# Realization of an Expandable Search Function for an E-Learning Web Portal

Maria Siebert
*Hasso Plattner Institut*
*Universität Potsdam*
*Potsdam, Germany*
*maria.siebert@hpi.uni-potsdam.de*

Christoph Meinel
*Hasso Plattner Institut*
*Universität Potsdam*
*Potsdam, Germany*
*christoph.meinel@hpi.uni-potsdam.de*

***Keywords*-search, plug-in architecture, Django, tele-teaching**

***Abstract*—When providing a huge amount of content to users, the search engine is an important part of the platform. On most web sites it is the most important entry point to the data archive. That is why it is an interesting research topic.**

**In this paper we present a solution how it is possible to make the search function expendable by using plug-ins. We show, why this is required in our web application, which is constantly growing through the creation of more searchable meta data. Therefore we show, how our requirements can be fulfilled and discuss the accompanying advantages and disadvantages.**

## I. Introduction

Most users of the internet use Google or Bing as start page. It is the easiest possibility to find a web page for a specific topic. Search pages are also used as short cut to known web sites. For example it is a common request to type *wikipedia* with the searched term, to get the link to topic at the wikipedia page.

Also when entering bigger web sites like web shops or video portals many users use the local search engine to find the desired content. This becomes more important, when the web page has a big amount of data and therefore is harder navigable through a normal menu structure. So when providing a huge amount of content to the users, the search engine becomes one of the most essential parts of the platform.

Search functions using semantic web technologies have been researched for years now [1]. Different approaches have been tried [2], even in e-lecture context [3][4]. There are other solutions for search enhancement, which uses user activity logs[5][6].

Our tele-teaching portal tele-TASK is capable of gaining structured meta data from a lot of different sources, like user generated data, log files or even the audio or video stream. Therefore we need the possibility to combine these different meta data fragments for search functions and to test them against each other. That is why we developed a flexible method for enhancement of the core search functionality, which is described in this paper.

Classical development strategies are rarely used when developing web applications. New design principles like SCRUM and DRY are used instead. The usage of frameworks increases [8]. This results in short development phases enabling the developers to create new functions rapidly. But it also results in a lack of usage of proven design principles used in classical application development. Using plug-ins for web applications is mostly uncommon. Most so called plug-ins should not be called like this. They are rather libraries than real plug-ins.

This paper will start with a short description of the underlying web application the tele-TASK portal and the design and implementation of the used plug-in architecture. Afterwards we have a look at the possibilities, which comes with the usage of the Django framework and we propose a solution for the implementation of our search functionality using the features of Django.

This paper will also focus on the obstacles we had to overcome when implementing the core search engine. We will also show the easy enhancement of the search engine with the example of adding the search of play lists. At the end there will be a short discussion on the advantages and disadvantages of the solution.

### A. About the tele-TASK portal

The implemented tele-teaching portal tele-TASK[1] provides a large amount of lectures and videos. The videos, which are produced using the tele-TASK system [9], [10], are captured with one audio and two video streams, allowing separated video streams for the lecturer and the digital presentation. The capturing system, which is used since 2002, allows to create a huge number of recordings of lectures without much effort.

The video portal itself is the platform to provide the generated video content to the whole world, allowing everybody to view a high percentage of the lectures held at our institute. Nearly 3000 lectures can be found. To make it easier for the user to find videos about particular topics, many of the lectures are split into handy video clips with extra meta data.

All this data establishes a good basis for doing research with real data and real users.

[1] http://www.tele-task.de

978-0-7695-4147-1/10 $26.00 © 2010 IEEE
DOI 10.1109/ICIS.2010.117

### B. About the architecture of the tele-TASK portal

The most important task of the portal is to provide the lecture videos. But furthermore it should be a possibility to gain more meta data. Therefore it is important to be able to extend the existing functionalities with new ones. This would result in a big connection overhead between the different modules. A shortened version of the old structure with its dependencies can be seen in figure 1. This image shows only small extract of the whole module architecture. Because of the high number of connections in the standard approach, we decided to implement a plug-in architecture.

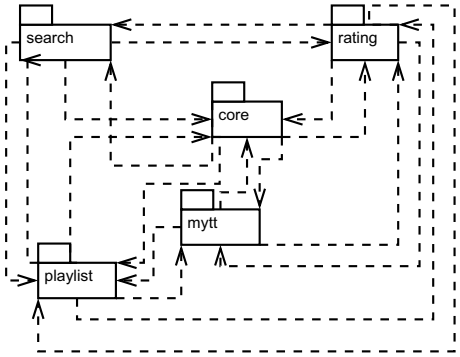


Figure 1. Extract of the structure without using plug-ins

Django itself does not provide any solution for building a plug-in architecture. This created the requirement to build up our own architecture. Therefore we implemented a plug-in architecture based on class inheritance[11]. This architecture allows to create flexible plug-in interfaces. Using this architecture a clear and maintainable structure can be developed, as seen in figure 2.

The underlying idea of the plug-in architecture can be shortly described as follows: The plug-in interface is described as a class. This class provides functions stubs for all functions which are required for the usage of the plug-in. For writing a plug-in it is only required to extend this base class. The plug-in will be detected automatically. Because the base class provides function stubs, each plug-in needs to override these functions if it wants to provide special functions.

With those architectural possibilities it was possible to implement the search functionality, by providing a few base classes with function stubs for creating parts of the database request for the search function. The modules, which know the data, provide the plug-ins by inheriting from these base classes and implementing the required functions.

## II. Searching data in Django

Writing a search request for a Django application is really easy. Because Django provides ORM for different databases like MySQL or PostgreSQL, the developer does not need to write raw SQL commands, which would create a database dependent application. In the following section there will be a short description of the Django ORM implementation and afterwards follows a short introduction to the possibilities of searching with Django.

### A. Object-Relational Mapping (ORM)

Django provides an Object Related Mapping (ORM), which allows the developer to use function calls for generating database requests. Because this paper focuses on search, only requests for fetching data are relevant.

In Django the models are written as classes, where each class maps to a database table. Each attribute of a class is stored as a column of this database table (see listing 1).

```
1 class I18n(models.Model):
2   text = models.TextField()
3   language = models.ForeignKey(Language)
4
5   def __str__(self):
6     return self.text
```

Listing 1. Database table class for i18n

Inside the Django application these classes are used for every data related work. Through the inheritance from *models.Model* the developer can use the *get* function to get each data row and the *save* function to save data changes. With the *delete* function rows can be deleted as well. The constructor of the class is used to generate new database table rows. These functions can be overridden to perform additional actions before or after changing the data.

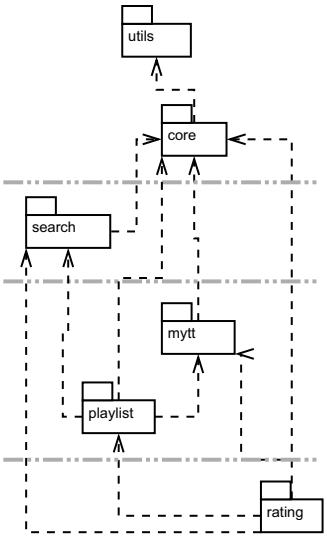


Figure 2. New structure of modules with plug-ins

With this construction it is easy to forget about the database itself and to write a database independent web application. Django contributes all the magic that is needed. It even simulates foreign keys for database engines, which do not provide such technology, like the SQLite database.

### B. Examples for search queries

Performing search request with Django using the ORM is really simple. Django provides an entry point for searching objects of a specific data type by using the ORM class attribute objects (see listing 2).

```
1 series = Series.objects.all()
2 for series in series:
3   # Do something with the data
```

Listing 2. Django search request

The execution of such a query is lazy, which means it is executed when the data itself is needed the first time. In the example listing 2 the query is executed in line 2, when generating the list for the iteration. This fact is important when enhancing the search queries, because it allows to create complex query objects.

These complex query objects can be generated by using the filter or exclude function. The request in listing 3 line 1 searches for all series with an id between 10 and 100.

```
1 series = Series.objects \
2   .filter(id__lt = 100) \
3   .exclude(id__gt = 10)
4 series = Series.objects \
5   .filter(series__lecture__id__lt = 100)
```

Listing 3. Django search request using the filter function

These filters can become really complex SQL statements, when using the automatic joins, provided by Django. For example it is possible to search all series with lectures with id less than 100. The lectures are connected to the series using a many to many relation. So the filter query in line 2 of listing 3 would generate at least two joins to other database tables.

When chaining filters and excludes, like in listing 3 line 1, the different parts of the where clause are combined using AND. For complex queries which should check for values in more than one field, like searching in title and description of the data object, it is necessary to be able to use OR instead. Therefore Django provides the Q object. In the listing 4 you can see a search request for a series, with id $< 100$ or id $> 200$.

```
1 q1 = Q(id__lt = 100)
2 q2 = Q(id__gt = 200)
3 series = Series.objects.filter(q1 | q2)
```

Listing 4. Django search request with Q object

When searching data, it is also important to order the data available in specified columns. For example it is nice to order the data by creation date. SQL provides the ORDER BY clause as solution. Django implements this class in the *order_by* function to enable the developer to use the SQL functionality. Therefore it is required to pass the name of the table columns to the function. It is also possible to pass names of joined table columns, but because there can be more than one result in a join, the result can be unexpected.

```
1 series = Series.objects.all() \
2   .order_by('start_date')
```

Listing 5. Django search request with order_by

SQL allows to create queries which calculate or count special data. Therefore Django provides the *annotate* function, which allows the use of the newly generated parameters in filters or the order by clause. This can be used for ordering the series by the number of lectures it offers.

```
1 series = Series.objects.all() \
2   .annotate(number_of_lectures=Count('
      lecture__id')) \
3   .filter(number_of_lectures__gt=2) \
4   .order_by(number_of_lectures) \
```

Listing 6. Django request with annotation

Using a combination of these different possibilities easily produces complex SQL statements. This results in the use of a large amount of resources and the risk, that the database server is not capable of answering the requests.

That is why Django also provides the possibility to have more influence on the query generation using the *extra* function. This function allows to manually add new tables and build own SQL statements. But it also compromises the clear partition between data class models and the database.

This brings the problem that the generated query is more database dependent than it would be, when using the more common Django database ORM functions. When using some extra functions for more complex queries, we discovered, that even small differences of the database version result in a big test effort. That is why we decided to try to surrender the usage of this function in the search context.

## III. Implementation of the search engine

For the implementation of the search engine, we wanted to use the flexible approach of our plug-in architecture. Therefore every function of the search should be expandable using plug-ins. In the following sections, there will be a description of the base plug-in interface and the implementation of the different search tasks. At the end follows an example describing how the search is enhanced with play list data.

### A. SearchBase class

The base class for the whole search functionality is the *SearchBase* class. It is an abstract class which provides some functionality to generate search queries. Most importantly it is the template for the generation of search type specific classes.

This template is used for every search type, e.g. for searching a series the class *SearchSeries* is created which inherits from *SearchBase*. This class will have the query *Series.objects.filter(isVisible=True)* pointing to the database model of series and performing some extra request to check if the series should be visible to the user. It also contains a list of possible order by parameters which can be extended

by subclasses. This class can also contain the basic search filter for searching for title or description.

```
class SearchBase(PluginBase):
  searchType = ''
  searchFieldsForm = ''
  query = None
  plugins = list()
  orderBy = list()

  def __init__(self, searchFieldsForm):
    [...]

  [...]

  def generate_q_object(self, fieldtype,
      fieldlogic, fieldtext):
    return None

  def prepare_order_by(self, query, order_by
      = None):
    for plugin in self.plugins:
      query, order_by = plugin.
          prepare_order_by(query, order_by)
    return query, order_by

  def generate_search_query(self,
      userIsIntern=False, order_by = None):
    [...]
    return query
```

Listing 7. Structure of the class SearchBase

### B. Filter/Exclude

The most important function for the search is the possibility to reduce the number of search results by using search terms for different searchable fields. These search fields must be defined by plug-ins, because it has to be possible to extend the list of searchable fields, when new searchable data is available.

Therefore the base class SearchType (see listing 8) is defined. This class collects the possible types, providing a name and a display name for each.

```
class SearchType(object):
  ''' Types available for search '''
  name = ''
  displayName = ''

  def __init__(self, name, dName, pos=1000):
    self.name = name
    self.displayName = dName
    self.position = pos
```

Listing 8. Class SearchType

After defining the search types it is also important to define, how the field should be searched. As basic search logics, there can be the following ones:

- **contains:** If it is selected, it is checked, whether the field contains the search string
- **does not contain:** If it is selected, it is checked, whether the field does not contains the search string
- **is:** If ii is selected, it is checked, whether the field is equal with the search string
- **is not:** If it is selected, it is checked, whether the field is not equal with the search string

```
class SearchLogic(object):
  ''' Logics available in search '''
  name = ''
  displayName = ''

  def __init__(self, name, dName, isP=True):
    self.name = name
    self.displayName = dName
    self.isPositive = True
```

Listing 9. Class SearchLogic

The base class for the search logic is defined analog to the design of the search types (see listing 9).

```
def generate_search_query(self, userIsIntern=
    False, order_by = None):
  query = self.query
  for form in self.searchFieldsForm.forms:
    try:
      [... (prepare data)]
      # Building Q-Objects for better
          chaining of filters
      if len(fieldtext) > 0:
        q = None
        for plugin in self.plugins:
          q_add = plugin.generate_q_object(
              fieldtype, fieldlogic,
              fieldtext)
          if q and q_add:
            q = q | q_add
          elif q_add:
            q = q_add
          if q:
            if checkSearchLogicIsPositiv(
                fieldlogic):
              query = query.filter(q)
            else:
              query = query.exclude(q)
    except Exception, e:
      print "Error while generating search
          query", e
  [...]
  return query
```

Listing 10. Function generate_search_query (Filter query)

With these two plug-in interfaces defined, it is possible to enhance the basic *generate_search_query* function of the *SearchBase* class. Therefore every plug-in of the base class, thus every subclass of this class, is called with the function *generate_q_object* (listing 10 line 10). The specialized function is able to generate a Q object, which is combined with other generated Q objects and added as filter or exclude to the existing query (listing 10 line 16).

### C. Order by

Another important function for the user when displaying the results is the possibility to display them in a specific

order. Therefore the Django *order_by* function is used. The default use case will be the ordering by an existing database field, like the name or the start date of the series or lecture. But it should also be possible to use annotations for the preparation of order by. Therefore the *prepare_order_by* function (see listing 11) is provided, which can be used to enrich the query with additional data, using Django annotations. This is useful for ordering by number of elements or an average value, for example when ordering lectures by their ratings.

```
24 def generate_search_query(self, userIsIntern=
       False, order_by = None):
25   query = self.query
26   [...]
27   if order_by:
28     query, order_by = self.prepare_order_by(
           query, order_by)
29     if order_by:
30       query = query.order_by(order_by)
31   return query
```

Listing 11. Function generate_search_query (Order_by)

With these two possibilities using filter and order by function for generating a search request, the most common operations are available. In the following paragraph there will be a short description, how the SQL search can be extended using pre and post processing of the data.

### D. Other enhancements for the search

On the one hand it is possible, that the normal search result creation is not finished after executing the SQL query. It may not be enough to filter all possible results and order them. Furthermore it is required to check for additional criteria or add kindred elements to the list of search results.

Therefore the base class can be extended by extra functions, which iterate the result list and add or remove entries of this list. These functions will create more overhead while gaining the results, but allow to provide database independent filters.

On the other hand it can be necessary to process the search terms before creating the database request. For example it is possible to find similar terms, which can be searched as well. Therefore a function can be provided, which scans all search terms and returns a post-processed list of these words.

### E. Example: play lists

Play lists provide the possibility to the users of the tele-TASK portal to create own compositions of the content in the portal. For example it can be used to provide a subset of a course or the combination the same topic lectured in different courses.

Play lists provide their own data structure using the same media entries as scenes and lecture. Through this connection it is possible to find the play lists connected with a specific lecture or series.

To search for series or lectures, which are part of a play list with a specific title, it is necessary to extend the search types with a search type for the play list title. Aside it is important to create the entry point for the play lists search be inheriting from the *SearchBase* class (see listing 12). This entry point will have the *Playlist* model class as starting point for the query.

```
1 class PlaylistSearch(SearchBase):
2   def __init__(self, searchFieldsForm)
3     super(PlaylistSearch, self).__init__(
          searchFieldForm)
4     query=Playlist.objects.all()
```

Listing 12. Structure of the class PlaylistSearch

With this class it is possible to write the search queries for all known basic search types, like the title of series as well as the new search type. Next to this, it is also essential to extend the base search classes for lectures and series, so they are capable of searching their content with this new search type (see listing 13).

```
1 class SearchSeriesPlaylist(SearchSeries):
2   def generate_q_object(self, fieldtype,
        fieldlogic, fieldtext):
3     q = None
4     if fieldtype == "search_playlist_title":
5       if fieldlogic == "search_Contains":
6         q = Q(lecture__media__playlistentry_
              _playlistgroup__playlist__name_
              _text__icontains = fieldtext)
7       [...]
8     return q
```

Listing 13. Extension of the search of series with play list title

With these few functions, the search is capable of displaying the search results for play lists next to the existing results and search for lectures, which are used in the play lists found with these search criteria.

## IV. Advantages and Disadvantages

As with every implementation there are some disadvantages and advantages. We made experiences after the implementation, which could not be foreseen at the beginning.

Disadvantages:

- Development of complex queries: Because of the invisibility of the complexity of the whole search query, a query can be too complex for the database to process. This is a big problem, when combining the queries, because during development the worst combinations are not tested properly. So an excessive load on the database server can happen. This can be avoided when testing each new statement accurately.
- Problems with SQL-optimization: The queries are generated by the plug-in architecture with some restrictions on what a developer of a plug-in can do. So there are less possibilities for doing SQL query optimization, then there would be using raw SQL statments.

This problem comes with the usage of the Django implementation for generating database queries. So it would also appear, if the Django ORM is used anyway. It can be solved by using more plug-ins for managing sub queries or saving reusable intermediate data.

Advantages:

- Easy embedding of new data: With new types of meta data being generated through new algorithms, it is easy to include this data inside the existing search. As seen in the play list example, it is not much work to enrich the standard search with new data.
- Module independencies: The search module does not depend on other more specific modules, like the play list or annotation module. Therefore it is possible to have a system without this function. Just by adding these modules, the new functions will be available for the users.
- Test possibilities: It is easy to exchange different implementations for testing. Sometimes it is not clear, which solution is the best for a specific search task, so it is possible to implement more than one solution and to test them against each other. It is also possible to use different search solutions parallel to each other to do a direct comparison of the search results.
- Combination of search tasks: Due to the Object Related Modelling, the different search tasks can be combined easily for a better search result. For example it is possible to search for a lecture with a specific title which is held by a specific person and which is used in a play list with a given name.
  Because the number of possible combinations will explode with the increasing of search types, this request is enabled through the automatic plug-in chaining.

Most of the problems we had to overcome are a result of the usage of the Django ORM and of the complexity of our data structure. Obviously we had these problems, because it was easy to produce more complex requests, but these requests are the results the users need. Therefore the architecture itself makes it easier to fulfil the wishes for new enhancements of the search function.

## V. Summary and future works

The basic search functions are implemented and used by the users of the portal. Therefore the design works fine. It is also possible to use the architecture for enhancements as seen when implementing the search for play lists and other data like audio data retrieval and annotations.

For the future there should be more data gained, like OCR data and data from the log files. More important than using more data is a better usage of the data. So the functions should be enhanced to use semantic technologies for the expansion of the search domain to find similar search terms or connections between different lectures.

It is also important to do more optimization of the search requests. Therefore it is required, to provide more or better functions in the search plug-in interface.

In the end, when we are capable of quickly finding appropriate results, these results should also be used to find related objects inside the portal, like related lectures or series. With this function it is possible to give the users informations about topics they could be interested in.

## References


[1] R. Guha, R. McCool, and E. Miller, "Semantic search," in *WWW '03: Proceedings of the 12th international conference on World Wide Web*. New York, NY, USA: ACM, 2003, pp. 700–709.

[2] J. Waitelonis and H. Sack, "Augmenting Video Search with Linked Open Data," in *Proc. of Int. Conf. on Semantic Systems 2009, i-Semantics 2009*, 2009.

[3] S. Linckels, S. Repp, N. Karam, and C. Meinel, "The virtual tele-task professor: semantic search in recorded lectures," in *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2007, pp. 50–54.

[4] L. Zhuhadar and O. Nasraoui, "Personalized cluster-based semantically enriched web search for e-learning," in *ONISW '08: Proceeding of the 2nd international workshop on Ontologies and nformation systems for the semantic web*. New York, NY, USA: ACM, 2008, pp. 105–112.

[5] Q. Cui and A. Dekhtyar, "On improving local website search using web server traffic logs: a preliminary report," in *WIDM '05: Proceedings of the 7th annual ACM international workshop on Web information and data management*. New York, NY, USA: ACM, 2005, pp. 59–66.

[6] J. Zhou, C. Ding, and D. Androutsos, "Improving web site search using web server logs," in *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*. New York, NY, USA: ACM, 2006, p. 22.

[7] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, 2002. [Online]. Available: http://dx.doi.org/10.1145/514183.514185

[8] M. Jazayeri, "Some trends in web application development," in *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 199–213. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.26

[9] V. Schillings and C. Meinel, "Tele-task – tele-teaching anywhere solution kit," in *Proceedings of ACM SIGUCCS*, Providence, USA, 2002.

[10] K. Wolf, S. Linckels, and C. Meinel, "Teleteaching anywhere solution kit (tele-task) goes mobile," in *SIGUCCS '07: Proceedings of the 35th annual ACM SIGUCCS conference on User services*. New York, NY, USA: ACM, 2007, pp. 366–371.

[11] M. Siebert, F. Moritz, and C. Meinel, "Establishing an Expandable Architecture for a tele-Teaching Platform," in *2010 Ninth IEEE/ACIS International Conference on Computer and Information Science Article*. Yamagata, Japan: IEEE Computer Society, 2010.