Establishing an Expandable Architecture for a Tele-Teaching Portal

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

Franka Moritz
Hasso Plattner Institut
Universität Potsdam
Potsdam, Germany
franka.moritz@hpi.uni-potsdam.de

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

***Abstract*—The complexity of web applications becomes a big problem, when applications are growing. Especially if new algorithms should be tested and compared, it is important to be able to separate the newly implemented functions from the core system. It is not enough to use standard frameworks to handle this, but to plan a maintainable structure for the application.**

This paper describes how a maintainable structure can be planned and implemented by using the Django framework while trying to avoid a big programming overhead.The new architecture for the main features and its advantages and disadvantages is described together with an example for a rating plug-in. An outlook on the possibilities for future developments using this architecture is provided as well.

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

I. Introduction

When designing a real world internet application, which is used by a big amount of users, it is important to have a stable basis to start with. It is important to assure at all time, that the primary functions work well and extra functions do not disturb the functionality.

As the tele-teaching portal is an academic project, it is also important to be able to compare different solutions and test new algorithm. For a tele-teaching portal there are two main research fields. On the one hand there are all kinds of interaction with the users. User interactions is one of the aspects for future developments in web applications [1]. Therefore there has to be an easy solution for the integration of community features like rating [2] or tagging [3]. Thereby this function should be integrated into the core functionality for a generation of a seamless usage experience.

On the other hand a lot of research focuses on improvement of the search of data. For the implementation of search algorithms for video lectures, a lot of meta data is needed. This data can be generated from the audio and video data [4] as well as be provided by the users. Even statistical data can be used. There is a lot of research which integrates these different approaches [5]. Nowadays a lot of implementations using semantic web technologies [6] are proposed.

Therefore this paper will start with some information about the application and a short definition of plug-ins. Afterwards the two approaches for the architecture, without and with plug-ins, are described and the advantages and disadvantages of the usage of plug-ins are named. At the end there is an overview of the future work, which is enabled through the architecture.

A. About the tele-TASK Portal

The implemented tele-teaching portal tele-TASK[1] provides a large amount of lectures and videos, most of them held at the HPI. The videos 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 for providing 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.

B. What is a Plug-in?

Many web-frameworks use the term plug-in, sometimes also called add-on, even if they don't offer anything which could be named plug-in. That is a reason why many web application developers have a wrong association for this term. In this paper plug-ins are defined as in classical software development.

A plug-in can be seen as a special interface description. This interface defines some functions or classes, which can be used by developers to extend the basic software. Big software with a lot of options for usage, use this concept to make their software more flexible. For example it is possible to write plug-ins for the web browser firefox, to enhance the functionalities of the web browser. There is even software, like eclipse which consists of pure plug-ins[7].

These interfaces enable a developer to write functions, which will automatically be included inside the system without any further changes on the core system. However

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

978-0-7695-4147-1/10 $26.00 © 2010 IEEE
DOI 10.1109/ICIS.2010.141

in the context of web applications the term plug-in is often used for libraries, which can be "plugged in" a system and used by the developer, which is not the original idea of a plug-in.

In fact plug-in architecture means an inversion of control. Not the author of the plug-in knows, when the plug-in is used, instead the developer, who writes the interface for the plug-in knows, what will be done with it and describes the needed data and functions. But he does not have to know, how these functions will be implemented. For example the interface developer writes an interface for data enhancement. He only knows, that he will get enhanced data, but he does not know, who will do it and what exactly will be done.

This concept of plug-ins is known from many other problems. Many projects[8][9] describe how they implement their own plug-in architecture or use existing interfaces. Nevertheless it is hardly known to web developers and most big web frameworks do not include mechanisms for using a plug-in architecture. So it is required to build it by hand.

II. Classical Approach for a Tele-Teaching Portal

The tele-TASK portal has been running for some years now, and has undergone some major changes. In summer 2009 there was the chance to do a complete restart using new technologies. This restart allowed to redesign the database. On top of the experiences with the old portal, there was also the decision to build more modules and separate the functions. In this section at first there is a short description of the used framework Django. Afterwards follows the description how the project would have been built without a plug-in architecture.

A. The Concept of Django

Django[2] is an open source web application framework. It is based on the python programming language and implements the MVC[3] pattern. To be precisely it is called MTV-pattern in Django, because the parts of the application are named model, template and view.

These three terms describe the most important parts of the Django system. The following short descriptions give an overview of the characteristics of Django to understand the following sections.

Models

Models are the representation of the data, which can be stored in different types of databases. Therefore it is a mapping of database tables to classes and table entries to objects. Models also provide the possibility to add helper functions. For example it is possible to create a function which converts a date into another format.

[2]http://www.djangoproject.com

[3]MVC-Pattern: Design pattern: Model-View-Controller postulates the separation of the application in data, program logic and display

They also help with the abstraction of some database design pattern and hide overhead database tables, like connection tables between two data tables.

Views

The task of the view is to collect the required data and provide it. Therefore it is possible to create a string containing the HTML code or use a template, which is a HTML file with special template syntax. The second approach is the usual one. Using templates, a view gets the data from the models and prepares it for the template, which will be rendered.

The views are also the access point for the URLs. In a special config file there are definitions which URL will call which view with the specified parameters. The config file also provides the possibility to create abstract names for the precise URLs, so that the developer does not have to know, which URLs will be used in the end. The developers only use the names for the URLs.

Templates

Normally templates are plain HTML files with a few template tags for variable output and loopings over lists. It is also possible to do some extra formatting on the variables using filters, such as transforming the format of a time stamp. Another possibility to extend templates is the usage of template tags. With the help of these tags, it is possible to insert every type of function inside the template.

Internally in Django, templates are interpreted as trees, consisting of different nodes. Each template tag creates a new node in the tree. It is possible that these template tags are nested. Thus a complex tree of tags can be produced. Because of this structure, each template tag has to produce exactly one node which implements a render function. Furthermore it is allowed, that this generated template node contains other template nodes as children.

B. Structure of a Modularized Application for the tele-TASK Portal Without a Plug-in Architecture

When designing an application it is a good concept to create modules for the different tasks. An important feature of these modules is access limitation and loosely coupling. As we tried to separate the modules by tasks, the result was modules like the login and profile managing module mytt, the search module or the module for rating (see figure 1). Using a non-plug-in solution, there will be a bidirectional connection between almost all modules.

This happens because on the one hand the core pages including the main entrance page of the application need to have references to the enhancement modules pages, so that the user of the application can reach all functions. A case in point is the search dialogue on the main page for an easy access of the search function. It is also possible that additional modules provide information, which should

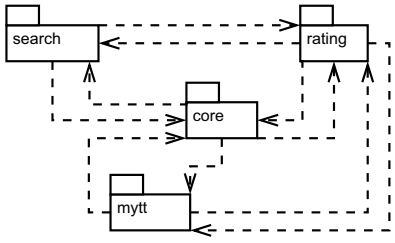

Figure 1. Extract of the structure without using plug-ins

be displayed on the core pages. For instance the ratings of a lecture should be visible and perhaps editable on the page of the lecture.

On the other hand the additional modules have to use the data of the basic modules. It is obvious that most modules use the data of videos or lectures for their own functions. They will also use functions of the core module for editing and displaying the data to get a consistent application.

The modules do not only communicate with the core module, but also with each other. For example there will be a search for play lists included inside the normal search for lectures and videos. It should also be possible to order the search results using the ratings of the content. The mytt module will be required as connection point by every other module, which provides special functions for logged-in users.

The resulting problem is, that for every new module you have to change existing other modules. It is obvious, that this will exponentially increase the complexity of the whole application. It is also not possible to remove a module or exchange it without touching the other components.

III. Concept for a plug-in architecture

The idea for the extendable application is, that the basic modules do not have to know anything about the specialized modules. In the end there will be a hierarchy of modules like in figure 2.

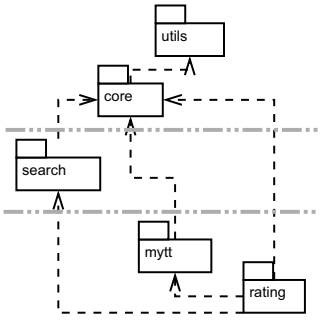

Figure 2. New structure of modules with plug-ins

As you can see, only specialized modules use data and interfaces of the more common modules. The basic modules only provide plug-in points, which can be used by the specialized modules.

With this structure there is no problem extending the application with new functions without touching any of the core modules. It is also easily possible to remove a plug-in module, which is not needed by any other modules, therefore is a leaf of the hierarchy tree.

To get such a structure, two changes of the concepts are needed. The first one is the extension of the views, by providing an interface for templates and a plug-in class mechanism. The second one is a design approach for the extension of data.

A. Extending the view

Django has no built-in concept for designing plug-ins for extending the views. It is a task for the developer to implement his own plug-in system.

A good approach to start with is to look at the programming language Python, in which Django is implemented. In Python there are a lot of ideas on how to design a plug-in system. An easy solution is to use the knowledge of a class about its subclasses. With this idea, everything you need is just a base class, which is capable of querying all its subclasses. An implementation of this approach can be found in listing 1.

```
class PluginBase(type):
  def __init__(cls, name, bases, attrs):
    pass

  def get_plugins(self):
    plugins = list()
    for sc in self.__subclasses__():
      plugins.append(sc)
      add = sc.get_plugins()
      for s in add:
        plugins.append(s)
    return plugins
```

Listing 1. Simple version of class PluginBase

This listing contains only a small snippet. Of course it is better, to do some extra checking if the subclass is really suitable and well formed.

```
import settings

for installedApp in settings.INSTALLED_APPS:
  try:
    __import__(installedApp + ".plugins")
  except Exception, e:
    [...]

  # Find template names of plug-ins
  [...]
```

Listing 2. Function for loading plug-ins

The problem is, that a class has knowledge only about those subclasses which have been used or included somewhere in the application. So the most convenient way to assure all subclasses are known is initializing them

when starting the Django server. Therefore we use the INSTALLED_APPS setting (see listing 2), which contains all available modules and is used for other initializations as well.

Then every known plug-in is accessible and can be used inside a Python function or class. For addressing a plug-in from the template it is required to write a special template tag.

Preparation for template tags: A template tag is a function which returns a template node object. These objects offer the necessary render function, which usually creates HTML data for displaying inside the web browser.

```
class PluginTemplateNode(template.Node):
  pluginTemplateName = "base"

  def __init__(self, data, tName = ''):
    if data:
      self.data = template.Variable(data)
    else:
      self.data = None
    self.templateName = tName

  def render(self, context):
    data = None
    if self.data:
      data = self.data.resolve(context)
    if self.templateName and len(self.
        templateName) > 0:
      t = get_template(self.templateName)
      return t.render(context)
        return ''

  __metaclass__ = PluginBase
```

Listing 3. class PluginTemplateNode

Therefore it is important, that each plug-in class, which is used for a template plug-in, inherits from the template node class. So it is convenient to write a base class like PluginTemplateNode (see listing 3), with some handy functions, which uses PluginBase as meta class.

```
{% show_plugin 'LectureInfo' lecture %}
```

Listing 4. Using plug-ins inside templates

The class variable pluginTemplateName is used, to get a unique identifier for using in the template tag. In listing 4 the usage of a plug-in for additional infos about lectures is show. The corresponding *PluginTemplateNode* class will have the value *LectureInfo* for pluginTemplateName.

Because plug-ins can create more than one template node but only one template node is allowed to be returned by a template tag, it is the easiest solution to have a template node which collects all of the plug-in template nodes and renders them, when its own render function is called (see listing 5).

With these two classes a general template tag can be written easily by retrieving all plug-in classes which inherit from *PluginTemplateNode* and append them to a *PluginColNode* object, which will be returned by the template tag.

```
class PluginColNode(Node):
  def __init__(self):
    self.plugins = list()

  def append(self, node):
    self.plugins.append(node)

  def render(self, context):
    renderresult = ''
    for p in self.plugins:
      if isinstance(p, PluginTemplateNode):
        res = p.render(context)
        if res is not None:
          renderresult = renderresult + res
    return renderresult
```

Listing 5. class PluginColNode

B. Extending the Models

It is not enough to only extend the view of the application, because it is a normal use case to have additional data. The normal database approach is to construct a new table and add a foreign key to the existing table to connect the data. As long as you only want to extend one table this is a valid construction. If you want to extend more than one table with the same feature you have to use a generic relation (see figure 3).

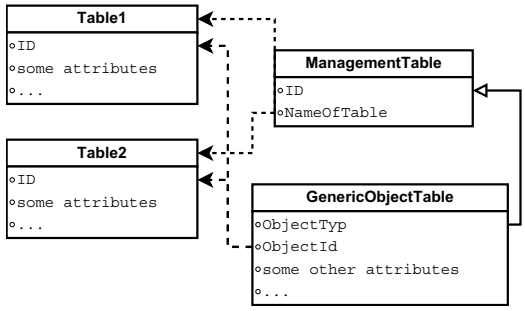

Figure 3. Database structure for generic relations

This concept, which is part of the Django distribution, allows to create a foreign key to multiple database tables by saving the type of the table and the id of the object. Because Django handles all database requests, the database structure and the management table are invisible to the application developer and generic foreign keys can be used almost like normal foreign keys.

C. Example: Rating-Functionality as Use Case for Plug-ins

The rating functionality is one feature that ought to be implemented as plug-in in a tele-teaching portal, because it is no core functionality and should therefore only be implemented in a module that can be switched on to work with the core application (see figure 4). In the context of the rating of media items rating is the quantification of the personally perceived quality of an item.

Figure 4. on the top: rating disabled - thereunder: rating enabled

In the tele-teaching context there are several layers where rating can be applied. Usually a tele-teaching portal consists of lecture recordings that are mostly embedded in a larger context, for example the course which runs a whole semester. Furthermore the lectures are often subdivided into smaller pieces. This is done in order to facilitate the usage of mobile players where the content needs to be downloaded, for pod-casting and also to simplify a more precise meta-data collection and search. As all the three layers include tele-teaching content, all of them should be rateable individually.

A rating will be stored in the database with the help of a new model that extends the core model base. The model *rating* combines the information of the rated content item, information about the user who posted the rating, the date when it was posted and the value of the rate (see figure 5). The content item is referenced via the combination of *contentType* and *objectID*. This so-called generic foreign key enables a flexible referencing of different content types on the application. That results in a rating functionality that could easily be adapted to not only work with series, lectures and segments but also with new content types like play lists.

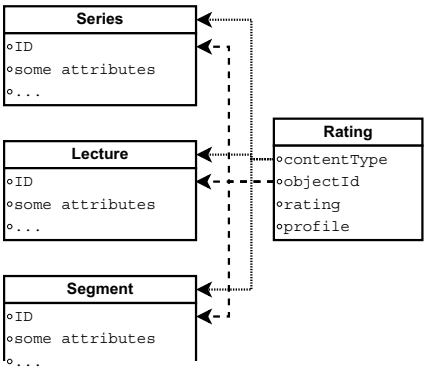

Figure 5. Database structure for rating

There are multiple places where the ratings can be used to enhance the user interface. The result of the rating should be shown in all places where the content items that can be rated are previewed, like the search page and overview pages in the video archive and its categories. The possibility to rate should also be given on all pages related to the content item, like the lecture details page and the video display page. This is necessary to ensure easy access and visibility of the functionality for the users. The rating interface checks the login status to forbid participation of anonymous users.

For the users to manage their actions in the personal area of the portal, called MyTT, an interface for managing ones own votes is required. Deleting and altering of votes should be allowed here.

The functionality for all plug-ins is implemented by extending the core view collection with new views. An interface from which all plug-ins might inherit is implemented within the core part of the application. There are two different plug-ins that offer the display of rating results and the rating interface with one interface implemented in the core application for each of them. One plug-in only displays a smaller version of the rating results on overview pages (see image 4). The second plug-in implements both a normal-sized display of the results and the rating interface. It is used on all detail pages for content items and videos.

The integration of the functionality into the standard templates works via the usage of template tags. At the place where the output of the rating is supposed to appear, a template tag which calls the rating plug-in and passes the content item whose rating should be displayed is included (see listing 6).

```
1 <div class="ratings">
2   {% show_plugin 'Rating' lecture %}
3 <div>
```

Listing 6. Integration of the rating plug-in via template tag

MyTT offers its own plug-in interface for the integration of sub-pages. One interface function realizes a menu link in MyTT, the other a visual link button in the interface. Both link to the plugged-in sub-page. Two rating plug-ins inherit from these functions to integrate the ”Manage my Ratings” interface into MyTT.

IV. Advantages and Disadvantages

As every technology plug-ins have their advantages and disadvantages and it depends on the use case if it is a good choice to use them. We gained these facts by collecting the experiences of the 10 developers in our teams, who started with different knowledge about architecture.

The following disadvantages became apparent within the project:

- Learning Tasks: The time for orientation is increased. A new developer has to learn the basic concepts of plug-ins and must be taught to follow the rules for new implementations. Therefore higher development skills are needed.

- Overhead for small tasks: A small link can need a whole function or class to be implemented instead of just changing the HTML file.
- High complexity for small projects: If there are only one or two functions, which are capable for plug-ins and it is known, that the application won't grow, plug-ins are not needed.

On the other hand there are a lot of advantages:

- Easy on and off switching of functionality: If any plug-in produces errors, it can easily be disabled. Disabling defective plug-ins works automatically as well. It is also possible to replace functions with better implementations.
- Different status for development and running system: It is possible to develop functions and have the code inside the project, but to disable them for the live application.
- Module separation: The modules are separated, therefore it is easier to outsource some tasks, by defining the interfaces. Furthermore this allows the production of smaller and larger versions of a system without any changes in the implementation.
- Less knowledge of the system needed: Developers for plug-ins need no knowledge of the core system, they only need the specification for the interfaces, they are using. This reduces the learning effort. So for bigger projects this overcompensate the time for understanding the plug-in concept.
- Decreasing complexity: In bigger projects the design overhead is relatively small. Furthermore it saves redundant tests inside the templates and views and decreases complexity of the project.
- Smaller influence on the different development tasks: Because not every developer has to use every plug-in it is easier to ignore developments in the initial stage.

In our project, we concluded, that the few disadvantages do not outweigh the advantages. It needs some time for the developers to get used to the paradigms. After that phase the developers think in a different way when designing their modules and would hardly miss the possibilities they gained.

V. Conclusion and Future Work

The actual implementation has been integrated into the portal and is used for over half a year now. The next step is to encapsulate the functionality of plug-ins into an easy-to-use project, which can be used project-independent. This project will be shared with the Django community. This would allow to gain more experiences with the plug-in architecture. These experiences should be collected so it can be checked, whether this approach is capable for other projects as well. Also one project team at our research group is waiting to test the architecture themselves.

Aside the plug-in architecture should be a a tool for future work. There are a lot of ideas, whose implementation is enabled through the architecture.

A big task is the implementation of the basic search features. There are two points, where plug-ins are needed. First there are a lot of different types of objects, which can be searched. In the core system there are series, lectures, segments and persons, but later there must be also a search for play lists or other data.

The second point is the definition of how the search should work. The core search will be capable to search the core meta data of a lecture or series. But with more meta data generated in different ways, like audio or OCR analysis, it is important to enhance the possibilities of the search engine and to improve the queries.

When this base search functionality is implemented, it should be used for further enhancements on the base of the different types of meta data available. It is planned to give more information about relations between different lectures or series.

There will be also other possibilities opened up by plug-ins, like the embedding of more community features, which will give new approaches for research. We plan to allow the users to provide more meta data using tags and to evaluate, if this data is useful for enriching the user experiences of the portal.

References

[1] 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.

[2] F. Moritz, M. Siebert, and C. Meinel, "Community rating in the tele-lecturing context," in *Proceedings of the IAENG International Conference on Internet Computing and Web Services (ICICWS'10)*. Hong Kong: IAENG, 2010.

[3] D. Liu, X.-S. Hua, L. Yang, M. Wang, and H.-J. Zhang, "Tag ranking." in *WWW*, J. Quemada, G. León, Y. S. Maarek, and W. Nejdl, Eds. ACM, 2009, pp. 351–360.

[4] S. Repp, A. Gross, and C. Meinel, "Dynamic browsing of audiovisual lecture recordings based on automated speech recognition." in *Intelligent Tutoring Systems*, ser. Lecture Notes in Computer Science, B. P. Woolf, E. Aïmeur, R. Nkambou, and S. P. Lajoie, Eds., vol. 5091. Springer, 2008, pp. 662–664.

[5] M. G. Noll and C. Meinel, "The metadata triumvirate: Social annotations, anchor texts and search queries." in *Web Intelligence*. IEEE, 2008, pp. 640–647.

[6] 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.

[7] D. Birsan, "On Plug-ins," *Queue*, no. March, 2005.

[8] N. Pinkwart, "A plug-in architecture for graph based collaborative modeling systems," in *Shaping the Future of Learning through Intelligent Technologies. Proceedings of the 11th Conference on Artificial Intelligence in Education*, F. V. . J. K. U. Hoppe, Ed. Amsterdam: IOS Press, 2003, pp. 535–536.

[9] 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.