

Università degli Studi dell'Insubria

Facoltà di Scienze MM.FF.NN.

Corso di Laurea Triennale in Informatica



Ristrutturazione di un sistema di gestione dei contenuti

Relatore: Dott. Ignazio Gallo

Tesi di Laurea di
Matteo Franceschi De Marchi
Matricola 725083

Anno Accademico 2016-2017

“What a liberation to realize that the “voice in my head” is not who I am. Who am I then? The one who sees that.” - Eckhart Tolle

Indice

1	Introduzione	1
1.1	Il Kirivo Network	1
1.2	Architettura del Kirivo Network	2
1.3	I team del Kirivo Network	4
1.4	Gestione delle homepage	4
1.5	La gemma cmsdealer	6
1.6	Obbiettivo	9
1.7	Obbiettivo secondario	9
2	Fase di formazione aziendale	11
2.1	Ruby e le sue librerie	11
2.2	Wordpress	12
2.3	Sviluppo agile	12
2.3.1	Test Driven Development	13
2.3.2	Pair Programming	13
2.3.3	Studio teorico	13
3	Preparazione dell'ambiente di sviluppo per Wordpress	15
3.1	Architettura per lo sviluppo	15
3.2	Procedura di deploy	16
3.3	Impostazione della macchine di sviluppo	16
3.4	Creazione dello script di deploy	17
4	Integrazione Page Builder	21
4.1	Page Builder by Site Origin	21
4.2	Widget	23
4.3	Inizializzazione	25
4.4	Implementazione Widget	25
5	API rendered prices	29
5.1	Creazione chiamata in Wordpress	31

5.2	Creazione risposta in Kiruby	32
5.3	Creazione chiamata in cmsdealer	36
6	Widgets di Origini	39
6.1	Origini - Slider prodotti	40
6.2	Origini - Banda tuttoschermo	41
6.3	Origini - Speciale	42
7	Widgets di Kirivo	43
7.1	Kirivo - Immagini Speciali	44
7.2	Kirivo - Immagini Newsletter	45
7.3	Kirivo - Prodotti in evidenza	46
7.4	Kirivo - Slider prodotti	47
8	Conclusioni	49
8.1	Sviluppi futuri	49

Elenco delle figure

1.1	Schema dell'architettura del Kirivo Network.	3
1.2	L'interfaccia visualizzata dai content managers per modificare la pagina.	5
1.3	Risultato renderizzato della componente compilata in figura 1.2.	5
1.4	Funzionamento della richiesta della homepage.	6
1.5	L'interfaccia visualizzata dai content managers per modificare un listing di prodotti.	7
1.6	Risultato renderizzato della componente compilata in figura 1.5.	8
1.7	Come dovrebbe essere editata la componente in figura 1.3.	9
1.8	Come dovrebbe essere editata la componente in figura 1.6.	10
3.1	Contenuto della cartella "wordpress".	17
4.1	L'interfaccia di editing di <i>Page Builder</i> per la homepage di Kirivo permette la suddivisione in componenti e la loro copia e modifica.	22
4.2	Il form che viene visualizzato per modificare i dati del Widget.	24
4.3	Porzione di sorgente di una pagina che utilizza il widget compilato come in 4.2.	24
4.4	Contenuto renderizzato dal widget <i>Origni - Speciale</i>	26
5.1	Nella modalità di <i>live editing</i> i prodotti non vengono visualizzati.	30
5.2	Con le renderedAPI i prodotti vengono visualizzati.	35
6.1	Contenuto mostrato dal widget "Origni - Slider prodotti".	40
6.2	Contenuto mostrato dal widget "Origni - Banda tuttoschermo".	41
6.3	Contenuto mostrato dal widget "Origni - Speciale".	42
7.1	Contenuto mostrato dal widget "Kirivo - Immagini Speciali".	44
7.2	Contenuto mostrato dal widget "Kirivo - Immagini Newsletter".	45
7.3	Contenuto mostrato dal widget "Kirivo - Prodotti in evidenza".	46
7.4	Contenuto mostrato dal widget "Kirivo - Slider prodotti".	47

1

Introduzione

Il progetto in discussione è stato sviluppato durante il periodo di alto apprendistato presso 7Pixel, dove sono stato assegnato al team Iguana, team che si occupa della gestione, principalmente front-end, dei siti del *Kirivo Network*

1.1 Il Kirivo Network

Il Kirivo Network (KN) è attualmente composto da due siti www.kirivo.it e www.origini.it:

www.kirivo.it è un negozio online che vende prodotti di tutte le categorie. Il marketplace dispone di un offerta di oltre 800.000 articoli in tutte le categorie tra cui elettrodomestici, prodotti per la casa, smartphone e TV, giocattoli, moda e altri.

www.kirivo.it è il marketplace ufficiale di www.trovaprezzi.it, il principale motore di ricerca italiano per la comparazione di prezzi.

www.origini.it è una divisione verticale di Kirivo. Il sito è specializzato nella vendita di vini e offre un ampia offerta di prodotti divisi per cantine e regioni. Il sito è online da Novembre 2016.

I siti del Kirivo Network fanno utilizzo di servizi di Back-End comuni che permettono di effettuare acquisti nei due siti utilizzando un unico account ed un unico carrello.

Con buone probabilità verranno aggiunti in futuro nuovi siti verticali per alcune categorie di Kirivo.

1.2 Architettura del Kirivo Network

Per l'erogazione dei siti del Kirivo network vengono usati diversi server:

- **Hybris:** una piattaforma Enterprise di e-commerce scritta in Java che offre una soluzione all-in-one per i siti di e-commerce comprendendo servizi quali la gestione del catalogo dei prodotti, degli utenti e la gestione sicura dei pagamenti. La scelta di utilizzare una piattaforma di e-commerce a pagamento è stata fatta principalmente per velocizzare i tempi di sviluppo in fase iniziale.

Hybris utilizza una database relazionale Postgres e il suo catalogo viene indicizzato dal motore di ricerca SolR.

- **SolR:** un motore di ricerca scritto in Java che permette di indicizzare i prodotti presenti a catalogo per una accesso più rapido.

Permette inoltre di filtrare in modo efficiente i prodotti presenti a catalogo ottimizzando le ricerche per categorie o caratteristiche del prodotto.

- **Kiruby:** un web-server Ruby che eroga le pagine web dei siti, si interfaccia con Hybris e Solr utilizzando i loro servizi di backend.

- **Wordpress:** usato per la creazione di pagine di contenuto che vengono incluse da Kiruby.

Il server di Wordpress si trova nella LAN aziendale ed è accessibile solamente dall'server Kiruby, la sua presenza è nascosta agli utenti finali.

- **Redis:** un database noSql che, salvando tutto il suo contenuto in RAM, garantisce alte prestazioni.

Viene usato da Kiruby come cache di contenuti, specialmente per le richieste di Kiruby a Wordpress.

- **Nginx:** un Reverse proxy usato per redirigere le chiamate fatte ai domini `www.kirivo.it` e `www.origini.it` ai server opportuni.



Figura 1.1: Schema dell'architettura del Kirivo Network.

1.3 I team del Kirivo Network

Lo sviluppo e la manutenzione dei siti del Kirivo Network viene effettuato da più team e questi sono:

- **Team Iguana:** si occupa dello sviluppo di Kiruby.
- **Team Nimbus:** si occupa dello sviluppo di Hybris SolR e Kitty.
- **Content Managers:** si occupano della comunicazione con i vendori, della gestione dei prodotti a catalogo e della creazione di pagine di speciali, lavorano interfacciandosi con Hybris e Wordpress.
- **Grafica:** lavora o direttamente su Kiruby o da al team Iguana grafiche HTML che vengono poi rese dinamiche ed integrate con i vari servizi.

1.4 Gestione delle homepage

Le homepage di Origini e Kirivo sono le pagine che, nei rispettivi siti, possono cambiare contenuto più frequentemente. Inoltre scegliere quali prodotti, quali offerte e quali contenuti vanno inseriti in homepage non è compito dei programmatori ma dei *content managers*, quindi si rivela importante dare la possibilità ai *content* di fare modifiche, come cambiare un prodotto da mettere tra quelli in evidenza in homepage o il testo di un pannello con la cantina del mese, senza dover passare dai programmatori per fare modifiche.

Per dare più libertà ai *content*, il contenuto della homepage, ovvero tutto quello che non è header e footer, non si trova nel server Kiruby ma in pagine Wordpress che i *content* possono direttamente modificare accedendo alla sezione *admin* di Wordpress.



The screenshot shows a WYSIWYG editor interface with a toolbar at the top containing buttons for 'Add Media' and 'Add Media From Azure', and various rich text tools like bold, italic, link, b-quote, del, ins, img, ul, ol, li, code, more, and close tags. Below the toolbar is a block of raw HTML code:

```
'<section id="hp_editorial_contents" class="bg_white">
<a href="/vino/cantina/azienda-vinicola-venturi">
<picture class="tile_article_image">

</picture>
</a>
<h5 class="subheader">La cantina del mese</h5>
<h3 class="dotted_bordered">Venturi</h3>
<h4 class="subheader"></h4>
<p class="hp_article_box_abstract">Obiettivo dell'azienda è la produzione di una limitatissima quantità di bottiglie dalle caratteristiche uniche per corpo, eleganza e profumi.</p>
<a href="/vino/cantina/azienda-vinicola-venturi" class="hollow secondary button content_cta">Scopri</a>
</section>      <section id="about_us" class="hp_box bg_verylightgrey">
```

Figura 1.2: L’interfaccia visualizzata dai content managers per modificare la pagina.



Figura 1.3: Risultato renderizzato della componente compilata in figura 1.2.



Figura 1.4: Funzionamento della richiesta della homepage.

Il server Kiruby, quando deve ricevere una richiesta per la homepage, chiede a Wordpress la sua pagina della home, ne estrae il contenuto e lo renderizza nell'HTML che restituisce. Il contenuto di Wordpress viene incluso nell'HTML tra Header e Footer, il cui codice invece resta in Kiruby in comune a tutte le altre pagine.

1.5 La gemma `cmsdealer`

Per visualizzare contenuti dinamici, come ad esempio un Box di 4 vini viene utilizzata una gemma (così vengono chiamate in Ruby le librerie) chiamata `cmsdealer`.

La gemma è necessaria per il fatto che le informazioni dei prodotti, come nome del venditore, prezzo, spese di spedizione ecc... non sono accessibili a Wordpress, che viene utilizzato come server per la gestione e la modifica di contenuti editoriali, ma sono accessibili a Kiruby che ha al suo interno diversi meccanismi per accedere alle informazioni dei prodotti andandosi ad interfacciare con Hybris e SolR.

La gemma `cmsdealer` usata dal server Kiruby scansiona la pagina di Wordpress da includere, se incontra un tag di nome `dynamic` ne legge l'attributo `type` e in base al valore di questo seleziona il corrispondente template, legge l'ID dei prodotti e sostituisce al tag `dynamic` l'HTML del box con i valori dei prodotti selezionati

Esempio: se processando la pagina HTML di Wordpress Kiruby trova

```
<dynamic type="KirivoListBox" ids="3422,2345,2872,2209" />
```



Figura 1.5: L’interfaccia visualizzata dai content managers per modificare un listing di prodotti.

allora verrà cercato il template di *kirivolistingbox.html.erb* e verrà popolato coi valori dei prodotti con gli identificativi specificati nell’attributo *ids*.

SAMSUNG GALAXY S8

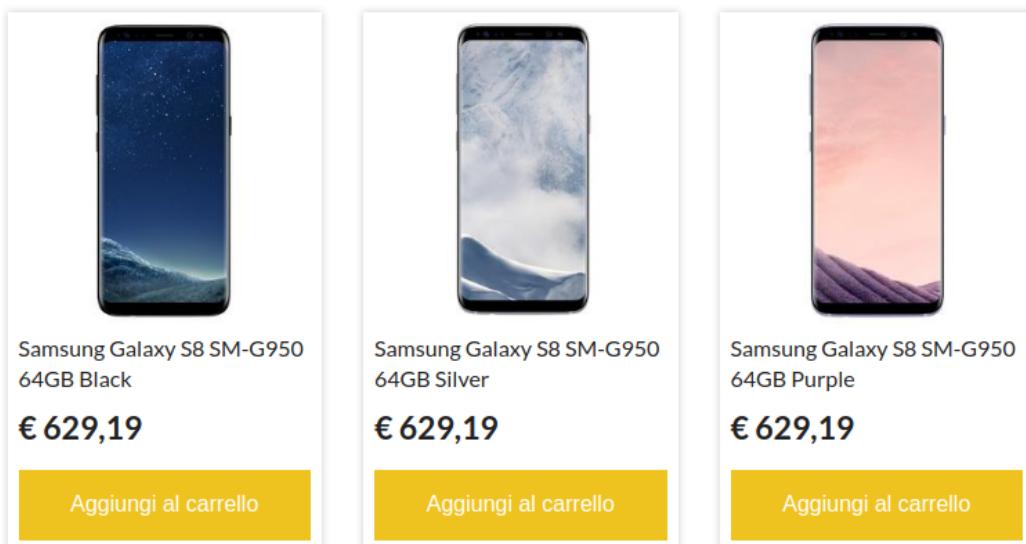


Figura 1.6: Risultato renderizzato della componente compilata in figura 1.5.

Origini - Speciale	
Sottotitolo:	La cantina del mese
Titolo:	Venturi
Descrizione:	Obiettivo dell'azienda è la produzione di una limitatissima quantità di bottiglie dalle caratteristiche uniche per corpo, eleganza e profumi.
Link:	/vino/cantina/azienda-vinicola-venturi
Image:	https://kn.kirivo.it/res/2017/02/viniventuri4.jpg
Upload or select image	

Figura 1.7: Come dovrebbe essere editata la componente in figura 1.3.

1.6 Obbiettivo

L'obbiettivo del progetto è quello di rendere l'edit delle pagine da parte dei content molto più semplice e flessibile, facendo in modo che i contenuti delle homepage possano essere editati visualmente e senza modificare direttamente il codice HTML (vedi figure 1.7 e 1.8).

Per farlo i content dovranno interagire con un interfaccia grafica web che permette la modifica delle informazioni necessarie e di poter spostare e copiare componenti della home con un click o con un drag and drop.

1.7 Obbiettivo secondario

Obbiettivo secondario del progetto è di rendere il sito più manutenibile.

La criticità del sistema è che contiene codice duplicato di componenti HTML nella gemma CmsDealer, in Kiruby e nei widget di Wordpress.

Per risolvere il problema sono state create le *RenderdPricesAPI* che restituiscono frammenti di HTML renderizzato per le varie componenti della home che contengono prodotti.

In questo modo il codice del template resta solamente in Kiruby e esponendo le API questo viene usato sia dai Widget di Wordpress sia dalla gemma CmsDealer.

Kirivo - Griglia prodotti

Titolo:

SAMSUNG GALAXY S8

Ids prodotti(min 4):

8806088722863,8806088815800,8806088722856

Figura 1.8: Come dovrebbe essere editata la componente in figura 1.6.

2

Fase di formazione aziendale

Durante i primi due mesi di tirocinio, la maggior parte del mio tempo è stato speso in formazione, l'obiettivo dello studio era di fornirmi le conoscenze adeguate per lavorare nel progetto, in particolare lo studio di Wordpress, e per integrarmi al Team Iguana per lo sviluppo e la manutenzione di Kiruby.

Il mio studio può essere diviso in tre argomenti principali:

- Ruby e le sue librerie
- Wordpress
- Sviluppo agile

Lo studio dei primi due è stato prevalentemente pratico, mentre lo studio e l'applicazione delle metodologie agili è stato anche in buona parte teorico.

2.1 Ruby e le sue librerie

Il primo argomento di studio una volta entrato in azienda è stato il linguaggio *Ruby*, essendo questo il linguaggio del web server *Kiruby* la quasi totalità del lavoro del mio Team viene svolto con questo linguaggio.

Per lo studio di Ruby mi sono affidato esclusivamente a risorse disponibili online, consigliate dal Team, in particolare il sito rubylearning.com[1].

In contemporanea a questo ho messo subito in pratica gli argomenti imparati risolvendo i *Koans*[2] di Ruby, ovvero una serie di test divisi per argomenti, come ad

esempio String, Symbols, Hashes, Regular Expression, che inizialmente falliscono e che vanno sistemati in modo da passare correttamente. Quindi una volta studiato un certo argomento veniva risolto il suo rispettivo *Koan*.

Una volta apprese le basi del linguaggio, per prendere maggiore confidenza ho iniziato a sviluppare vari Kata agili.

I Kata agili sono dei programmi da sviluppare iterativamente, ovvero prima vengono fornite delle specifiche, una volta implementate se ne aggiungono di nuove o si modificano le precedenti.

Questo tipo di esercizi serve a far abituare lo sviluppatore a scrivere codice in modo manutenibile, ovvero in modo che l'aggiunta o il cambiamento di una qualche funzionalità richieda il minimo sforzo grazie alla qualità del codice scritto.

Oltre alle basi del linguaggio ed ai Kata agili ho studiato anche l'utilizzo di alcune *gemme* (così vengono chiamate le librerie Ruby) che vengono usate frequentemente in Kiruby tra cui Test-Unit come framework di testing, librerie per l'integrazione con SolR e Redis e librerie per lo scambio di dati JSON attraverso HTTP come *HTTPParty*.

2.2 Wordpress

Parallelamente allo studio di Ruby mi sono dedicato allo studio di Wordpress.

Wordpress è il Content Management System più diffuso al mondo, è un applicazione web scritta in PHP che deve essere servita da un web-server, solitamente Apache, nel nostro caso Nginx.

Nella fase di studio mi sono concentrato nell'analizzare e studiare come è strutturata l'applicazione e dove un programmatore deve mettere mano per fare modifiche e personalizzare la propria installazione.

Oltre allo studio di Wordpress ho fatto un approfondimento del linguaggio PHP del quale avevo qualche nozione ed esperienza di utilizzo, non troppo approfondita, già prima del mio arrivo.

2.3 Sviluppo agile

In 7Pixel si sviluppa utilizzando metodologie agili: il software viene sviluppato iterativamente, nuove feature e aggiornamenti vengono pubblicati quotidianamente. Nel periodo di studio le varie esercitazioni sono state effettuate utilizzando due metodologie tipiche dello sviluppo agile.

- Test Driven Development
- Pair Programming

2.3.1 Test Driven Development

Il TDD Test Driven Development è una tecnica utilizzata ovunque in azienda per lo sviluppo, nel TDD prima di aggiungere una qualsiasi nuova funzionalità si scrive un test che passerebbe solo se quella funzionalità fosse implementata correttamente.

Una volta scritto il test che fallisce, si cerca nel modo più veloce e semplice possibile di fare passare il test.

Una volta implementato il codice per far passare il test si fa del refactoring per rendere il codice più leggibile e soprattutto manutenibile, cercando di eliminare il più possibile la presenza di codice duplicato.

2.3.2 Pair Programming

Pair Programming significa sviluppo in coppia, ovvero due membri del Team lavorano contemporaneamente allo stesso codice sulla stessa macchina, utilizzando due schermi speculari, due tastiere e due mouse

Il Pair Programming si rivela molto efficace, perchè si riescono ad evitare molti errori di distrazione che possono costare caro in termini di tempo e soprattutto si ha molto spesso la possibilità di confrontarsi con punti di vista diversi che possono portare ad un analisi più approfondita del problema e a soluzioni migliori.

Dal mio punto di vista di apprendista il Pair Programming ha portato inoltre il vantaggio di poter lavorare con gente più esperta e quindi, durante il lavoro, di imparare tecniche, metodologie e *best practices* per lo sviluppo.

2.3.3 Studio teorico

Oltre alla messa in pratica delle tecniche di sviluppo agile sono stati anche effettuati studi teorici su le come sviluppare codice pulito e manutenibile.

Gli argomenti principali sono stati:

- Studio dei principali design pattern, tra cui i pattern GRASP e SOLID.
- Metodologie di refactoring.
- Metodologie per mantenere il codice ordinato e leggibile.

3

Preparazione dell'ambiente di sviluppo per Wordpress

Wordpress, essendo un Content Management System open source, è altamente personalizzabile e dispone di un innumerevole quantità di *plugin*, gratuiti e a pagamento, per ogni tipo di funzionalità.

Al mio arrivo veniva usata un'installazione base di Wordpress, con la sola aggiunta di un plugin chiamato *Microsoft Azure for Wordpress* che fa in modo che tutte le immagini che vengono caricate dal pannello di admin di Wordpress vengono caricate e servite da un server di Microsoft Azure.

Questo serve principalmente, come accennato precedentemente, ad oscurare il dominio del server di Wordpress, infatti se in una pagina del KN fossero linkate le immagini con l'URL di Wordpress questo potrebbe comportare dei problemi di sicurezza.

Per iniziare i miei lavori su Wordpress era necessario impostare un ambiente di sviluppo e dei modi per automatizzare la distribuzione delle modifiche.

Inoltre era necessario aggiungere un sistema di versionamento che fino a quel momento per Wordpress, a differenza degli altri progetti, non veniva utilizzato.

3.1 Architettura per lo sviluppo

In 7Pixel per lo sviluppo di tutte le applicazioni, come tipico delle aziende che fanno sviluppo agile, viene usata la seguente architettura basata su tre macchine:

- **Macchina di produzione:** è la macchina da cui vengono servite l'applicazione per l'utente finale.
- **Macchina di LAB:** è un ambiente identico a quello di produzione. Viene utilizzato per testare le nuove funzionalità prima di venire deploiate in produzione.
- **Macchina di sviluppo:** è la macchina dove lavorano gli sviluppatori, non vengono usati dati reali per i prodotti, ma solo un numero ridotto di prodotti fake utili ai fini di testing.

3.2 Procedura di deploy

Per la distribuzione delle modifiche si usa il seguente procedura

- **Sviluppo in locale:** viene editato il codice per aggiungere una nuova funzionalità usando Test Driven Development, una volta visto in locale che la funzionalità è stata implementata correttamente si passa alla fase successiva
- **Test in lab:** le nuove modifiche vengono deploiate in LAB, dove, sfruttando un ambiente simile a quello di produzione, vengono fatti ulteriori controlli, se si riscontra qualche problema si ritorna alla fase precedente e si corregge altrimenti si passa alla fase successiva
- **Deploy in produzione:** una volta effettuati i controlli in LAB, le modifiche vengono pubblicate sulle macchine di produzione e saranno disponibili agli utenti finali.

Nei minuti successivi si tiene sotto controllo **New Relic**, un'applicazione di monitoraggio degli errori, per vedere se le modifiche pubblicate fanno generare degli errori inaspettati. In caso di errori si fa *rollback* alla versione precedente altrimenti la nuova funzionalità viene considerata pubblicata con successo

Prima dei deploy, sia in LAB che in produzione, vengono fatti girare tutti i test, unitari e di integrazione, e il codice viene pubblicato solo se tutti questi sono *verdi*, ovvero passano correttamente.

3.3 Impostazione della macchine di sviluppo

Prima del mio arrivo il Team Iguana non si occupava dello sviluppo di Wordpress, non era quindi presente l'applicazione nelle macchine di sviluppo locale.

È stato mio compito quindi, prima di iniziare a sviluppare, di installare su tutte le macchine di sviluppo un'istanza di Wordpress, servita dal server Nginx[3], lo stesso server già presente nelle macchine locali per la reindirizzazione delle chiamate a Kiruby.

Name	Size	Type
wp-admin	87 items	Folder
wp-content	3 items	Folder
wp-includes	185 items	Folder
index.php	418 bytes	Program
license.txt	19.9 kB	Text
readme.html	7.4 kB	Text
wp-activate.php	5.4 kB	Program
wp-blog-header.php	364 bytes	Program
wp-comments-post.php	1.6 kB	Program
wp-config-sample.php	2.9 kB	Program
wp-cron.php	3.3 kB	Program
wp-links-opml.php	2.4 kB	Program
wp-load.php	3.3 kB	Program
wp-login.php	34.3 kB	Program
wp-mail.php	8.0 kB	Program
wp-settings.php	16.2 kB	Program
wp-signup.php	29.9 kB	Program
wp-trackback.php	4.5 kB	Program
xmlrpc.php	3.1 kB	Program

Figura 3.1: Contenuto della cartella "wordpress".

È stata poi creata un repository di git per il versionamento di Wordpress. Alla radice della cartella *wordpress* (vedi Figura 3.1) troviamo alcuni file di configurazione e le cartelle *wp-content*, *wp-admin* e *wp-includes*[4] di queste solo la cartella *wp-content* viene versionata perchè qui troviamo tutti i file rilevanti per la programmazione delle varie funzionalità, mentre nelle altre cartelle troviamo file di configurazione e altre funzionalità standard di *wordpress* che non vengono modificati dagli sviluppatori e rimangono intatte in tutte le macchine.

3.4 Creazione dello script di deploy

Lo script di deploy, essendo tutte la macchine di sviluppo e produzione macchine Linux, è stato scritto in un file eseguibile *kirCMS-autodeploy.sh* utilizzando il linguaggio Bash

ed utilizza una procedura simile allo script di deploy utilizzato da Kiruby.

Prima di eseguire il deploy è necessario fare commit sul master della repository di Wordpress in modo da avere i sorgenti git aggiornati.

Una volta committate le modifiche da pubblicare bisogna eseguire lo script *kirCMS-autodeploy.sh* passandogli come argomento *lab* o *pro* a seconda della macchina su cui si vuole deploiare.

Lo script per prima cosa chiede le credenziali di autenticazione, apre un tunnel verso la macchina dove si vuole deploiare, scarica l'ultima versione zippata della repository e la invia alla macchina collegata.

Una volta collegato in remoto alla macchina su cui si deve deploiare, viene dezippatata la repository e rinominata come *wp-content*, mentre quella che precedentemente si chiamava *wp-content* viene rinominata *wp-content-bkp* e quella che era *wp-content-bkp* viene cancellata. Dentro *wp-content* vengono eseguiti tutti i test di PHPUnit, eseguendo lo script presente all'interno della repository chiamato *test.sh*.

Se i test falliscono viene stampato a console un messaggio con lo stack dell'errore, *wp-content* viene cancellata e *wp-content-bkp* viene rinominata in *wp-content*. In questo modo la procedura termina, lasciando online la versione precedentemente pubblicata.

Se tutti i test passano invece la procedura termina e il server inizierà a utilizzare le ultime modifiche.

La cartella *wp-content-bkp* dove troviamo la precedente versione pubblicata serve a tenere quest'ultima ancora disponibile semplificando la procedura per un eventuale rollback.

Se vengono individuati degli errori con il nuovo deploy, per ritornare alla situazione precedente è sufficiente eliminare la cartella *wp-content* e rinominare *wp-content-bkp* in *wp-content*.

```
$ cd /media/www/wordpress  
$ rm wp-content && mv wp-content-bkp wp-content
```

comandi da eseguire a terminale per effettuare rollback alla versione precedente

Listing 3.1: Script di deploy: kirCMS-autodeploy.sh

```
#!/bin/bash

set -e

RED_COLOR='tput setaf 1'
GREEN_COLOR='tput setaf 2'
RESET_COLOR='tput sgr0'

CHECKOUT_DIR=wplib
TARFILE=wplib.tar.gz
TARGET_DIR=/media/www/wordpress
WP_CONTENT=wp-content
WP_CONTENT_BKP=wp-content-bkp
UPLOADS=wp-content/uploads

LOGFILE=/media/www/wordpress/wp-content/themes/basic/test/.log.txt
LOAD=/media/www/wordpress/wp-load.php
TESTS=/media/www/wordpress/wp-content/themes/basic/test/.

#CHIEDI USERNAME E CONTINUA DOPO AVER APERTO IL TUNNEL
if [ ! "$1" ]
then
    echo "${RED_COLOR}Dimmi dove vuoi deployare! LAB \
o PRO?${RESET_COLOR}"
    exit
fi
echo -n "Username: "
read username
gnome-terminal -e "$1.sh $username" 2>&1 > /dev/null &
echo "${GREEN_COLOR}Tunnellizzato e premi un tasto \
per continuare${RESET_COLOR}"
read

#SCARICA LA REPO DA GIT, COMPRIME E MANDA DOVE APERTO IL TUNNEL
cd /tmp && git clone git@gitlab.p7intranet.it:wplib.git
```

```

tar -czvf $TARFILE $CHECKOUT_DIR
scp -P 20059 /tmp/${TARFILE} root@localhost:${TARGET_DIR}

rm $TARFILE
rm -rf $CHECKOUT_DIR

echo "${GREEN_COLOR}== ${TARFILE} copied in remote at \
${TARGET_DIR} ==${RESET_COLOR}"

echo "==== switch WP-CONTENT and create WP_CONTENT_BKP ==="
#COLLEGA AD HOST REMOTO
ssh -C -l root -p 20059 -o UserKnownHostsFile=/dev/null -o \
StrictHostKeyChecking=no -R 8765:192.168.254.140:22 localhost \
-i /home/xpuser/shoppydoo/kiruby/key.ssh -t << HERE
    #ESEGUE IN REMOTO
    cd /media/www/wordpress; bash --login
    tar -xzf ${TARFILE}
    [ -d $WP_CONTENT_BKP ] && rm -rf $WP_CONTENT_BKP
    rm ${TARFILE}
    mv $WP_CONTENT $WP_CONTENT_BKP && mv $CHECKOUT_DIR \
$WP_CONTENT
    chmod 777 -R $UPLOADS
#ESEGUE I TEST E STAMPA ESITO IN FILE DI LOG
    phpunit --bootstrap $LOAD $TESTS > $LOGFILE
    if grep -q 'OK (' $LOGFILE;
    then
        echo '${GREEN_COLOR}All tests passed!!';
    else
        echo '${RED_COLOR}Tests not passed!! :(';
        cat $LOGFILE; echo '${RESET_COLOR}';
        rm -rf $WP_CONTENT && mv $WP_CONTENT_BKP \
$WP_CONTENT;
    fi
HERE
echo "${GREEN_COLOR} Success , CMS is online!${RESET_COLOR}"

```

4

Integrazione Page Builder

Per passare ad una gestione più semplice delle pagine da parte dei Content mi è stato chiesto di cercare qualche sistema che permetesse una suddivisione della pagina in componenti facilmente editabili e riutilizzabili.

Le esigenze principali erano:

- modificare il contenuto delle componenti da interfaccia grafica e non editando il codice HTML.
- poter creare semplicemente copie delle componenti già create
- poter spostare le varie componenti con *drag and drop* e avere feedback visivo immediato della modifica della pagina

Per questo mi è stato consigliato di fare una ricerca tra le varie soluzioni disponibili nella ampia libreria di plugin di Wordpress.

Dopo aver analizzato i vari plugin disponibili è stata individuata una soluzione open source chiamata *Page Builder by Site Origin*[5] che soddisfava le esigenze.

4.1 Page Builder by Site Origin

Il plugin *Page Builder* permette la suddivisione della pagina in colonne all'interno delle quali si possono inserire delle componenti.

Queste componenti possono essere duplicate, modificate aprendo il form della componente e spostate con drag and drop.

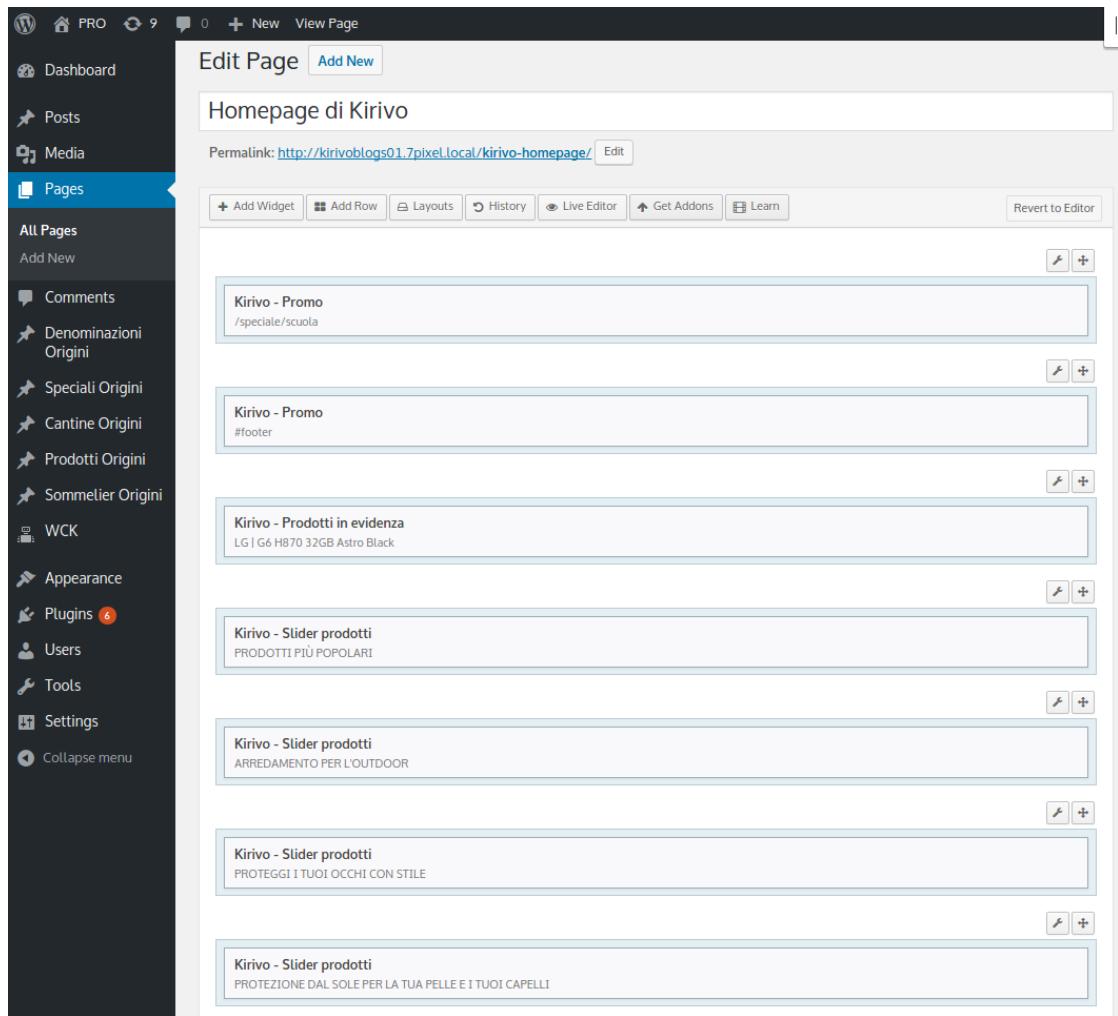


Figura 4.1: L'interfaccia di editing di *Page Builder* per la homepage di Kirivo permette la suddivisione in componenti e la loro copia e modifica.

Il vantaggio di questo plugin rispetto ad altri plugin che permettono la suddivisione della pagine è che all'interno delle colonne oltre a poter inserire delle componenti standard fornite dal plugin si possono anche inserire i Widget di Wordpress.

4.2 Widget

I *Widget* sono delle componenti riutilizzabili che possono essere aggiunte a qualsiasi tipo di pagina Wordpress.

Classici esempi di Widget sono le icone per i link ai social network, o un anteprima di un post creato, tuttavia è possibile creare Widget personalizzati.

Per creare un widget bisogna implementare una sottoclasse di `WP_Widget`[6] e sovrascrivere il metodo `widget` dove viene restituito il contenuto HTML che la pagina deve restituire quando quel widget viene incluso e, se si vuole del contenuto dinamico, bisogna sovrascrivere `form` dove viene creato il form HTML che verrà visualizzato dai content per modificare le parti dinamiche delle componenti.

Le componenti compilate nel form vengono poi utilizzate dal metodo `widget`.

```

1  <?php
2  class simpleWidget extends WP_Widget {
3      function __construct() {
4          parent::__construct( false , 'Kirivo - Simple widget' );
5      }
6
7      function widget( $args , $instance ) { ?>
8          <div>
9              <h1>Titolo del widget : <?php echo $instance[ 'title' ]?></h1>
10             </div>
11             <?php
12     }
13
14     function form( $instance ) {?>
15         <p>
16             <label for="<?php echo $this->get_field_id( 'title' ); ?>"
17                 'Titolo:');</label>
18             <input id="<?php echo $this->get_field_id( 'title' );?>"
19                 name="<?php echo $this->get_field_name( 'title' ); ?>"
20                 type="text" value="<?php echo $instance['title']; ?>" />
21         </p>
22         <?php
23     }
24 }
```

Una semplice implementazione di un widget che stampa un titolo dinamicamente

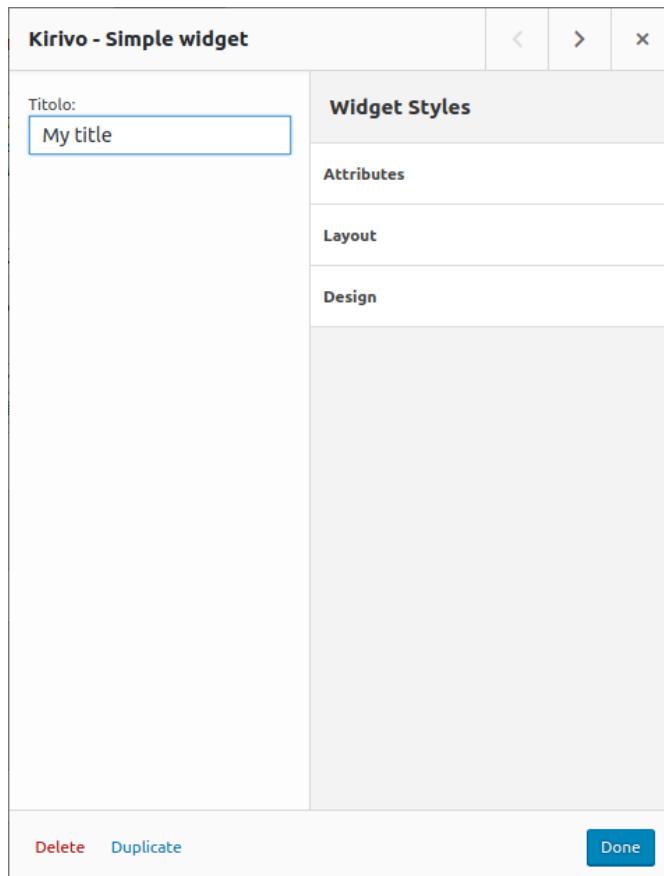


Figura 4.2: Il form che viene visualizzato per modificare i dati del Widget.

```
<div>
  <h1>Titolo del widget : My title</h1>
</div>
```

Figura 4.3: Porzione di sorgente di una pagina che utilizza il widget compilato come in 4.2.

4.3 Inizializzazione

Per inizializzare i Widget bisogna aggiungere una action[7], ovvero un meccanismo offerto da Wordpress per aggiungere dei comportamenti prestabiliti, chiamata *init_widget* dove viene aggiunto ogni Widget creato.

La sintassi per aggiungere un action in wordpress è la seguente

```
add_action('nome_action','nome_della_funzione_implementata')
```

Per far in modo di non dover aggiungere manualmente un Widget al file di inizializzazione ogni volta che se ne crea uno nuovo, è stato implementato un script di inizializzazione che va all'interno delle cartelle *widget_orgini* e *widget_kirivo* e aggiunge tutti i Widget che trova all'interno di queste cartelle, per individuare un widget all'interno della cartella viene fatto pattern-matching per i file che terminano con *Widget.php*.

```

1 <?php
2 function getKirivoWidgets() {
3     $res = array();
4     foreach(glob( dirname(__FILE__)."/widget-kirivo/*Widget.php") 
5             as $filename) {
6         array_push($res,$filename);
7     };
8     return $res;
9 }
10 function getOrginiWidgets() {
11     $res = array();
12     foreach(glob( dirname(__FILE__)."/widget-orgini/*Widget.php") 
13             as $filename) {
14         array_push($res,$filename);
15     };
16     return $res;
17 }
18 function my_register_widgets() {
19     foreach(getKirivoWidgets() as $filename) {
20         require_once $filename;
21         register_widget(str_replace('.php','',str_replace(dirname(__FILE__).
22             '/widget-kirivo/','', $filename)));
23     }
24     foreach(getOrginiWidgets() as $filename) {
25         require_once $filename;
26         register_widget(str_replace('.php','',str_replace(dirname(__FILE__).
27             '/widget-orgini/','', $filename)));
28     };
29 }
30 add_action('widgets_init','my_register_widgets');
```

Il file initialize_widget.php

4.4 Implementazione Widget

Per implementare i Widget necessari per le Homepage sono state isolate le componenti HTML necessarie e per ognuna di queste è stato creato un Widget poi, per ognuna di queste è stato creato il suo form per editare le componenti necessarie.

Ci sono due categorie principali di widget



LA CANTINA DEL MESE

FEDERICO FERRERO

L'azienda agricola Ferrero nasce dal lavoro e dalla passione di mio bisnonno Giacomo che alla fine dell'800 insedia la sua azienda a Mango, in località Gala, e inizia a vinificare uve dolcetto e moscato. Il vino diviene rapidamente la fama di quest'azienda che cresce e si espande grazie all'impegno e all'opera di mio nonno Luigi Pietro Michele, di mio padre e dei suoi fratelli tanto che nel 1948 venivano prodotti 200 quintali di uva all'anno.

[Scopri](#)

Figura 4.4: Contenuto renderizzato dal widget *Origni - Speciale*.

- Widget che non contengono prodotti.
- Widget che contengono prodotti.

Per implementare i Widget che non contengono prodotti, come ad esempio il widget *Origni - Speciale*, è stato messo all'interno del metodo *widget* direttamente l'HTML di quella componente con eventuale contenuto dinamico stampato in php e nel metodo *form* aggiunto l'HTML per gli input necessari.

Listing 4.1: Codice del widget *Origni - Speciale*

```

1  <?php
2  class specialWidget extends WP_Widget {
3      function __construct() {
4          parent::__construct( false , 'Origini - Speciale' );
5      }
6
7      function widget( $args , $instance ) {
8          $image = ! empty( $instance[ 'img-link' ] ) ? $instance[ 'img-link' ] : '';
9      ?>
10     <section id="hp_editorial_contents" class="bg_white">
11     <div class="row paragraph_vertically_spaced">
12         <div class="small-12 medium-6 columns">
13             <div class="hp_box">
14                 <a href=<?php echo $instance['link']?>>
15                     <picture class="tile_article_image">
16                         <img src=<?php echo $image?> alt="descrizione dell'immagine">
17                     </picture>
18                 </a>
19             </div>
20         </div>
21         <div class="small-12 medium-6 columns">
22             <div class="hp_box">
23                 <div class="hp_article_box">
24                     <h5 class="subheader"><?php echo $instance[ 'subtitle' ]?></h5>
25                     <h3 class="dotted_bordered"><?php echo $instance[ 'title' ]?></h3>
26                     <h4 class="subheader"></h4>
27                     <p class="hp_article"><?php echo $instance[ 'description' ]?></p>
28                 </div>

```

```

29      <div class="row hp_box_more text-center">
30          <a href=<?php echo $instance['link']?> class="hollow ">Scopri</a>
31      </div>
32  </div>
33  </div>
34 </div>
35 </section><?php
36 }
37
38 function form( $instance ) {
39     $title = esc_attr($instance['title']);
40     $image = ! empty( $instance['img-link'] ) ? $instance['img-link'] : '' ;?>
41     <p><label for=<?php echo $this->get_field_id('subtitle')?>></label>
42         Sottotitolo: </p><input id=<?php echo $this->get_field_id('subtitle')?>
43             name=<?php echo $this->get_field_name('subtitle')?>" type="text"
44             value=<?php echo $instance['subtitle']; ?>" />
45     </p>
46     <p><label for=<?php echo $this->get_field_id('title')?>></label>
47         Titolo:</p> <input class="required" id=<?php echo $this->get_field_id('title')?>
48             name=<?php echo $this->get_field_name('title')?>" type="text"
49             value=<?php echo $title; ?>" />
50     </label>
51     <p><label for=<?php echo $this->get_field_id('description')?>></label>
52         Descrizione:</p> <input id=<?php echo $this->get_field_id('description')?>
53             name=<?php echo $this->get_field_name('description')?>" type="text"
54             value=<?php echo $instance['description']; ?>" />
55     </p>
56     <p><label for=<?php echo $this->get_field_id('link')?>></label>
57         Link:</p> <input class="required" id=<?php echo $this->get_field_id('link')?>
58             name=<?php echo $this->get_field_name('link')?>" type="text"
59             value=<?php echo $instance['link']; ?>" />
60     </p>
61     <p>
62         <label for=<?php echo $this->get_field_id('img-link'); ?>></label>
63         <input id=<?php echo $this->get_field_id('img-link'); ?>
64             name=<?php echo $this->get_field_name('img-link'); ?>" type="text"
65             value=<?php echo esc_url( $image ); ?>" />
66         <button class="upload_image_button button button-primary">
67             Upload or select image
68         </button>
69     </p>
70     <?php
71 }
72 }

```

Per i widget che contengono prodotti invece all'interno di *widget* non è stato inserito il codice HTML con il template di ogni prodotto ma è stato messo il tag *dynamic* i cui attributi sono popolati dinamicamente in base al contenuto del form. Una volta che il server Kiruby prende una pagina Wordpress e trova il dynamic tag allora provvede a sostituire il tag con le *tile* di tutti i prodotti specificati.

Listing 4.2: Il metodo *widget* di *Origini - Slider prodotti* stampa il dynamic tag che verrà letto da Kiruby e sostituito con l'HTML dei prodotti

```

1 function widget( $args, $instance ) {?>
2     <dynamic type="OriginiHomeSpecialProductsBox" product_ids=<?php echo $instance['ids']?>">
3         title=<?php echo $instance['title']?>" subtitle=<?php echo $instance['subtitle']?>">
4         link=<?php echo $instance['scopri-link']?>" no_slack='true' />
5     <?php}

```


5

API rendered prices

Una volta implementati i vari Widget delle componenti delle homepage le specifiche di poter modificare il contenuto delle componenti da interfaccia grafica e senza editare il codice HTML, di poter creare copie delle componenti già create, e di poter spostare semplicemente con drag and drop le varie componenti risultavano soddisfatte.

L'unico problema rimanente dunque era quello di avere un feedback visivo immediato durante l'editing della pagina.

Il feedback visivo era già presente per quelle componenti senza prodotti il cui HTML risiedeva in Wordpress, per le componenti che fanno uso di prodotti invece l'utente che usava Wordpress non vedeva niente renderizzato.

Questo perchè una volta compilato il Widget, Wordpress renderizza il tag *dynamic* e non i prodotti, questi compaiono solo una volta che la pagina Wordpress viene processata da Kiruby, che trovando il tag *dynamic* lo sostituisce con l'HTML.

Per risolvere questo problema è stato deciso di implementare nel server Kiruby delle API chiamate *renderedAPI* che fatte delle richieste con gli ID dei prodotti e altri parametri come titolo e descrizione ritorna l'HTML con tutte le schede prodotto ed eventuali intestazioni.

Queste API poi vengono chiamate da Wordpress che riceve subito l'HTML e lo renderizza immediatamente, le stesse API vengono anche chiamate da *cms_dealer* per sostituire i tag *dynamic*.

Per spiegare il meccanismo delle rendered API prendiamo in considerazione il meccanismo utilizzato per il widget *Kirivo - Slider Prodotti* (vedi figura 1.6).

The image shows a user interface for a web editor, likely Kirivo, displaying a layout grid. The grid contains several content blocks, each with a title and some descriptive text or an image. To the right of the editor, there are three examples of how these blocks might appear on a live website:

- SI RIPARTE!** (Promotional banner for school supplies): A girl holding binoculars that look like school buses.
- ISCRIVITI ALLA NEWSLETTER** (Newsletter sign-up): A teal-colored box with a yellow button labeled "5€" and a "Iscriviti ora >>" button.
- PRODOTTI IN EVIDENZA** (Product highlight): A box featuring a large image of a cat and a dog, with the text "SPECIALE CANI&GATTI".
- PRODOTTI PIÙ POPOLARI** (Popular products): A box with a light gray background.

Figura 5.1: Nella modalità di *live editing* i prodotti non vengono visualizzati.

Altri Widgets come *Origini - Slider Accessori*, *Origini - Slider Prodotti* e *Kirivo - Griglia Prodotti* utilizzano un meccanismo praticamente identico ma utilizzano differenti template HTML.

5.1 Creazione chiamata in Wordpress

Inanzitutto per tutti quei Widget che fanno chiamate alle API è stata creata una superclasse PHP chiamata *AbstractProductWid*. Qui nel metodo *widget*, che stampa l'HTML del Widget, viene stampato il contenuto che si riceve facendo la chiamata alle API. L'indirizzo a cui viene fatta la chiamata viene composta con il metodo *composeUrl*.

Listing 5.1: I metodi *widget* e *callApiAndRender* di *AbstractProductWid*

```

1 function widget($args, $instance){
2     $this->callApiAndRender($this->composeUrl($instance));
3 }
4
5 function callApiAndRender($url) {
6     $api_response = wp_remote_get($url);
7     $api_data = json_decode( wp_remote_retrieve_body
8         ( $api_response ), true );
9     echo str_replace('\n', '', $api_data['rendered_html']);
10 }
```

Il metodo *composeUrl* della classe *AbstractProductWid* inizia a comporre la chiamata con quei campi comuni a tutti i Widget che fanno chiamate alle API ovvero: gli ID dei prodotti, e il titolo della sezione.

Listing 5.2: Il metodo *composeUrl* di *AbstractProductWid*

```

1 function composeUrl($instance) {
2     $api_request = getHost();
3     $api_request .= '/iguana/api/cms/' . $this->iguanaApiRoute() .
4         '?product_ids=' . $instance['ids'];
5
6     if(!is_null($instance['title'])) {
7         $api_request .= "&title=".urlencode($instance['title']);
8     }
9
10    return $api_request;
11 }
12
```

Le classi che estendono *AbstractProductWid* poi devono definire il metodo *iguanaApiRoute* che specifica a quale rotta il Widget deve fare la chiamata ed eventualmente aggiungere campi alla chiamata sovrascrivendo *callApiAndRender* con la chiamata al metodo *composeUrl* della superclasse come prima istruzione.

Ad esempio il codice del Widget *Kirivo - Slider Prodotti* che deve fare la chiamata alla rotta */iguana/api/cms/rendered_box_prices* e aggiungere anche i campi link allo scopri altro e sottotitolo sarà:

Listing 5.3: Viene sovrascritto *iguanaApiRoute* da *ProductsWidget* per specificare la chiamata da effettuare

```
1 function iguanaApiRoute() {
2     return 'rendered_box_prices';
3 }
```

Listing 5.4: Ridefinendo *composeUrl* vengono aggiunti ulteriori campi specifici del Widget *ProductsWidget*

```
1
2 function composeUrl($instance) {
3     $api_request = parent::composeUrl($instance);
4     if(!is_null($instance['subtitle'])) {
5         $api_request .= "&subtitle=".urlencode($instance['subtitle']);
6     }
7     if(!is_null($instance['scopri-link'])) {
8         $api_request .= "&link=". urlencode($instance['scopri-link']);
9     }
10    return $api_request;
11 }
```

5.2 Creazione risposta in Kiruby

Il widget *ProductsWidget* quindi quando deve essere renderizzato farà una chiamata del tipo

https://origini.it/iguana/api/cms/rendered_box_prices?product_ids=id1,id2,id3&title=carouselttitle

ne prelevava il contenuto e lo stampa.

Vediamo ora come sono state implementate le risposte in Kiruby.

Per implementare delle rotte bisogna estendere una classe già presente in Kiruby chiamata *AbstractRoute*.

Dopodichè è stato creato un modulo chiamato *RenderedCmsApiRouteModule* dove vengono implementati metodi comuni a tutte le rotte delle renderedAPI, come l'estrazione del titolo, e la logica per dividere il valore passato nel parametro *product_ids* e da questo prendere le informazioni di tutti i prodotti.

Listing 5.5: Il metodo *title* che preleva il parametro *title* se presente

```
def title
  params[:title].present? ? params[:title] : ''
end
```

Listing 5.6: Il metodo *product_ids* trasforma gli ids passati nel parametro in un array

```
def product_ids
  params[:product_ids].gsub(/\s+/, '').split(',')
end
```

La risposta di una sottoclasse di *AbstractRoute* viene implementata nel metodo *response_hash*, le rotte restiscono un JSON così strutturato:

```
{rendered_html: "<section>....(conenuto html da renderizzare)..</section>"}
```

Listing 5.7: Il metodo *response_hash* dove viene creato il JSON da restituire

```
def response_hash
  session = create_session
  prices = session.filter_by_code(product_ids).get_results
  prices = ordered_prices(prices)
  {rendered_html: rendered_html(prices, title, subtitle, link)}
end
```

Il metodo *response_hash*, preleva tutte le informazioni dei prodotti chiamando il metodo

```
session.filter_by_code(product_ids).get_results
```

e il contenuto restituito dal JSON viene fornito chiamando il metodo *rendered_html*.

Listing 5.8: Il metodo *rendered_html* restituisce il contenuto dell'erb compilato con i parametri passati con la chiamata *FileTemplate.for_kirivo*

```
def rendered_html(prices, title, subtitle, link)
  FileTemplate.for_kirivo('kirivo_products_box.erb',
    OpenStruct.new({products: prices, title: URI.decode(title)})
end
```

Una volta presi i prodotti vengono passati insieme al titolo all'erb *origini_products_box.erb*, che a sua volta per ogni prodotto compila l'erb *products_template.erb*.

Listing 5.9: Il template *origini_products_box.erb*

```
<section id="hp_specials" class="bg_white">
  <div class="row text_center">
    <h5 class="subheader text_center"><%= title %></h5>
  </div>

  <div class="row small-up-1 medium-up-3 large-up-4" data-equalizer="box_offer_height">
    <% products.each do |product| %>
      <%= FileTemplate.for_kirivo('product_template.erb', { price: product }) %>
    <% end %>
  </div>
</section>
```

Listing 5.10: Il template *products_template.erb*

```
<% price = model[:price] %>
<div class="columns product_tile_end" data-price-container>
  <div class="deal_offer_tile" data-merchant-id="<%= price.merchant_id %>">
    <div class="feedback_mask">
      <a href="<%= Origini::ProductSheetRoute.url_for(price) %>">
        <div class="upper_tile">
          <div class="image_box">
            <picture>
              
            </picture>
          </div>
        </div>
      </a>
      <div class="bottom_tile">
        <div class="tile_description">
          <small class="tile_quantity text_grey"><%= price.capacity %></small>
          <a href="<%= Origini::ProductSheetRoute.url_for(price) %>">
            <h4 class="product_name" data-price-name>
              <strong><%= price.name %><% if price.is_year_defined %>
                <%= price.production_year %>
              <% end %>
              </strong></h4>
            </a>
          </div>
        </div>
        <div class="more_info_hover">
          <% if price.winery %>
            <a href="<%= Origini::WineryRoute.url_for(price.winery) %>">
              <h5 class="tile_winery_name text_grey"><%= price.winery %></h5>
            </a>
          <% end %>
          <h5 class="tile_docg_name"><%= price.denomination %></h5>
        </div>
        <div class="cart_feedback">
          <i class="fa fa-3x fa-check"></i> Aggiunto al carrello
        </div>
      </div>
      <div class="tile_hover">
        <form class="add_to_cart_form" action="<%= Cart.add_cheapest_to_cart_url %>" method="post">
          <button id="add_to_cart_<%= price.id %>" class="button_add_to_cart white" data-add-to-cart>
            <i class="fa fa-shopping-cart"></i> AGGIUNGI AL CARRELLO
          </button>

          <input name="qty" class="qty" value="1" type="hidden">
          <input name="productCodePost" value="<%= price.id %>" type="hidden" data-price-id>
        </form>
        <input name="ean" value="<%= price.ean_codes.first %>" type="hidden" data-price-ean>
      </div>
    </div>
  </div>
</div>
```

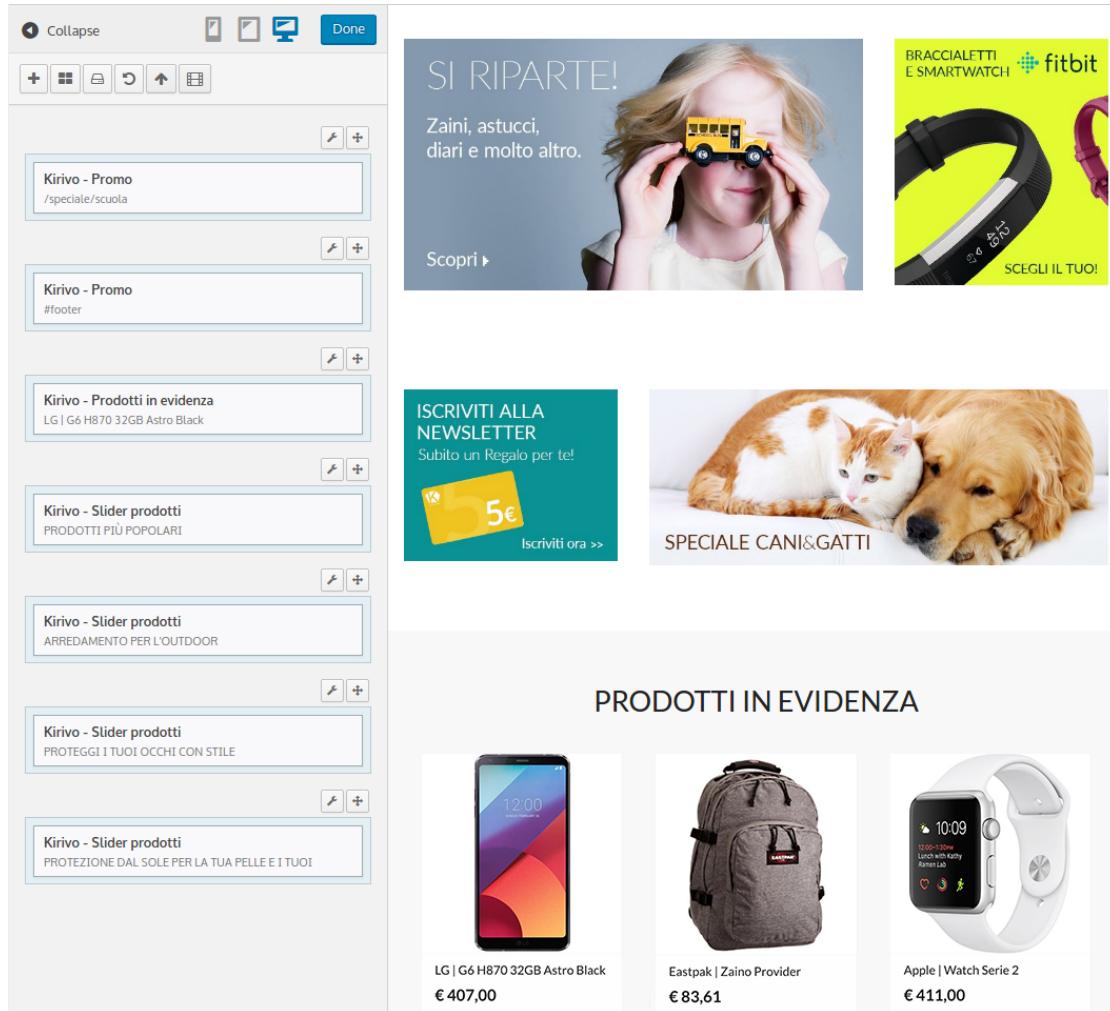


Figura 5.2: Con le renderedAPI i prodotti vengono visualizzati.

5.3 Creazione chiamata in cmsdealer

La gemma *cmsdealer* per trasformare i dynamic tag in contenuto renderizzato usava una procedura simile a quella delle renderedAPI con la differenza che invece dei parametri della chiamata venivano letti i parametri del tag *dynamic*, e invece che restituire l'HTML della componente in un JSON questo veniva direttamente scritto nella pagina HTML da restituire dove veniva letto il tag *dynamic*.

Nonostante la creazione delle renderedAPI *cmsdealer* veniva ancora utilizzato, questo perchè nei siti a parte le homepages ci sono ancora pagine compilate scrivendo direttamente l'HTML e che usano i dynamic tag.

Per evitare duplicazione di codice, in particolare la presenza doppia degli erb dei template dei prodotti, si è fatto in modo che quando *cmsdealer* legge un tag *dynamic*, sostituisce questo con il contenuto che viene restituito dalle renderedAPI.

Per fare questo, in modo simile ai Widget, è stata implementata una sottoclasse di *DynamicTag* chiamata *ApiCallingTag*.

In questa classe viene implementato il meccanismo di composizione della chiamata, la lettura della risposta e la sostituzione del contenuto di quest'ultima con il tag *dynamic*.

Il metodo che restituisce il contenuto da sostituire *replaced_html* è stato ridefinito in modo da ritornare il contenuto della risposta delle API.

Listing 5.11: Il codice della classe *ApiCallingTag*

```
class ApiCallingTag < DynamicTag
  def replaced_html(original_url = nil)
    params = {url: original_url}
    allowed_api_parameters.each do |p|
      params[p] = eval(p) if self.respond_to?(p)
    end
    json = JSON.parse(Connection.new.get(url_for(params)).body)
    json['rendered_html']
  end

  def url_for(params)
    URI.add_params("www.origini.it/iguana/api/cms/#{{get\_route}}",
      params)
  end
end
```

Le sottoclassi di *ApiCallingTag* invece devono implementare due metodi ovvero *get_route* dove viene definito quale rotta delle renderedAPI chiamare e *allowed_api_parameters* dove viene definita un array con i parametri che possono essere letti dal tag *dynamic*.

Per esempio il dynamic tag `OriginiListBox` quando legge il tag può individuare ed inoltrare nella chiamata i parametri `product_ids`, `title`, `description`, `number`, `listing_link` e aggiungerà questi parametri alla chiamata alla rotta:

`www.origini.it/iguana/api/cms/rendered_origini_listing_box`

Listing 5.12: Il codice della classe che legge e trasforma il tag con type `OriginiListBox`

```
class OriginiListBox < ApiCallingTag
  def allowed_api_parameters
    ['product_ids', 'title', 'description', 'number', 'listing_link']
  end

  def get_route
    'rendered_origini_listing_box'
  end
end
```

Cmsdealer quindi quando troverà un tag del tipo

```
<dynamic type="OriginiListBox" product_ids="id1,id2,id3" title="products"
listing_link="/speciale/spedizione_gratuita">
```

Sostituirà a questo il contenuto ritornato dalla chiamta alla rotta

```
www.origini.it/iguana/api/cms/rendered_origini_listing_box&product_ids=id1,id2,id3
&title=products&listing_link=%2Fspeciale%2Fspedizione_gratuita
```

Così facendo tutti i requisiti del progetto risultano soddisfatti. Nei due capitoli successivi vengono elencati i Widget creati per i due siti, specificando quali valori sono stati resi personalizzabili.

6

Widgets di Origini

LE SELEZIONI

I CONSIGLI DEL NOSTRO SOMMELIER


Cuveè Spumante Extra Brut

Montello e Colli Asolani DOC
Giusti Wine
📍 Veneto

Bottiglia 0,75 lt **16,47 €**


Syrah

Terre Siciliane IGT
Tenuta San Giaime
📍 Sicilia

Bottiglia 0,75 lt **15,00 €**


Traminer Aromatico

Friuli Grave DOC
Fossa Mala
📍 Friuli-Venezia Giulia

Bottiglia 0,75 lt **9,80 €**


Cortona

Cortona DOC
Palazzo Vecchio
📍 Toscana

Bottiglia 0,75 lt **11,00 €**

[Scopri Altro](#)

Figura 6.1: Contenuto mostrato dal widget ”Origini - Slider prodotti”.

6.1 Origini - Slider prodotti

Il Widget ”Origini - Slider prodotti” visualizza un box contenente 4 vini, con titolo, sottotitolo e link per ”Scopri altro”.

Il form permette di modificare i seguenti campi (riferirsi a Figure 6.1):

- Sottotitolo: il testo ”Le selezioni”
- Titolo: il testo ”I consigli del sommelier”
- Ids prodotti: la lista degli ID dei prodotti da visualizzare separati da virgola
- Link: l’URL a cui punta il tasto *Scopri Altro*



Figura 6.2: Contenuto mostrato dal widget ”Origini - Banda tuttoschermo”.

6.2 Origini - Banda tuttoschermo

Il Widget ”Origini - Banda tuttoschermo” visualizza quella porzione di HTML usata attualmente per lo ”Speciale regione” (vedi Figure 6.2).

Il form permette di modificare i seguenti campi (riferirsi a Figure 6.2):

- Sottotitolo: il testo ”Speciale regione”
- Titolo: il testo ”Lombardia e cultura del buon vino”
- Descrizione: il testo della descrizione ”La varietà dei territori...”
- Link: l’URL a cui punta il tasto *Scopri Altro*
- Image-Link: l’URL dell’immagine di sfondo



Figura 6.3: Contenuto mostrato dal widget "Origini - Speciale".

6.3 Origini - Speciale

Il Widget "Origini - Speciale" visualizza quella porzione di HTML usata attualmente per lo "Speciale del mese" (vedi Figure 6.3).

Il form permette di modificare i seguenti campi (riferirsi a Figure 6.3):

- Sottotitolo: il testo "La cantina del mese"
- Titolo: il testo "Federico Ferrero"
- Descrizione: il testo della descrizione "L'azienda agricola Ferrero nasce..."
- Link: l'URL a cui punta il tasto *Scopri Altro*
- Image-Link: l'URL dell'immagine da visualizzare

7

Widgets di Kirivo



Figura 7.1: Contenuto mostrato dal widget "Kirivo - Immagini Speciali".

7.1 Kirivo - Immagini Speciali

Il Widget "Kirivo - Immagini Speciali" permette di modificare il contenuto delle prime due immagini della Homepage, ovvero le immagini solitamente dedicate allo speciale del mese e alle offerte

I campi che si possono modificare sono (riferirsi a Figure ??):

- Link speciale: l'URL dove si viene indirizzati quando si schiaccia sull'immagine dello speciale
- Url immagine speciale desktop: l'url dell'immagine da mettere nello speciale per desktop
- Url immagine speciale mobile: l'url dell'immagine da mettere nello speciale per mobile
- Link offerte: l'URL dove si viene indirizzati quando si schiaccia sull'immagine delle offerte
- Url offerte speciale desktop: l'url dell'immagine da mettere nelle offerte per desktop
- Url offerte speciale mobile: l'url dell'immagine da mettere nelle offerte per mobile



Figura 7.2: Contenuto mostrato dal widget "Kirivo - Immagini Newsletter".

7.2 Kirivo - Immagini Newsletter

Il Widget "Kirivo - Immagini Newsletter" permette di modificare il contenuto delle prime due immagini della Homepage, ovvero le immagini solitamente dedicate allo speciale del mese e alle offerte

I campi che si possono modificare sono (riferirsi a Figure 7.2):

- Url immagine newsletter desktop: l'url dell'immagine da mettere nella sezione "iscriviti alla newsletter"
- Url immagine newsletter mobile: l'url dell'immagine da mettere nella sezione "iscriviti alla newsletter"
- Link offerta: l'URL dove si viene indirizzati quando si schiaccia sull'immagine dell'offerta
- Url offerta desktop: l'url dell'immagine da mettere nelle offerte per desktop
- Url offerta mobile: l'url dell'immagine da mettere nelle offerte per mobile

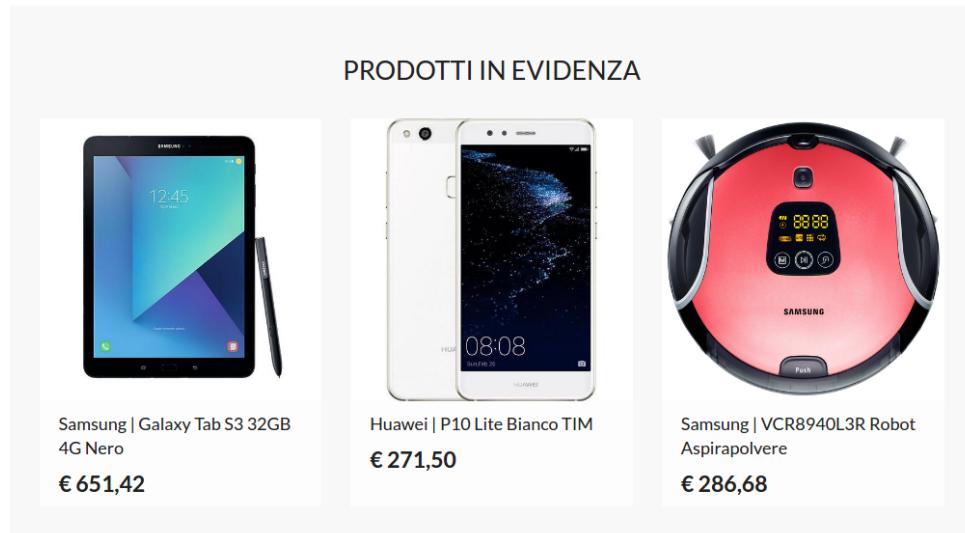


Figura 7.3: Contenuto mostrato dal widget "Kirivo - Prodotti in evidenza".

7.3 Kirivo - Prodotti in evidenza

Il Widget "Kirivo - Prodotti in evidenza" visualizza quella porzione di HTML usata per visualizzare i prodotti in evidenza (vedi Figure 7.3).

I campi che si possono modificare sono (riferirsi a Figure 7.3):

- Titolo: il testo "PRODOTTI IN EVIDENZA".
- Ids: l'ID dei prodotti da visualizzare separati da virgola. I prodotti devono essere almeno tre, se sono più di tre verranno usati i primi 3 ID validi.

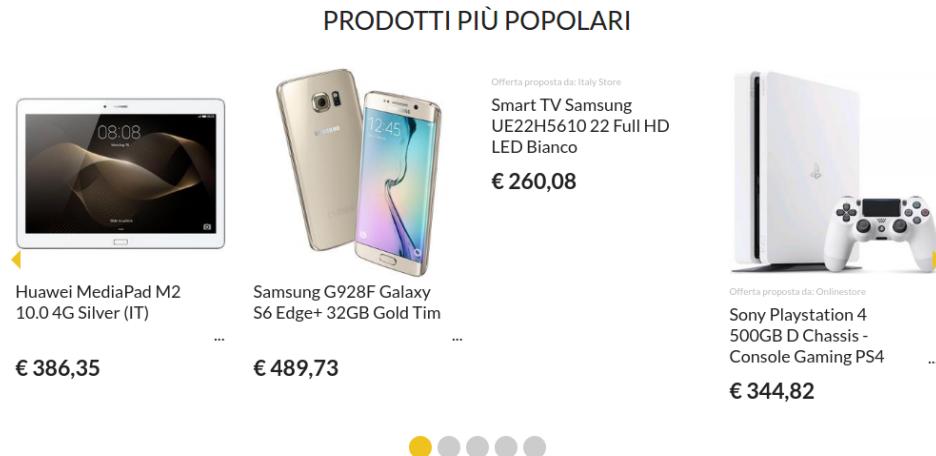


Figura 7.4: Contenuto mostrato dal widget "Kirivo - Slider prodotti".

7.4 Kirivo - Slider prodotti

Il Widget "Kirivo - Slider prodotti" visualizza quella porzione di HTML usata per visualizzare uno slider di un insieme di prodotti (vedi Figure 7.4).

I campi che si possono modificare sono (riferirsi a Figure 7.4):

- Titolo: il testo "PRODOTTI PIÙ POPOLARI".
- Numero max: il numero massimo di prodotti da visualizzare. Se lasciato vuoto non viene imposto alcun limite.
- Ids: l'ID dei prodotti da visualizzare separati da virgola. I prodotti devono essere almeno quanti specificati in numero max o quattro se non viene specificato.

8

Conclusioni

Grazie allo sviluppo del progetto, i content managers possono ora editare le homepages dei siti in maniera più semplice e più efficiente.

Ora viene garantita maggiore robustezza e copertura in caso di errore durante la modifica delle pagine, rendendo l'editing estremamente più sicuro, infatti prima andando modificare direttamente il codice HTML c'era il rischio di eliminare un tag o parti strutturali del codice HTML rischiando di andare a compromettere come veniva visualizzata l'intera pagina, ora con un editing mirato solo sulle componenti necessarie questo rischio è scomparso.

8.1 Sviluppi futuri

La creazione dei Widget, e l'integrazione del plugin *Page Builder* in Wordpress sono stati inizialmente effettuate per una creazione e modifica più semplice delle homepages di www.kirivo.it e www.origini.it.

Una volta impostato il sistema questo da la possibilità di creare qualsiasi tipo di pagina in modo modulare.

È stato programmato dal Project Manager del Kirivo Network di creare nuovi Widget e con questi, insieme a quelli già esistenti, creare gli speciali di Kirivo.

Gli speciali sono pagine che vengono create periodicamente per mettere in evidenza determinate categorie di prodotti, oppure durante feste come natale, pasqua, halloween, festa della mamma, festa del papà.

Queste pagine ad oggi vengono ancora create dai content editando dall'interfaccia di Wordpress il codice HTML che viene creato dalla grafica.

Oppure per creare un nuovo speciale si copia l'HTML di uno speciale già esistente e se ne modificano testi ed ID di prodotti.

Per la creazione di speciali modulari quindi andranno sviluppati nuovi Widget, come un modulo di intestazione e nuovi caroselli con differenti layout e poi le pagine potranno essere create in modo più rapido e robusto.

Colophon

La tesi è stata scritta utilizzando il linguaggio LaTeX.

Le immagini sono state create appositamente usando screenshots delle applicazioni ed eventualmente editate con GIMP.

Bibliografia

- [1] “Core ruby programming,” <http://rubylearning.com/satishtalim/tutorial.html>.
- [2] “Learn ruby,” <http://www.rubykoans.com/>.
- [3] “Install wordpress with nginx on ubuntu,” <https://www.digitalocean.com/community/tutorials/how-to-install-wordpress-with-nginx-on-ubuntu-14-04>.
- [4] “Beginner’s guide to wordpress file and directory structure,” <http://www.wpbeginner.com/beginners-guide/beginners-guide-to-wordpress-file-and-directory-structure/>.
- [5] “Siteorigin page builder,” <https://siteorigin.com/page-builder/>.
- [6] “Widgets api,” https://codex.wordpress.org/Widgets_API.
- [7] “Plugin api/action reference,” https://codex.wordpress.org/Plugin_API/Action_Reference.