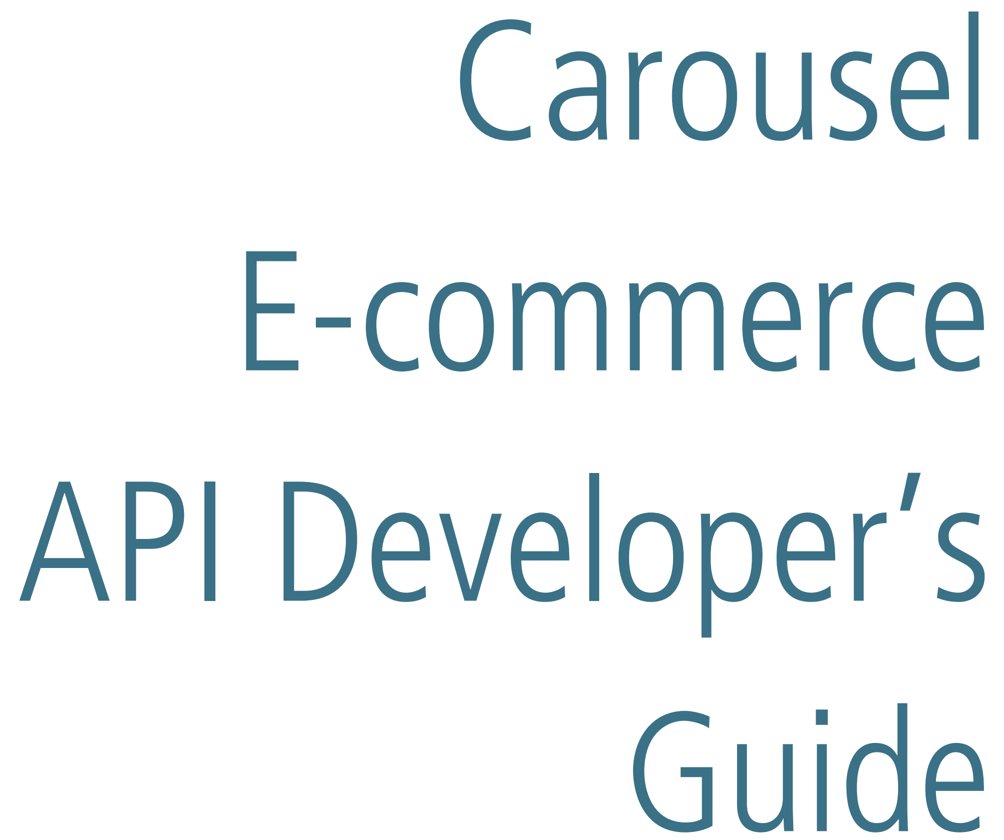


Carousel E-commerce API Developer's Guide

by Noesis Srl
http://www.noesis-research.com

Version 1.0

CAROUSEL

Table of Contents

Carousel E-commerce API Developer's Guide ... *2*

Technical Notes ... **2**

Site Configuration ... **2**

Site creation ... 2

Creating the sections ... 4

Creating the zones ... 5

Creating the profiles ... **7**

Offer Creation ... **8**

Campaign Configuration ... **13**

Value creation ... **17**

Checking the accuracy of the configuration ... **19**

Optimization Process ... **20**

The GetRecommendationRules – SetFeedback loop ... 20

The SetFeedback method ... 21

Carousel E-commerce API Developer's Guide

This document describes how to use the Carousel E-commerce API to programmatically access the Carousel Optimization Engine through a web service (Carousel Service).

Technical Notes

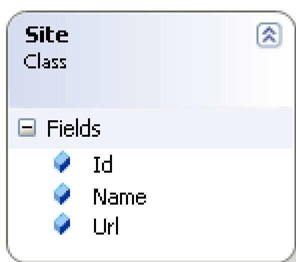
LANGUAGE: The Carousel API *Web Service* interface supports a growing number of languages - including Java, .NET, Perl, PHP, Python, OCAML, Ruby and XML. The sample code provided in this guide is in C#. The Java version is also provided at the end of the document. You can easily translate the code in any language of your choice.

HTTP/SOAP ERRORS: It must be stressed that all the procedures described below are concerned with the calls to a *Web Service*, and so they must always be encapsulated within a *try-catch* block which can manage any exceptions generated both by possible communications problems and by parameters being passed on incorrectly.

Site Configuration

To be able to use the service there is a preliminary configuration stage to create the various entities which describe the activity of the user. These will be the basis upon which the rules for the visualization of the offers will be created. The first step is to define the working environment by creating the various sites, each one with its sections and zones.

Site creation



Each user can run more sites, each one characterized by one or more *Sections*.
Using the method

AddSite(string userName, string password, Site site)

it is possible to add various *Sites*. It is up to the user to create the object that it describes by specifying a name (*Name*) and a web address (*Url*). The field *Name* is mandatory while the field *Url* not. The field *Id* is the identification assigned automatically by the application. Whenever one is specified by the user, this will in any case be ignored.
The user can retrieve a given site at any time and consult the list of existing sites, deciding to cancel or update them.

Example of a code – creating a new site

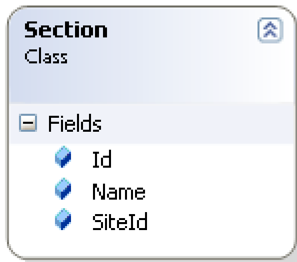
```
// This code sample creates a new site without sections and zone.
// To create them, see the procedure AddSection and AddZone.
public void AddSite()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();
```

```
    // Credentials
    string user = "INSERT_USERNAME_HERE";
    string password = "INSERT_PASSWORD_HERE";

    // Create new site structure.
    Site newSite = new Site();
    newSite.Name = "NewSite";
    newSite.Url = "http://NewSite.com";

    try
    {
        // Add the new site if there are no error
        Site addedSite = service.AddSite(user, password, newSite);

        // Display new site
        Console.WriteLine("New site with url " + addedSite.Url
                          + " and id " + addedSite.Id + " was created.");
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine("New site was not created due to the following
                          error: " + ex);
    }
}
```

Example of a code – retrieving the list of existing sites

```
// This code sample retrieves information about all sites that belongs to the
customer issuing the request.
public void GetAllSites()
{
    // Set up service connection
    EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

    // Credentials
    string user = "INSERT_USERNAME_HERE";
    string password = "INSERT_PASSWORD_HERE";

    try
    {
        // Get all sites
        Site[] sites = service.GetSites(user, password);

        // Display site info
        foreach (Site s in sites)
        {
            Console.WriteLine("Site name is " + s.Name + " URL is "
                              + s.Url + " and id is " + s.Id);
        }
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine("Sites info no retrieved due to the following
                          error: " + ex);
    }
}
```

Creating the sections



Each *Section* belongs to one and only one *Site* and is characterized by one or more *Zones*.
Using the method

AddSection(string userName, string password, int siteId, Section section)

it is possible to add various *Sections* to the identified site through the ID by parameter. It is up to the user to create the object that it describes by specifying a name (*Name*).
The field *Name* is mandatory; the field *Id* is the identification assigned automatically by the application and *SiteId* is the identification of the site which the section belongs to. Whenever one is specified by the user, this will in any case be ignored.
The user can retrieve a given section at any time, consult the list of all existing sections or only those of a particular site, deciding whether to cancel or update them.

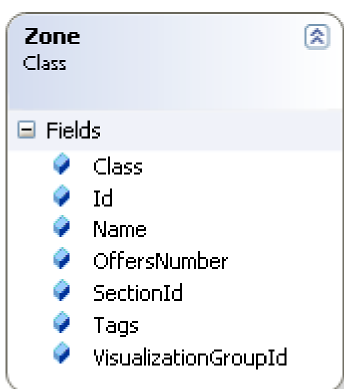
Example of a code – creating a new section

```
// This code sample creates a new section without zone for a site.
// To create them, see the procedure AddZone.
public void AddSection()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      int siteId = 1; // site with ID=1 must be exist

      // Create new section structure
      Section newSection = new Section();
      newSection.Name = "NewSection";

      try
      {
            // Add the new section if there are no error
            Section addedSection = service.AddSection(user, password, siteId,
                                                      newSection);

            // Display new section
            Console.WriteLine("New section with id " + addedSection.Id +
                              " was created for the site " +
                              addedSection.SiteId);
       }
       catch (ArgumentException ex)
       {
            Console.WriteLine("New section was not created due to the following
                              error: " + ex);
       }
}
```

Example of a code – retrieving the list of sections for a certain site

```
// This code sample retrieves information about sections of a particular sites.
public void GetSectionBySite()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      int siteId = 1; // site with ID=1 must be exist

      try
      {
            // Get sections
            Section[] sections = service.GetSectionsBySite(user, password,
                                                          siteId);

            // Display sections info
            Console.WriteLine("Sections for site with Id: " + siteId + " are ");
            foreach (Section s in sections)
            {
                  Console.WriteLine("sectionId " + s.Id);
            }
       }
       catch (ArgumentException ex)
       {
            Console.WriteLine("Sections info no retrieved due to the following
                                   error: " + ex);
       }
}
```

Creating the zones



Each *Zone* belongs to one and only one *Section*.
Using the method

AddZone(string userName, string password, int sectionId, Zone zone)

it is possible to add various *Zone* to a section identified through the ID by parameter. It is up to the user to create the object that it describes by specifying a name (*Name*), a class (*Class*), a tag list (*Tags*) and a visualization group (*VisualizationGroupId*).
The class identifies a group with shared characteristics: the zones which make up a class should have similar characteristics and defining a series of categories/groups coherently makes it possible to propagate knowledge between them, *Zones* with the same class share information.
The field *OffersNumber* is mandatory and represent the number of item to display in this zone.
The field *Tags* is not compulsory and covers any necessary tags which apply to the zone.
The field *VisualizationGroupId* is not compulsory but, if used, is useful to stop equal content being shown at the same time. Zones which share the same visualization group are shown simultaneously,

for example, on the same page of the same site; if it is specified that they belong to the same *VisualizationGroupId* it will be possible to avoid the same content being shown in them.
The field *Name* is mandatory while the field *Class* not; the field *Id* is the identification assigned automatically by the application and *SectionId* is the identification of the section which the zone belongs to. Whenever one is specified by the user for the zone, this will in any case be ignored.
The user can retrieve a given zone at any time, consult the list of all existing zones or simply those of a particular section, deciding to cancel or update them.

Example of a code – creating a new zone

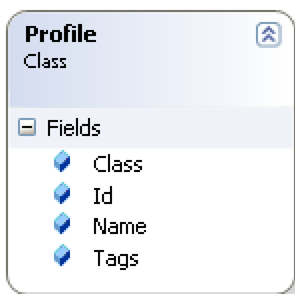
```
// This code sample creates a new zone for a section.
public void AddZone()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      int sectionId = 1; // section with ID=1 must be exist
      int offersNumber = 10;

      // Create new zone structure
      Zone newZone = new Zone();
      newZone.Name = "NewZone";
      newZone.Class = "Class_for_Zone";
      // this field is not mandatory
      newZone.Tags = new ArrayOfString { "tag1", "tag2", "tag3" };
      newZone.OffersNumber = offersNumber;

      // Zones that have the same VisualizationGroup are showed at the same time
      newZone.VisualizationGroupId = 2; // this field is not mandatory

      try
      {
            // Add the new zone if there are no error
            Zone addedZone = service.AddZone(user, password, sectionId,
                                             newZone);

            // Display new zone
            Console.WriteLine("New zone of " + addedZone.Category +
                              " category and id " + addedZone.Id +
                              " was created for the section " +
                              addedZone.SectionId);
       }
       catch (ArgumentException ex)
       {
            Console.WriteLine("New zone was not created due to the following
                              error: " + ex);
       }
}
```

Example of a code – retrieving a given zone

```
// This code sample retrieves information about a specified zone.
public void GetZone()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      int zoneId = 3; // zone with ID=3 must be exist

      try
      {
            // Get zone information if there are no error
            Zone zone = service.GetZone(user, password, zoneId);

            // Display zone informaztion
            Console.WriteLine("Zone in section with Id: " + zone.SectionId +
                              " of " + zone.Category + " category and id " +
                              zone.Id);
       }
       catch (ArgumentException ex)
       {
            Console.WriteLine("Zone information was not retrieved due
                               to the following error: " + ex);
       }
}
```

Once the working environment is concluded, profiles and offers should be defined, and you should create and organize the campaigns that will then be activated and which algorithms will decide the selection of the content and the rules to pass on to the Ecommerce Server.

Creating the profiles



Using the method

AddProfile(string userName, string password, Profile profile)

it is possible to add various *Profiles*. It will be the user's job to create the object that it describes by specifying a name (*Name*), a class (*Class*) and a tag list (*Tags*).
As in the case of the *Zones*, the class defines a similarity group: the profiles which are a part of the same class should have similar characteristics and defining a series of categories/groups coherently makes it possible to propagate knowledge between them. *Profiles* with the same class share information.
The field *Tags* is not compulsory and covers the tag list which may be necessary to qualify the profile.
The field *Name* is mandatory while the field *Class* not; the field *Id* is the identification assigned automatically by the application. Whenever one is specified by the user for the profile, this will in any case be ignored The user can retrieve a given profile at any time or consult the list of all existing profiles, deciding to cancel or update them.

Example of a code – creating a new profile

```
// This code sample creates a new profile.
public void AddProfile()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      // Create new profile structure
      Profile newProfile = new Profile();
      newProfile.Name = "NewProfile";
      newProfile.Class = "Class_for_Profile";
      // this field is not mandatory
      newProfile.Tags = new ArrayOfString { "tag1", "tag2", "tag3" }

      try
      {
            // Add the new profile if there are no error
            Profile addedProfile = service.AddProfile(user, password,
                                                      newProfile);

            // Display new profile
            Console.WriteLine("New profile of " + addedProfile.Category +
                              " category and id " + addedProfile.Id + " was
                              created.");
      }
      catch (ArgumentException ex)
      {
            Console.WriteLine("New profile was not created due to the following
                              error: " + ex);
      }
}
```

Offer Creation

Every campaign is made up of one or more *Offers*.
Using the method

AddOffer(string userName, string password, Offer Offer)

it is possible to add an *Offer* to a given campaign identified by means of parameter. It will be the user's job to create the object which will describe them, specifying a name (*Name*), a class (*Class*) and a tag list (*Tags*).
As for the *Zones*, the class identifies a similarity group: offers which are a part of the same class should have similar characteristics and defining a series of categories/groups coherently makes it possible to propagate knowledge between them. *Offers* with the same class share information.
The field *Tags* is not compulsory and represents any tag list which may be necessary for an *Offer*.
The field *Name* is mandatory while the field *Class* not; the field *Id* is the ID assigned automatically by the application; whenever one is specified by the user for the offer, this will in any case be ignored.

The user can retrieve a given offer at any time or consult the list of all existing offers, or only those for a single campaign, deciding to cancel or update them.

Example of a code – creating a new offer

```
// This code sample creates a new offer.
public static void AddOffer()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      // Create new Offer structure
      Offer newOffer = new Offer();
      newOffer.Name = "NewOffer";
      newOffer.Class = "Class_for_Offer";
      // this field is not mandatory
      newOffer.Tags = new ArrayOfString { "tag1", "tag2", "tag3" };

      try
      {
            // Add the new offer if there are no error
            Offer addedOffer = service.AddOffer(user, password, newOffer);

            // Display new offer
            Console.WriteLine("New Offer of " + addedOffer.Class +
                             " class and id " + addedOffer.Id + " was created");
      }
      catch (ArgumentException ex)
      {
            Console.WriteLine("New Offer was not created due to the following
                               error: " + ex);
      }
}
```

Example of a code – retrieving the list of offers for a given campaign

```
// This code sample retrieves information about offers of a particular campaign.
public void GetOfferByCampaign()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      int campaignId = 1; // campaign with ID=1 must be exist

      try
      {
            // Get offers
            Offer[] offers = service.GetOffersByCampaign(user, password,
                                                         campaignId);
```

```
            // Display offers info
            Console.WriteLine("Offers for campaign with Id: " +
                              campaignId + " are ");
            foreach (Offer o in offers)
            {
                  Console.WriteLine("offer of "+o.Class+" class and Id "+o.Id);
            }
      }
      catch (ArgumentException ex)
      {
            Console.WriteLine("Offers info no retrieved due to the following
                              error: " + ex);
      }
}
```

In some cases it can be a good idea to consider deactivating an *offer* within a campaign, because, for example, it is not currently available in the warehouse, or simply because we want to add some kind of boost to the distribution of another *offer*.
To obtain this, it is enough to reconfigure the campaign in question by using the method

UpdateCampaign(string userName, string password, Campaign campaign)

taking care to exclude the desired *offer* from the list of those which are active (*OfferIds*) for this campaign.
At a later time when it needs to be reactivated simply make the same call, this time including the identification code of the item to be reinserted within the field *OfferIds* .
The following is an example of this (as regards the details on the creation of the campaign, see the later relevant paragraph).

```
Campaign newCampaign = new Campaign();
newCampaign.Name = "NewCampaign";
newCampaign.Start = DateTime.Now;
newCampaign.End = DateTime.Now.AddDays(3);
newCampaign.OptimizationMetric = 0;
newCampaign.ProfileIds = new ArrayOfInt() { };
newCampaign.ZoneIds = new ArrayOfInt() { 1, 2, 3 };
newCampaign.OfferIds = new ArrayOfInt() { 1, 2, 3};
```

The campaign *newCampaign* (to which the software has automatically assigned the identification code 9) will work with the *offers* defined by the user as *1*, *2* and *3*. Let us suppose that we want to temporarily deactivate *offer #2* and insert a new *offer*, *#4*. We will use the call *UpdateCampaign*. Only the fields to which a new value will be assigned need to be changed. All the others will stay the same.

```
Campaign updatedCampaign = new Campaign();
updatedCampaign.Id = 9;
updatedCampaign.Name = null;
updatedCampaign.Start = null;
updatedCampaign.End = null;
updatedCampaign.OptimizationMetric = null;
updatedCampaign.ProfileIds = null;
updatedCampaign.ZoneIds = null;
updatedCampaign.OfferIds = new ArrayOfInt() { 1, 3, 4 };
```

Offer #2 is now deactivated.

If and when the user wants to reactivate it, they must make a new call and include it again in the campaign list. Please note that all the previous statistics which have been memorized for the *offer* (clicks and impressions obtained) remain and will be reloaded and used to calculate the algorithm only when the *offer* is active.

```
Campaign updatedCampaign2 = new Campaign();
updatedCampaign2.Id = 9;
updatedCampaign2.Name = null;
updatedCampaign2.Start = null;
updatedCampaign2.End = null;
updatedCampaign2.OptimizationMetric = null;
updatedCampaign2.ProfileIds = null;
updatedCampaign2.ZoneIds = null;
updatedCampaign2.OfferIds = new ArrayOfInt() { 1, 2, 3, 4 };
```

After this call the *offer #2* is again activated for the campaign.

An *offer* can mean different things depending on the merchandise category that we want it to represent; for example, both the general category “cell phone” and the specific product “Nokia N76” can be described.
The object fields always stay the same but the relevant meaning should be interpreted based on the *offer*. Let us look at some examples.

Let us suppose that we have to run a window on a site in which the user only wants to show the merchandise categories, a visitor’s click can then redirect him or her to the place where only the specific products are shown.
The user has to define as many *offers* as there are categories which must compete to be shown in the window; in this case the field *Class* represents the super-category to which the defined category belongs, in practice, if it exists, it corresponds to the higher level of the merchandise hierarchical structure.
Below, we show the code which should be used in the above situation.

```
Offer newOfferT = new Offer();
newOfferT.Name = "telephones";
newOfferT.Class = "electronics";
newOfferT.Tags = new ArrayOfString { "tag1", "tag2" };

Offer newOfferN = new Offer();
newOfferN.Name = "notebooks";
newOfferN.Class = "electronics";
newOfferN.Tags = new ArrayOfString { "tag1", "tag2", "tag3" };

Offer newOfferS = new Offer();
newOfferS.Name = "shoes";
newOfferS.Class = "clothes";
newOfferS.Tags = new ArrayOfString { "tag1" };

Offer newOfferL = new Offer();
newOfferL.Name = "books";
newOfferL.Class = "educational";
newOfferL.Tags = new ArrayOfString { "tag4", "tag5" };

Offer newOfferC = new Offer();
newOfferC.Name = "body_care";
newOfferC.Class = "";
newOfferC.Tags = new ArrayOfString { "tag5" };
```

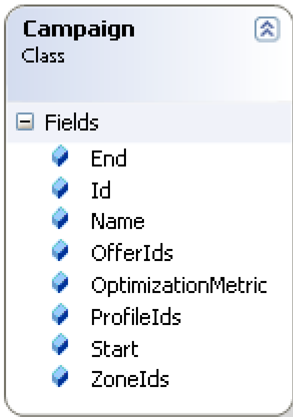
Let us now suppose that we have to run a window on a site in which the user wants to show the specific products themselves. A visitor's click on the product can then redirect them to the specific product page or can allow them to proceed with the purchase of it.
The user has to define as many *offers* as there are products which must compete to be shown in the window. In this case the field *Class* represents the merchandise category which it belongs to.
Below, there is the code that should be used in the above situation.

```
Offer newOfferN1 = new Offer();
newOfferN1.Name = "Nokia_N76";
newOfferN1.Class = "cell_phones";
newOfferN1.Tags = new ArrayOfString { "tag1", "tag2", "tag3" };

Offer newOfferN2 = new Offer();
newOfferN2.Name = "Nokia_N95";
newOfferN2.Class = "cell_phones";
newOfferN2.Tags = new ArrayOfString { "tag1", "tag2" };

Offer newOfferN3 = new Offer();
newOfferN3.Name = "Sony Ericsson J110i";
newOfferN3.Class = "cell_phones";
newOfferN3.Tags = new ArrayOfString { "tag1", "tag2" };

Offer newOfferA = new Offer();
newOfferA.Name = "Acer_Aspire";
newOfferA.Class = "notebooks";
newOfferA.Tags = new ArrayOfString { "tag1" };

Offer newOfferA2 = new Offer();
newOfferA2.Name = "Sony_VAIO";
newOfferA2.Class = "notebooks";
newOfferA2.Tags = new ArrayOfString { "tag4", "tag5" };

Offer newOfferT = new Offer();
newOfferT.Name = "TomTom";
newOfferT.Class = "GPS";
newOfferT.Tags = new ArrayOfString { "tag5" };

Offer newOfferH = new Offer();
newOfferH.Name = "HTC";
newOfferH.Class = "smartphones";
newOfferH.Tags = new ArrayOfString { "tag6" };
```

Finally, let us suppose that we have to run a window on a site in which the user wants to show specific products, but because of privacy policies regarding the user or simply because a decision has been made to use the software in this way, the user does not want to pinpoint a specific *offer* for each product but rather prefers *Carousel* to note only the merchandise categories.
The user should proceed as in the first case described above, defining as many *offers* as there are categories which must compete to be shown in the window. The rules generated by *Carousel* will show which *offer* to show in a given *profile-zone* pair in a completely transparent way from the real value attributed to the offer. It is clear that if the user has defined *offers* that are really categories, he or she will then need to interpret the rules and decide which of the real products in the indicated category to show.

When the various entities necessary to describe the user's activity have also been defined, the final step is to define the additional characteristics which allow us to specify a series of details for carrying out the campaign. At this point, therefore, we will move on defining the constraints, the costs and the rewards.

Campaign Configuration



Each user can run a campaign made up of one or more *Offers*.
Using the method

AddCampaign(string userName, string password, Campaign campaign)

it is possible to add various *Campaigns*. It will be the user's job to create the object that it describes by specifying a name (*Name*), a start and an end (*Start, End*), a list of the IDs of the profiles and offers that the campaign will work with (*ProfileIds*, *OfferIds*), a list of the IDs of the zones that the *OfferIds* in the campaign can be shown in (*ZoneIds*) and the type of optimization metric that will be used (*OptimizationMetric*).
The field *Start* is compulsory, whereas *End* is not; the end date "31/12/2050" has been entered as the default setting, in effect indicating that the campaign is infinite.
The field *ProfileIds* is not compulsory while the field *ZoneIds* is; a campaign without particular profiles defined can exist, but not one without zones to work in. If they are defined, the profiles and zones listed must exist and have been previously created. It should be noted that the same zone or profile can be affected by different campaigns.
The field *OptimizationMetric* is not compulsory; it indicates what the objective function of the algorithm will be based on, and can have one of the following values:

Clicks = 0,
Revenue = 1
The default setting is 1;

The field *Name* is compulsory.
The field *Id* is the identification assigned automatically by the application. Whenever one is specified by the user for the campaign, this will in any case be ignored. The user can retrieve a given campaign at any time or consult the list of all existing campaigns, deciding to cancel or update them; it is also possible to know the current state of a campaign, using the following code system:

Initialized = 0	*if the campaign has already been created but not starter yet*
Paused = 1	*if the campaign has been temporarily paused*
Started = 2	*if the campaign is running*
Stopped = 3	*if the campaign has finished*

Example of a code – creating a new campaign

```
// This code sample creates a campaign with a specified duration, for a set of
profiles over a set of zones.
// To create profiles and zones see the procedure AddProfile and AddZone.
// To create e add advertisement to campaign see the procedure AddAdvertisement.
public void AddCampaign()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      // Create new campaign structure
      Campaign newCampaign = new Campaign();
      newCampaign.Name = "NewCampaign";
      newCampaign.Start = DateTime.Now;
      // end time must not be specified if the campaign is infinite
      newCampaign.End = DateTime.Now.AddDays(3);

      // this field has a value chosen from above:
      // 0 if we want to optimize the number of click,
      // 1 if we want to optimize the revenue
      newCampaign.OptimizationMetric = 0;

      // profile ids must be exist
      // this field is no mandatory
      newCampaign.ProfileIds = new ArrayOfInt() { };

      // zone ids must be exist
      // this field is mandatory
      newCampaign.ZoneIds = new ArrayOfInt() { 1, 2, 3 };

      // offer ids must be exist
      // this field is mandatory
      newCampaign.OfferIds = new ArrayOfInt() { 1, 2, 3};

      try
      {
            // Add the new campaign if there are no error
            Campaign addedCampaign = service.AddCampaign(user, password,
                                                         newCampaign);

            // Display new campaign
            Console.WriteLine("New campaign with name " + addedCampaign.Name +
                              " and id " + addedCampaign.Id + " was created.");
      }
      catch (ArgumentException ex)
      {
            Console.WriteLine("New campaign was not created due to the following
                              error: " + ex);
      }
}
```

Example of a code – retrieving the list of existing campaigns

```
// This code sample retrieves information about all campaigns that belongs to
the customer issuing the request.
public void GetAllCampaigns()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      try
      {
            // Get all campaigns
            Campaign[] campaigns = service.GetCampaigns(user, password);

            // Display campaign info
            foreach (Campaign c in campaigns)
            {
                  Console.WriteLine("Campaign name is " + c.Name +
                                    " and id is " + c.Id);
            }
      }
      catch (ArgumentException ex)
      {
            Console.WriteLine("Campaigns info no retrieved due to the following
                              error: " + ex);
      }
}
```

Example of a code – updating a campaign

```
// This code sample change a campaign updating only the field specified and not
null.
public void UpdateCampaign()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      // Create the updated campaign structure

      // In this campaign only the name changes, all the other fields
      // are set to null
      Campaign campaign1 = new Campaign();
      campaign1.Id = 3; // Campaign with ID=3 must exist
      campaign1.Name = "UpdatedName";
      campaign1.Start = null;
      campaign1.End = null;
      campaign1.OptimizationMetric = null;
      campaign1.ProfileIds = null;
      campaign1.ZoneIds = null;
      campaign1.OfferIds = null;

      // In this campaign start date and optimization metric are changed
```

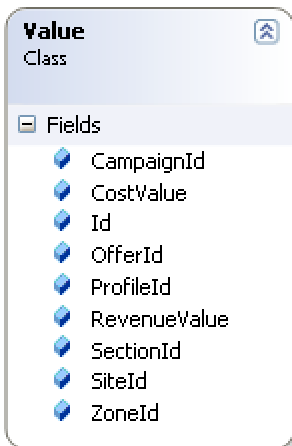
```
        Campaign campaign2 = new Campaign();
        campaign2.Id = 7; // Campaign with ID=7 must exist
        campaign2.Name = null;
        campaign2.Start = DateTime.Now.AddDays(5);
        campaign2.End = null;
        campaign2.OptimizationMetric = 1;
        campaign2.ProfileIds = null;
        campaign2.ZoneIds = null;
        campaign2.OfferIds = null;

        // In this campaign profile list, zone list  and offer list are changed
        Campaign campaign3 = new Campaign();
        campaign3.Id = 9; // Campaign with ID=9 must exist
        campaign3.Name = null;
        campaign3.Start = null;
        campaign3.End = null;
        campaign3.OptimizationMetric = null;
        // to remove all the profile, create new empty list
        campaign3.ProfileIds = new ArrayOfInt();
        // to change the zone list, create new list and add the specified zones
        campaign3.ZoneIds = new ArrayOfInt() { 1, 2, 3 };
        // to change the offer list, create new list and add the specified offers
        campaign3.OfferIds = new ArrayOfInt() { 1, 2, 3 };

        try
        {
              Campaign updateCampaign1 = service.UpdateCampaign(user, password,
                                                                  campaign1);

              Campaign updateCampaign2 = service.UpdateCampaign(user, password,
                                                                  campaign2);

              Campaign updateCampaign3 = service.UpdateCampaign(user, password,
                                                                  campaign3);

              // Display result
              Console.WriteLine("Campaigns were updated.");
         }
         catch (ArgumentException ex)
         {
              Console.WriteLine("Campaigns was not updated due to the following
                                   error: " + ex);
         }
}
```

Example of a code – checking and modifying the state of a campaign

```
// This code sample retrieves information about the current status of a
specified campaign, resume a campaign and change status of another one pausing
it.
public void CampaignStatus()
{
        // Set up service connection
        EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

        // Credentials
        string user = "INSERT_USERNAME_HERE";
        string password = "INSERT_PASSWORD_HERE";

        // campaigns must exist
        int campaignId1 = 1;
```

```
      int campaignId2 = 2;
      int campaignId3 = 3;

      try
      {
            // Get campaign status
            int status1 = service.GetCampaignStatus(user,password,campaignId1);

            // Pause campaign 2
            int status2 = service.PauseCampaign(user, password, campaignId2);

            // Resume campaign 3
            int status3 = service.PauseCampaign(user, password, campaignId3);

            // Display campaign status
            Console.WriteLine("Campaign with Id: " + campaignId1 + " is " +
                                   status1);
            Console.WriteLine("Campaign with Id: " + campaignId2 + " is " +
                                   status2);
            Console.WriteLine("Campaign with Id: " + campaignId3 + " is " +
                                   status3);
       }
       catch (ArgumentException ex)
       {
            Console.WriteLine("Campaign status no retrieved due to the following
                                  error: " + ex);
       }
}
```

Value creation



It is possible to identify a series of values for an offer which can be applied to either an entire campaign, to single zones only, or to sections or entire sites, either a given profile, or also a combination of more of these. Using the method

AddValue(string userName, string password, Value Value)

it is possible to add a *Value*. It will be the user's job to create the object that describes it specifying the relevant entities and the value (*Value*). *OfferId* is mandatory while the other fields that identify the entities (*CampaignId*, *ZoneId*, *SectionId*, *SiteId* and *ProfileId*) can all be completed or less of them according to the extent of restriction required for the value. For example, if a *value* is required which is in effect for the *offer* displayed in a certain *zone*, then only the fields *OfferId* and *ZoneId* need to be completed.
The field *Id* is the ID assigned automatically by the application. Whenever one is specified by the user for the *value*, this will in any case be ignored.
The field *CostValue* and *RevenueValue* are respectively the cost payed and the gain obtained when the *offer* is showed in the particular situation described by *CampaignId*, *ZoneId*, *SectionId*, *SiteId* and *ProfileId*. Both the field are mandatory. Note that *CostValue* must be greater or equal to 0, *RevenueValue* must be strictly greater than 0, and also their difference (*RevenueValue - CostValue*) must be greater than 0.
The user can retrieve a given *Value* at any time or consult the list of all existing *Values*, deciding to cancel or update them.

Example of a code – creating new values

```
// This code sample create some Values.
// Note that not all the fields are indicate, in this case the Value specify the
value in a particular zone,section or site,for a particular campaign or profile.
public static void AddValues()
{
	// Set up service connection
	EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

	// Credentials
	string user = "INSERT_USERNAME_HERE";
	string password = "INSERT_PASSWORD_HERE";

	// Create new Value structure
	// the offer,campaign,zone,section,site and profile indicated must exist

	// This Value is valid wherever for the offer with ID=3 of the campaign
	// with ID=10 of profile with ID=5
	Value newValue1 = new Value ();
	newValue1.OfferId = 3;
	newValue1.CampaignId = 10;
	newValue1.SectionId = null;
	newValue1.SiteId = null;
	newValue1.ProfileId = 5;
	newValue1.CostValue = Convert.ToDouble("INSERT_VALUE_HERE");
	newValue1.RevenueValue = Convert.ToDouble("INSERT_VALUE_HERE");

	// This Value is valid for the offer with ID=5 shown in site with ID=1
	Value newValue2 = new Value();
	newValue2.OfferId = 5;
	newValue2.CampaignId = null;
	newValue2.SectionId = null;
	newValue2.SiteId = 1;
	newValue2.ProfileId = null;
	newValue2.CostValue = Convert.ToDouble("INSERT_VALUE_HERE");
	newValue2.RevenueValue = Convert.ToDouble("INSERT_VALUE_HERE");

	// This Value is valid for the offer with ID=7 of the campaign with ID=3
	// only in the zones of section with ID=2
	Value newValue3 = new Value();
	newValue3.OfferId = 7;
	newValue3.CampaignId = 3;
	newValue3.SectionId = 2;
	newValue3.SiteId = null;
	newValue3.ProfileId = null;
	newValue3.CostValue = Convert.ToDouble("INSERT_VALUE_HERE");
	newValue3.RevenueValue = Convert.ToDouble("INSERT_VALUE_HERE");

	Value newValue = new Value();
	newValue.OfferId = 1;
	newValue.CampaignId = 10;
	newValue.SectionId = 2;
	newValue.SiteId = 3;
	newValue.ProfileId = 5;
	newValue.CostValue = Convert.ToDouble("INSERT_VALUE_HERE");
	newValue.RevenueValue = Convert.ToDouble("INSERT_VALUE_HERE");

	try
	{
		// Add new Values if there are no error
		service.AddValue(user, password, newValue1);
```

```
            service.AddValue(user, password, newValue2);
            service.AddValue(user, password, newValue3);
            service.AddValue(user, password, newValue);

            // Display new Value
            Console.WriteLine("New Costs were created.");
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("New Values were not created due to the following
                                    error: " + ex);
         }
}
```

Checking the accuracy of the configuration

There are always the main checks for the accuracy of data inputted by the user; the most typical one is on whether something referred to by only its ID actually exists, for example, when trying to create a new *Zone* in a non-existent *Section*.
At the end of configuration it is in any case a good idea to test the accuracy of the operations so far; in particular, by using the method

CheckConfiguration(string userName, string password)
it is possible to check the accuracy of some aspects.

The method carries out the tests and returns any applicable error messages; all messages are in the form of a string, and the fields are separated by “;”.
The fields indicated are:

#	internalErrorNumber
tag	a tag describing the type of error: "warning", "error" or "info"
table	table name
field	field name
idsList	list of the IDs on record that have caused the error (a string of Ids separated by commas)
message	message

Example of an error message

```
101;warning;Zones;Category;15,17,28;Not all zones have a class
105;error;Profiles;Category;10,20;Error in profiles
```

Example of a code – checking the configuration

```
// This code sample check the accuracy of the configuration done.
public void CheckConfiguration()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();
```

```
        // Credentials
        string user = "INSERT_USERNAME_HERE";
        string password = "INSERT_PASSWORD_HERE";

        try
        {
              ArrayOfString checks = service.CheckConfiguration(user, password);

              if (checks.Count == 0)
                    Console.WriteLine("No error in configuration");
              else
              {
                    Console.WriteLine("Errors in configuration: ");
                    foreach (string c in checks)
                          Console.WriteLine(c);
              }
         }
         catch (ArgumentException ex)
         {
              Console.WriteLine("Check not done due to the following error: "+ex);
         }
}
```

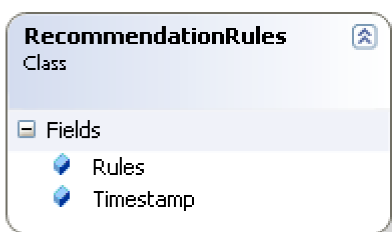
Once the entire system has been tested and the accuracy of it has been verified, it will be possible to start using the service by means of periodic calls which allow the user to retrieve the rules for the display of offers and for saving feedback data.

Optimization Process

After the sites, the offers and campaigns have been configured, the E-commerce Server can begin a permanent communication process with Carousel Ecommerce Service, to receive the instructions to optimize the campaign in real time. As explained in the introduction, this communication process is a sequence of calls alternate to the methods *GetRecommendationRules* and *SetFeedback*.
Carousel generates all the associations between profile-zone and best offer are generated in terms of profitability. The algorithm takes all the zones and all the offers in all the active campaigns which have been previously defined by the user into consideration. The service identifies and returns which will be the best offers to display.

The GetRecommendationRules – SetFeedback loop

Carousel calculates and identifies which will be the best offers to display in a given context. The algorithm takes all the zones and all the offers in all the active campaigns which have been previously defined by the user into consideration.
Every so often the user requests these rules and the E-commerce Server satisfies the requests on the different pages.



Using the method

GetRecommendationRules (string userName, string password, string lastTimestamp)

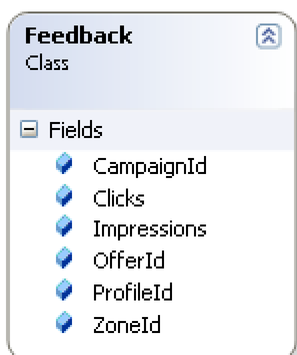
the user requests a list of all the rules updated since the date indicated by the parameter and receives in return a list of the *RecommendationRule*. In all of these, which offers to display (*Offers*) is specified, in which zone (*ZoneId*) they are displayed, and to a certain user profile (*ProfileId*). Timestamp represents the creation date of the rules and is necessary to understand to what extent the information has been updated.
If there isn't any new rules, the method doesn't give an object but NULL.

Example of a code – retrieving the rules

```
// This code sample retrieves the recommendation rules which determine the Offer
to visualize during the next time.
public static void RequestRecommendationRules()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      string lastRequestTime = DateTime.Now.ToString();

      try
      {
            RecommendationRules rules = service.GetRecommendationRules(user,
                                                password, lastRequestTime);
      }
      catch (ArgumentException ex)
      {
            Console.WriteLine("Recommendation Rules not retrieved due to the
                                  following error: " + ex);
      }
}
```

The SetFeedback method

The feedback data represents what has happened on the sites where Carousel's suggestions have been applied; in particular, it is concerned with the number of impressions and clicks received by the ads selected for the various profile-zones.
This data is of fundamental importance because it is by actually using it that the algorithm learns, evolves and generates ever more suitable and optimal rules; because of this, the more frequently that feedback is sent, the easier it will be for Carousel to improve the suggested rules.
Feedback that is sent on time and frequently ensures that Carousel will function at its best.



Using the method

SetFeedback(string userName, string password, Feedback[] feedback)

the user can periodically send a list of all the feedback gathered up until that point.
All *Feedback* must indicate the number of impressions and clicks (*Impressions*, *Clicks*) obtained by every offer displayed (*OfferId*) in a specified zone (*ZoneId*) displayed to a particular user profile (*ProfileId*).

Every time the user sends feedback the application recalculates and updates the rules; for this reason, until new data has been sent, in memory we maintain the old rule list previously calculated with the usual value for the field *TimeStamp*.

Further requests for the rules will not give an object *RecommendationRules* but NULL.
Note that recalculating and updating the rules are asynchronous and require some minutes.

Example of a code – sending feedback

```
// This code sample prepare and send the periodically list of feedback data that
// the server needs to compute rules.
public static void SetFeedback()
{
      // Set up service connection
      EcommerceServiceSoapClient service = new EcommerceServiceSoapClient();

      // Credentials
      string user = "INSERT_USERNAME_HERE";
      string password = "INSERT_PASSWORD_HERE";

      // Create the list of feedback
      // the Offer, zone and profile indicated must exist
      Feedback[] feedback = new Feedback[]
      {
            new Feedback
            {
                  OfferId = 1,
                  ProfileId = 10,
                  ZoneId = 3,
                  Impressions = 1500,
                  Clicks = 75
            },
            new Feedback
            {
                  OfferId = 2,
                  ProfileId = 5,
                  ZoneId = 6,
                  Impressions = 2300,
                  Clicks = 126
            }
      };

      try
      {
            service.SetFeedback(user, password, feedback);
            Console.WriteLine("Feedback sent");
      }
      catch (ArgumentException ex)
      {
          Console.WriteLine("Feedback not send due to the following error:"+ex);
      }
}
```

About Noesis

Noesis is an Italian software house based in Pisa, Italy and has been founded by Academic professors of Pisa University.

Noesis' background assets include ideas, skills and technologies developed in research projects since 1997, involving large companies, small technology-oriented firms, and several universities. These projects were partially funded by the European Union and the Italian Government, partially self-financed by the industrial partners.

The research projects developed technologies of artificial intelligence, adaptive systems, data mining, and text mining applied to **decision making**, **product/content recommendation**, **revenue management**, *and* **dynamic pricing**.

Noesis currently operates truly as a web company having the software development management in house and cooperating with external companies both in the Pisa area and on worldwide base for the engineering power and resources.

Noesis has established a network of partners both as developers and as distributors at a worldwide level for exploiting the solutions developed.

Learn more about Noesis at www.noesis-research.com

For additional information, please contact:

Noesis s.r.l.

Corso Italia, 89
56125 Pisa
Italy

Telephone *+39 050 9911080*
Fax *+39 050 9911588*
Email info@noesis-research.com

www.noesis-research.com