulación por software usando el programa *Google Authenticator*. En el momento que terminemos la configuración IAM para acceder a AWS vamos a necesitar la contraseña que dimos de alta y la contraseña que nos indicará el MFA cada vez que entremos; esta contraseña después de unos segundos será sustituida por otra nueva y dejará de ser válida. El siguiente paso es controlar el acceso al sistema, para ello se crean usuarios y grupos de usuarios con distintos permisos. Lo recomendable es que el usuario root que se ha creado en primera instancia no se use y se cree otro usuario con menos privilegios.

Todas las acciones descritas y el resto de configuraciones de seguridad se hacen desde la consola de services IAM.

[51] Multi-Factor Authentication, sistema de seguridad de gestión de contraseñas únicas, puede ser físico o virtual, las contraseñas las genera a partir de un token o semilla, compartido.

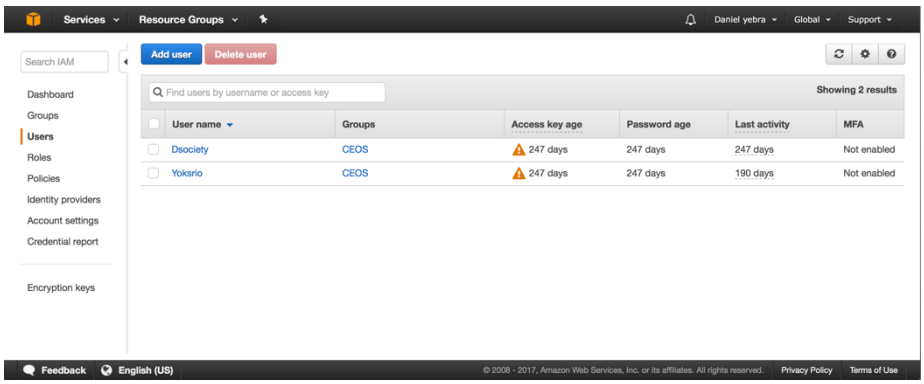


Ilustración 12Consola IAM seguridad AWS

Acceso a las herramientas del sistema

Dado que el servicio de la capa gratuita de Amazon permite acceder a todas sus funcionalidades, aunque estas estén limitadas, podemos hacerlo desde la consola web en el apartado *Services*; nos listará todas las opciones divididas en bloques con subniveles.

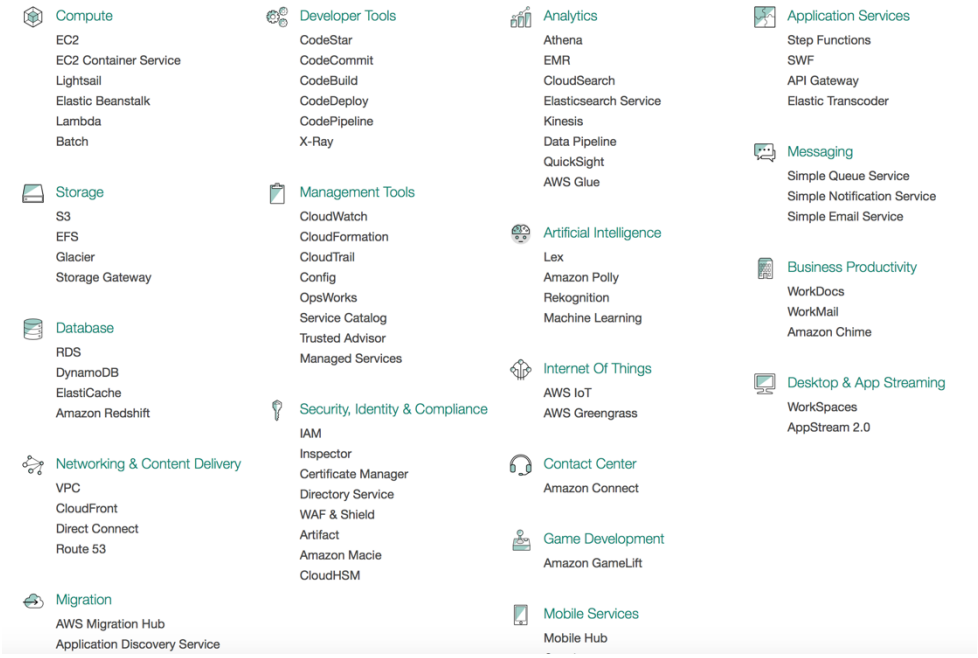


Ilustración 13 Captura del escritorio de servicios en AWS

El sistema también permite crear un *Resource Grups*, que son accesos directos a las aplicaciones que más usamos. Algo bastante útil dada la cantidad de opciones que tenemos.

Instancia Ec2

El siguiente paso que tendremos que hacer es crear la instancia en la que vamos a alojar nuestro servidor. Para ello tendremos que irnos al dashboard del ECD y lanzar la máquina. El primer paso que nos va a pedir es que decidamos que SO va a correr, y nos ofrece múltiples opciones, desde Ubuntu, Windows, Amazon linux… Hay que reseñar que no todos los SO están disponibles en la capa gratuita y hay que fijarse en que especifique "*Free tier*" debajo del logo del SO. A lo largo de la creación de los servidores nos encontraremos con muchas decisiones en las que podríamos salirnos de la capa gratuita y empezarían a facturarnos directamente, por lo que hay que poner especial cuidado a estos detalles.

Las elecciones desde aquí son bastante sencillas, ya que la capa gratuita no da muchas opciones y para nuestro proyecto nos es suficiente; no obstante, siempre se puede subir al siguiente nivel y tendremos las opciones para hacerlo de manera automática o manual. Las elecciones son las siguientes: Instance Type, propósito general t2.micro; almacenamiento 50 GiB SSD; reglas de seguridad. Antes de lanzar la instancia hay que crear un Key Pair, que consta de una clave pública guardada por AWS y una clave privada guardada por nosotros. Vamos a necesitar el archivo privado para poder hacer la conexión por ssh.

AWS nos da varias alternativas para conectarnos a la instancia que acabamos de crear. La forma más sencilla y recomendable es mediante el **cliente ssh de java,** que se ejecuta desde el navegador. Para ello tenemos que ir a las instancias del EC2 y pulsar conectar, se nos abrirá una ventana en el navegador que nos pedirá que adjuntemos la ruta de nuestra clave privada. Y ya nos mostrará la Shell con la que podemos interactuar. También podemos conectarnos desde nuestro terminal mediante las APIs que facilitan y de un cliente de ssh.

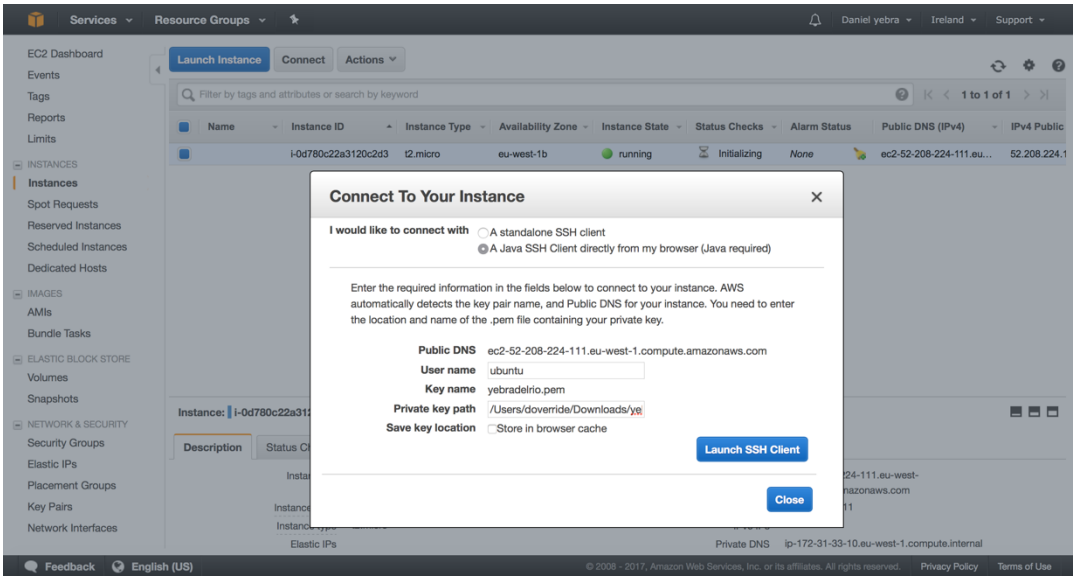


Ilustración 14 Conexión ssh con EC2 AWS

Instancia RDS

De la misma forma que con el EC2 accedemos al panel de control del RDS, seleccionamos la BD que queremos instanciar y seleccionamos DEV/TEST, que es la única opción que entra dentro de la capa gratuita. Luego tendremos que elegir el tipo de instancia, que continuaremos con el t2.micro, el espacio de almacenamiento y las configuraciones de superusuario.

Cuando tengamos la configuración completa, el siguiente paso es conectar nuestra base de datos a nuestro sistema; por ejemplo, si estamos usando Django, esta acción se realizaría en el archivo *settings.py* y tendríamos que especificar el *endpoint* de la base de datos con el puerto y el nombre de usuario y contraseña creado. Para conocer el endpont y el puerto tendremos que mirar en la consola de AWS RDS en *Configurations Details*. Hecho esto, la configuración de las replicaciones de la BD y las acciones sobre ella se efectúan en el desplegable de Instance Actions.

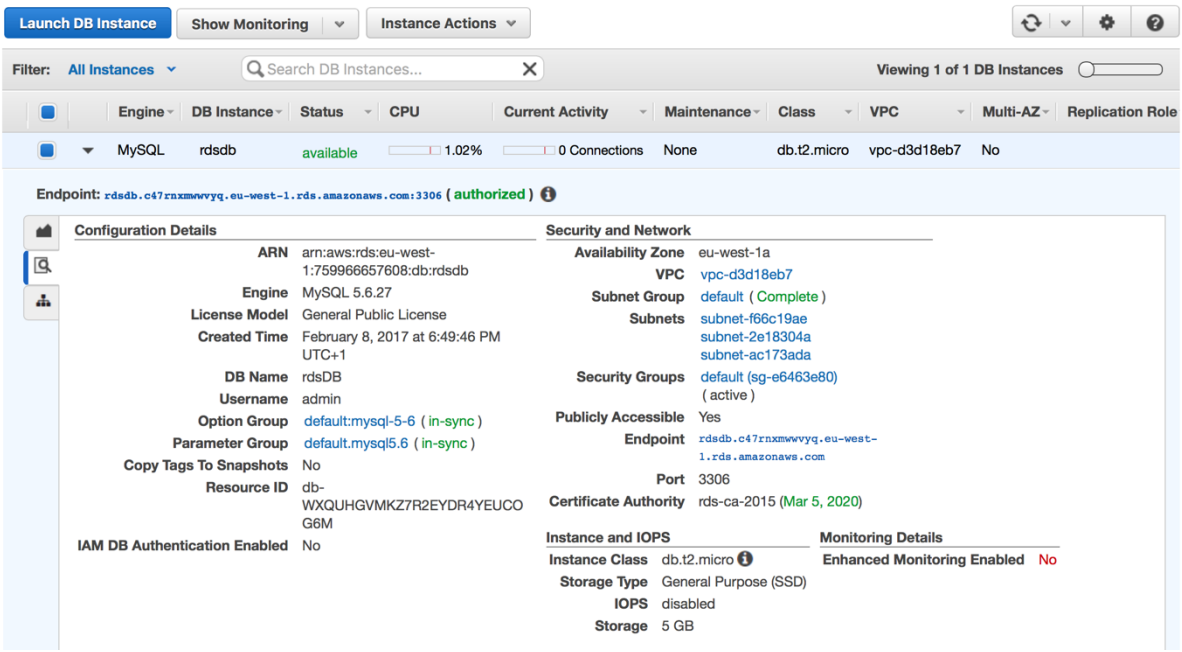


Ilustración 15 Consola de configuracion RDS Details

En la Ilustración de la consola RDS podemos comprobar el estado de la BBDD (BD), el consumo de la CPU, además de conocer detalles importantes de la máquina, como la fecha de creación, grupos de seguridad, espacio…

Programación

Explicación del proceso

El proceso de codificación es la parte que atañe solo al equipo de desarrollo. No es una parte que esté muy determinada en las guías de metodologías ágiles, sin embargo, la metodología elegida Programación Extrema(XP) en sus pilares fundamentales nos habla de la filosofía que tenemos que mantener: simplicidad, comunicación, respeto. Ambas muy centradas en mantener las buenas prácticas a la hora de programar. Hay que recordar que los equipos de desarrollo son muy pequeños y que es habitual que los desarrolladores cambien de puesto. Por lo tanto, hay que ser muy estrictos y metódicos a la hora de programar y pensar siempre en la persona que va a venir detrás y en la legibilidad del código. Para formalizar estas cuestiones existen ya ciertas guías que podemos seguir para facilitarnos el trabajo.

Buenas prácticas

Se consideran buenas prácticas a un conjunto coherente de acciones que han tenido éxito en un determinado contexto y se espera que vuelvan a dar los mismos resultados en contextos similares. Este conjunto de acciones es de los más variado, vamos a comentar los puntos más relevantes:

- Indentación y estilo: Mantener una indentación coherente a lo largo del proyecto, seguir un estilo marcado de espacios, sangrados y posiciones de puntuadores[52] en definitiva, mantener un estilo uniforma mejora la legibilidad del código.
- Elección de nombres: Usar nombres descriptivos y claros. Es importante que los nombre revelen intenciones. Un buen nombre de una variable, función y clase tiene que responder a las siguientes cuestiones, por qué existe, qué hace y como se usa. Una buena guía para saber si estamos eligiendo bien, es qué si necesita de un comentario para explicarse, no es un buen nombre. Se considera también una buena práctica poner nombres de acciones a los métodos.
- Creación de funciones: La primera regla es que el tamaño de las funciones tiene que ser reducido. Las funciones sólo tienen que tener un propósito y hacerlo bien, evitando que se centren en más de una cosa para ellos cambiaremos el nivel de abstracción si es necesario.
- Comentarios: Los comentarios no compensan un código mal escrito, lo ideal es que el código se auto-explique pero si es necesario clarificar algo del código o explicar la intención es importante ser directos y escuetos si se puede explicar en tres palabras no hacerlo en cuatro. También, comentar al inicio de las funciones la condiciones, entradas y salidas además de advertir de consecuencias, además de incluir notaciones TO-DO para comunicarnos con el resto de desarrolladores y con nuestros yo futuros.
- Orden en clase: Mantener las convención de estilo de los lenguajes de POO. Las variables tienen que ir en primer lugar, siguiendo un orden como: las constantes publicas, variables estáticas privadas, variables de instancia privada, y por último las públicas. Al conjunto de variables les siguen las funciones públicas, y privadas. Además de seguir el principio de encapsulación tienen que ser reducidas.

Como podemos ver todos los puntos coinciden en que hay que aplicar la lógica y buscar los simple y estándar. Codificando siempre pensando en que el código sólo se escribe una vez, pero se lee muchas.

Formalizar guías de estilo

A relación con las buenas prácticas, cada lenguaje tiene elaborado unas guías de estilo que marcan las buenas prácticas. Es función del equipo encontrar de forma conjunta una guía de estilo que satisfaga a todos, y ser estricto en su cumplimiento. Las guías de estilo a seguir nos son cerradas y es habitual que cada empresa cree las suyas propias a partir de alguna existente.

Pair programing

Una de las formas de programación que contempla la metodología XP es el Pair programing. Consiste en una programación conjunta, con dos roles establecidos entre los desarrolladores. **El controlador** es el encargado en centrarse en resolver el código que se está desarrollando en ese

[52] **Puntuadores :** [ ] ( ) { } , ; : ... * = # ! % ^ & – + | ~ \ ' " < > ? . /

momento, también es el responsable de comunicar el punto de vista que tiene sobre la resolución de un problema y constatar la opinión con su compañero. **El navegador** se encarga de hacer una revisión completa del código, prestando atención a errores de programación (centrados en la algoritmia o en la codificación no errores sobre la guía de estilo o errores tipográficos, los cuales se revisarán al final del propósito o lo advertirá el IDE de desarrollo); es el encargado de mantener la atención fija en el propósito actual y saber identificar cuando el controlador está cansado y hay que hacer un relevo de roles. También de forma conjunta serán los encargados de hacer el diseño y la planificación del desarrollo, así como de discutir las mejores ideas para la implementación.

Las ventajas de este tipo de programación para equipos de desarrollo junior FS son las siguientes: Transmitir conocimientos entre ellos de una forma cómoda y natural; una fácil adaptación a nuevos entornos y lenguajes al tener el apoyo de un compañero el efecto abrumador decrece; Mejora el estilo y la legibilidad tanto del proyecto, como a la larga de los programadores, ya que el desarrollo del código habrá pasado por el filtro de dos personas distintas.

Como puntos negativos están que no siempre es fácil encontrar equipos que trabajen bien juntos, que ambos tienen que estar igual de cómodos en cualquiera de los dos roles, es probable que en programadores experimentados se rinda más trabajando en paralelo.

Organización propia

Un punto muy importante a la hora de enfrentarse a una iteración de un proyecto, es tener la capacidad de organizarse uno mismo y saber priorizar dentro de las tareas designadas y conocer por cual empezar. Para ello es recomendable que se hagan pequeñas iteraciones con aproximaciones temporales y llevar un historial personal de la iteración, anotando las debilidades y las fortalezas, con la finalidad de capturar los problemas y poder tratarlos .

Caso práctico

En el desarrollo de nuestra aplicación se ha realizado pair programing en cada inicio de herramienta nueva del sistema, y se establecieron sesiones esporádicas con la finalidad de continuar la cohesión entre el equipo y compartir resoluciones. Además se establecieron bases conjuntas de buenas prácticas.

Desarrollo y explicación de la tecnología usada

Como herramienta para la codificación se han utilizado editores de texto y código fuente especializados para el desarrollo software. A estos editores se les pueden añadir funcionalidades típicas de los IDEs de desarrollo como, autocompletado y marcado de llaves, coloreado de la sintaxis para todos los lenguajes existentes, y la posibilidad de incorporar añadidos para mejorar sus características. En nuestro desarrollo se han utilizado dos editores que prácticamente no tienen ninguna diferencia y su uso recae en gustos personales sobre su apariencia: SublimeText y Atom.

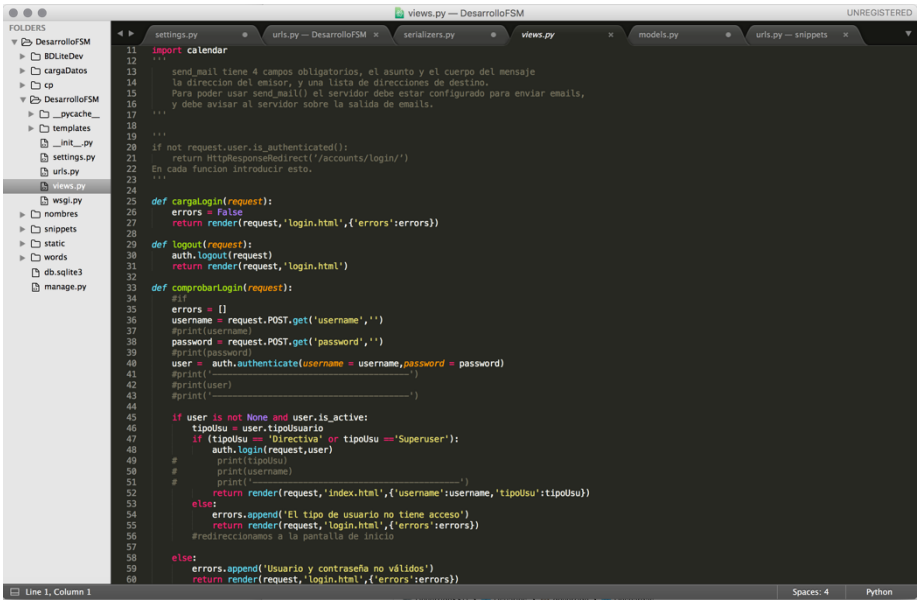


Ilustración 16Interfaz SublimeText

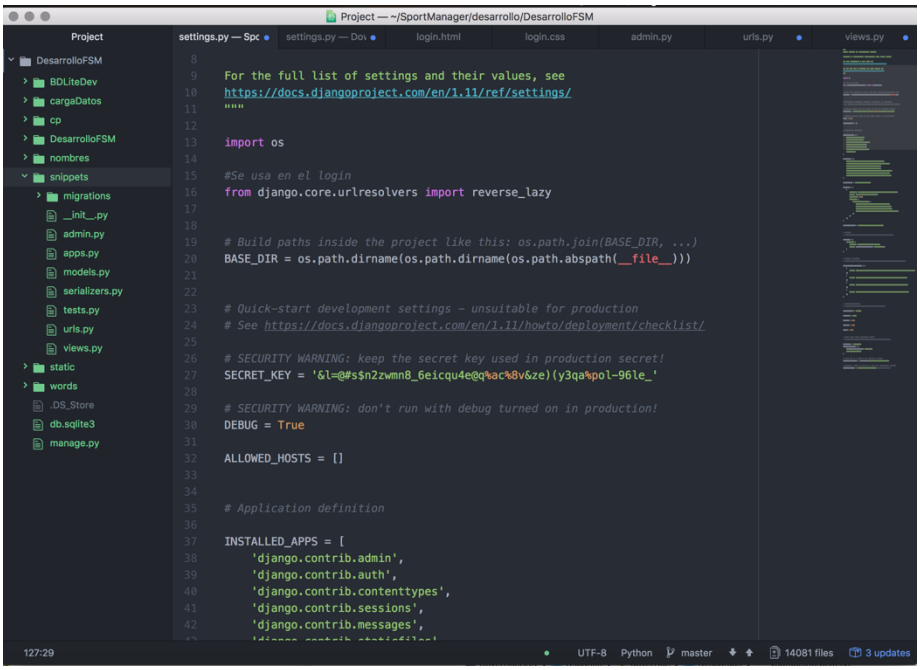


Ilustración 17Interfaz Atom

Pruebas

Explicación del proceso

Es una fase de vital importancia para la metodología elegida y para el conjunto de las metodologías ágiles. Uno de los propósitos de esta metodología es el que cada vez que se cierra un sprint o una tarea, no volver a abrirlo y esta es la base de la planificación del proyecto centrarse en proceso actual sin tener que estar pendiente de funcionalidades anteriores. Es por esto por lo que hay que garantizar, con el mayor grado de seguridad posible, que las pruebas sobre la funcionalidad que se está implementando estén bien diseñadas y sean correctas.

El diseño de las pruebas en metodologías ágiles se realiza siempre antes de empezar la codificación, este es uno de los puntos que más difiere con la metodología tradicional. Aun así, la metodología elegida que hemos elegido, XP, pone aún más énfasis en el proceso de pruebas, usando la técnica **TDD** (Test Driven Development) y **ATDD** (Aceptance Test Driven Develepment).

La diferencia fundamental entre estos dos estilos radica en la intervención del cliente y en diseñador en sí, de la prueba.

En el sistema **TDD**, es el propio desarrollador encargado de esa tarea el que diseña y programa las pruebas a partir de la información que el estime. En una primera instancia se crea la prueba que se va a realizar. La prueba va a fallar, ya que la funcionalidad es inexistente por lo que el siguiente paso del programador es crear una función lo más simple y rápido posible para que pase la prueba. Como último paso hay que refactorizar el código, atendiendo a la redundancia y eficiencia. Cuando damos por finalizada la prueba se pasa a la siguiente prueba hasta terminar toda la iteración.

Por el contrario, el sistema **ATDD** se basa en criterios de aceptación[53], lo que le define más como metodología que como procedimiento práctico. En él se busca tanto la aceptación como la creación de la prueba teniendo al usuario como actor determinante en el proceso. El diseño de la prueba se hace a partir de la historia de usuario y/o de reuniones con el cliente y no se cierra la tarea hasta que éste da el visto bueno a la resolución de la prueba. De esta manera se sabe que el cliente estará satisfecho en el propio proceso de creación.

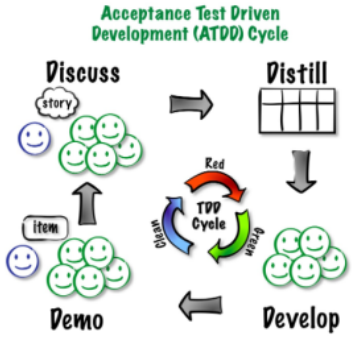


Ilustración 18Por James Shore, Grigori Melnick, Brian

[53] Se define en la IEEE como **prueba de aceptación** a: "aquellas pruebas formales con respecto a las necesidades, requerimientos y procesos de negocio del usuario, conducidas para determinar ya sea si un sistema satisface o no los criterios de aceptación y habilita al usuario, cliente o entidad autorizada para determinar si el sistema se acepta o no" IEEE Std 610-1991, IEEE Computer Dictionary

En muchos casos estos dos conceptos tienden a mezclarse o a confundirse, determinando que la TDD son sólo pruebas unitarias[54] y las ATDD sólo pruebas de integración[55], pero el matiz principal entre ellas es la relación con el cliente en este proceso.

Caso práctico

En nuestro proyecto hemos tenido el defecto de no seguir estas normas desde el principio a causa del desconocimiento y observamos las consecuencias derivadas de esto en nuestro propio desarrollo. En las primeras fases de realización del servidor se crearon las pruebas una vez se había desarrollado las interfaces y no se fue muy exigente al realizarlas, la consecuencia fue que una fase posterior hubo que cambiar el modelo ORM y deshacer bastante trabajo sobre la base de datos. A día de hoy estamos implementando ambos modelos tanto. Dejando TDD uno para las pruebas unitarias y de integración de más bajo nivel y ATDD para pruebas de integración más generales a un nivel más cercano a la interfaz.

Herramientas y las tecnologías

Cada lenguaje de programación e incluso cada IDE suele tener unas librerías específicas para realizar test unitarios, como *unittest* en Python o *XCTestCase* en Swift.

La complejidad de los test unitarios radica más en elegir la prueba que en la codificación, no obstante, la gran mayoría de test unitarios TDD tienen una codificación parecida, usando una clase contenedor, y una clase antecedida por la palabra test, que es un marcador para indicar a la librería que es la función de prueba, dentro de esta clase se llaman a las funciones a comprobar y mediante funciones de la librería de testing nos comprueba si el resultado es el esperado. Muchos lenguajes también permiten usar librerías **mock** que crean objetos fake, con la finalidad de romper relaciones con otros objetos del ámbito y probar así el requerimiento de manera independiente.

[54] "Escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto."

[55] "Una vez que se han aprobado las pruebas unitarias, las pruebas de integración lo que prueban es que todos los elementos unitarios que componen el software, funcionan en conjunto."

Documentación

Explicación del proceso

Generalmente se tiende a pensar que las metodologías ágiles están exentas de documentación, pero no es la realidad.

Durante todo el proceso es necesario ir generando la documentación en la mayoría de los casos como representación del trabajo realizado y para poder trabajar con ello e ir creando las iteraciones. El primer documento con el que tratamos surge con la **planificación inicial de la entrega**, aunque la metodología genere mucha incertidumbre, es necesario planificar un plan para el proyecto en el que se creen unas estimaciones de costo, dinero… aunque dada la naturaleza de la metodología todo lo generado es necesariamente impreciso y flexible, aun así, al terminar esta fase se genera un **documento de requerimientos iniciales** que va a ser la base del proyecto y se podrá determinar como *el core*, al que implementar funcionalidades. Los siguientes documentos son los que se realizan **durante las iteraciones**, se genera documentación antes de empezar con el sprint, en el que se decide el alcance de la iteración, se dividen las tareas, se marcan fechas, etc. Los primeros documentos en generarse son las **historias de usuarios** que será lo que de forma a la iteración. Con el fin de poder cuantificar la evolución de los sprint se genera el ***iterationPlan***. El conjunto de estos documentos, tendrían que generar una ampliación mucho detallada del documento de requerimientos.

Más allá de lo explicado, que se resumen en una documentación de captura de necesidades y de seguimiento muy básico. El resto de documentación es **generalmente volátil** y suele estar basada en diagramas de diseño que se crea antes de la codificación y se utilizan medios como pizarras, bocetos en papel, por lo que no perduran en el tiempo. No obstante algunos de estos diseños generales como el diagrama de despliegue o el de estados, se suelen guardar como parte de la documentación.

Además, también hay que considerar documentación a la que se crea mientras se codifica, siguiendo las directrices de las buenas practicas, y realizar comentarios antes de las funciones, al inicio de las clases y en el código que no es evidente o que genere ambigüedades

El manual de usuario, si se requiere, también es considerado documentación y si este va asociado a una historia de usuario en el sprint tendría que especificarse ex profeso y generalmente irse trazando a medida que se genera el código.

No sólo hay que prestar atención a la documentación que se crea, también es importante el tratamiento que recibe la que se consume, por lo que es recomendable compartir la documentación de la tecnología con la que estemos trabajando con el resto del equipo, almacenándola en un lugar accesible por todos.

Caso práctico

En primera instancia y mientras nos íbamos acomodando a la metodología ágil se creó multitud de documentación más típica de la metodología tradicional. Pero mientras avanzábamos con el proyecto fuimos dejando a un lado la creación de documentos y nos acogimos a la filosofía Agil usando la documentación para negociar las iteraciones mediante models Trello

Además, dado que tenemos una parte del proyecto que es la creación de una API, es más que probable que tengamos que generar la documentación de la parte pública de la API si en algún momento del desarrollo introducimos compatibilidad con aplicaciones ajenas a nuestro desarrollo.

Herramientas y las tecnologías

Lo más común además del uso de editores de texto tradicionales como Pages, Word y Libreoffice es la creación de una WIKI del proyecto, de hecho, lo habitual es crear una especialmente para el detallado de funcionalidades de cada versión que se ha sacado de la aplicación y otra para ideas futuras y otras necesidades.

También se han utilizado como se ha comentado anteriormente, las herramientas omnigraffle y sketch, muy útiles para diseñar entornos de usuario y realizar diagramas.

En cuanto a la creación de la documentación del código existen algunas herramientas o añadidos de las IDs para extraer todos los comentarios y crear un DOC sobre nuestro programa ya sea en archivos XML, muchos tan sencillos como irse a archivo y pulsar en generar doc.

Para el consumo de documentación se ha utilizado un programa que aúna en un solo lugar todas las documentaciones, permitiendo además su descarga desde la propia aplicación, añadir notas, agrupar códigos, etc. Se llama Dash, y su uso es completamente intuitivo.

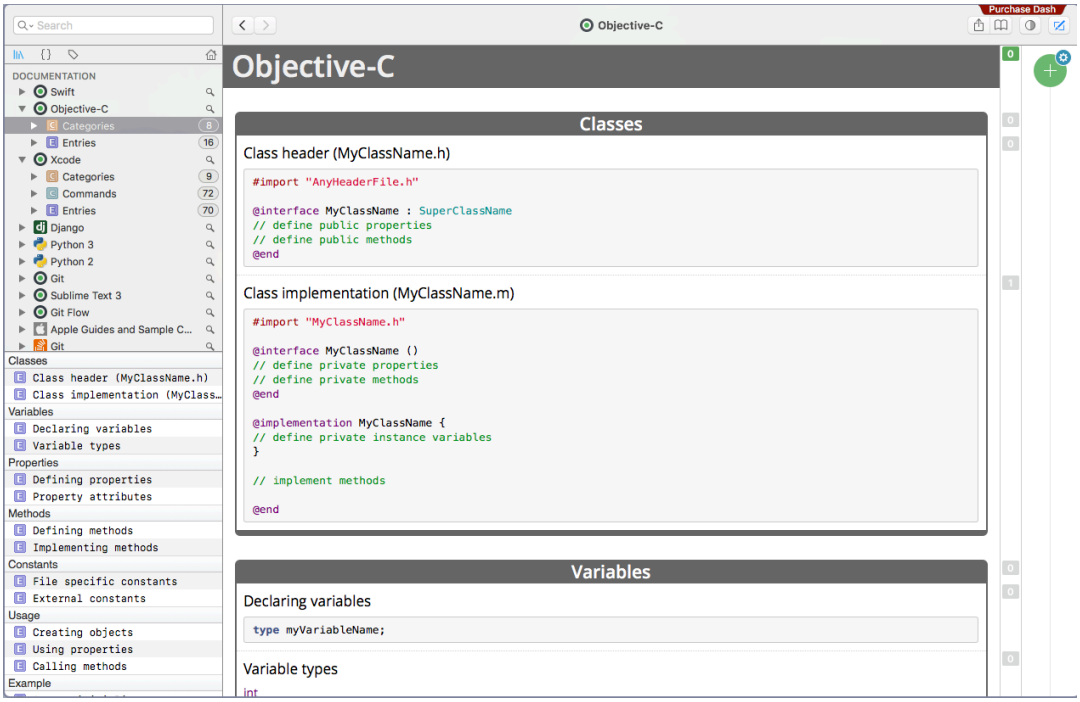


Ilustración 19Captura del funcionamiento de Dash.

CIERRE

Explicación del proceso

Varias pueden ser las razones por las que se termine un proyecto Startup, algunas de ellas vienen dadas por el éxito del producto, que hace que la empresa tenga un gran crecimiento y por el camino pierda la filosofía; puede ser que se quede a medio camino del éxito, pero que otra empresa mayor sepa ver el potencial y desee absorberla; también se puede dar el caso de que por alguna razón ya sea problema del producto o de la gestión no se obtuvieron los resultados esperados y se tuviera que cerrar el proyecto y el más común que es la falta de moral que acompaña a la inexperiencia.

Si profundizamos en la parte más negativa, vemos que los principales motivos que impiden el éxito de un STUP son los siguientes:

Problemas de tesorería típicos:

- CAC<LTV: Se produce cuando se dedica el mayor esfuerzo económico a captar clientes que los que se dejan estos en tus productos. Analizar el coste de adquisición del cliente (CAC) es fundamental para no desviarnos del objetivo de buscar la rentabilidad. Es bastante común que la STUP tienda a obsesionarse en captar clientes a cualquier precio. El LTV es el valor que obtenemos en el tiempo vida del cliente, analizar el LTV no es tarea sencilla ya que el valor puede no estar en una única compra y existan otros marcadores como el potencial de una nueva comprar, el marketing que genera, etc. No obstante, lo más recomendable es considerar el LTV desde un punto

de vista más negativo para mantener siempre una holgura sobre posibles variaciones no esperadas.

- Estimaciones erróneas: Cuando se calcula mal un proyecto, y se gasta más de lo calculado, (aunque esté todo bien dimensionado y el gasto este justificado) puede hacer que se agote el dinero antes de llegar a una próxima ronda de financiación.

- Morir de éxito: Es posible que el éxito del producto genere una demanda que la STUP no pueda complacer, ya que no disponga de los fondos necesarios para aguantar el ritmo de crecimiento que supone tener una oferta por debajo de las exigencias del mercado.

Problemas internos:

- Desmotivarse: El ansia emprendedora y la ilusión, son sensaciones que ayudan a superar la incertidumbre y a sobreponerse a los problemas que surjan manteniendo una visión positiva. Pero, no son sensaciones que se mantengan constantes. Por lo que hay que estar continuamente buscando la inspiración y mantener la visión.

- Conflictos internos: Uno de los problemas principales de este tipo de empresa es la jerarquía cuando hay más de un socio fundador. El tiempo y las discusiones sobre estrategias y modelos de negocios pueden acabar en enfrentamientos irresolubles. Para ello, la mejor opción es crear políticas de arbitraje y resolución de conflictos intentando buscar distintos mecanismos de gestión para encontrar una solución aceptable.

En definitiva, cualquier flaqueza en la trinidad de Visión Equipo y Dinero pueden hacer peligrar la STUP.

Existe una frase anónima que dice lo siguiente: "*Un hombre inteligente aprende de sus propios errores mientras un sabio aprende errores de los demás*." Está en nuestra mano intentar todo lo posible por conseguir el éxito y en la era de la información en la que estamos, tenemos al alcanza de un *click* la experiencia de muchos emprendedores y las historias de sus éxitos y fracasos. Si somos inteligentes seremos capaces de esquivar muchos baches que otros ya han superado antes y han dejado constancia de ello.

Caso práctico

Por fortuna, en el tiempo que se lleva desarrollando la STUP no han existido problemas relevantes que pusieran el riesgo la integridad del proyecto. Si es cierto, que la carencia de financiación y la falta de experiencia han retrasado todas las fechas estimadas, por otro lado, variación completamente esperable dada la situación.

Si nos centramos en el futuro inmediato, podemos ver que es bastante incierto, ya que tenemos un abanico de posibilidades que hay que explorar, ya que estamos viendo que la aplicación está despertando interés en algunos inversores y hemos tenido ya acercamientos con tentativas de compras. No obstante, el foco continúa puesto en realizar el proyecto de manera independiente hasta

la etapa de venta final, momento en el que es posible que se realice una ronda de financiación para obtener ayuda comercial y publicitaria.

5. CONCLUSIONES Y TRABAJOS FUTUROS

Este proyecto surgió antes como idea de negocio entre dos compañeros de universidad, que como proyecto de fin de grado, pero nos pareció interesante el poder plasmar lo aprendido a través de una memoria que perdurara y pudiera ser de utilidad tanto para nosotros en un futuro, como para cualquier persona que necesite una guía para empezar con un proyecto. Observando el resultado, creo que se mantiene el propósito de querer dar una idea general del proceso, incluyendo algunas pistas en forma de experiencias personales y herramientas.

Otro de los motivos de realización del TFG ha sido el de experimentar con el máximo número de tecnologías que actualmente se usan en desarrollos reales y poder obtener así una ventaja laboral a la hora de buscar empleo; objetivo que ha sido todo un éxito ya que es el principal motivo por el que están contactando con nosotros varias empresas, con el fin de que trabajemos con ellos.

El tiempo de realización del proyecto ha sido muy bien aplicado, ya que se ha podido poner a prueba prácticamente todos los campos que se han enseñado a lo largo de los cuatro cursos del grado; afianzando así muchos conocimientos y sirviendo de calentamiento para el inmediato futuro laboral.

Como puntos de mejora podríamos aclarar que hubiera sido recomendable y mucho más productivo empezar por la búsqueda de la metodología adecuada en lugar de aventurarnos directamente a programar, hubiéramos ahorrado mucho más tiempo y esfuerzo. Sacamos en claro también, que hay que sobredimensionar absolutamente todas las estimaciones, ya que la falta de experiencia hace que el tiempo sea un factor de incertidumbre muy difícil de prever.

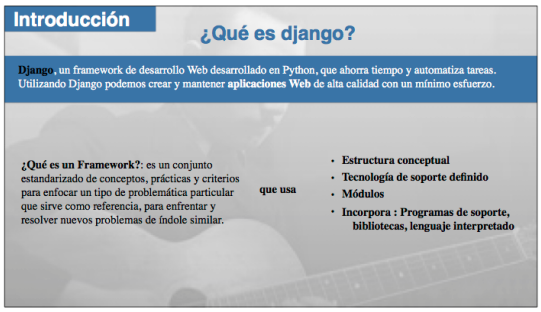
Como una de las conclusiones de más valor que obtenemos, ya no de este trabajo sino de la realización de la carrera, es que la fuerza de voluntad y la perseverancia, son el motor que nos lleva a conseguir nuestros objetivos, y que por muy lejos que parezcan estar al final se acaban alcanzando.

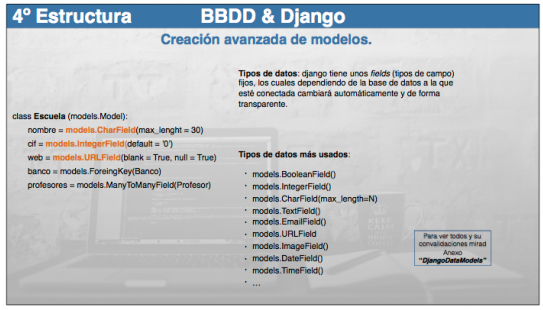
6. ANEXOS

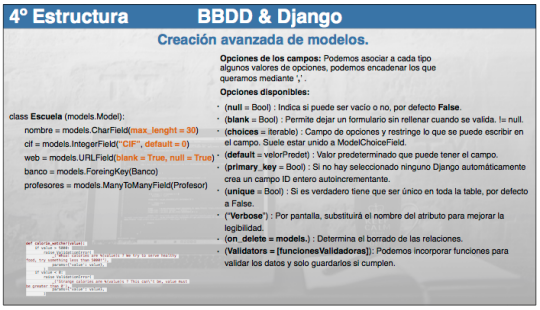
Durante este documento se han hecho referencias a varios cursos realizados con la finalidad de compartir lo aprendido con el resto del equipo, y además poder crear una base de conocimientos para las nuevas incorporaciones. En el siguiente anexo se muestra una representación de ellos:

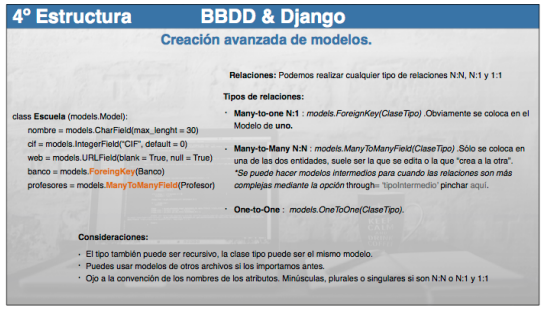
Curso de Django

Curso preparado para el grupo de desarrollo. Es necesario tener amplios conocimientos en programación.



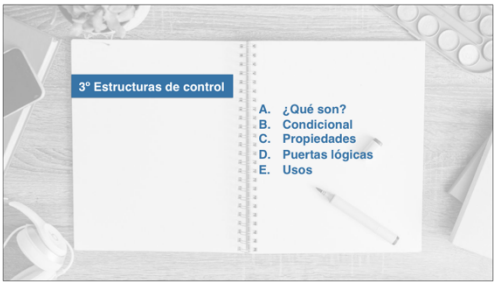


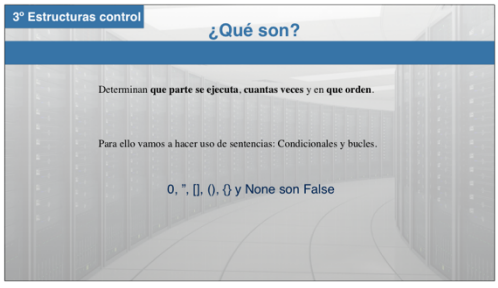


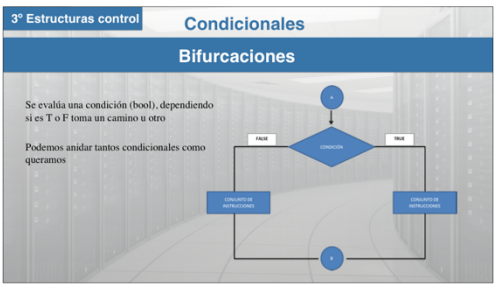


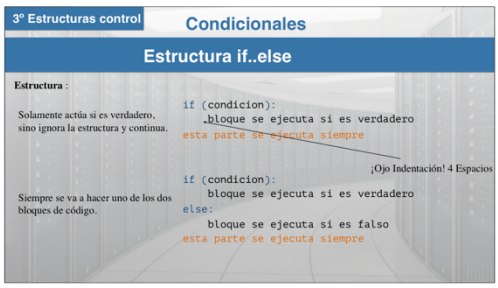
Curso de Python:

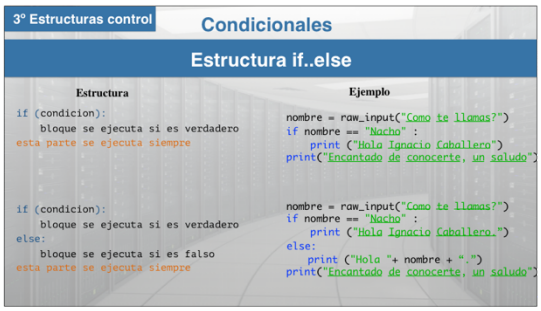
Tiene como finalidad la de introducir a la programación a una persona con pocos conocimientos técnicos, con la finalidad de darle las herramientas para saber comprender fragmentos de código y poder hacer variaciones sobre este y pruebas unitarias.

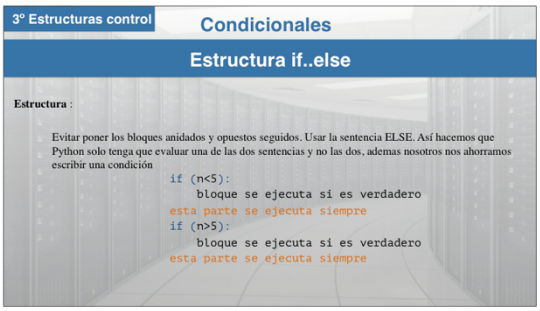


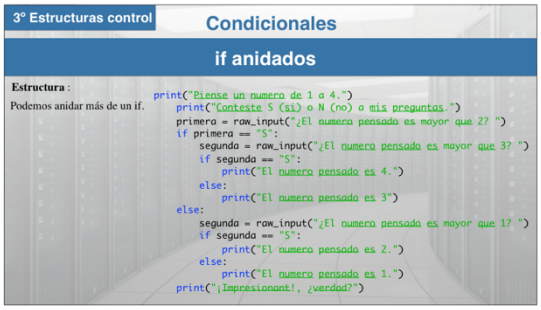


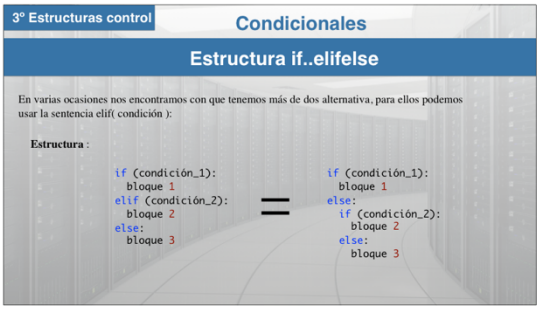






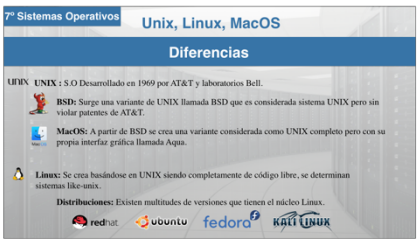






Introducción a la informática:

Siguiendo el principio de que todo trabajador de la STUP tiene que ser conocedor del negocio y de tecnologías TI se ha ideado un curso básico de introducción a la informática para formar a personal no técnico.



5º Software Hardware

Tipos

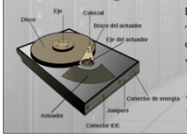
Hardware

Unidad de Disco Duro:

Es el dispositivo de almacenamiento masivo de un ordenador para almacenar archivos digitales.
Típicamente dos tamaños 3,5 ordenadores de escritorio y 2,5 portátiles.

Lo más habitual son los Discos duros magnéticos y los de estados sólido(SSD).

Discos magnéticos:



Tiempo medio de acceso: tiempo medio que tarda la aguja en situarse en la pista y el sector deseado.

Tiempo de lectura/escritura: tiempo medio que tarda el disco en leer o escribir nueva información.

Velocidad de rotación: Es la velocidad a la que gira el disco duro:

5400RPM, 7200RPM…

Tasa de transferencia: velocidad a la que puede transferir la información a la computadora.

5º Software Hardware

Tipos

Hardware

Unidad de Estado Sólido: SSD (Solid-State Drive). No utiliza soporte físico como los discos magnéticos, sino una basada en integrados de puertas NAND llamada Flash parecida a la RAM.

Diferencias: SSD: Mayor velocidad, más duración física, menos duración de escritura, coste más elevado.

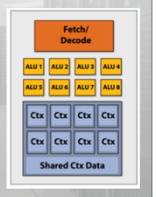
5º Software Hardware

Tipos

Hardware

Unidad de procesamiento gráfico, GPU, Tarjeta Gráfica : Está dedicado al procesamiento de gráficos para liberar de la carga a la CPU. La arquitectura es muy distinta a la CPU, más simple y mucha más cantidad de ALU.

Actualmente se puede encontrar integrada en la propia CPU o de manera dedicada, como un componente más.

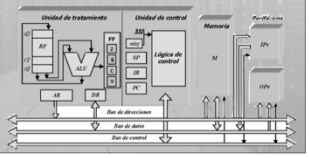


5º Software Hardware

Tipos

Hardware

Bus de datos: Es, un sistema digital, el que transfiere datos entre los distintos componentes de una computadora o entre varias computadoras. Está formado por cables o pistas en un circuito impreso, dispositivos como condensadores además de circuitos integrados.



7. BIBLIOGRAFÍA

Área de Negocio

Roles STUP:


Quién es quién en el organigrama de una startup

La irrupción de startups ha traído consigo una ristra de acrónimos procedentes, en su mayoría, del mundo anglosajón. Recogemos aquí las equivalencias al castellano y las principales responsabilidades que asume cada uno de los cargos.

Principales cargos

http://www.emprendedores.es/gestion/organigrama-startup-quien-es-quien-principales-cargos-funciones

Arturo Batanero Actualidad STUP.


Full-stack Developers, unicornios y otros seres mitológicos

por Arturo Batanero

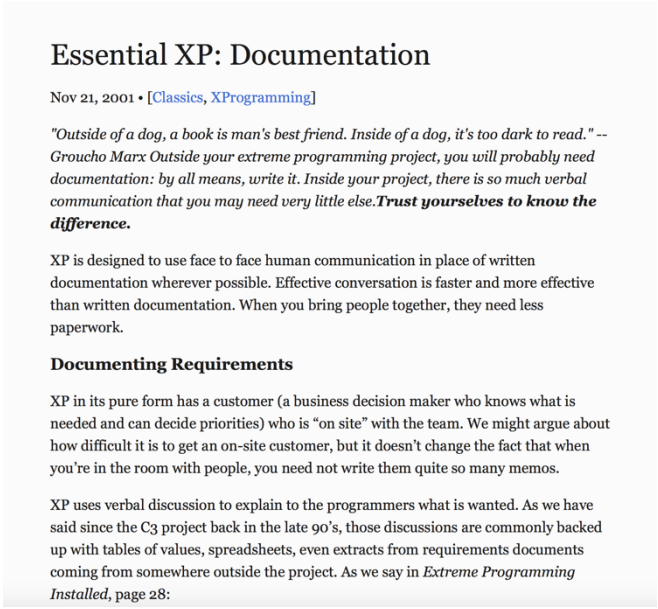
Según la "Developer survey 2016" de StackOverflow, una mayoría de desarrolladores se consideran a sí mismos "full-stack developers", un perfil intrínsecamente senior, muy exigente en conocimientos y con alta demanda en el sector.

Hablamos de un **desarrollador capaz de diseñar e implementar proyectos tanto en el lado servidor como en el lado cliente**, lo que implica conocer los puntos fuertes y débiles de cada tecnología y mantenerse actualizado según estas vayan evolucionando y/o aparezcan otras nuevas.

Con la evolución tecnológica del Web stack, mantenerse actualizado y productivo en el front y en el back-end resulta cada vez más complicado, hasta el punto de cuestionarnos si poder cumplir esas expectativas ya es más un mito que una realidad.

https://www.paradigmadigital.com/dev/full-stack-developers-unicornios-otros-seres-mitologicos/

Documentación XP:



Essential XP: Documentation

Nov 21, 2001 • [Classics, XProgramming]

*"Outside of a dog, a book is man's best friend. Inside of a dog, it's too dark to read." -- Groucho Marx Outside your extreme programming project, you will probably need documentation: by all means, write it. Inside your project, there is so much verbal communication that you may need very little else.**Trust yourselves to know the difference.***

XP is designed to use face to face human communication in place of written documentation wherever possible. Effective conversation is faster and more effective than written documentation. When you bring people together, they need less paperwork.

Documenting Requirements

XP in its pure form has a customer (a business decision maker who knows what is needed and can decide priorities) who is "on site" with the team. We might argue about how difficult it is to get an on-site customer, but it doesn't change the fact that when you're in the room with people, you need not write them quite so many memos.

XP uses verbal discussion to explain to the programmers what is wanted. As we have said since the C3 project back in the late 90's, those discussions are commonly backed up with tables of values, spreadsheets, even extracts from requirements documents coming from somewhere outside the project. As we say in *Extreme Programming Installed*, page 28:

http://ronjeffries.com/xprog/articles/expdocumentationinxp/

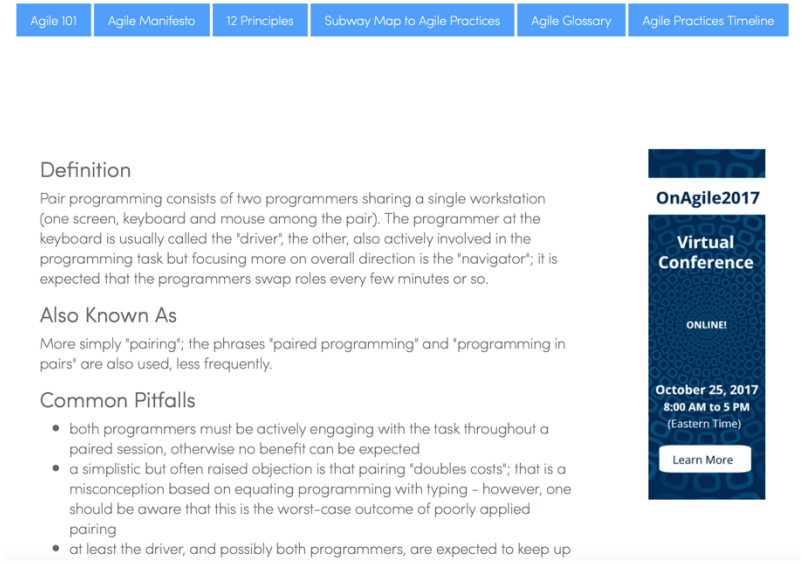
Alberto Mena artículo sobre FS Developer
https://www.contunegocio.es/tecnologia/un-programador-full-stack-para-tu-empresa/

Inicio de proyectos, conceptos:
http://concepto.de/mision-y-vision/#ixzz4r2qXS5TV
http://www.grandespymes.com.ar/2013/09/07/guia-para-elaborar-correctamente-la-vision-y-mision-de-la-empresa/
http://javiermegias.com/blog/2014/04/vision-startup-pollo-sin-cabeza/

Proceso y finalización:
http://www.kewlona.es/2016/10/causas-del-fracaso-de-una-startup/
http://demiumstartups.com/causas-cierre-startups-encaje-producto-mercado/
http://javiermegias.com/blog/2014/09/emociones-miedos-montar-startup/

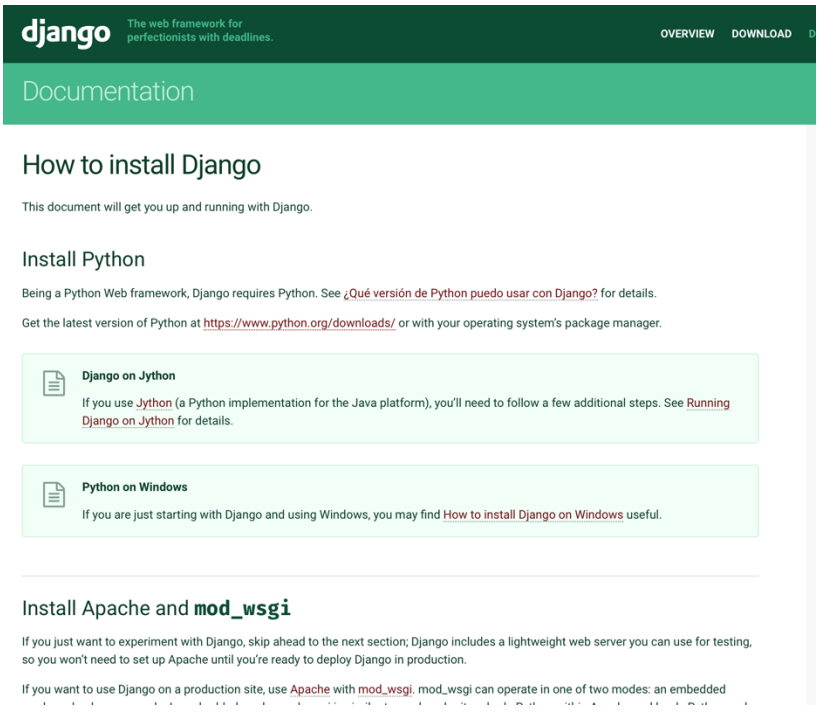
Técnica

Pair programing:



https://www.agilealliance.org/agile101/agile-glossary/

Django documentación y tutorial oficial.



https://docs.djangoproject.com/es/1.11/

Control de Versiones:
http://rogerdudlerhub.io/git-guide/index.es.html
https://git-scm.com/book/es/v1/Fundamentos-de-Git-Guardando-cambios-en-el-repositorio

Programación:

Clean code: Robert Cecil Martin ISBN 9780132350884
https://wiki.python.org/moin/BeginnersGuide
http://www.mclibre.org/consultar/python/
http://restfulwebapis.com

BBDD:
http://www.silicon.es/bases-datos-no-relacionales-nosql-cuando-usarlas-2324948
https://www.bisente.com/documentos/mysql-postgres.html

AGRADECIMIENTOS

El mayor agradecimiento a mi padre, que me ha apoyado, aconsejado y animado durante estos años y además ha depositado una confianza en mí completamente desmerecida. Nunca podría haber terminado esta etapa tardía de mi vida sin su ayuda. Gracias.

A mi amigo y compañero Fran, por compartir tantas frustraciones juntos y por tantas otras que aún nos quedan.

A mi familia y a mi pareja Yuri, por sufrirme, ayudarme y seguir a mi lado durante todo el camino.

A cada uno de los compañeros que he tenido el placer de conocer y a todos los profesores que me han acompañado en la carrera, especialmente a los que continúan enseñando con pasión y con compromiso después de muchos años.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá