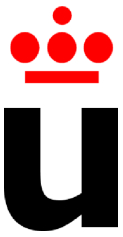


UNIVERSIDAD
REY JUAN CARLOS

INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Curso Acádemico 2018/2019

Trabajo Fin de Grado

PYCARDIO: PAQUETE PARA ANÁLISIS DE SEÑALES CARDÍACAS

Autor : Javier Fernández Morata

Tutor : Dr. Felipe Ortega

Trabajo Fin de Grado

PYCARDIO: PAQUETE PARA ANÁLISIS DE SEÑALES CARDÍACAS

Autor : Javier Fernández Morata

Tutor : Dr. Felipe Ortega

La defensa del presente Proyecto Fin de Carrera se realiza el día de de 20XX, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 20XX

*_Escribimos Dedicatoria_*

Agradecimientos

[AQUÍ VAN LOS AGRADECIMIENTOS]

Resumen

Aquí viene un resumen del proyecto.Ha de constar de tres o cuatro párrafors, donde se presente de manera clara y concisa de qué va el proyecto. Han de quedar respondidas las siguientes preguntas:

- ¿De qué va este proyecto? ¿Cuál es el objetivo principal?
- ¿Cómo se ha realizado? ¿Qué tecnologías están involucradas?
- ¿En qué contexto se ha realizado el Proyecto? Es un proyecto dentro de un marco general?

Índice general

Capítulo 1

Soluciones tecnológicas

Este capítulo constará de dos partes. Una primera donde se muestra una visión general de las tecnologías que usaremos para la creación de la página web, y una segunda donde explicamos brevemente cada tecnología utilizada.

Arquitectura general

En cualquier proyecto de desarrollo web existen dos tipos de tecnologías, BackEnd y FrontEnd. Por tanto, antes de exponer la visión general de nuestra web. ¿Qué significan estos dos términos?:

- **BackEnd:** Tecnologías que trabajan en el lado del servidor,es decir, tecnologías en toma de datos, procesarlos, envío al usuario. Sin estas, las tecnologías del *frontEnd* no tendrían nada que mostrar.

- **FrontEnd:** Tal y como hemos comentado, son las tecnologías que se encargan de mostrar el contenido, es decir, tecnologías centradas en el lado de la aplicación como CSS, HTML5, JavaScript.

Una vez diferenciada estas capas, en nuestro *BackEnd* podemos diferencias tecnología donde se alojara tanto nuestro contenido web como nuestro código fuente en *GitHub*, tecnología que además proporcionará otras como soporte de problemas (ITS), tecnologías sobre cobertura de código.

En la capa *FrontEnd* observamos como todas las tecnologías que colaborarán son aquellas ya conocidascomo HTML5, CSS; donde cuyo objetivo es mostrar contenido y otras nuevas

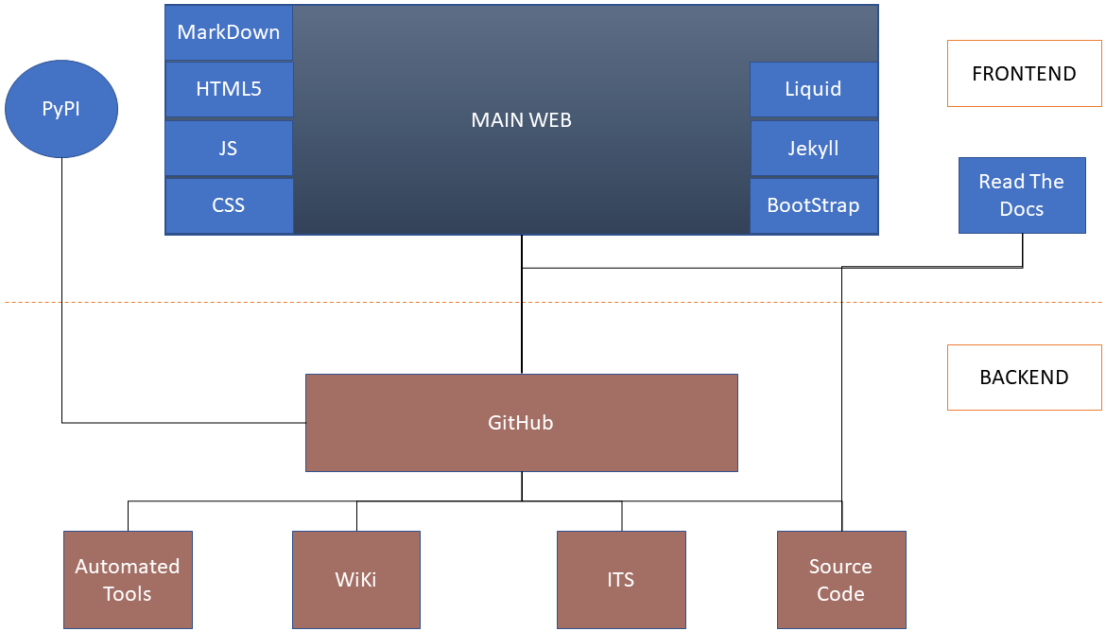


Figura 1.1: Arquitectura General de la Web

como *Jekyll*. Observando la figura 1.1 vemos que otras dos tecnologías nos apoyarán en mostrar el funcionamiento del módulo *PyCardio*, una para mostrar la documentación de los módulos(*ReadTheDocs*) y otra donde también se alojará el código fuente siendo así una aplicación de terceros de Python(PyPI).

BackEnd

Control de versiones con Git. GitHub

Este proyecto al ser un trabajo de *software libre*, significa que el código sufrirá cambios de distintos autores, por lo que es necesario un sistema de control de versiones. Para que un sistema sea de control de versiones debe proporcionar un mecanismo de almacenamiento de los elementos que se desea gestionar, posibilidad de realizar cambios sobre los elementos almacenados y un registro histórico de las acciones realizadas con cada elemento o conjunto

de elementos.

El sistema que se ha escogido debido a nuestra arquitectura de almacenamiento de código y por su seguridad, comodidad y velocidad, es *Git*. Una gran diferencia de *Git* respecto a los demás sistemas de control de versiones es la manera de llevar el registro de cambios sobre sus datos, mientras que la mayoría de sistemas almacenan la información como una lista de cambios, en *Git*, cada vez que se realiza un cambio sobre un archivo *Git* hace una instantánea, y guarda una referencia sobre el archivo sin estos cambios. Para ser mas eficiente, si los archivos no han sido modificados, *Git* no almacena el archivo de nuevo. Esta distinción influye en uno de los mayores beneficios de *Git*, las ramificaciones, tema que trataremos a continuación, viendo como trabajar en un proyecto siguiendo una serie de reglas impuestas por *GitFlow*.

¿Cómo funciona *Git*?

Todo el funcionamiento de este sistema de control de versiones es mayoritariamente local, es decir, no se necesita información de ningún servidor haciendo así que este sistema sea muy rápido respecto a otros que necesitan información remota para sus operaciones.

Git tiene tres estados para los archivos que trabajamos:

- **Committed**: Este estado indica que los archivos que han sido modificados, se han almacenado de manera segura en la base de datos local.
- **Modified**: Indica que los archivos que han sido modificados, no han sido confirmados, es decir, los archivos no se han almacenado de forma segura en la base de datos local.
- **Staged**: Estado que marca archivos modificados para la siguiente confirmación.

Estos tres estados nos lleva a diferenciar las siguientes áreas de trabajo:

- **Working Directory**: Este área es donde obtenemos la copia del proyecto, lista para usarse o modificarse, sin influir en la copia original.
- **Staged Area**: Área en la que se recoge los cambios que van a realizarse sobre los archivos antes de confirmarse.
- **Git Directory**: Estado de los archivos originales del proyecto, incluidos metadatos. Cuando se confirma un archivo, *Git* toma los cambios del área de preparación (Staged Área) y almacena los cambios nuevos en el directorio *Git*.

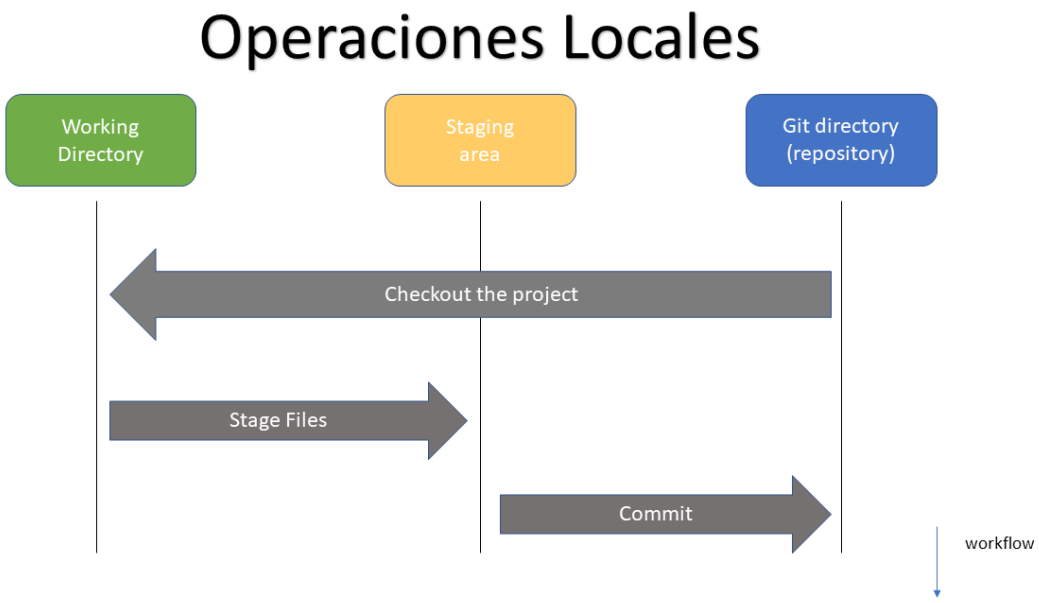


Figura 1.2: Esquema de los estados de un repositorio Git

Por tanto, el flujo que hay que seguir para trabajar con este sistema de control de versiones es el representado en la figura 1.2, donde como vemos tras modificar una serie de archivos en el directorio de trabajo preparamos los archivos añadiéndolos al staged area, se confirman los cambios realizados sobre estos, añadiendo así las instantáneas (copias de los archivos con los cambios confirmados) en el directorio *Git*.

Una vez que hemos visto como trabaja *Git* nos queda responder la pregunta de *¿Qué es GitHub? GitHub* es una plataforma de desarrollo colaborativo de software para alojar proyectos utilizando los mencionados repositorios *Git*, es decir, aloja el código en la nube y brinda herramientas para el trabajo de equipo en el proyecto. Otra de las características de *GitHub* es el poder contribuir en el desarrollo de software de los demás, algo esencial en cualquier proyecto de software libre. *¿Qué herramientas proporciona para el trabajo en equipo? GitHub* proporciona un amplio abanico de funcionalidades , entre ellas, destaca una wiki para el mantenimiento de las versiones de las paǵinas, un sistema de seguimiento de problemas permitiendo al equipo detallar los problemas con lo desarrollado y una herramienta de revisión de código.

Antes de dar por finalizada la sección sobre *Git*, debemos hablar sobre *GitFlow*, extensión de *GitHub* que permite un mantenimiento y estrategia para las ramificicaciones en un proyecto de desarrollo. Esta estrategia y mantenimiento se basa en una serie de reglas,

donde se destacan dos ramas principales: rama *master*, y rama *develop*.La *master* se usa cada vez que tengamos código para producción y la rama *develop* para el código que conformará la nueva versión del proyecto

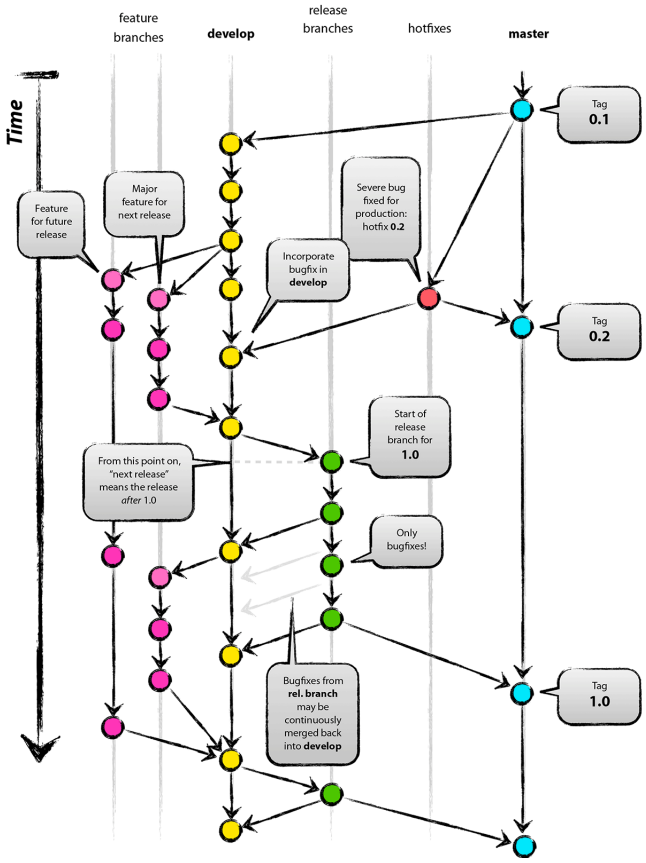


Figura 1.3: WorkFlow impuesto por la estrategia y mantenimiento de *GitFlow*

Por tanto, cada vez que se incorpora código a la rama *master*, tenemos nueva versión. A parte de las ramas principales, observando la figura 1.3 podemos distinguir tres tipos de ramas auxiliares, donde cada una sigue sus propias reglas:

- **Feature:** Se originan a partir de la rama *develop*, y se utilizan para incorporar nuevas características a la aplicación.
- **Release:** Rama que se crea al igual que *feature*, a partir de la rama *develop*. En ellas se depura el código el cual se va a pasar a la producción, es decir, a la rama *master*.
- **Hotfix:** Rama cuyo objetivo es parecido al de la rama *feature*, pero esta se origina a partir de la *master*, y los errores que corregimos en esta rama no están planificados

a diferencia de la rama *feature*, que sí lo están.

De esta manera el uso que daremos de *GitHub* es el de construir un repositorio donde alojaremos el código de nuestro paquete, permitiendo así que cualquiera pueda aportar ideas al proyecto o mejorar el software.

ITS (Issue Tracking System)

GitHub proporciona un sistema de seguimiento de incidencias (en inglés *issue tracking system*), funcionalidad que permite administrar y mantener una lista de incidentes, al tratarse de *software*, una incidencia será una solicitud de modificación, corrección o mejora. Esta herramienta integrada en *GitHub* permite a cualquier usuario (si el repositorio es público) agregar una incidencia mediante un *ticket*. Estos, se crean mediante una interfaz gráfica que permite escribir la incidencia en texto plano y etiquetar esta según de donde provenga (bug,error de sintaxis,etc). Cualquier usuario puede editar estos *tickets*, pudiendo así cerrarlos, añadir etiquetas.
Las ventajas de incorporar un ITS en un proyecto de *software* son las siguientes:

- Tener un único lugar para observar tanto las incidencias creadas por los colaboradores del proyecto como las creadas por los usuarios del *software*.
- Categorizar las incidencias, como ya hemos mencionado, mediante etiquetas.
- Tener un acceso rápido sobre el historial de una incidencia. Además, herramientas como *GitHub* incorpora un cuadro de búsqueda, permitiendo así que una incidencia no se repita.

Así, esta herramienta nos proporcionará corregir aquellos errores o bugs detectados por usuarios de *PyCardio*, donde estos nos informaran mediante la apertura de *tickets*.

Wiki

Aunque en nuestro proyecto para la creación de documentación usaremos otra herramienta, no hay que dejar pasar por alto, la que nos proporciona *GitHub*.
Todo repositorio de *GitHub* viene equipado con una sección para alojar documentación,

llamado wiki.[1].Estas wikis se generan mediante texto de marcado [2] o cualquier otro formato que soporte la creación de wikis. El objetivo por tanto de esta sección es informar de como se ha diseñado, cómo instalar, manifiestos sobre sus principios básicos, objetivos y como usar detalladamente el código al cual pertenece el repositorio, escribir una buena documentación ayuda a otros a usar y a que se extienda dicho proyecto.

Automated tools

GitHub además de las funcionalidades que hemos tratado, este ofrece un gran abanico de APIs para la gestión de proyecto, donde estas son automáticas. *GitHub* ofrece estas de manera categórica, cada categoría se centra en una funcionalidad distinta aunque más de una herramienta ofrece diversos servicios, las más importantes son las siguientes:

Categoría	**Descripción**
Code quality	Automatiza revisión de código con revisión de estilos y coberturas de pruebas .
Code review	Herramientas que aseguran que el código sigue un estandár de calidad adecuado
Continuous Integration	Para construir y probar el código para cada push que se realiza, evitando errores en la producción
Dependency management	Protege y gestiona las dependencias de terceros
Monitoring	Monitoiza el impacto de los cambios en el código, mide rendimiento y errores
Security	Para encontrar, arreglar y prevenir

Cuadro 1.1: Categorías de herramientas para la gestión de proyectos en *GitHub*

Una herramienta que engloba varias de estas funcionalidades es Code Coverage (*CodeCov*). Es una herramienta que nos permite medir la cobertura de código, es decir, mide en que grado el código fuente de un programa ha sido comprobado (testeado). Además esta cobertura es mostrada mediante representaciones visuales y gráficos de evolución. Cuando usamos *CodeCov* usa un porcentaje indicando cuanto código está cubierto. Para el cálculo de esta medida *CodeCov* se distinguen tres terminos que influyen de manera distinta en el cálculo de este:

- **Hit:** Indica que código fuente ha sido ejecutado por pruebas de test.

[1]Wiki alude al nombre que recibe una comunidad virtual, cuyas páginas pueden ser editadas directamente desde el navegador, donde los mismos usuarios crean, modifican, corrigen o eliminan contenidos que, generalmente, comparten

[2]Este lenguaje puede ser *MarkDown*, lenguaje utilizado para la escritura de ficheros de documentación como `README.md`

- **Partial:** Parte del código fuente que no es comprobado (*testeado*) por un bloque de pruebas, es decir, ramificaciones de código (p.ej. sentencia *if-else*).

- **Miss:** Código no comprobado por un bloque de pruebas.

Así el cálculo de cobertura es la relación: $coverage = hits/(sum\ of\ hit + partial + miss)$
Tras este cálculo tal y como hemos mencionado, *CodeCov* ofrece una medida visual de esta cobertura, con la interfaz el desarrollador puede ver que parte de código debe cubrir en sus siguientes bloques de pruebas.
Otro servicio en el que podemos integrar nuestro proyecto y que no debemos pasar por alto es *Travis-CI*, servidor de integración continua en la nube donde podemos vincular nuestra cuenta de GitHub, donde se encarga de realizar las tareas planificadas por cada push & commit que hagamos en alguno de nuestros repositorios de nuestra cuenta. La mayor ventaja que aporta *Travis-CI* es la de probar nuestras librerías y/o aplicaciones con distintas configuraciones sin tener que instalarlas localmente.

FrontEnd

HTML5

HTML (Hiper Text Markup Language) hace referencia al lenguaje de marcado para la elaboración de páginas web. Es un estándar que sirve de referencia para creación de páginas web. Este estándar está a cargo de la W3C, organización dedicada a la estandarización de casi todas las tecnologías ligadas al desarrollo web, sobre todo, referente a la escritura y su interpretación. Es un lenguaje sencillo, basado en utilizar etiquetas , que definen el contenido que se mostrará al usuario, a parte de tener estas etiquetas con su respectivo contenido, tenemos que destacar otro elemento en este lenguaje y son los atributos de etiquetas, que nos permitirán configurar los elementos o ajustar su comportamiento. Los atributos más importantes son *class* e *id*, que se usarán para darle estilo al contenido web mediante CSS y Bootstrap.

HTML5 es la quinta revisión importante de HTML. Lo más importante de esta revisión es que se especifican dos variantes de sintaxis: HTML y XHTML que esta estará servida como lenguaje XML. Las principales novedades que podemos destacar de la nueva revisión son:

- Se mejora el elemento *canvas*
- API que permite almacenar datos en el lado del cliente.
- Mejora de formularios.
- API para la geolocalización.
- Nuevos elementos como `<video>` y `<audio>` que permite la inclusión de contenido multimedia.

CSS

Cascading Style Sheets (hojas de estilo en español) es un lenguaje usado para definir y crear la presentación de un documento estructurado escrito en HTML o XML2. La W3C se encarga de elaborar el estándar que finalmente han optado por seguir todos los navegadores. La sintaxis es muy sencilla , se basa en asignar un valor a una propiedad, estas reglas se pueden establecer en un documento aparte o en la cabecera de un documento HTML tras el elemento `<style>`. La idea fundamental de CSS reside en separar la estructura del documento de la vista del mismo.

Bootstrap

Bootstrap son plantillas echas en base HTML5, CSS3, JQuery y JavaScript que permite hacer una página responsiva, es decir, nos permite que una página web se adapte al tamaño del navegador o en el dispositivo que está siendo mostrada. Este *framework* trae varios elementos con estilos predefinidos fáciles de configurar como botones. La plantilla que utilizamos en el proyecto usa elementos *Bootstrap*.

MarkDown

MarkDown es un lenguaje de marcado que facilita la aplicación de formato a un texto empleando una serie de caracteres de una forma especial. En principio fue diseñado para elaborar sitios web haciendo que elaborarlos sea más rápido y sencillo que si estuviésemos empleando directamente HTML, pero hoy en día se utiliza para cualquier tipo de texto independientemente de cual vaya a ser su destino, ya que el principal objetivo de la sintaxis de *MarkDown* es hacerla lo más legible posible.
Esto hace que *MarkDown* sea realmente dos cosas, por un lado, un lenguaje y por otro, una herramienta de software tal y como dice "Jhon Gruber"(uno de los creadores) en su web oficial [3]. Las ventajas de usar *MarkDown* son las siguientes:

- Escribir webs es más rápido y cómodo. Ventaja que aprovecharemos bastante a la hora de crear actualizar la web del proyecto.
- Es más difícil cometer errores de sintaxis.

[3] `https://daringfireball.net/projects/markdown/`

- Perfecto para usarlo con editores de textos minimalistas.

Liquid

Liquid es un lenguaje de plantillas de código abierto creado por Shopify y escrito en Ruby. Se utiliza para cargar contenido dinámico en los contenidos de las páginas web.

¿Cómo funciona *Liquid?*

El código de *Liquid* se puede categorizar en tres partes:

- **Objetos:** estos objetos indican donde mostrar el contenido en una página. Los objetos y variables estarán denotados por {{ }}

Código 1: Input. Objetos

```
{{ page.title }}
```

Código 2: Output. Objetos

```
{{ Liquid Basics }}
```

- **Etiquetas:** Crean la lógica y el control de flujo para las plantillas, vendrán entre llaves y el caracter de porcentaje ({%%}). El lenguaje utilizado en las etiquetas no produce ningún texto visible, esto significa que puede asignar variables y crear condiciones o bucles sin mostrar nada en la página.

Código 3: Input. Etiquetas

```
{% if user %}
    Hello {{ user.name }}!
{% endif %}
```

Código 4: Output. Etiquetas

```
Hello Adam
```

- **Filtros:** Los filtros cambian la salida de un objeto de *Liquid*. Se utilizan dentro de una salida, es decir, entre la doble llave ({{ }}), y vienen separados por |.

Código 5: Input. Filtros

```
{{ "/my/fancy/url" | append: ".html" }}
```

Código 6: Output. Filtros

```
/my/fancy/url.html
```

Como ya hemos mencionado *Liquid* es usado en diversos *frameworks* de contenido web. En nuestro caso, para *Jekyll*, se crearon sus propios objetos, etiquetas y filtros, estableciendo así *Jekyll Liquid*. Para conocer estos, vienen detalladamente explicados en la página web de documentación sobre sus plantillas [4].

Read the Docs

Read the Docs es un portal web gratuito que permite alojar la documentación de un proyecto (principalmente para proyectos *Python*) haciéndola fácil de encontrar y ofreciendo una opción de búsqueda. Simplifica la documentación de manera que este portal te permite generarla, versionar, y alojarla.
Las dos principales ventajas que proporciona *Read the Docs* son que nunca estaremos desincronizados con nuestro código y que permite alojar distintas versiones de nuestra documentación. Estas dos ventajas se aprovechan del sistema de control de versiones, donde para la primera, cuando hacemos un 'push' de nuestro código se genera automáticamente la documentación, estando así la documentación actualizada y para la segunda ventaja es tan fácil como tener una rama para cada versión nueva de la documentación.
Para ver como generar nuestra propia documentación con *Read The Docs* vamos a seguir los siguientes pasos:

1. Vamos a ver que tecnología usaremos para la generación de documentación.

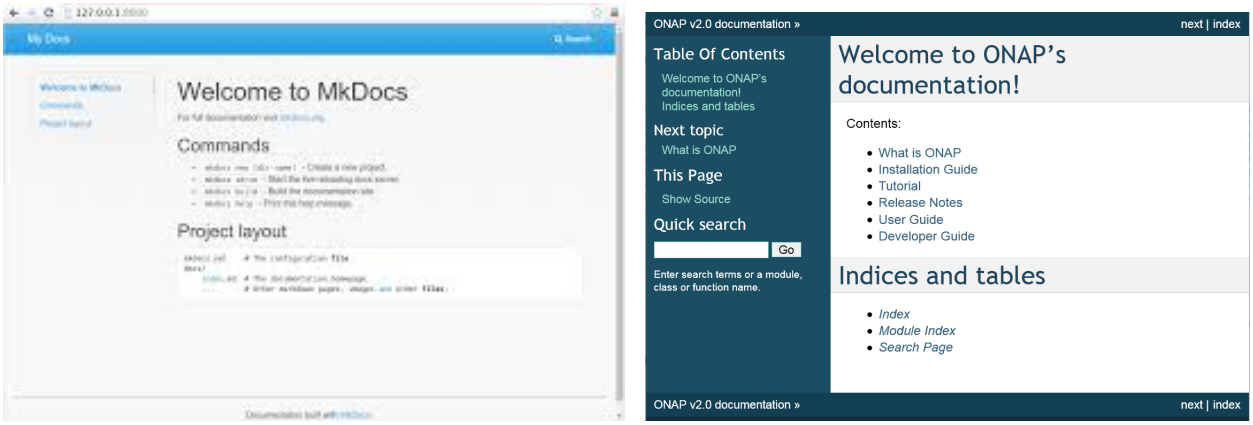
2. Como importar nuestra documentación.

[4] `https://jekyllrb.com/docs/liquid/`

3. Como versionar la documentación.

Generadores de Documentación

Read the Docs usa para generar la documentación *Sphinx* y/o *MkDocs*. *MkDocs* es un generador que se centra en la simplicidad y la rapidez, permite una personalización sencilla mediante extensiones y temas, utiliza *MarkDown* [5]. *Sphinx* en cambio te permite generar páginas web, PDFs, documentos para e-readers, además de la propia documentación; utiliza *ReStructuredText*(otro lenguaje de marcado).



(a) MkDocs Example (b) Sphinx Example

Figura 1.4: Ejemplos de los generadores de la documentación

Importar nuestra Documentación

A la hora de importar nuestra documentación debemos tener una cuenta , tras ello, podemos vincular dicha cuenta a la de *GitHub* permitiendo así que *Read the Docs* muestre una lista de repositorios que se pueden importar a *Read the Docs*. A la hora de importar uno de nuestros repositorio, es hora de configurar el *webhook*[6] de nuestro repositorio, se encargaría de notificar cuando hay un cambio en nuestro código fuente para generar de nuevo la documentación.

Una vez importado el repositorio y configurado el *webhook*, nuestra documentación se generará automáticamente. ¿Como lo genera *Read the Docs* a partir del repositorio?

Lo primero que hace es mirar dos cosas, el tipo de documentación (*Sphinx* o *MkDocs*) y el repositorio. Dependiendo del tipo *Read the Docs* actuará de una manera u otra.

[5] ¿Qué es *MarkDown* 1.3.4

[6] Método de alterar una aplicación o página web, con callbacks personalizados.En este caso son gestionados por *GitHub*, con cada commit o merge, este se lo notifica a *Read the Docs*

- **Sphinx:** Cuando se escoge este tipo, *Read the Docs* mirará el archivo `conf.py`, si este no existe, se creará uno. Tras ello, se buscarán archivos con extensión `.rst` en el directorio `doc` o `docs`, si estos no existen, se buscarán en los directorios restantes del proyecto.

- **MkDocs:** Al optar por este tipo, se lee el archivo `mkdocs.yml`, archivo que debería encontrarse en la raíz del repositorio, de manera que, si no se encuentra se generará uno. En él se indica el directorio donde se encuentran los archivos con extensión `.md`, archivos a partir de los cuales se creará la documentación. Cuando este archivo se generá solo, *Read the Docs* busca automáticamente de forma iterativa en los siguientes directorios `docs, doc, Doc o book`, una vez que se encuentra el archivo con la extensión mencionada, se añade en `mkdocs.yml` el directorio desde donde se creará la documentación.

Como versionar la documentación

Tal y como se mencionó al inicio de este capítulo, *Read the Docs* soporta múltiples versiones de su repositorio. Cuando se importa por primera vez el proyecto se creará la version *latest* apuntando a la rama por defecto de nuestro sistema de control de versiones, en nuestro caso, rama `master` de nuestro repositorio de *GitHub*.

En los casos normales la versión *latest* apuntará al código más actualizado, si en el proyecto se desarrolla en otra rama, se deberá establecer esta como rama por defecto en la interfaz de configuración de *Read the Docs*.

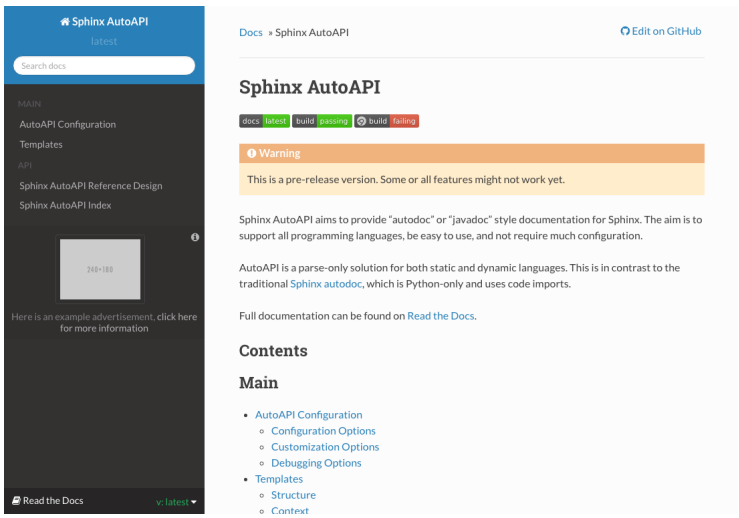


Figura 1.5: Ejemplo de documentación generada por *Read the Docs*

Con todo esto tenemos una manera sencilla de generar la documentación de nuestro proyecto de una manera rápida, sencilla, sincronizada con el desarrollo y automática.

PyPi

PyPI (o Python Package Index) es el repositorio de *software* oficial para aplicaciones de terceros. Los desarrolladores de Python pretenden que sea un catálogo exhaustivo de todos los paquetes de código abierto que hay de Python. Normalmente, si un proyecto escrito en Python no está empaquetado o almacenado en *PyPI* es difícil que este sea encontrado por otros desarrolladores y que este se utilice en sus proyectos, es más, si un proyecto no se encuentra en esta plataforma puede considerarse como una sospecha sustancial de estar mal gestionado, no estar listo para su puesta en libertad o que ha sido abandonado.

Para poder alojar un proyecto en *PyPI* hay que seguir unas ciertos pasos a la hora de crear el módulo python. Estas pasos consisten en añadir los archivos necesarios, una estructura necesaria para crear el módulo y finalmente subirlo al repositorio *PyPI*. Se pueden resumir en lo siguiente:

1. Se crea una estructura de archivos, donde `example_pkg` es mi proyecto de ejemplo: Donde *__init__.py* es utilizado para marcar que es un paquete de Python. Es decir,

```
example_pkg
├── example_pkg
│   ├── __init__.py
│   └── module.py
├── README.md
├── LICENSE.md
└── setup.py
```

Figura 1.6: Estructura de proyecto para PyPI

que puedo realizar un `import module.py`.

2. Se crea el archivo `setup.py`. Este archivo es el script de compilación para *setuptools*.

3. Creación de `README.md`, dando una breve descripción del proyecto y en qué consiste.

4. Creación de una Licencia, ya que es importante indicar a los usuarios que instalan el módulo bajo que condiciones pueden usar el paquete.

5. El siguiente paso es generar paquetes de distribución para el paquete.

6. El último paso por tanto es la carga de los archivos de distribución.

Al ser nuestro proyecto un módulo de código libre, deberemos seguir estos pasos para tener *PyCardio* en el repositorio oficial.

GitHub Pages y Jekyll

GitHub Pages y *Jekyll* se usan conjuntamente para generar sitios web estáticos , donde *Jekyll* se encarga de generar los sitios web y *GitHub Pages* de servicio de alojamiento. Ambos servicios están integrados. Para explicar como trabajan ambos servicios, vamos a tratarlos primero por separado.

GitHub Pages

Es un servicio de alojamiento web que ofrece *GitHub*, se usa para alojar sitios web estáticos usando directamente un repositorio *Git*. Para aclarar la principal ventaja de *GitHub Pages* frente a otros servidores vamos a definir que es una pagina web estática y que se diferencia respecto a una página web dinámica.

¿Qué es una página web estática?

Una página web estática es aquel documento web que muestra el mismo contenido para todos los usuarios, es decir, que no es personalizable o no hay elección de interactuar con ella para ordenar, ocultar o modificar contenido (páginas dinámicas). Una de las ventajas principales de este tipo de páginas es lo económico que resulta crearlas, sin tener que usar ningún tipo de programación especial. Estas páginas son útiles para el objetivo de simplemente mostrar contenido. Hay que destacar que la mayor desventaja de generar sitios web estáticos son su actualización haciendo de esta algo tedioso.

Una vez aclarado, la ventaja principal, a parte de todas las proporcionadas por ser un repositorio *Git*, es que al no necesitar base de datos para servir el contenido, solo se requiere un servidor web que vaya sirviendo el contenido que se solicita.

Jekyll

Jekyll es un generador de sitios web estáticos, en vez de trabajar con base de datos, *Jekyll* trabaja con texto plano (MarkDown o Textile) y lenguaje de plantilla (Liquid) generando la página web. La principal ventaja que podemos inferir de *Jekyll* es que no necesitamos saber HTML, para generar nuestro contenido web, ventaja que contra resta sobre lo tedioso que es tener actualizado un sitio web estático.

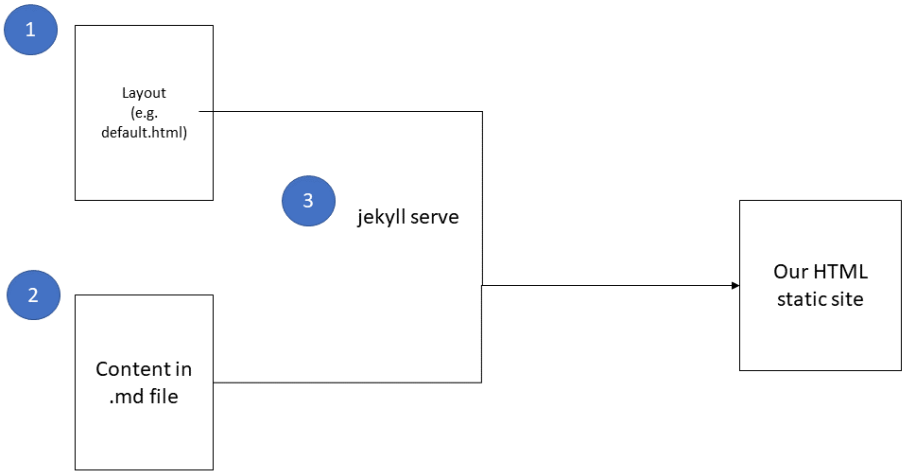


Figura 1.7: Proceso para web básica con Jekyll

Tal y como observamos la figura 1.7 el proceso de trabajo para construir webs con *Jekyll* es:

1. Creamos un layout o diseño de nuestra página HTML que incluirá código del lenguaje de programación *Liquid*.

2. Escribimos nuestro contenido en texto plano (*MarkDown* o *Textile*) donde tendremos un preámbulo indicando el diseño a utilizar para ese post, así como otras características para nuestro contenido HTML.

3. Ejecutamos *Jekyll* generando nuestro sitio web estático.

Para entender como funcionan estos tres pasos vamos a tratar cada uno por separado dando un breve resumen apoyado en ejemplos sencillos. Antes de empezar hay que destacar que cuando ejecutamos *Jekyll* sobre nuestro directorio , cuyo fin será el que contenga los archivos de nuestro sitio web estático, establece una estructura creando en nuestra raíz del proyecto una serie de directorios, cuya función son las siguientes:

_includes: En el irán las porciones de código que se repetirán para todos el sitio web (p.ej. footer), se incluirán en nuestro layouts con includes.

_layouts: Directorio que contendrá todas las plantillas creadas para el sitio web.

_posts: Lugar donde irán todos los artículos que iremos escribiendo en texto plano.

_drafts: En esta carpeta guardaremos los borradores de nuestros artículos, es decir, se utiliza para la prueba de código.

_site: Está se genera no al crear el proyecto *Jekyll*, si no al lanzar el servidor, está contendrá todos los HTML montados así como un *index.html* (archivo principal de un sitio web).

_config.yml: Esto no es un directorio, es una archivo, pero también es generado al crear un proyecto *Jekyll* donde en él encontraremos datos de configuración así como indicaciones de qué directorio obtener los datos.

Una vez que tenemos claro a que corresponde cada directorio en nuestro proyecto , tratamos a continucación lo mencionando anteriormente, los pasos de un proyecto *Jekyll*.

Diseño de Layout

Layout es una plantilla donde *Jekyll* incrustará el contenido escrito en texto plano. Tal y como hemos comentado antes, estas plantillas deberán almacenarse en el directorio `_layout`. En ellas haremos uso de nuestro código guardado en `_include`, un ejemplo como el de incluir un footer al final de mi plantilla `default.html` es el siguiente:

Código 1: default.html

```
<!-- default.html -->
<footer>
    {{include footer.html}}
</footer>
```

Código 2: footer.html

```
<!-- footer.html -->
<p>Posted by: Hege Refsnes</p>
<p>Contact information: <a href="mailto:someone@example.com">
  someone@example.com</a>.</p>
```

La herramienta que mas optimiza el trabajo a la hora de elaborar estas plantillas de la que hace uso *Jekyll* es *Liquid*. Este lenguaje que trataremos en el siguiente punto hace uso de variables, donde estas son etiquetas, objetos y filtros. Además cualquier archivo *Jekyll* es accesible mediante variables vía *Liquid*. Estas variables pueden ser el título del sitio web (establecido en `_config.yml`). La variable más importante es {{`content`}}, ya que en ella es donde *Jekyll* incrustará el contenido escrito en texto plano. Un ejemplo sencillo de un layout sería el siguiente:

Código 3: Layout Básico

```
<!doctype html>
<html lang="en">
  <head>
    {{include head.html}}
  </head>
  <body>
    <nav>
        {{include nav.html}}
    </nav>
    <h1>{{ page.title }}</h1>
```

```
11      <section>
12        {{ content }}
13      </section>
14      <footer>
15          {{ include footer.html }}
16      </footer>
17    </body>
18  </html>
```

Escribiendo contenido

Una vez que ya tenemos nuestro layout, es hora de crear el contenido con el que poder construir nuestro sitio web. Como ya hemos mencionado, no es necesario tener conocimientos de HTML, solo escribir contenido en texto plano.

A la hora de escribirlo la parte más importante es la denominada como *Front Matter*, ya que cualquier documento que lo contenga, *Jekyll* lo tratará como un fichero especial. Esté debe ser el primer elemento de nuestro fichero, y debe seguir la forma de lenguaje *YAML* entre tres guiones.

```
---
layout: post
title: Blogging Like a Hacker
---
```

Entre los guiones se pueden definir variables personalizadas o variables ya definidas que son accesibles en el contenido usando etiquetas *Liquid* y lo serán tanto en los layouts como en el contenido que estamos creando. La más importante de las variables globales predefinidas que existen es la denominada como **layout**, con ella indicamos que plantilla vamos a utilizar para el contenido que se va a crear tras el *Front Matter*

No podemos finalizar esta sección sin mencionar la regla más importante a la hora de crear posts en nuestra sitio web estático. El fichero que creamos con nuestro texto plano debe alojarse en el directorio ya comentado como , _**layout**, el nombre de éste debe seguir una regla de formato, que vendrá dada por: `YEAR-MONTH-DAY-title.MARKUP`, ya que para organizar el contenido de nuestra web *Jekyll* lo organizará por un sistema de directorios con orden cronólogico.

Generando la web

Creada la plantilla y creado el contenido solo queda generar el sitio web, para ello basta con ejecutar *Jekyll*.

Una vez que creamos un directorio como nuestro proyecto *Jekyll*, cada vez que queramos generar el contenido HTML, se ejecuta el comando con la opción de lanzar el sitio localmente o solo construirlo teniendo así nuestro sitio web estático.

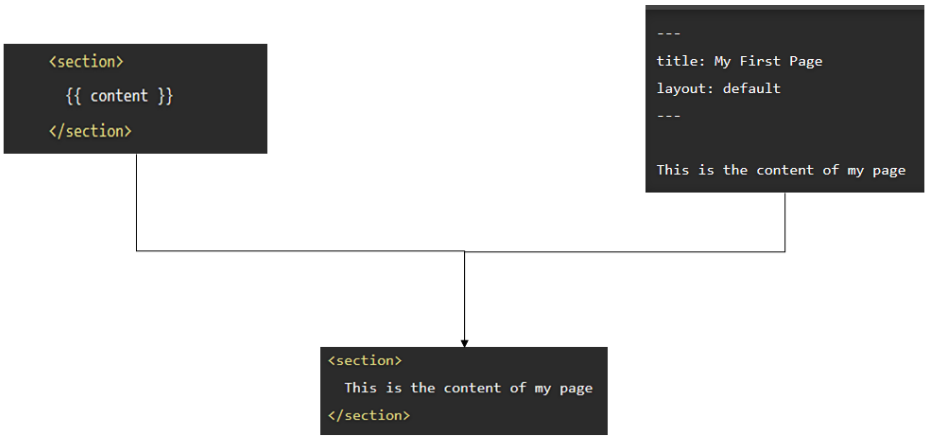


Figura 1.8: Proceso de Contenido en Jekyll

```
jekyll new #Crea un proyecto Jekyll
jekyll serve #Construye el sitio web y lo sirve localmente en
            #"localhost:1400"
jekyll build #Genera el contenido web
```

Jekyll en repositorio GitHub

Ya tenemos nuestro servidor mediante el servicio *GitHub Pages*, y para el contenido *Jekyll*, pero ¿Cómo actualizamos nuestra web?¿Cada vez que creamos contenido es necesario actualizar repositorio y ejecutar *Jekyll* por separado?.

GitHub funciona con Jekyll, lo que quiere decir que cuando creamos contenido y este lo comprometemos (committed), *Jekyll* se encargaŕa de generar el contenido tal y como hemos mencionado.