Come creare un modulo per Magento

lunedì, 06 febbraio 2017

Magento è già di per sé un CMS molto potente e completo, ma come sempre quando si parla di software, può nascere l'esigenza di estenderne le capacità per aggiungere una o più funzionalità non previste dalla versione di default. Se non sai da dove partire, questa guida è pensata appositamente per te.

Sommario

Il progetto: cosa realizziamo?

Per spiegarti come realizzare un modulo personalizzato per Magento, prenderemo in esempio un progetto piuttosto banale. Ipotizza di voler annotare in un file di log ogni modifica apportata ai prodotti del tuo e-commerce, per tenere traccia dei cambiamenti che avvengono nel tuo shop.

Il progetto: come lo realizziamo?

Per ottenere quanto descritto, c'è bisogno che io ti annoi con un po' di teoria. Lo so che a nessuno piace, ma è necessario, sii paziente.

Magento utilizza un sistema di estensione molto utile agli sviluppatori per iniettare del codice che alteri il risultato di alcune funzioni di Magento stesso senza modificarne i files. Com'è possibile?
La magia è operata dal pattern **Event Observer**. Un Observer è una classe i cui metodi vengono dichiarati "in ascolto" sulle funzioni di default di Magento. Ogni qual volta una di queste funzioni di default viene richiamata ed eseguita, il codice del metodo Observer agganciato alla funzione viene a sua volta eseguito.
Nel nostro caso, equivale a dire:

Esiste una funzione nel core di Magento che esegue l'update dei prodotti sul db ogni volta che l'utente clicca sul pulsante "Aggiorna". Quando questa funzione viene richiamata, esegui il metodo dell'Observer agganciato su di essa

Prepariamo il terreno

Prima di iniziare, considera un aspetto fondamentale. È necessario che la cache di Magento sia disabilitata, così da non incorrere in problemi di lettura dei files giusti. Se non sai come fare, vai su:

Sistema > Gestione della Cache

clicca sul pulsante in alto a sinistra:

SELEZIONA TUTTO

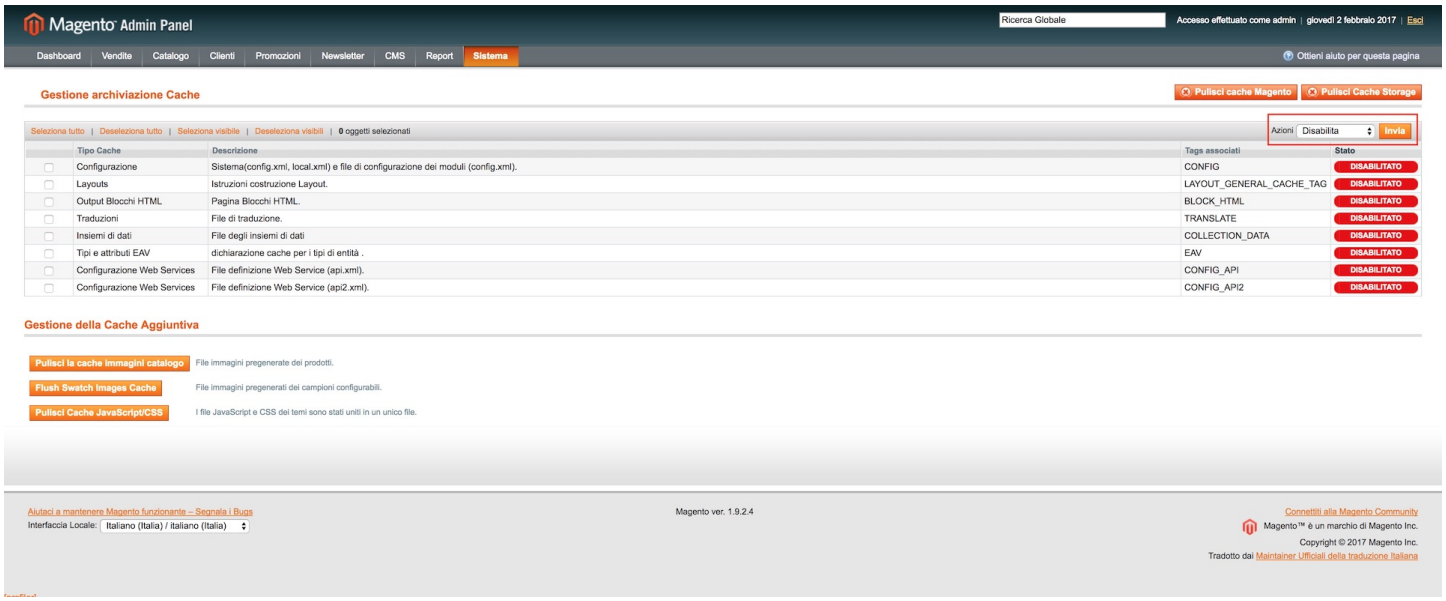
e poi a destra, nel menu a tendina "Azioni", scegli:

DISABILITA

ed infine clicca su:

INVIA

Fatto?



Il minimo sindacale c'è, ma la dritta che posso darti, per agevolarti nel lavoro, è di cambiare qualche impostazione di Magento... Se ti va di accettare qualche suggerimento, leggi l'articolo:

Come fare il debug in Magento

Creiamo le cartelle

La prima cosa da fare, è creare le cartelle che ospiteranno i files del tuo modulo. Se non sai ancora come fare o se hai le idee confuse su come sono strutturate le directory di Magento, ti consiglio di dare una rinfrescata leggendo l'articolo Il filesystem di Magento.

Ora che è tutto più chiaro, possiamo stabilire il nome del Vendor (supponiamo **Code4Life**) ed il nome del Modulo (supponiamo **ProductUpdateHistory**).

Procedi quindi col creare all'interno della cartella:

app/code/local/

una cartella chiamata **Code4Life** ed al suo interno una ulteriore cartella chiamata **ProductUpdateHistory** ed ancora al suo interno, la cartella **etc**, ottenendo il seguente percorso:

app/code/local/Code4Life/ProductUpdateHistory/etc/

Queste sono le cartelle che conterranno i files necessari allo sviluppo del tuo primo modulo per Magento.
All'interno del percorso appena creato, inserisci il file di configurazione config.xml:

app/code/local/Code4Life/ProductUpdateHistory/etc/config.xml

al cui interno scriverai:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
 <modules>
  <Code4Life_ProductUpdateHistory>
   <version>1.0.0</version>
  </Code4Life_ProductUpdateHistory>
 </modules>
</config>
```

Il file di configurazione, non fa altro che comunicare a Magento alcuni dati inerenti il modulo, come ad esempio la versione e il path secondo la nomenclatura di Magento:

NomeVendor + '_' + NomeModulo

Fin qui tutto semplice, no? Ma aspetta un attimo... come fa Magento a capire che esiste un modulo nuovo da caricare?
Giusta osservazione. Ed in effetti c'è bisogno di comunicarglielo! Per farlo, portati nella cartella

app/etc/modules/

e crea il file Code4Life_ProductUpdateHistory.xml, che rispetta sempre la nomenclatura **NomeVendor + '_' + NomeModulo** ed al suo interno scrivi:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
 <modules>
  <Code4Life_ProductUpdateHistory>
   <active>true</active>
   <codePool>local</codePool>
  </Code4Life_ProductUpdateHistory>
 </modules>
</config>
```

La cartella:

app/etc/modules/

contiene i files relativi ai moduli di Magento, il path in cui trovare i file che li compongono ed il flag di attivazione. È a questo punto che Magento è cosciente dell'esistenza del tuo modulo! Non ci credi? Entra nel backend del tuo Magento e vai su:

Sistema > Configurazione > Avanzate

e nell'unica tab presente (vale a dire: "Disabilita output dei moduli") troverai:

Code4Life_ProductUpdateHistory

Ecco, è in questo preciso istante che puoi urlare al mondo "Ho realizzato il mio primo modulo per Magento!!".
Dì la verità, non senti il tuo orgoglio crescere almeno un po'? Sono sicuro di sì, ma non è finita!

Il modulo esiste, Magento ne riconosce l'esistenza, ma bisogna programmarlo per quello che deve realmente fare: scrivere un file di log ad ogni aggiornamento di prodotto, intercettando la funzione di default di Magento che gestisce l'update.

L'Event Observer

Bene, recuperiamo la concentrazione.
Abbiamo anticipato che dovrai cercare la funzione che si occupa di effettuare l'update dei prodotti.
Per questa volta, ti aiuto io. L'evento a cui ti aggancerai è:

catalog_product_save_after

Riprendiamo adesso il file:

app/code/local/Code4Life/ProductUpdateHistory/etc/config.xml

precedentemente scritto ed aggiungiamo le righe necessarie a dichiarare l'**Event Observer**. Per completezza, ti riporto direttamente il file completo, evidenziando le righe aggiunte:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
 <modules>
  <Code4Life_ProductUpdateHistory>
   <version>1.0.0</version>
  </Code4Life_ProductUpdateHistory>
 </modules>
 <global>
  <events>
   <catalog_product_save_after>
    <observers>
     <code4life_productupdatehistory>
      <class>code4life_productupdatehistory/observer</class>
      <method>writeLogOnProductUpdate</method>
      <type>singleton</type>
     </code4life_productupdatehistory>
    </observers>
   </catalog_product_save_after>
  </events>
 </global>
</config>
```

Fermi tutti, facciamo un po' di chiarezza.

- **Riga 8**: Definisce il campo di applicazione del comportamento del modulo, indica appunto globale
- **Riga 9**: Indica che stiamo per definire un *Event Observer*

• **Riga 10**: Stabilisce su quale evento di Magento vogliamo mettere in ascolto il nostro Observer
• **Riga 11**: Inizializza l'Observer per l'evento indicato
• **Riga 12**: Contiene l'identificatore univoco all'interno del nodo *catalog_product_save_after*. Per convenzione, si utilizza la solita nomenclatura **NomeVendor + '_' + NomeModulo**, ma questa volta in lowercase (tutto minuscolo).
• **Riga 13**: Definisce la classe Model da istanziare, che contiene il metodo che verrà richiamato
• **Riga 14**: Contiene, finalmente, il nome del metodo da richiamare, il quale esegue quanto ci siamo promessi

Adesso Magento ha tutte le informazioni necessarie. Sa quale metodo eseguire, dove reperirlo, a quale classe appartiene e a quale evento deve agganciarlo. Non ti resta che scrivere il codice vero e proprio.

La classe Observer

Come stabilito nel file:

app/code/local/Code4Life/ProductUpdateHistory/etc/config.xml

Magento si aspetta una classe in cui cercare il metodo da eseguire ed è nostro compito crearla.

All'interno della cartella del nostro modulo, crea una cartella chiamata Model ed al suo interno crea un file Observer.php, ottenendo:

app/code/local/Code4Life/ProductUpdateHistory/Model/Observer.php

con il seguente contenuto:

```
class Code4Life_ProductUpdateHistory_Model_Observer {
 public function writeLogOnProductUpdate( Varien_Event_Observer $oObserver ) {
  $oProduct = $oObserver->getEvent()->getProduct();
  $sUserId = Mage::getSingleton( 'admin/session' )->getUser()->getUserId();
  Mage::log( 'Il prodotto "' . $oProduct->getName() . '" (ID: ' . $oProduct->getId() .
') Ã¨ stato aggiornato [USER ID: ' . $sUserId . '].', null, 'product_updates.log' );
 }
}
```

ed il gioco è fatto!
In particolare, nella riga 5 viene utilizzato il metodo statico *log()* che si occupa di creare una riga di log in cui abbiamo inserito il nostro messaggio di aggiornamento avvenuto, con particolare riferimento al prodotto aggiornato e a quale utente lo ha modificato.

Finalmente, il tuo primo modulo per Magento è davvero completo, non ti resta che provarlo. Entra nel backend del tuo Magento e prova ad aggiornare un prodotto, vedrai il risultato dei tuoi sforzi prendere forma nel file:

var/log/product_updates.log

che conterrà le informazioni generate dal metodo **writeLogOnProductUpdate** della classe Observer.

Non ti resta che fare tesoro di questa guida, costruendo moduli via via più complessi ed adeguati alle tue esigenze.

Una precisazione

Nel caso specifico preso in esame, ci siamo occupati di scrivere un file di log. Qualora non dovessi trovare il file nel percorso indicato dopo l'aggiornamento dei prodotti, assicurati di aver abilitato la possibilità di scrivere file di log. Nel backend del tuo Magento, vai su:

Sistema > Configurazione > Sviluppatore > Impostazioni log

ed assicurati che l'opzione ***Abilitato*** sia impostata su ***Si***.