Technical documentation

**Summary**

[1 Getting started COLIN 7](#_Toc416085531)

[1.1 Before you start (technical requirements) 7](#_Toc416085532)

[1.2 Download and install on IIS (scripts) 7](#_Toc416085533)

[1.3 Source code getting started (compile front, backoffice and services) 7](#_Toc416085534)

[2 Architecture 7](#_Toc416085535)

[2.1 Global schema 7](#_Toc416085536)

[2.2 Web applications: Front, ABO, API (ASP.net MVC and ASP.net WEB API) 8](#_Toc416085537)

[2.2.1 Front Office and ABO: ASP.net MVC 4.5 8](#_Toc416085538)

[2.2.2 API: ASP.net API 9](#_Toc416085539)

[2.3 Technologies and frameworks 10](#_Toc416085540)

[2.3.1 ORM: NHibernate 10](#_Toc416085541)

[2.3.2 Dependency injection/instances lifecycle: Structure map 11](#_Toc416085542)

[2.3.3 Windows service and jobs: Quartz.NET 12](#_Toc416085543)

[2.3.4 The other frameworks 12](#_Toc416085544)

[2.4 Windows service and jobs (Quartz.NET) TODO 16](#_Toc416085545)

[2.5 Database and ORM (mappings, HILO for ID generation, LINQ) TODO 16](#_Toc416085546)

[2.6 Shared components (DAO, business, media manager) 16](#_Toc416085547)

[2.6.1 Business 17](#_Toc416085548)

[2.6.2 DAO 18](#_Toc416085549)

[2.7 Database??? 18](#_Toc416085550)

[2.8 Localization??? 18](#_Toc416085551)

[3 Compilation and deployment Colin 18](#_Toc416085552)

[3.1 Compile using visual studio 18](#_Toc416085553)

[3.2 Package using our NANT scripts (msbuild based) 18](#_Toc416085554)

[3.3 Configure test environment and production parameters 18](#_Toc416085555)

[3.4 Deploy using powershell 18](#_Toc416085556)

[4 Coding standards 18](#_Toc416085557)

[4.1 Formatting 18](#_Toc416085558)

[4.1.1 Source files layout 18](#_Toc416085559)

[4.1.2 Indentation & Braces 19](#_Toc416085560)

[4.1.3 Line Length 19](#_Toc416085561)

[4.1.4 Blank Lines 19](#_Toc416085562)

[4.1.5 Blank Spaces 19](#_Toc416085563)

[4.2 Naming 20](#_Toc416085564)

[5 Parameters 21](#_Toc416085565)

[5.1 Parameters system (attributes, and table) 21](#_Toc416085566)

[5.1.1 Database Model 21](#_Toc416085567)

[5.1.2 Object Model 22](#_Toc416085568)

[5.2 Change parameters value in ABO 23](#_Toc416085569)

[6 Themes TODO 24](#_Toc416085570)

[6.1 Theme mechanism: custom view engine 24](#_Toc416085571)

[6.2 Emails templates 24](#_Toc416085572)

[6.3 Custom helpers 24](#_Toc416085573)

[6.4 Getting started on a custom theme creation 24](#_Toc416085574)

[7 Security 24](#_Toc416085575)

[7.1 Database Model 24](#_Toc416085576)

[7.2 Object Model 25](#_Toc416085577)

[7.3 Users and roles model 25](#_Toc416085578)

[7.4 User Account Information 26](#_Toc416085579)

[7.5 Securing Passwords 26](#_Toc416085580)

[7.6 Authentication mechanism TO COMPLETE 27](#_Toc416085581)

[8 Account model TO COMPLETE 28](#_Toc416085582)

[8.1 Database Model 28](#_Toc416085583)

[8.2 Object Model 29](#_Toc416085584)

[8.3 Merchants 30](#_Toc416085585)

[8.3.1 Database Model 30](#_Toc416085586)

[8.3.2 Object Model 30](#_Toc416085587)

[8.4 Customers 31](#_Toc416085588)

[8.4.1 Database Model 31](#_Toc416085589)

[8.4.2 Object Model 32](#_Toc416085590)

[8.5 Newsletter TO COMPLETE 32](#_Toc416085591)

[8.5.1 Database Model 32](#_Toc416085592)

[8.5.2 Object Model 33](#_Toc416085593)

[9 Catalog model 33](#_Toc416085594)

[9.1 Global database model 33](#_Toc416085595)

[9.2 Products 35](#_Toc416085596)

[9.2.1 Database Model 35](#_Toc416085597)

[9.2.2 Object Model 36](#_Toc416085598)

[9.3 Offers 37](#_Toc416085599)

[9.3.1 Database Model 37](#_Toc416085600)

[9.3.2 Objet Model 38](#_Toc416085601)

[9.4 Stocks 38](#_Toc416085602)

[9.4.1 Database Model 38](#_Toc416085603)

[9.4.2 Object Model 39](#_Toc416085604)

[9.5 Categories 39](#_Toc416085605)

[9.5.1 Database Model 39](#_Toc416085606)

[9.5.2 Objet Model 40](#_Toc416085607)

[9.6 Delivery methods, zones and rates TO COMPLETE 40](#_Toc416085608)

[9.6.1 Database Model 40](#_Toc416085609)

[9.6.2 Object Model 41](#_Toc416085610)

[9.7 Cross selling 42](#_Toc416085611)

[9.7.1 Database Model 42](#_Toc416085612)

[9.7.2 Objet Model 43](#_Toc416085613)

[9.8 Visible Products 43](#_Toc416085614)

[10 Geolocation 43](#_Toc416085615)

[10.1 The classic geolocation 44](#_Toc416085616)

[10.1.1 Database Model 44](#_Toc416085617)

[10.1.2 Object Model 44](#_Toc416085618)

[10.2 Geocoding 44](#_Toc416085619)

[10.2.1 Database Model 45](#_Toc416085620)

[10.2.2 Object Model 46](#_Toc416085621)

[10.2.3 The “getGeocodingInfo()” method: 46](#_Toc416085622)

[10.2.4 The search process 47](#_Toc416085623)

[11 Cart: cart component, profile and new order 48](#_Toc416085624)

[11.1 Cart 48](#_Toc416085625)

[11.1.1 Database Model 48](#_Toc416085626)

[11.1.2 Object Model 48](#_Toc416085627)

[11.2 Orders and merchant orders 49](#_Toc416085628)

[11.2.1 Database Model 49](#_Toc416085629)

[11.2.2 Object Model 50](#_Toc416085630)

[11.3 Payment modules 50](#_Toc416085631)

[11.3.1 Atos 51](#_Toc416085632)

[11.3.2 Be2Bill 51](#_Toc416085633)

[11.3.3 Check 51](#_Toc416085634)

[11.3.4 Mercanet 51](#_Toc416085635)

[11.3.5 Ogone 51](#_Toc416085636)

[11.3.6 Paybox 52](#_Toc416085637)

[11.3.7 Payname 52](#_Toc416085638)

[11.3.8 Paypal 52](#_Toc416085639)

[11.3.9 PayU 52](#_Toc416085640)

[11.3.10 Systempay 52](#_Toc416085641)

[11.3.11 Transfer 52](#_Toc416085642)

[11.3.12 WebAffaires 52](#_Toc416085643)

[11.4 Merchant billing 52](#_Toc416085644)

[11.4.1 Database Model 52](#_Toc416085645)

[11.4.2 Object Model 53](#_Toc416085646)

[11.5 Merchant commissions and payments 53](#_Toc416085647)

[12 Comments and ratings model 54](#_Toc416085648)

[12.1 Products 54](#_Toc416085649)

[12.1.1 Database Model 54](#_Toc416085650)

[12.1.2 Object Model 55](#_Toc416085651)

[12.2 Merchants 56](#_Toc416085652)

[12.2.1 Database model 56](#_Toc416085653)

[12.2.2 Object Model 56](#_Toc416085654)

[12.3 Orders 57](#_Toc416085655)

[12.3.1 Database Model 57](#_Toc416085656)

[12.3.2 Object Model 57](#_Toc416085657)

[13 Complaints 58](#_Toc416085658)

[13.1 Database Model 58](#_Toc416085659)

[13.2 Object Model 60](#_Toc416085660)

[13.1 Workflow 61](#_Toc416085661)

[13.2 Manual and automatic refund 61](#_Toc416085662)

[14 URL rewriting 62](#_Toc416085663)

[14.1 Rewriting mechanism 62](#_Toc416085664)

[14.1.1 Dabase Model 62](#_Toc416085665)

[14.2 Object Model 63](#_Toc416085666)

[14.3 Automatic URL generation 63](#_Toc416085667)

[15 Jobs: long running/recurrent tasks TODO 64](#_Toc416085668)

[15.1 In-depth architecture 64](#_Toc416085669)

[15.2 Existing jobs 64](#_Toc416085670)

[15.3 Create a new job 64](#_Toc416085671)

[16 API TODO 64](#_Toc416085672)

[16.1 Streams schema 64](#_Toc416085673)

[16.2 Entities and methods 64](#_Toc416085674)

[16.3 Formats 64](#_Toc416085675)

[16.4 Authentication 64](#_Toc416085676)

[16.5 Submit a new catalog import job 64](#_Toc416085677)

[17 Cache TODO 64](#_Toc416085678)

[17.1 Architecture (lazy caching, aspects) 64](#_Toc416085679)

[17.2 In-memory cache 65](#_Toc416085680)

[17.3 Shared cache 65](#_Toc416085681)

[18 EAV 65](#_Toc416085682)

[18.1 Database Model 65](#_Toc416085683)

[18.2 Object Model 67](#_Toc416085684)

[18.3 Load EAV TODO 67](#_Toc416085685)

[18.4 Edit EAV TODO 67](#_Toc416085686)

[18.5 Apply Filter on EAV TODO 67](#_Toc416085687)

[19 CMS for custom text pages TODO 68](#_Toc416085688)

[20 Extending the solution with custom modules TODO 68](#_Toc416085689)

# Getting started COLIN

## Before you start (technical requirements)

## Download and install on IIS (scripts)

## Source code getting started (compile front, backoffice and services)

# Architecture

## Global schema

Client (Browser)

Front

Client (Browser)

ABO

Client

API

ASP.NET MVC

ASP.NET WEB API

HTML (Razor)

JSON/XML/CSV

NHibernate

SQL Server

## Web applications: Front, ABO, API (ASP.net MVC and ASP.net WEB API)

### Front Office and ABO: ASP.net MVC 4.5

#### Classic ASP.NET MVC structure:



**Presentation**: In this part there as 3 layers and an utility. When http request arrives, the first layer summoned by ASP.NET MVC is the Controller. The controller part is to get models domain by the Domain Services in the domain layout. It sends those models to the view (View Model). The conversion between Domain Models and View Model is usually done by Mappers.

**Domain**: The most important layout. Domain Services provides the data access under the form of Domain Models. Domain Models contains the business logic of the application. Communication between the logic and the persistence part is done by the Repositories.

**Persistence**: It’s the part where the data’s are persist. They can be persisting in a relational database, XML files, in web services etc …. Conversion between Domain Model and storage model is done by Data Mappers.

**Infrastructure**: This part contains all the services used by the others such as security, logging and caching.

#### Our ASP.NET MVC structure:



Compared to the classic MVC structure, there is a business layout to process the business validation. The Data Access Model is the equivalent of the Domain services it is associated with the NHibernate ORM that will be describe above.

The Structure framework is used for IOC, it will be describe lower.

### API: ASP.net API



**API:** Accepting requests from any website, and also available as .ddl to use in any windows/web application.

**Services:** Accepting request only from API. Business Logic and database fetching methods should be written here.

**Domain:** Above.

**Persistence:** Above.

## Technologies and frameworks

### ORM: NHibernate

NHibernate is an object-relational mapping (ORM) solution for the Microsoft .NET platform. It provides a framework for mapping an object-oriented domain model to a traditional relational database.

Through Nhibernate it is possible to use linq langage to make request on the database, the framework will build the SQL request (it is possible to see the sql request generate by NHbiternate with NHibernate Profiler).

Follow [this link](http://www.codeproject.com/Articles/21122/NHibernate-Made-Simple) to understand how NHibernate really works.

Nhibernate mapping files, written in xml are in this following path: MPf.DAO.NHibernate\Mappings (except for modules which have their own Mapping repository). Each files map a domain Model with a table and this is how they look like:

<?xml version="1.0" encoding="utf-8" ?>

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" namespace="MPf.Domain" assembly="MPf.Domain" >

<class name="GeocodingInfo" table="dbo.members\_geocoding">

<!--<cache usage="nonstrict-read-write" />-->

<id name="ID" column="members\_geocoding\_id" unsaved-value="0">

<generator class="native" />

</id>

<property name="UserId" column="members\_geocoding\_userid" not-null="true"/>

<property name="Country" column="members\_geocoding\_country" not-null="true"/>

<property name="AdministrativeAreaLevel2" column="members\_geocoding\_administrative\_area\_level\_2"/>

<property name="AdministrativeAreaLevel1" column="members\_geocoding\_administrative\_area\_level\_1"/>

<property name="Locality" column="members\_geocoding\_locality"/>

<property name="PostalCode" column="members\_geocoding\_postal\_code"/>

</class>

</hibernate-mapping>

Mpf.DAO.NHibernate\Mappings\GeocodingInfo.hbm.xml

Although request in NHibernate are written with linq, it doesn’t support all request. So, to work around those few exceptions or to improve the performance indeed, it is also possible to write SQL request and execute them, ex: MPf.DAO.NHibernate.ProductDAO. It’s also possible to call TSQL function and store procedure. Calling stored procedure or TSQL function can improve the performance strongly (but it is harder to maintain). You can also extend linq request in order to use custom commands all over the DAO’s. It was done in MPf.DAO.NHibernate/Extensions folder where “DatesGreaterThan” and “ParseDate” has been added ([here](https://devio.wordpress.com/category/nhibernate/) another sample).

NHibernate Session and NHibernate SessionFactory are managed by the SessionWrapper and the SessionFactoryWrapper. Their lifecycles, as all business Models and DAO models are managed by IOC through the Structure Map framework.

### Dependency injection/instances lifecycle: [Structure map](http://docs.structuremap.net/index.html)

Structure Map allows us to create dependence between classes. To make that, interfaces will be manipulated instead of classes. So, classes have a constructor with parameters: interfaces of the classes which it depends:

public class GeocodingInfoBusiness : Contract.IGeocodingInfoBusiness

{

IGeocodingInfoDAO \_geocodingInfoDAO;

public GeocodingInfoBusiness(IGeocodingInfoDAO geocodingInfoDAO)

{

\_geocodingInfoDAO = geocodingInfoDAO;

}

MPf.Business/GeocodingInfoBusinss.cs

At compile, Structure Map will load an instance of the classes linked with the interface. Interface and classes are linked in registry files:

For<DAO.IGeocodingInfoDAO>().LifecycleIs(Lifecycles.GetLifecycle(instanceScope)).Use<GeocodingInfoDAO>();

MPf.DAO.NHibernate/IOC/Registry.cs

The instanceScope that we’re using is Hybrid.

Here are the different lifecycles that Structure Map support:

**PerRequest**: The default operation. A new instance will be created for each request.

**Singleton**: A single instance will be shared across all requests

**ThreadLocal**: A single instance will be created for each requesting thread. Caches the instances with ThreadLocalStorage.

**HttpContext**: A single instance will be created for each HttpContext. Caches the instances in the HttpContext.Items collection.

**Hybrid**: Uses HttpContext storage if it exists otherwise uses ThreadLocal storage.

**HttpSession**: A single instance will be created for each HttpSession. Caches the instances in the HttpContext.Session collection. Use with caution.

**Unique**: A new object instance is created every time, even within the same object graph

**Transient**: Only one object instance will be created for the currently executing Thread

### Windows service and jobs: [Quartz.NET](http://www.quartz-scheduler.net/)

Quartz .Net is a .NET library of a scheduling framework. It is used in our solution to execute jobs with a windows service. It is included in the MPf.Service.Job.Windows project, the MPf.Service.job project and in the MPf.Batch.Scheduler.Quartz project.

In the solution, we use Quartz to launch a service which gets the programmed jobs in database. We also used the scheduler to execute the jobs at their corresponding CRON.

For the service, the configuration is set in the MPf.Service.Job.Windows.exe.config :

|  |
| --- |
| <quartz>  <add key="quartz.scheduler.instanceName" value="@quartz.scheduler.instanceName@"/>  <add key="quartz.scheduler.jobFactory.type" value="@quartz.scheduler.jobFactory.type@" />  <add key="quartz.threadPool.type" value="@quartz.threadPool.type@"/>  <add key="quartz.threadPool.threadCount" value="@quartz.threadPool.threadCount@"/>  <add key="quartz.threadPool.threadPriority" value="@quartz.threadPool.threadPriority@"/>  <!-- SERVER PROPERTIES -->  <add key="quartz.scheduler.exporter.type" value="@quartz.scheduler.exporter.type@"/>  <add key="quartz.scheduler.exporter.port" value="@quartz.scheduler.exporter.port@"/>  <add key="quartz.scheduler.exporter.bindName" value="@quartz.scheduler.exporter.bindName@"/>  <add key="quartz.scheduler.exporter.channelType" value="@quartz.scheduler.exporter.channelType@"/>  <add key="quartz.scheduler.exporter.channelName" value="@quartz.scheduler.exporter.channelName@"/>  <!-- JOBSTORE CONFIGURATION -->  <add key="quartz.jobStore.type" value="@quartz.jobStore.type@"/>  <add key="quartz.jobStore.useProperties" value="@quartz.jobStore.useProperties@"/>  <add key="quartz.jobStore.dataSource" value="@quartz.jobStore.dataSource@"/>  <add key="quartz.jobStore.tablePrefix" value="@quartz.jobStore.tablePrefix@"/>  <add key="quartz.jobStore.lockHandler.type" value="@quartz.jobStore.lockHandler.type@"/>  <add key="quartz.dataSource.default.connectionString" value="@mpf\_connectionstring@"/>  <add key="quartz.dataSource.default.provider" value="@quartz.dataSource.default.provider@"/>  <add key="quartz.plugin.repository.type" value="MPf.Batch.Scheduler.Qwartz.Plugins.RepositorySchedulingProcessorPlugin, MPf.Batch.Scheduler.Qwartz"/>  <add key="quartz.plugin.repository.scanInterval" value="10" />  </quartz> |

Quartz schedules jobs but in our solution we also use IBatch interface to create them, that’s why we have created a specific job factory where the type is specified in the parameter quartz.scheduler.jobFactory.type.

quartz.dataSource parameters are the information for the connection to the database.

quartz.jobStore.dataSource and quartz.jobStore.tablePrefix are the parameters which specify where are the Quartz’s tables. For more information on quartz.jobstore configuration, you can [click here](http://www.quartz-scheduler.net/documentation/quartz-2.x/tutorial/job-stores.html).

Example of a Quartz configuration :

|  |
| --- |
| <quartz>  <add key="quartz.scheduler.instanceName" value="APIQuartzSchedulerTest" />  <add key="quartz.scheduler.jobFactory.type" value="MPf.Batch.Scheduler.Qwartz.IOC.IOCJobFactory, MPf.Batch.Scheduler.Qwartz" />  <add key="quartz.threadPool.type" value="Quartz.Simpl.SimpleThreadPool, Quartz" />  <add key="quartz.threadPool.threadCount" value="10" />  <add key="quartz.threadPool.threadPriority" value="2" />  <!-- SERVER PROPERTIES -->  <add key="quartz.scheduler.exporter.type" value="Quartz.Simpl.RemotingSchedulerExporter, Quartz"/>  <add key="quartz.scheduler.exporter.port" value="555"/>  <add key="quartz.scheduler.exporter.bindName" value="QuartzScheduler"/>  <add key="quartz.scheduler.exporter.channelType" value="tcp"/>  <add key="quartz.scheduler.exporter.channelName" value="httpQuartz"/>-->  <!-- JOBSTORE CONFIGURATION -->  <add key="quartz.jobStore.type" value="Quartz.Impl.AdoJobStore.JobStoreTX, Quartz"/>  <add key="quartz.jobStore.useProperties" value="true"/>  <add key="quartz.jobStore.dataSource" value="default"/>  <add key="quartz.jobStore.tablePrefix" value="dbo.QRTZ\_"/>  <add key="quartz.jobStore.lockHandler.type" value="Quartz.Impl.AdoJobStore.UpdateLockRowSemaphore, Quartz" />  <add key="quartz.dataSource.default.connectionString" value="Server=YOURSERVER;Database=YOURDATABASE;Uid=YourUserId;Pwd=yourpassword"/>  <add key="quartz.dataSource.default.provider" value="SqlServer"/>  </quartz> |

### The other frameworks

#### Postsharp

Postsharp is a weaver which interfer as post-compiler. It allows using AOP in your C# code with a very simple and intuitive syntax.

Example: apply the SerializableAttribute to a class

[Serializable]

public class Merchant : Account

{

//Primary Key

public override Guid UserId { get; set; }

MPf.Domain.Merchant

\* Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

#### [log4net](http://logging.apache.org/log4net/)

log4net is a tool to help the programmer output log statements to a variety of output targets. In case of problems with an application, it is helpful to enable logging to find where the problem can be located. With log4net it is possible to enable logging at runtime without modifying the application binary.

In our solution, log4net is used to collect logging information generated by NHibernate. The configuration is set in the web.config:

<log4net>

<!-- Define some output appenders -->

<appender name="trace" type="log4net.Appender.TraceAppender, log4net">

<layout type="log4net.Layout.PatternLayout,log4net">

<param name="ConversionPattern" value="%d{ABSOLUTE} %-5p %c{1}:%L - %m%n"/>

</layout>

</appender>

<appender name="console" type="log4net.Appender.ConsoleAppender, log4net">

<layout type="log4net.Layout.PatternLayout,log4net">

<param name="ConversionPattern" value="%d{ABSOLUTE} %-5p %c{1}:%L - %m%n"/>

</layout>

</appender>

<appender name="eventViewer" type="log4net.Appender.EventLogAppender">

<!-- ATTENTION: verifier que cette application existe dans le event viewer, si l'application n'a pas les autorisations pour en creer une nouvelle -->

<logName value="MPf"/>

<applicationName value="@application@" />

<layout type="log4net.Layout.PatternLayout,log4net">

<param name="ConversionPattern" value="%d{ABSOLUTE} %-5p %c{1}:%L - %m%n"/>

</layout>

</appender>

<appender name="exceptionFileAppender" type="log4net.Appender.RollingFileAppender">

<param name="File" value="@log.file.path@\@application@\errors-file.txt" />

<param name="AppendToFile" value="true" />

<rollingStyle value="Size" />

<maxSizeRollBackups value="10" />

<maximumFileSize value="10MB" />

<staticLogFileName value="true" />

<layout type="log4net.Layout.PatternLayout,log4net">

<param name="ConversionPattern" value="%d{ABSOLUTE} %-5p %c{1}:%L - %m%n"/>

</layout>

</appender>

<appender name="logFileAppender" type="log4net.Appender.RollingFileAppender" >

<param name="File" value="@log.file.path@\@application@\log-file.txt" />

<param name="AppendToFile" value="true" />

<rollingStyle value="Size" />

<maxSizeRollBackups value="10" />

<maximumFileSize value="10MB" />

<staticLogFileName value="true" />

<layout type="log4net.Layout.PatternLayout,log4net">

<param name="ConversionPattern" value="%d{ABSOLUTE} %-5p %c{1}:%L - %m%n"/>

</layout>

</appender>

<root>

<priority value="WARN"/>

<appender-ref ref="trace"/>

</root>

<logger name="ExceptionLogger">

<level value="ALL"/>

<appender-ref ref="exceptionFileAppender"/>

<appender-ref ref="eventViewer"/>

</logger>

<logger name="InfoLogger">

<level value="ALL"/>

<appender-ref ref="logFileAppender"/>

<appender-ref ref="eventViewer"/>

</logger>

</log4net>

#### [Automapper](https://github.com/AutoMapper/AutoMapper)

As it’s called, AutoMapper is used to get rid of code that maps an object to another.

namespace MPf.Web.Front.Models

{

public class Mapper

{

static Boolean init = false;

public static T Map<TSource, T>(TSource source)

{

if (!init)

{

//… codes before

//map the attributes with the same name in both Models

AutoMapper.Mapper.CreateMap<Domain.GeocodingInfo,Models.Shops.GeocodingInfoModel>();

//map the attributes when the names are different

AutoMapper.Mapper.CreateMap<Domain.Product, Models.Wishlist.WishlistProduct>()

.ForMember(dest => dest.Price, o => o.MapFrom(x => x.BestPrice))

.ForMember(dest => dest.Brand, o => o.MapFrom(x => x.Manufacturer.Name))

.ForMember(dest => dest.Name, o => o.MapFrom(x => x.Model))

.ForMember(dest => dest.PricePromo, o => o.MapFrom(x => x.BestPromoPrice))

//… codes after

init = true;

}

return AutoMapper.Mapper.Map<TSource, T>(source);

}

}

}

MPf.Web\Front\Models\Mapper.cs

Sample call:

MPf.Domain.GeocodingInfo DGeoInfo = Domain.GeocodingInfo();

MPf.Web.Front.Models.Shops.GeocodingInfoModel GeoInfos =

MPf.Web.Front.Models.Mapper.Map<MPf.Domain.GeocodingInfo, MPf.Web.Front.Models.Shops.GeocodingInfoModel>(DGeoInfo);

//with list

IList<MPf.Domain.GeocodingInfo> DGeoInfos = new List<MPf.Domain.GeocodingInfo>();

List< MPf.Web.Front.Models.Shops.GeocodingInfoModel> GeoInfos =

MPf.Web.Front.Models.Mapper.Map<IList<MPf.Domain.GeocodingInfo>,

List<MPf.Web.Front.Models.Shops.GeocodingInfoModel>>(DGeoInfos);

#### [NewtonSoft](http://www.newtonsoft.com/json)

It’s a .NET Framework to serialized and deserialize JSON Objects.

Serialization Sample:

Product product = new Product();

product.Name = "Apple";

product.Expiry = new DateTime(2008, 12, 28);

product.Sizes = new string[] { "Small" };

string json = JsonConvert.SerializeObject(product);

// {

//   "Name": "Apple",

//   "Expiry": "2008-12-28T00:00:00",

//   "Sizes": [

//     "Small"

//   ]

// }

Deserialization sample :

string json = @"{

  'Name': 'Bad Boys',

  'ReleaseDate': '1995-4-7T00:00:00',

  'Genres': [

    'Action',

    'Comedy'

  ]

}";

Movie m = JsonConvert.DeserializeObject<Movie>(json);

string name = m.Name;

Source <http://www.newtonsoft.com/json>

#### [Razorengine](https://github.com/Antaris/RazorEngine)

With this framework, it is possible to use razor.

Sample:

<a href="@Model.Link">@Model.Name</a>

#### [Rhino Mocks](http://hibernatingrhinos.com/oss/rhino-mocks)

Unit test

#### [Couchebase](http://www.couchbase.com/)

Cache MPf.Cache.CacheProvider.CouchbaseCacheProvider

#### [CSVHelper](http://joshclose.github.io/CsvHelper/)

CSVHelper is a .NET library for reading and writing CSV files. This package is mainly use to import/export catalogs.

#### [MbUnit](http://www.codeproject.com/Articles/6060/MbUnit-Generative-Unit-Test-Framework)

Unit test

#### [StemmersNet](https://stemmersnet.codeplex.com/)

Stemmer is a framework which is used to determine related word. It is used in our solution on import catalog jobs in order to generate tag on a product when it’s import.

MPf.Batch\Batchs\CatalogueJob.cs

## Windows service and jobs (Quartz.NET)

Jobs are executed by a windows service which allows you to automatically execute tasks on your marketplace. They work with Quartz, an open source job executing system. This last one allows you to schedule job executions with a chosen CRON to program the executing at a specific date.

To generate the windows service executable, you need to add the Mpf.Batch assembly in your project. It contains preprogrammed batchs you can use to import products or other datas in your database. For more details, you can refer to the part 15.2. This one allows you to create your own batch by implementing the IBatch interface.

MPf.NHibernate.DAO assembly is also required because the MPf.Batch needs to access to the database to get access to the batch tables describe in the section 15.1. We advise you to use the MPf.Service.Job.Windows project, it is the windows service which starts the batchs by launching the Quartz scheduler. Note that the MPf.Service.Job.Windows requires the MPf.Service.Job to create the service.

To install and to start the service, you need to create a windows service on your server, which executes the executable file generated “MPf.Service.Job.Windows.exe”.

## Database and ORM (mappings, HILO for ID generation, LINQ)

**NHibernate classes**

The ORM used in the solution is NHibernate, the classes related to him are available in the MPf.DAO.NHibernate assembly. All of them implements their corresponding interface in the MPf.DAO assembly. For example ComplaintDAO implements IComplaintDAO interface.

**Mapping**

As explained above, NHibernate needs to be mapped to find tables in your database. It was made by XML files in MPf.DAO.NHibernate.Mapping folder. Each table has a XML file which rely it on the DAO class.

**Hilo**

In order to increase performance, we use HILO for the primary key generation on several tables. Hilo is composing the identifier from two parts, named high and low. You can find the next high value for each table in the nhibernate\_hilo table.

Tables using HILO:

* products
* quantities
* shipping\_delivery\_modes
* manufacturers
* categories\_link
* shipping\_merchant\_products
* batch\_catalog\_item
* products
* merchant\_products

**LINQ**

In the most of case, we use LINQ requests to insert, get and delete datas in tables. The DAO classes respect the contract of IDAO, so they contains save(), get(), getAll() methods. Note you can specify criteria with the associated criteria classes available in MPf.DAO.Criteria for the get methods.

## Shared components (DAO, business, media manager)

All business and DAO follow this architecture and share all over the solution (Front/Admin/Jobs).



Domain

DAO

Business

### Business

Business classes are doing the business validation and they will call DAO’s classes to get/update/save data’s. Each domain classes don’t have their own business, a same business, when it’s relevant may do the validation and call the DAO for several domain classes.

Each Business classes implements his interface. Each of them has his own select criteria’s which inherits from MPf.Business.Criteria.Select. This class contains the basic criterias, such as the sort parameters, paginations, or EAV filters. Those criterias are transmitted to the DAO, usually there is a method called “ApplyCriteria”, in each business which operates the transmission (in the case of the business and the DAO contain the same attributes, this method may not exist and the AutoMapper is used instead).

### DAO

DAO classes implement their own interface. All those interfaces inherit from the IDAO interface which contains the essential method that each DAO must have (save/delete/get). Each domain class must have his own DAO class. The requests to get retrieve data from the database are built here in linq language by using the DAO criteria’s.

## Database???

## Localization

In the solution, you have several parameters related to the localization.

### Globalization parameters

In the marketplace, you can set the localization for culture dependent functions such as date or numbers or resources files. In your web.config, you can complete the parameter globalization which contains two attributes:

* culture, for dependent functions (date, numbers, …)
* uiculture, to select ressources files in your MPf.Web.Front.Themes.{your theme}.Ressources folder. These last ones correspond to the resource files where all static texts are saved.

Example:

|  |
| --- |
| <globalization culture="fr-FR"  uiCulture="fr-FR"  fileEncoding="utf-8"  responseEncoding="utf-8"  requestEncoding="utf-8"  /> |

### Mail language

In your web.config, you can complete the uiculture parameter in appSettings section, it corresponds to the parameters which selects the folder mail language for your mail templates. By default for the front office, the templates in your MPf.Web.Front.Themes.{your theme}.Mails are selected if no mail templates are found in your MPf.Web.Front.Themes.{your theme}.Mails.{your appSettings uiCulture} folder.

Example:

|  |
| --- |
| <appSettings>  <add key="uiCulture" value="fr-FR"/>  </appSettings> |

### HTML

The localization for your HTML Lang attribute is set in a localization parameter of the marketplace available in the parameters table. The parameter name is htmlLang and its category is localization. We advise you to call this parameter in your view page with this following way:

|  |
| --- |
| @{  String htmlLang = MPf.Web.Services.Parameterization.GetParameter<String>("htmlLang", "localization", "en");  }  <!DOCTYPE html>  <html lang="@htmlLang">  […]  </html> |

# Compilation and deployment Colin

## Compile using visual studio

## Package using our NANT scripts (msbuild based)

## Configure test environment and production parameters

## Deploy using powershell

# Coding standards

## Formatting

### Source files layout

Source files have the following ordering:

* Using statements
* Class declaration (with namespace in usually)
* Interfaces attributes
* Common attributes
* Constructor
* Parameters attributes (inside a region called “Parameters”). **Each parameter is separated by a line break.**
* Methods declaration.

### Indentation & Braces

Braces, when necessary, are placed directly below and aligned with the statement that begins a new scope of execution.

### Line Length

No lines longer than 140 characters since they're not handled well by many terminals and tools.

### Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines are always be used in the following circumstances:

* Between sections of a source file
* Between class and interface definitions

One blank line is always used in the following circumstances:

* Between methods
* Between the local variables in a method and its first statement
* Before a block or single-line comment
* Between logical sections inside a method to improve readability

### Blank Spaces

Blank spaces are used in the following circumstances:

         A keyword followed by a parenthesis is separated by a space. Example:

while ( true ) {

...

}

Note that a blank space must be used between a method name and its opening parenthesis. This convention helps to distinguish keywords from method calls.

         A blank space must appear after commas in argument lists.

         All binary operators must be separated from their operands by spaces. Blank spaces are never separated unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

a += c + d;

a = ( a + b ) / ( c \* d );

while ( d++ = s++ ) {

n++;

}

printSize( "size is " + foo + "\n" );

         The expressions in a for, foreach, or if statement are separated by blank spaces. Example:

for ( expr1; expr2; expr3 )

## Naming

Follow the standard [Naming Guidelines](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnamingguidelines.asp) set by the .NET framework team by using only three capitalization styles: **Pascal**, **Camel**, and **Upper** casing.

Examples:

|  |  |  |
| --- | --- | --- |
| Identifier Type | Capitalization Style | Example(s) |
| Abbreviations | Upper | ID, REF |
| Namespaces | Pascal | AppDomain, System.IO |
| Classes & Structs | Pascal | AppView |
| Constants & Enums | Pascal | TextStyles |
| Interfaces | Pascal | IEditableObject |
| Enum values | Pascal | TextStyles.BoldText |
| Property | Pascal | BackColor |
| Variables, and Attributes | Pascal (public)  Camel (private, protected, local) | WindowSize  windowWidth, windowHeight |
| Methods | Pascal (public, private, protected)  Camel (parameters) | ToString()  SetFilter(string filterValue) |
| Local Variables | Camel | recordCount |

Guidelines:

In **Pascal** casing, the first letter of an identifier is capitalized as well as the first letter of each concatenated word. This style is used for all public identifiers within a class library, including namespaces, classes and structures, properties, and methods.

In **Camel** casing, the first letter of an identifier is lowercase but the first letter of each concatenated word is capitalized. This style is used for private and protected identifiers within the class library, parameters passed to methods, and local variables within a method.

**Upper** casing is used only for abbreviated identifiers and acronyms of four letters or less.

# Parameters



Parameters are managed in the “MPf.Parameterization” assembly. They are giving the possibility to change easily some behavior (ex: amount of products on some page, showing or not sold out products etc…) on the marketplace. When they exist, the parameters are declared in a region on the top of the class. They can have 3 different types: integer, String or Boolean.

## Parameters system (attributes, and table)

### Database Model



parameters\_id: primary key.

parameters\_category: the class in which the parameters are used (ex: ProductBusiness, BackOfficeMerchantController).

parameters\_context: It precises the scope of the parameters. It can have 3 values, “API”, “Front” or ”BOAdmin”. This parameter is facultative, if it’s empty, the value of the parameter will be the same for the 3 projects.

parameters\_type: “TextBox” if the parameter is a String or an Integer. “CheckBox” if it’s a Boolean.

parameters\_content: ???

parameters\_value: value of the parameter

parameters\_description: description of the parameters (usually null)

### Object Model



ParameterizationAttribute: It’s the parameter Model which is mapped with the database table dbo.parameters.

InterceptionRegistry: parameters structure map registry.

Datareflector: implements 2 interfaces:

* IParameterizationService which contains the method GetParameter. This method is getting loading the parameter from the database or creating it, if it does not exist yet.
* IDataReflector which contains the method fillObject. This method is setting the value of the parameters in the different classes.

**How to create a parameter:**

To create a parameter in the parameters table, you just need to use the attribute Parameterization above a field of the class where you want to use the parameter, you can see an example below:

[Parameterization("DeliveryZoneMandatoryToShow", "OfferBusiness", "true")]

public Boolean DeliveryZoneMandatoryToShow { get; set; }

DeliveryZoneMandatoryToShow *is the name of the parameter, it has to be the same then the attribute which follows.*

OfferBusiness *is the* parameters\_category

*“true” is the default value (avoid majuscule when it’s a Boolean or the parameters\_type will be a textbox instead of a checkbox).*

After the next compilation, when you will launch the solution, the parameter will be added automatically in the parameters table of your database thanks to the FillObject() method of DataReflector class. This last one is called by InterceptorIOC class.

## Change parameters value in ABO



Parameters can be edit from the ABO: Administration->Parameters. There’s cache on parameters (refresh every 2hours). If you want you want to change the behavior immediately you have to refresh it.

# Themes

## Theme mechanism: custom view engine

The theme mechanisms are managed in the MPf.Web project by the ThemeViewEngine class which inherits of RazorViewEngine. It is added in the ViewEngines of the MVC framework by the method Application\_Start() of the MVCApplication class in the Global.asax.

The constructor of the ThemeViewEngine class defines the VirtualPathProviderViewEngine properties. These properties are used by a view engine which is called by the MVC framework to render view pages. These parameters will be used by the MVC framework to make view calls in controllers for example by using View() method.

**Properties**

* MasterLocationFormats: contains the locations where the view engine will search to find a master view in an area-enabled application. By default, in the front office the path is :
  + - ~/Themes/{your theme’s name}/Views/Shared/{0}.cshtml
* PartialViewLocationFormats: contains the locations where the view engine will search to find a partial view. By default in the front office, the values are :
  + - ~/Themes/{your theme’s name}/Views/{1}/{0}.cshtml
    - ~/Themes{your theme’s name}/Views/Shared/{0}.cshtml
  + ViewLocationFormats: contains the locations where the view engine will search to find a view. By default in the front office, the path is:
    - ~/Themes/{your theme’s name}/Views/{1}/{0}.cshtml
  + AreaPartialViewLocationFormats: contains the locations where the view engine will search to find a partial view in an area-enabled application. By default in the front office, the paths are:
    - ~/Themes/{your theme’s name}/Modules/{2}/Views/{1}/{0}.cshtml
    - ~/Modules/{your theme’s name}/Views/{1}/{0}.cshtml
  + AreaViewLocationFormats: contains the locations where the view engine will search to find a view in an area-enabled application. By default in the front office, the paths are:
    - ~/Themes/{your theme’s name}/Modules/{2}/Views/{1}/{0}.cshtml
    - ~/Modules/{2}/Views/{1}/{0}.cshtml

## Emails templates



Mails are managed in the MPf.Message project. It is possible to create custom emails for your theme, those one have to be stored in a folder, named “Mails”, at the root of the theme. When an email is sent, the method LoadTemplate of the MPf.Message.Sender will load the template corresponding to the theme.

The emails content is dynamic, the Send method of the Sender is taking a generic Model as parameter and build emails considering it. You can manipulate models in your emails templates in razor, as for common views.

## Custom helpers

You can create HTML helpers which can be used in your views. We advise you to put them in a MPf.Web.Front.Themes.{your theme’s name}.Helpers folder.

## Getting started on a custom theme creation

# Security

## Database Model



## Object Model



## Users and roles model

The users and roles models are following the asp.net membership architecture of the ASP.NET framework.

The Membership and Roles of the frameworks are designed such that a single user and role store can be shared among many different applications. An ASP.NET application that uses the Membership or Roles frameworks must specify what application partition to use. In short, multiple web applications can use the same user and role stores.

The aspnet\_Applications table defines these partitions. Each application that uses the database to store user account information is represented by a row in this table. The aspnet\_Applications table has four columns: ApplicationId, ApplicationName, LoweredApplicationName, and Description. ApplicationId is of type uniqueidentifier and is the table's primary key; ApplicationName provides a unique human-friendly name for each application.

The other Membership and Role-related tables link back to the ApplicationId field in aspnet\_Applications. For example, the aspnet\_Users table, which contains a record for each user account, has an ApplicationId foreign key field; ditto for the aspnet\_Roles table. The ApplicationId field in these tables specifies the application partition, the user account or role belongs to.

In our solution you will find 6 different roles:

* Merchant
* Customer
* Administrator
* Super-merchant
* Salesman
* Operator

## User Account Information

User account information is housed in two tables: aspnet\_Users and aspnet\_Membership. The aspnet\_Users table contains fields that hold the essential user account information. The three most pertinent columns are:

* UserId
* UserName
* ApplicationId

UserId is the primary key (and of type uniqueidentifier). UserName along with the password, makes up the user's credentials (A user's password is stored in the aspnet\_Membership table). ApplicationId links the user account to a particular application in aspnet\_Applications. There is a composite UNIQUE constraint on the UserName and ApplicationId columns. This one ensures that in a given application each UserName is unique, yet it allows for the same UserName to be used in different applications.

The aspnet\_Membership table includes additional user account information, like the user's password, email address, the last login date and time, and so forth. There is a one-to-one correspondence between records in the aspnet\_Users and aspnet\_Membership tables. Like the aspnet\_Users table, aspnet\_Membership includes an ApplicationId field that ties this information to a particular application partition.

## Securing Passwords

Password information is stored in the aspnet\_Membership table. The SqlMembershipProvider allows passwords to be stored in the database using one of the following three techniques:

**Clear** - the password is stored in the database as plain-text. I strongly discourage using this option. If the database is compromised - be it by a hacker who finds a back door or a disgruntled employee who has database access - every single user's credentials are there for the taking.

**Hashed** - passwords are hashed using a one-way hash algorithm and a randomly generated salt value. This hashed value (along with the salt) is stored in the database.

**Encrypted** - an encrypted version of the password is stored in the database.

The password storage technique used depends on the SqlMembershipProvider settings specified in Web.config. We will look at customizing the SqlMembershipProvider settings in Step 4. The default behavior is to store the hash of the password.

The columns responsible for storing the password are Password, PasswordFormat, and PasswordSalt. PasswordFormat is a field of type int whose value indicates the technique used for storing the password: 0 for Clear; 1 for Hashed; 2 for Encrypted. PasswordSalt is assigned a randomly generated string regardless of the password storage technique used; the value of PasswordSalt is only used when computing the hash of the password. Finally, the Password column contains the actual password data, be it the plain-text password, the hash of the password, or the encrypted password.

Table below illustrates what these three columns might look like for the various storage techniques when storing the password *MySecret!*

|  |  |  |  |
| --- | --- | --- | --- |
| Storage Technique<\_o3a\_p /> | Password<\_o3a\_p /> | PasswordFormat<\_o3a\_p /> | PasswordSalt<\_o3a\_p /> |
| Clear | MySecret! | 0 | tTnkPlesqissc2y2SMEygA== |
| Hashed | 2oXm6sZHWbTHFgjgkGQsc2Ec9ZM= | 1 | wFgjUfhdUFOCKQiI61vtiQ== |
| Encrypted | 62RZgDvhxykkqsMchZ0Yly7HS6onhpaoCYaRxV8g0F4CW56OXUU3e7Inza9j9BKp | 2 | LSRzhGS/aa/oqAXGLHJNBw== |

*[Source:* [*http://www.asp.net/web-forms/overview/older-versions-security/membership/creating-the-membership-schema-in-sql-server-cs*](http://www.asp.net/web-forms/overview/older-versions-security/membership/creating-the-membership-schema-in-sql-server-cs)*]*

Merchants and customers password are encrypted and admin passwords are stored in clear in the database.

## Authentication mechanism

The class MPf.Security.FormsAuthenticationService has Authentication methods which manage form authentication, when a user has to enter his username and his password. It is possible to bypass the authentication by setting the web.config the value of the appsetting security.bypass.authentication to true. Although merchants and admin log on by their username plus their password, customers authenticating log on by their email and their password.

The class LogonAuthorize manages authorizations once a user is log on.

The class MPf.Security.APIAuthorize manages the security of API’s.

On the front side, the AuthenticationController inherited of LayoutController class which contains the methods to login and logout users. The controller has methods to redefine password or send a new password. These controllers call the IAuthenticationService interface from the MPf.Security project in order to use the methods for the authentication.

In the MPf.Security project, FormsAuthentificationService implements the IAuthentification interface. That’s why this class manages the users with its methods.

**FormsAuthentificationService’s methods:**

* Authenticate(string username) : log in user with his username
* Authenticate(string username, string password): log in user with username and password (create cookie)
* AuhtenticateNoCookie(string username, string password): log in user with username and password without creating cookie and return session id
* Logout(): log out user
* PasswordForgotten(string username, out bool? reset): send the password user if it finds it else it sends and changes a new generated password of eight characters
* GetUser(): return the membership user by using cookie
* GetUser(string username): return the membership user by using username parameter
* GetUser(Guid userid): return the membership user by using user id
* IsAuthenticated(): return true if the user is connected
* IsUserInRole(String rolename): return true if the rolename parameter corresponds to the connected user role
* GeneratePassword(): return a generated password of eight characters which contains two non-alphanumeric characters
* GetRoleForUser(Guid userId): return the user role corresponding to the user id parameter
* GetRolesForUser(String username): return the list of the user roles

# Account model TO COMPLETE

## Database Model



## Object Model



AccountBusiness will be in charge to make the validation (as checking if the username or email is already used or not) when an account is created or edited.

The AccountDAO communicates with the database, it will read/write on the different tables concerned by the account, merchant or customer and the corresponding asp.NET tables. You won’t find any DAO for asp.NET table because we use the model of the framework, so the different methods which manage those tables are included.

BaseModel has Patchs and EAV attributes, shared by every entity, so all of them inherit of it.

## Merchants

### Database Model



### Object Model



By default a merchant, when it is created does not have his account activated. The “activation” considers 2 parameters, the field is\_approve of the table aspnet\_membership and the field active\_status of the merchant table. Both of them have to be set to true without what, the offer of this merchant won’t appear on the front.

When a merchant is created threw the ABO or the Front (MerchantRegistation page), an email is sent to the merchant, asking him to accept the Terms of Use. Once it’s done, the field is\_approve is set to true and the merchant can log to his back office. The active\_status is given by the ABO on the edit merchant page.

## Customers

### Database Model



### Object Model



Customers are enabled by default. To authenticate, they have to use their email. By default, after registration, a customer is not log on, except on the purchasing tunnel. If you want them to be logged after a registration, set the parameter AutoAuthenticate to true.

## Newsletter

### Database Model



### Object Model



The newsletters table contains email and phone number which are registered to the newsletter. Thanks to the NewsLetterUsersBusiness, you have the RegisterMailOrPhone method which allows you to save a mail, a phone number or the both information in the newsletter table. Note that the newsletter\_email\_status or the newsletter\_phone\_status fiels contains a false value if the email address or the phone number is null or empty or only composed of white spaces.

# Catalog model

## Global database model



## Products

### Database Model



 All fields won’t be details because most of them are enough explicit:

**Products table**

products\_id: id, primary key

products\_extern\_id: extern id, used when we import products with job, products may be informed

products\_model: title of the product

manufacturer\_id: manufacturer\_id, foreign key

products\_validation: validation status (0 pending, 1 cancel, 2 activated).

products\_source: 0 if it has been created handly (in ABO or MBO), 1 if it has been imported with a job

products\_picture: url picture of the product

userId: merchant GUID

products\_type: (0 unassigned, 1 virtual, 2 physical, 3 fake)

products\_wanted: works as a flag, “wanted products” will have a special display such as a heart.

products\_votes: amount of votes (as facebook like)

products\_priority: can be use to set the order on which products will be deplayed on the front

products\_subtype: 0 product, 1 gift card, 2 lesson, 3 credit, 4 unique service, 5 variable

products\_metakeywords: metakeywords, not intended to be displayed but used to for perform research

products\_metadescription: metadescription, not intended to be displayed but used to for perform research

### Object Model



Products, as each catalog item, can be import by using jobs or can be created by the merchant in their MBO or by administrators in the ABO. When they are created they are activated by default. The product modification table permits the moderation of a product by the merchant. After each change, a new products modification row will be created, and when administrators will accept the change(s), the product will be updated. The product modification fields represent all products fields that can be updated by the merchant.

## Offers

### Database Model



merchant\_products\_option\_name: usually color or size

merchant\_products\_status: 1 deactivate, 2 activate, 3 pending

quotation\_id: if this field is filled, it means that the offer is created to respond to a specific quotation. So only the customer and the merchant who has created this offer will be able to see it.

### Objet Model



Offer, as each catalog item, can be import by using jobs or can be created by the merchant, in their MBO or by administrators in the ABO. In this last case, when they are created, they are not activated by default. There is no moderation on offers, the merchant is free to edit them as he wants.

## Stocks

### Database Model



quantities\_id: id, primary key

merchant\_products\_id: foreign key

quantities\_quantity: quantity

quantities\_options\_values: ???

quantities\_is\_enabled: enable/disable quantities for an offer

### Object Model



Quantities, as each catalog item, can be import by using jobs or can be created by the merchant, in their MBO or by administrators in the ABO.

## Categories

### Database Model



### Objet Model



The categories\_categories table is used to add information on categories: getting all child or parents of a category, know if a category has validated product to display it or not on the front. This table is filled by a stored procedure “sp\_buildCategoriesCategories” in the CategoriesDAO. This last one is executed each time that the job CatalogJob is launched or by calling the method “UpdateCategoriesCategories()”.

The categorie\_activ\_status field gives the possibility to enable/disable a category easily. Once the category is disabled, the products belong to this category won’t appear on the front office.

## Delivery methods, zones and rates

### Database Model



### Object Model



### Zones

In the marketplace, you can define zones which will be used for your delivery rate for example. In these zones, you can put countries which are saved in the shipping\_delivery\_country. The corresponding business is the CountryBusiness class.

By default a delivery zone is defined by an alphabetic character and it is saved with the syntax “Zone {zone character}” in the shipping\_delivery\_zone table (ex: Zone A). The corresponding business is the DeliveryZoneBusiness class, you can find methods to get, count, create, save, update and delete a country.

You can find the links between the zones and the countries in the table shipping\_delivery\_country\_zone. The corresponding business is the DeliveryCountriesZoneBusiness class.

These zones are defined by the administrator in the back office administration, it’s the ZoneManaementController class which allows him to manage them. By default in the controller, you can’t put a country in many zones.

### Delivery mode

You can add delivery modes in your marketplace, the business class which manages that is the class DeliveryModeBusiness. These modes are saved in the table shipping\_delivery\_modes.

### Delivery Rate

Each offer has a specific delivery rate, because each merchant defines the delivery rate zones for his offers. The class business which manages that is the class ShippingMerchantProductBusiness and the table where the information are saved is the table shipping\_merchant\_products. By default, the fee’s value is set to 0 and the business prevents the duplicate fees for each product.

## Cross selling

There are different types of cross selling: by categories, by merchant or by products. The cross selling by categories and by merchant don’t call any special process on the page where the product is loaded. In order to practice we just have to load the products to belong to the same merchant or the same categories. Check the product controller in order to see how it is done (#region “caterogy's products” and “merchant's products”). The cross selling by product is different, it gives the possibility to the administrator to link products between them through his back office.

### Database Model



products\_link\_id: id, primary key

products\_id: product id, foreign key

linked\_product\_id: the id of the product on which it is linked

products\_link\_type\_id: at the moment it can have only one value: 1, which means cross selling. It represents the aim of the association between both products.

### Objet Model



In order to load the cross selling products, you can call the GetProductsLinkedList of the product business by passing the product id and the cross-selling type in parameters (always 1 at the moment).

## Visible Products

Even if the product is activated and has and activated offer, he has to satisfy others conditions in order to be display on the front. Refer to functional documentation for more details on this aspect.

# Geolocation

In our solution, there is 2 different types of geolocation: the classic one which is based on the distance between 2 coordinates, and the second one is based on a concept of zone set by user (usually the merchant) in which he will exercise (we’ll call this one geocoding).

## The classic geolocation

### Database Model



userId: GUID, primary key

gLatitude: Latitude

gLongitude: Longitude

### Object Model



The geolocalisationbusiness class contains methods to determine coordinates of an address. By this way, it determines the distance between a customer and the merchants, and applies filters.

Example: show merchants in a perimeter or 20km.

## Geocoding

“Geocoding search” is another type of geolocation search. This one is not based on the distance between a client and a merchant determinated with the coordinates (latitude and longitude) of both parts. With the ”Geocoding search” feature, a merchant can choose a zone (countries, departments, localities) where he can practice his activity. The research will be applied in terms of the zone he specified.

The [Google Geocoding API](https://developers.google.com/maps/documentation/geocoding/) : originally, has been designed to get coordinates from an input address but, it provides the different component of an address that will be used develop our feature.

### Database Model



Members\_geocoding\_id: id, primary key

Members\_geocoding\_userid: id of the merchant at the moment but this table can be used for all members in the future.

Members\_geocoding\_country: country

Members\_geocoding\_administrative\_area\_level\_1: region in France, states in USA

Members\_geocoding\_administrative\_area\_level\_2: department in France, county in USA

Members\_geocoding\_locality: locality

Member\_geocoding\_postal\_code: postcode

*\*The primary key is not in the userId so a user can have several geocoding records.*

### Object Model



### The “getGeocodingInfo()” method :

**Process :**

* The method taking a string as input.
* Initialize a new Geocoding domain object.
* It requesting the geocoding API using the method GeoAPICall(string URL) which is return the response of the API, in xml (follow this [link](http://maps.googleapis.com/maps/api/geocode/xml?address=Bordeaux,France) in order to have a sample).
* Once it’s done, it parses the xml response by using the System.Xml library and fills the geocoding Domain Model.
* It determines the geocoding info type by counting the amount of elements in the “address\_components” we use.
* If it’s a locality (at least 4 elements) with the postcode field filled, it returns the geocoding domain object.
* Else it makes a reverse call of the google geocoding API by using the coordinates that it gets by parsing the Xml response of the first call. Then it parses the response in order to get the postcode needed: select the “address\_component” element which has the type “postal\_code“ in the response, then get the postal\_code. Once it’s done, it returns the geocoding domain object.

\**There is a parameter called “GeoCodingCountry” in order to add a parameter when requesting the google geocoding API to improve the relevance of the result.*

### The search process

When a research is done by using the geolocation information, the getGeocodingInfo method is called in order to create a geocoding Domain Model corresponding to the search string. The domain model is used to fill the selectCriterias corresponding to the GeocodingInfos (locality, postcode, administrativearealvl1, administrativearealvl2, country).

The filter follows the logic below:

If a user is looking for a region, it will return:

- any merchants who have this region in their geocoding information.

- any merchants who have the same country but do not have any region information (it means that they offer their service in the whole country)

Here a sample with 3 different merchants:

- "A" with this GeocodingInfos: *AdministrativeArealvl1: Aquitaine / country: France*

- "B" with *AdministrativeArealvl2: Gironde / AdministrativeArealvl1: Aquitaine / country: France*

- "C" with *country: France*

If a user is looking for Dordogne, the criterias that will be generated will be: *AdministrativeArealvl2: Dordogne / AdministrativeArealvl1: Aquitaine / country: France*

“A” has the same *country / administrativeArealvl1* and does not have *AdministrativeArealvl2,* so it is found.

“B” has the same *country / AdministrativeArealvl1* but has an *AdministrativeArealvl2* which is different, so it is not found.

“C” has the same country and does not have any *AdministrativeArealvl1* and *AdministrativeArealvl2,* so it is found.

\* *A merchant can have several localities, country etc ...Cart and orders model*

# Cart: cart component, profile and new order

## Cart

### Database Model



### Object Model



Cart are stored by using asp.net profiles, you can see the field which are persist in database. Using profile is more secured and we can easily manage the time that the cart is available (it vanishes when the customer log off or after a delay which is customizable). A cart is a dictionary of CartItems, and you can see the attributes upon. Note that the cart project is using their own catalog item, they are almost the same than the common ones and mapped with them in the MPf.Cart.Model.Mapper.

## Orders and merchant orders

### Database Model



### Object Model



*\*Only domain objects appear on this figure but each of them got their DAO and their business class.*

An order can be composed by products from different merchants because the solution is multivendor, that’s why we distinguish orders and OrderMerchant. Following this Model, an order can be composed by several OrderMerchant and an OrderMerchant can be composed by several OrderProducts.

An order with the corresponding order\_merchant and order\_product are created (with the status unpaid) as soon as the user chooses a payment method to keep traceability, even if he doesn’t pay.

If the payment has been done with success, an order payment is created, the status of different orders are updated and the status history tables as well.

order\_merchant are created by default with the status waiting (customizable). Then it is up to the merchant to update the status in his MBO. Check the functional documentation for more details about the process.

## Payment modules

Payment modules project are in folder module.

### Atos

### Be2Bill

#### Database Model



### Check

### Mercanet

### Ogone

#### Database Model



### Paybox



### Payname



### Paypal

### PayU

### Systempay

### Transfer

### WebAffaires

## Merchant billing

### Database Model



### Object Model



The bill is generated when the status is set to the status for which the bill is supposed to be generated. This status is parameterizable, the name of the parameter is BillingGenerationStatus and his default value is 4 which means that the order has been shipped by the merchant. The bill is generated by the method Notify of the billing business which is not called explicitly. The method is called automatically by the IOC when the order status changes.

## Merchant commissions and payments

The table order\_merchant\_commission contains the amount of the commission reserved to the owner to the marketplace and the amount due to the merchants. The amount of the commission is specific to each merchant, it corresponds to the field merchant\_our\_commission\_rate of the merchant table. It is possible to set up a default commission rate by changing the value of the parameter DefaultCommissionRate. By this way any merchant that will register will have this commission rate.

# Comments and ratings model

## Products

### Database Model



* **comments**:

comment\_id: comment id, primary key

comment\_userid: GUID of the comment author

comment\_date: date of the comment

comment\_text: the comment message

comment\_status: status activated (1) or not (0)

* **products** (the table contains other field but they don’t concern comments):

products\_id : product id, primary key

products\_note: rate of the product

* **comments\_products** (association table)

comment\_product\_id: id, primary key

comment\_id: comment id, foreign key

moodValue: rate given by user

### Object Model



Comments on products are not loaded by default, in order to load them the attribute “GetComments” of the MPf.Business.Criteria.ProductSelect has to be set to true and then the “LoadProductComment” method of the CommentProductBusiness will be called:

if (selectCriteria.GetComments.HasValue && selectCriteria.GetComments.Value)

{

\_commentProductBusiness.LoadProductComments(listProducts, CommentState.Visible);

}

MPf.Business\ProductBusiness.cs l369

Only activated comments are loaded through this method. By default when they are added, they are already visible (comment\_status = 1). If they do not respect the website TOS, their status can be change in the ABO.

The field “products\_note” of the database is updated as soon as a comment is added.

## Merchants

### Database model



merchant\_comments\_id: id, primary key

merchant\_comments\_text: comment message

merchant\_comments\_status: comment status (0: pending, 1: Rejected, 2: Validated)

merchant\_comments\_merchant\_id: merchant GUID

merchant\_comments\_date\_created: creation date

merchant\_comments\_date\_modified: modification date

merchant\_comments\_author\_name: name of the author

merchant\_comments\_author\_email: email of the author

merchant\_comments\_author\_id: GUID of the author

merchant\_comments\_mark: mark

### Object Model



As for comments on Products, comments on merchants are not loaded by default. In order to load them, the attribute LoadMerchantValidatedComments of the MPf.Business.Criteria.MerchantSelect has to be set to true and the LoadMerchantComments method of the MerchantCommentBusiness will be called:

if (selectCriteria. LoadMerchantValidatedComments)

{

\_merchantCommentsBusiness.LoadMerchantComments(merchants, Domain.MerchantCommentsStatus.Validated);

}

MPf.Business\MerchantBusiness.cs l176

When a comment is given on a user the default status is “Pending”, so it won’t load. When a comment is given, administrators of the website are notified with an email, asking them to validate the comment or to refuse it.

## Orders

### Database Model



userId: Merchant GUID

orders\_merchant\_comment: comment message

orders\_merchant\_rating: rate, default value is -1.

### Object Model



Customer can give a comment only after having made an order and received it. The comment and the rate are stored in the order\_merchant\_table. After each rate given, the field merchant\_average\_rating is updated.

# Complaints

## Database Model



complaints\_subject(INTEGER) :

* 1 : Damaged
* 2 : Delayed
* 3 : Different from order
* 4 : Return 7 days
* 5 : Other
* 6 : Return 14 days
* 7 : Lost
* 8 : Defective
* 9 : Improper

complaints\_status(INTEGER) :

* 1 : Opened
* 2 : Reimbursed
* 3 : Sent again
* 4 : OtherSolving
* 5 : Solved
* 6 : PendingReimbursed
* 7 : PartiallyReimbursed
* 8 : PendingSentAgain
* 9 : PartiallySentAgain
* 10 : Canceled

complaints\_subreason :

* 0 : None
* 1 : Bad color
* 2 : Bad Quality
* 3 : Different From Picture
* 4 : Other

## Object Model



## Workflow



To get more details, get the functional documentation.

## Manual and automatic refund

Refunds are managed by the method « refund » in the ComplaintBusiness. Here are the different steps of a refund:

* Refund amount is calculated considering the eventual delivery fees and promos.
* The quantity of order product is updated
* The amount of the order merchant is updated (the refund amount is subtract)
* A negative payment in orders payment is created

**When the payment module accepts automatic refund**, the refund is done, the complaint status is updated (“Reimbursed”) and the client is notify by email.

**When the payment module doesn’t accept automatic refund**, the complaint status becomes “PendingReimbursed”. The client is notified in order to aware him the refund is in process. Then once the merchant or the administrator made the refund, he has to update the complaint status by clicking on a button on his MBO or ABO. The complaint status is then set to “Reimbursed” and the client is notified by email.

# URL rewriting

By default, url have this form: http://domain/controller/action/parameters. But it is possible to custom them.

## Rewriting mechanism

### Dabase Model



**url:**

url\_id: primary key

url: the url path(without the domain part)

controller: name of the controller

action: action name

**url\_parameter:** (an action can have several parameter, that’s why they are on another table)

url\_parameter\_id: primary key

url\_id: url id, foreign key

parameter: parameter value

parameter\_name: parameter name

## Object Model



Rewriting mechanism allowed to replace the default url by a custom one. To afford this, an url will be associated to the controller, the action and the parameters called on the page on which we wants to custom the URL. The method which manages the creation of custom url is CreateorUpdateUrl in the MPf.Web.Urls.RewritingService.

## Automatic URL generation

The automatic URL generation is set on product, categories and merchant index pages. For each of them, there is a method, in the rewriting service named SaveUrl that will create automatically the custom url, based on the name of the product, category or the merchant. These methods will be called in the corresponding business, when a product is updated or created.

To enable the automatic url, the parameter ActivateCustomUrls has to set to true.

Url rewriting provokes a lot of request on the database, that’s the reason why there is cache on them which is refreshed automatically every hour, it is managed by the MPf.Web.Urls.RewritingCacheService.

# Jobs: long running/recurrent tasks

## In-depth architecture

**Database Model :**

|  |
| --- |
| **batch** |
| *Contains informations of jobs reports* |
| batch\_id : int, primary key  batch\_extern\_id : batch id  batch\_datecrea : date of batch creation  batch\_datefin : date of the end of execution  batch\_datedem : datetime  batch\_merchantid : merchant id of the batch  batch\_status : status (Pending = 1, Processing = 2, Done = 4, Error = 8)  batch\_type : name of the batch type |

|  |
| --- |
| **batch\_scheduling** |
| *Contains informations to schedule jobs* |
| batch\_scheduling\_id : batch scheduling id  batch\_scheduling\_name : batch scheduling name  batch\_scheduling\_description : batch description  batch\_scheduling\_type : name of batch type  batch\_scheduling\_cron : batch CRON  batch\_scheduling\_date\_modified : date of last modification on the batch scheduling |

|  |
| --- |
| **batch\_scheduling\_parameters** |
| *Contains parameters associated to the scheduled jobs* |
| batch\_scheduling\_parameters\_id  batch\_scheduling\_parameters\_name : name of the parameter  batch\_scheduling\_parameters\_value : value of the parameter  batch\_scheduling\_id : batch scheduling id |

|  |
| --- |
| **batch\_catalog\_item** |
| catalog\_id  batch\_id  catalog\_status  catalog\_productid  catalog\_offerid  catalog\_msg  catalog\_sku |

|  |
| --- |
| **batch\_articles** |
| id  url  imageurl  title  text |

The project MPf.Windows.Service.Job contains the Program class which calls JobControlers class of the MPf.Service.Job project. This class starts the scheduler with the “Start()” method by calling the main interface of the Quartz.

Quartz is added in the solution with a plugin. By default, it uses the RepositorySchedulingProcessorPlugin class which implements ISchedulerPlugin, but you can change that and use your own ISchedulerPlugin inherited class by modifying the parameter quartz.plugin.repository.type in the config file MPf.Service.Job.Windows.exe.config. The method start() of this plugin executes the execute method of the RepositoryScanJob job, because this last one which implements IJob interface. In the RepositoryScanJob, you can get the batchs scheduled in the batch\_scheduling table. It creates a trigger and a job for the scheduler by getting the information of the batch. Then, the batch is scheduled in quartz.

Note that to use IBatch to create job, the solution instantiates the jobs with the class IOCJobFactory which implements IJobFactory. If you want to use your own job factory, you need to modify the parameter quartz.scheduler.jobFactory.type in the config file.

Then the jobs are executed following their CRON recorded in the batch\_scheduling table. A new record in batch table is saved to have a report of the batch state.

By using log4net, all exceptions of the process can be found in the log file and in the error file localized in the path which is given in the config file.

## Existing jobs

The assembly MPf.Batch contains several preprogrammed batches which allow you to import data on your marketplace. Note that all of them implement the IBatch interface.

### ImportJob

In the BAO, you can schedule this job. This batch collects information present in the CSV which is given by the target URL parameter of the batch. The information are on the products and offers, you have to fill in your CSV with the following columns. Note that all columns are not required.

Information you can put in your CSV file must respect this columns numeration:

* TITLE : column 0
* BRAND : column 1
* IMAGE : column 2
* CATEGORY ID : column 6
* CATEGORIES NAME : column 7
* DESCRIPTION : column 8
* DESCRIPTION\_SHORT : column 9
* PARTNER ID : column 10
* EAN : column 11
* MPN : column 12
* ISBN : column 13
* UPC : column 14
* MPID : column 15
* SKU : column 16
* PRICE : column 17
* PRICE PROMO : column 18
* DATE START PROMO : column 19
* DATE END PROMO : column 20
* STOCK : column 21
* SHIPPING DELAY : column 22
* DELIVERY MODE : column 23
* DELIVERY ZONE : column 26
* DELIVERY ZONE PRICE : column 29

When the CSV is deserialized, the job creates a Batch.Model.BatchRequest and calls the batch CatalogJob with the batch request in parameter.

### CatalogJob

The CatalogJob is the batch which saves the information on products, or offers in the corresponding table in the database. It contains constraints and conditions by default to respect in order to record a product or an offer in your marketplace.

To use it in your own job, you need to call an instance of the ICatalogBatchHandler and you need to use “Create()” method which takes BatchRequest in parameter.

Example :

|  |
| --- |
| //where batchRequest is a MPf.Batch.Model.BatchRequest object  \_catalogBatchHandler.Create(batchRequest); |

BatchRequest contains Guid for user id, ProductTypology and List<CatalogItem>.

## Create a new job

Thanks to the MPf.Batch assembly, you just need to create a class which implements IBatch interface. This DLL file can scan its calling assembly to find classes which use the IBatch interface. This last one contains the name, the description and the “execute()” method called when the job is launched.

The code you want to execute must be placed or called in “execute()” method.

MPf.DAO.NHibernate assembly allows you to use the database tables. You can record information on the jobs in “batch” table of your database by using BatchDAO methods.

If you want schedule jobs by using code, you can use BatchSchedulingDAO class and BatchSchedulingParametersDAO class and add records in these tables.

**Parameters**

If you want to add Job parameters, you can add them in your IBatch inherited class and completed their values for each corresponding job in you database.

For example:

In your IBatch inherited class, you can use this attribute with the following way:

|  |
| --- |
| [Model.BatchParameter(Required = Model.Required.Facultative,                                             Type = Domain.BatchSchedulingParameter.ParameterType.Guid)]  public Guid UserId { get; set; } |

You need to specify “required” parameter (Facultative or Mandatory) and “type” parameter (Date, String, URL, Guid, Int, Decimal, Email, Char, Boolean, None).

# API TO COMPLETE

## Streams schema

## Entities and methods

The API is a restful application based on JSON and XML, it uses System.Web.Http assembly. The controllers you can call, inherited of APIController. The list of controllers is:

* CatalogController: contains methods to put your catalog in your marketplace (products and offers)
* CategoriesController: contains methods to get categories and subcategories
* MerchantsController: contains a method to get merchants
* OffersController: contains a method to get offers
* OrdersController: contains methods to get orders, to put orders and to get status orders
* ProductsController: contains a method to get products

You can access to these functionalities by using url of this type: {your domain}/{controller}/{action}/{id}

This last one is set in the RouteConfig class.

## Formats

The methods of these controllers return objects of the model which are JSON data. You can define the return format of your request by using attribute in your models. In the solution, we use NewtonSoft.JSON to send an answer with JSON format.

Example of a model using NewtonSoft:

|  |
| --- |
| [JsonObject]  [DataContract(Name = "catalogItem", Namespace = NameSpace.NAME)]  public class CatalogItem  {  [JsonProperty(PropertyName = "product")]  [DataMember(Name="product")]  public Product Product { get; set; }  [JsonProperty(PropertyName = "offer")]  [DataMember(Name = "offer")]  public Offer Offer { get; set; }  } |

To apply that, the class must have the JsonObject attribute. We also place the attribute JSonProperty above the class’s field by specifying the property name to set the name of the JSON object.

## Authentication

The access to the API is restricted to users which have the role specified for the action.

For example:

|  |
| --- |
| [APIAuthorize(Roles = "power-user, merchant")]  public HttpResponseMessage PostUpdate(OrdersRequest requestUpdate)  {  [...]  } |

We use the attribute APIAuthorize to define roles which are authorized to access to the methods. You need to place it above your method and you have to set the roles with their respective name separated by commas as on the example.

The authentication user is done with an API key which is available in the backoffice administration when you activate API for a merchant. You have to specify this key in your request to authenticate you.

?????????????????????????

## Submit a new catalog import job

You can import a catalog with this API rest, more precisely you can import a list of products and offers. You have to your disposition a class named CatalogController which is a controller class. In this one, there is the PostCatalog method, you can call it and send your catalog.

????????????????????

# Cache TO COMPLETE

## Architecture (lazy caching, aspects)

In the solution, there is a MPf.Cache project which manages the cache on your marketplace. You can distingue several ways to apply cache. The solution execute the cache service when it is launched by calling the Init() method of ICacheRefreshService. Note that this execution is available if the Enable parameter (parameter recorded in parameter table of your database) is set to true. A new thread is launched to manage that, it calls the Refresh method with an interval defined in the parameter CacheRefreshFrequencyInMinutes. It’s the Refresh() methods of ICacheRefreshModule inherited classes which are impacted by this behavior.

**Lazy Caching**

In the solution, you can make lazy caching by using attributes on your methods. It is the class CacheableAttribute which allows you to save data when the user’s actions call a method.

You can use this attribute with the following way:

|  |
| --- |
| [Cache.Cacheable(CacheMode = Cache.CacheMode.Dual)]  public Domain.Offer GetDetail(Int32 offerId, Int32? quotationId)  { |

Place the Cache.Cacheable attribute above of the method declaration. You have to specify the cache mode.

Cache.Cache contains three values:

* Simple: ???
* Standart: ???
* Dual: ????

## In-memory cache

The in-memory cache is used by the provider class named RuntimeCacheProvider. This one inherits of ICacheProvider. The cache that it uses is the .NET framework cache of the System.Runtime.Caching namespace.

## Shared cache

The shared cache is made by the CouchBase cache which is started by the CacheService class. CouchBase is a NoSQL key/value base where the value is a JSON document.

To register the information in your CouchBase server, you have to convert data in JSON format, that’s why you have a JSON folder in the MPf.Cache project which contains serializers and deserializers. To add your own class, you have to use IDeserializer or ISerializer interfaces.

The corresponding provider class is the class CouchbaseCacheProvider. You have several parameters to permit access to the cache:

|  |
| --- |
| [MPf.Parameterization.Parameterization("URIs", "Couchbase", "http://127.0.0.1:8091/pools")]  public String URLs { get; set; }  [MPf.Parameterization.Parameterization("Bucket", "Couchbase", "cache")]  public String Bucket { get; set; }  [MPf.Parameterization.Parameterization("BucketPassword", "Couchbase", "")]  public String BucketPassword { get; set; }  [MPf.Parameterization.Parameterization("Password", "Couchbase", "")]  public String Password { get; set; }  [MPf.Parameterization.Parameterization("UserName", "Couchbase", "")]  public String UserName { get; set; }  [MPf.Parameterization.Parameterization("Design", "Couchbase", "cache")]  public String Design { get; set; }  [MPf.Parameterization.Parameterization("DependenciesView", "Couchbase", "dependencies")]  public String DependenciesView { get; set; } |

# EAV

## Database Model

EAV’s are giving the possibility to associate an extra attribute to a model. Then the model will be associated to an entity (customer, merchant, product, offer, etc …).



**eav\_attributes:**

eav\_attributes\_id: primary key

eav\_attributes\_name: name of the attribute

eav\_attributes\_label: name of the label

eav\_attributes\_is\_required: when it is set to true, the eav attribute HAS to be filled when the entity on which it is associated is created.

eav\_attributes\_input:

* 1: Boolean
* 2: TextBox
* 3 : TextArea
* 4 : Drop Down List
* 5 : Date
* 6 : Int
* 7 : Decimal

eav\_entity\_type:

* 1: Product
* 2: Offer
* 3: OfferProduct
* 4: Merchant
* 5: Order
* 6: Customer
* 7: OrderMerchant

\*The list of entity type is not exhaustive, it is possible to add more entity type as long as the enumeration “EavEntityType” is updated.

eav\_attributes\_is\_visible: allow (or not) displaying the eav attribute on the front in order to give the possibility to the administrator (in the ABO) or the merchant (in the MBO) to edit it.

eav\_attributes\_is\_filterable: allow (or not) filters on EAV’s.

**eav\_attributes\_option**: options of eav when his type is a dropdownlist

eav\_attribute\_id: id of the eav attribute the option is on.

eav\_attribute\_option\_value: dropdownlist value

eav\_attribute\_option\_text: dropdownlist text

**eav\_models:**

eav\_models\_id: id, primary key

eav\_models\_name: name of the eav model

eav\_models\_label: label of the eav model

**eav\_models\_attributes:** association table between between eav\_models and eav\_attributes

**eav\_entity\_model**: associate an eav\_model to an entity

**eav\_entity\_attributes\_values**: containing the values of eav\_attributes for each entity.

## Object Model



## Load EAV TODO

To load EAV, call the LoadEntityAttributeValues method of the EavBusiness. This method is generic so it works for any entity. By default the EAV attributes values are not loaded. In order to load them, set the parameter LoadEntityAttributeValues in the ABO. Once that is done, it will load the EAV in the corresponding business.

## Edit EAV TODO

It is possible to edit/save an EAV by calling the method “Save” of the MPf.Web.EAV.EAVService which allows to save a list of EAV’s. But a lot of fields have to be informed. Use the GetEditable method of the MPf.Web.EAV.EAVService to this effect. It will return a list of EAV.Models.EntityAttributesValues with all the fields fields (except the value that will have the default value depending on the type of the attribute). When you have a lot of EAV’s to save or edit, it is highly recommended to use EditorTemplates. They will avoid you to add manually all the fields in your view. As sample check the creation/edition page of the ABO.

## Apply Filter on EAV TODO

EAV filters work almost like edit or save. Call the method GetCriteriaFilters instead of the GetEditable method this time. Fill the model in the view by using editors templates. Then transmit the EAV Filters to the DAO. In each DAO, if EAV Filters are filled, it will call the MPf.DAO.NHibernate.EAVModel.EAVQueryBuilder which builds the query on EAV. If you need to add custom filters on EAV attributes, you will have to edit this class.

# CMS for custom text pages TODO

# Extending the solution with custom modules

You can add your custom modules in your marketplace, they are managed by the project MPf.Module. A folder Modules has been created at the root of the solution in order to regroup all module projects which are added, you can add yours inside. If your module has views, images or scripts, you can add it dedicated folder in the module folder of MVC project of the solution. For example, in the MPf.Web.Front.Themes.{your theme’s name}.Modules.

The IOC of the MPf.Module allows to find and to use modules which implements IModule interface. It calls the business which adds the module if it doesn’t exist in the table *modules* of your database. In the back office administration, you can activate its.