Resource management for plugins in the Dynamic Presentation Generator

Kelly Alexander Teigland Whiteley

Department of Informatics
University of Bergen
Norway

Long Master thesis
November 2011

Foreword

This document is the result of my master degree in informatics at University of Bergen.

The thesis evaluates and improves the plugin architecture of Dynamic Presentation Generator 2.1.

I would like to use this opportunity to give special thanks to my supervisors Khalid A. Mughal and Torill Hamre for their great cooperation, dedication and advice throughout my whole Master degree. Thanks to Haakon Nilsen for great help with technical issues and advice. Thanks also to my fellow Master student Aleksander Waage for the cooperation in this Master thesis and the courses leading up to it.

Thanks to my fellow Master students Øystein Lund Rolland, Morten Høiland, Aleksander Vines and Jostein Bjørge for supporting me and making this all a very good and memorable experience.

Finally, I would like to give thanks to my family for all the support throughout my studies, with special thanks to my wife, Lene Whiteley, for all her loving support and encouragement.

Kelly Alexander Teigland Whiteley
Bergen, 19. November 2011

Contents

List of Figures

List of Tables

Source Code

Definitions

ACID:	Atomicity, Consistency, Isolation and Durability
API:	Application Programming Interface
CLOB:	Character Large OBject
CMS:	Content Management System
CRUD:	Create, Read, Update and Delete
CSS:	Cascading Style Sheet
DAL:	Data Access Layer
DAO:	Data Access Object
DBMS:	DataBase Management System
DPG:	Dynamic Presentation Generator
HTML:	HyperText Markup Language
HQL:	Hibernate Query Language
IoC:	Inversion of Control
JAFU:	JAva i Fjern Undervisningen
JAR:	Java ARchive
JCR:	Java Content Repository
JDBC:	Java DataBase Connectivity
JDOM:	Java Document Object Model
JPA:	Java Persistence API
JPG:	Java Presentation Generator
JPQL:	Java Persistence Query Language
JSR:	Java Specification Request
LOB:	Large OBject
MVC:	Model View Controller
ORM:	Object Relational Mapping
OXD:	OpenX Data
PCE:	Presentation Content Editor
PM:	Presentation Manager
PV:	Presentation Viewer
RDBMS:	Relational DataBase Management System
SQL:	Structured Query Language
TDD:	Test Driven Development
UiB:	Universitetet i Bergen (eng:University of Bergen)

UML:	Unified Modeling Language
URL:	Uniform Resource Locator
WYSIWYG:	What You See Is What You Get
XML:	Extensible Markup Language
XP:	eXtreme Programming
XSLT:	Extensible Stylesheet Language Transformations

1 Introduction

1.1 Background

Java in distant learning (JAFU)(Norwegian: *Java i Fjernundervisningen*) is a project at the Department of Informatics at the University of Bergen (UiB), which was started in 1999. Its goals are to offer web-based courses for students who do not have the opportunity to attend campus lectures. JAFU has mainly offered the two Java programming courses; *INF-100F* and *INF-101F*. These courses are composed mostly of the same curriculum as their campus counterparts; *INF-100* and *INF-101*. Lecture notes, assignments and other resources are made available on a web site for the students. The lecturer and teaching assistants of the course manage these resources, publish news and answer questions from students. To make this easier, tools for web based publishing and dynamic content management have been developed as part of the JAFU project.

Khalid A. Mughal presented the concept of *presentation patterns* in the article *Presentation Patterns: Composing Web-based Presentations* [75]. This concept describes how content can be separated from the presentation, promoting reuseability. The first implementation of this concept was in the system *Java Presentation Generator*, developed under the JAFU project in Kevin Chruickshank's Master thesis [23]. The goal was to design a web site for courses.

In 2004, the *Dynamic Presentation Generator* was developed. This took over from JPG as a new and improved implementation of the presentation pattern concept. It was intially developed by Yngve Espelid during his Master thesis [28], and has since been the focus of development at JAFU by many Master students. DPG is a generic *Content Management System* which is used for building web pages dynamically and managing their content.

In 2008, Karianne Berg [17], Bjørn Ove Ingvaldsen [60] and Bjørn Christian Sæbak [93] evaluated DPG in their Master theses. Their conclusion was that DPG proved difficult to use and was very error prone, because it required knowledge of the system and technologies to publish and manage content. It was therefore decided to redesign DPG from scratch, leading to version 2.0. This version provides with an intuitive web interface for publishing and managing content, as well as a modern, modular system architecture.

DPG was further developed in 2010 by Tobias Olsen [78] and Peder Skeidsvoll [94], leading to the current version of 2.1. This included a vast improvement of the plugin architecture of DPG, as well as general improvements of functionality and robustness of the system.

Morten Høiland [52] and Øystein Lund Rolland [91] developed complex plugins for data collection using XForms [112], in their Master theses. This tested DPG's support for user interaction and collection of large amounts of data, and ultimately proved that DPG does not facilitate for effective and efficient functionality for data collection.

This Master thesis evaluates and improves the current plugin architecture of DPG 2.1, with the main focus of improving persistent plugin data and resource handling. Parts of the thesis will be work in collaboration with Aleksander Waage [114], which is specified at the start of the relevant chapters.

1.2 Motivation

DPG has proved to be successful for content management and designing web-based presentations. Unfortunately, it has been lacking in support for data collection and user interaction, which are the plugins' responsibilities in DPG. The current plugin architecture limits plugins to very simple data collection. To expand DPG's applications, the plugin architecture needs to be improved. This is the main motivation for this Master thesis.

The trigger for this motivation was discussions in the JAFU project regarding mi-

gration of data with other systems, specifically *OpenX Data* (OXD) [79] and its large amounts of geographic data. This required complex and dynamic map plugins, as well as good support for plugin resources.

1.3 Goals

1.3.1 Overall goal

The overall goal of this Master thesis is to expand the plugin architecture of DPG to provide a better plugin resource solution and facilitate for more complex plugins.

1.3.2 Subgoals

The overall goal is composed of the following sub-goals:

- Evaluate the current plugin architecture of DPG 2.1
- Propose and evaluate solutions for improvement of the plugin architecture in DPG, and implement the best solutions
- Evaluate the current solution for persistent storage of DPG plugin data
- Propose and evaluate new solutions for plugin resources in DPG
- Implement an improved plugin resource solution for DPG
- Provide guidelines for using the new plugin resource solution

1.4 Development methodologies

Throughout the development, the candidate has tried to follow *agile* development methodologies. Principles were mainly taken from *eXtreme programming* (XP) [67], such as *test driven development*(TDD), *pair programming*, *collective code ownership*, *refactoring* and *modular design*. Martin Fowler's guidelines for clean and agile code development [68] have also been followed. This made a lot of sense, given that DPG is being worked on, and will continue to be worked on by multiple Master students in the JAFU project.

1.5 Development tools

The standard development tool used at JAFU is *Eclipse* [39], and this was the tool used to write all the Java code in this thesis. To build DPG and manage its dependencies, the build tool *Apache Maven* was used and run as an Eclipse plugin using *M2Eclipse* [8]. DPG has been deployed and tested with the *web containers* *Apache Tomcat* [38] and *Jetty* [22]. Jetty was mostly used for testing simple changes during development and was run as a Maven plugin, while Tomcat was used for deployment testing.

The generated HTML and JavaScript [119] code was tested with the popular browsers; *Mozilla Firefox* [38], *Google Chrome* [44], *Microsoft Internet Explorer* [71] and *Opera* [97]. For script debugging, mostly Opera was used.

LaTeXwas used to write this thesis, and *Microsoft Visio* [72] and *Google Drawings* [46] were used to make figures and diagrams.

For collaboration, backup and version control of both code and LaTeXfiles, *Subversion* [36] was used.

1.6 Structure of thesis

Chapter 2: Background and Problem Description This chapter first explains the DPG 2.1 and its main components, as well as the concept of presentation patterns. Lastly, the problem description of the thesis is presented.

Chapter 3: Evaluation of DPG's plugin architecture and entity list handling

This chapter presents the current plugin architecture, as well as an evaluation, with focus on plugins' role in the presentation pattern. The discovered weaknesses are presented, and serve as the basis of chapter 4.

Chapter 4: Improvements of DPG's plugin architecture

This chapter discusses proposed improvements of the plugin architecture weaknesses discovered in chapter 3. The final plugin architecture implementations are also presented here.

Chapter 5: Proposed solutions for plugin resource management in DPG

This chapter presents and discusses the current architecture of the plugin resource

solution of DPG. Goals for a new architectural solution are presented, and proposed solutions are evaluated on the basis of these goals.

Chapter 6: Evaluation of data models and persistence technologies for plugin resources in DPG

There are multiple models and technologies for persistent storage of data. This chapter evaluates the most popular data models and technologies supporting them. The focus of the evaluation is on their applications as solutions for DPG plugin resources. Goals for the technologies are therefore presented, and the technologies are finally evaluated on the basis of these goals.

Chapter 7: Implementing new plugin resource management in DPG

This chapter presents the implementation based on the conclusion of the evaluations in chapter 5 and 6. The implementation choices are presented and thoroughly explained throughout the chapter. Finally, a conclusion is presented, evaluating the fulfillment of the goals and comparing the new implementation with the old one.

Chapter 8: Evaluation, Experiences, further development and conclusion

The last chapter presents the evaluation of the goals the candidate set for this thesis. Experiences with both development methodologies and technologies are presented, as well as suggestions for further development based on discovered weaknesses throughout the development. Finally, a conclusion for the thesis is given.

Appendix A : Guidelines for using the new plugin resource interface

The appendix contains guidelines for plugin developers using the new plugin resource presented in chapter 7. This includes conventions, examples, references and general hints for managing plugin resources.

2

Background and Problem Description

2.1 Web Content Management System

A *Content Management System* (CMS) is a system for managing content effectively. Content can be anything from simple text to multimedia like images or videos. Its main functions are to control data access and structure the data in a central storage.

A *Web Content Management System* is a CMS which focuses on the publishing of content on a web page. This typically includes web-based administration tools to allow users to publish or manage content on a web site, without the knowledge of programming languages or HTML. Examples of popular web CMS are Joomla! [6], Drupal [1] and Wordpress [14]. There also exists web CMS tailored for more specific applications, such as the educational course management system Moodle [10].

2.2 Presentation Patterns

In 2003, Khalid A. Mughal introduced the concept of *presentation patterns* [75]. The concept promotes reuse of content by defining how it should be *structured*, *rendered* and *navigated*. This concept has since been implemented and further developed to the current version used in DPG 2.1.

2.2.1 Presentation Pattern Specification

A *presentation pattern specification* is a set of rules for how a presentation pattern should be structured syntactically. It specifies how elements can be placed and which values these elements can contain in a presentation pattern.

2.2.2 Presentation patterns and presentations

A *presentation pattern* is developed from the presentation pattern specification, and defines the data structure of the content in a *presentation*. Multiple *presentations* can be instantiated from a single presentation pattern, which promotes the reuse of content structure. An example of this is how DPG 2.1 is used for distant learning at the Department of Informatics, UiB. Figure 2.1 shows how presentations for each available course are based on a single course pattern.

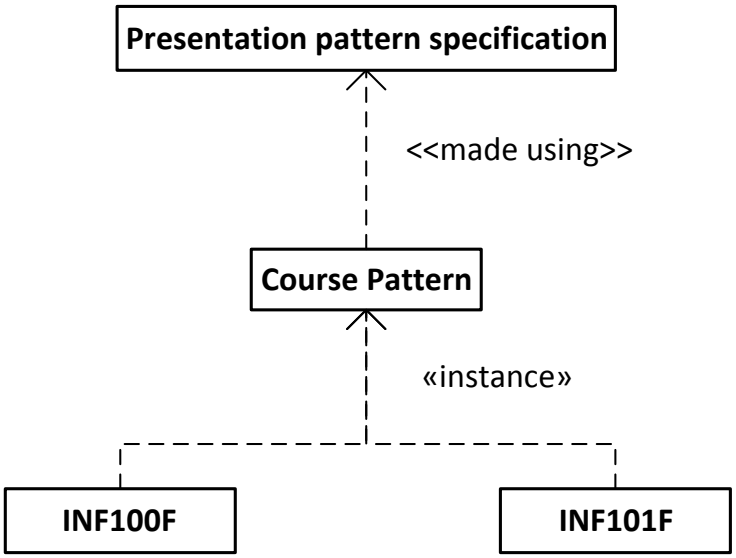

Figure 2.1: INF100F and INF101F are presentations which are instances of the same course pattern.

Content, such as the syllabus for a course, can be presented in multiple ways and places. To achieve this, a presentation uses Cascading Style Sheets (CSS) [19], Ve-

locity Templates [37] and Extensible Stylesheet Transformations (XSLT) [113].

2.2.3 The structure of a presentation pattern

The current version of the presentation pattern specification specifies four main components: *entity*, *entity-instance*, *view* and *page*. The relationship between these components are presented in figure 2.2. This structure promotes reuse of both content and its presentation.

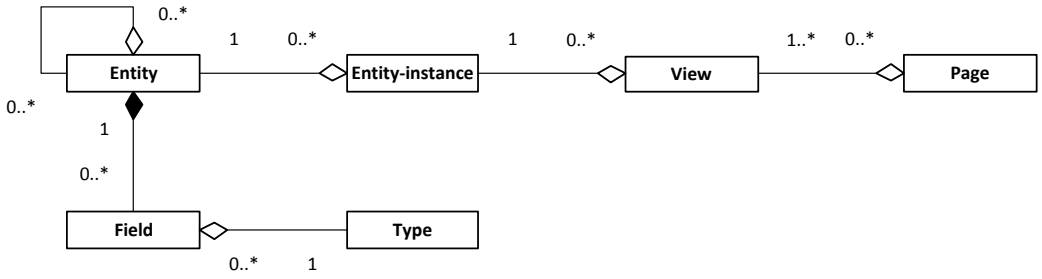

Figure 2.2: The relationships between the components defined by the presentation pattern specification.

An *entity* defines the actual structure of data. Specifically, it contains a list of *fields* attributed with specific *types*. There is a composition relationship between fields and entities, meaning that each field is defined, and exists, only in an entity. An entity containing contact information, may contain the fields `name`, `address` and `email`. In DPG 2.1, each field type specifies which *field plugin* should handle the content contained in the field. The fields in the contact entity would naturally be of the type `string`, which means the `string` plugin would handle their presentation.

The actual content is mapped to an *entity-instance*, which is an instance of an entity. Multiple entity instances can be made from the same entity, meaning they each contain their own content, but the content is structured in the same way.

A *view* specifies how an entity-instance will be presented. An entity-instance can be mapped to multiple views, meaning the content can be presented in multiple ways without affecting the content and its structure. Specifically in DPG, a view maps an entity-instance to an XSLT file.

A *page* is a composition of views which are to be presented on a web page. A *page-template* is used to structure the views on the page. A view can be mapped to

multiple pages.

In DPG, each pattern is defined in its own `pattern.xml` configuration file, using *Extensible Markup Language* (XML) [111]. This will be presented in more detail in chapter 3.

2.3 Dynamic Presentation Generator

Dynamic Presentation Generator (DPG) is a web CMS developed by the JAFU project. DPG is built around the concept of presentation patterns, and is at version 2.1 at the time of this thesis.

DPG follows the *Model-View-Controller* (MVC) [42] software architecture pattern. This pattern isolates domain logic from the presentation, supporting modularity and *separation of concerns* [120] of the components. DPG is built on Spring framework [104] components, such as Spring MVC [103] and Spring Security [105]. Some features that Spring can provide are an *Inversion of Control* (IoC) container and facilities for data access, messaging, testing, secure authentication and authorization.

Currently, DPG uses either JackRabbit [32] or a simple file system structure for storing and retrieving persistent content. This solution will be discussed further in sections 5.1 and 6.1.

DPG 2.1 is made up of four main components:

- Lobby
- Presentation Manager (PM)
- Presentation Content Editor (PCE)
- Presentation Viewer (PV)

These components will be further explained in the following subsections.

2.3.1 Lobby

Lobby is a subsystem of DPG which handles the authentication and authorization of users. This is where the users log in and get access to the three other components of DPG, depending on their user roles. Currently, the lobby uses a system called *Webucator* for administration of users and their roles, which was developed in Kristian Løvik's Master thesis [65].

The user roles are split into three categories:

- **Reader** Gets access to the PV, which means it can see the web pages and the presented content in authorized presentations
- **Publisher** Gets access to the PV, as well as the PCE, meaning it can both publish, edit and view content of authorized presentations
- **Admin** Gets full access to all the DPG components. This means it can do the same as a publisher and a reader, but can also create or delete presentations

The lobby also includes a dashboard for DPG, which presents DPG status information and options for reloading field plugins at runtime, and clearing caches.

2.3.2 Presentation Viewer (PV)

The PV handles the rendering of presentation content. It transforms the content into presentable HTML using XSLT, Velocity page templates and CSS. This HTML content is then presented to anyone with a reader role or higher.

2.3.3 Presentation Content Editor (PCE)

The PCE handles the actual content of a presentation. It also provides a publisher or admin with an interface for publishing or editing the content through an intuitive web interface.

2.3.4 Presentation Manager (PM)

The PM handles the creation, configuration and deletion of presentations based on presentation patterns. It also provides a web interface for these operations for users with an admin role.

2.4 Problem Description

The candidate will evaluate the current plugin architecture of DPG and determine the potential weaknesses and problems associated with it. The candidate will then evaluate the current solution for handling plugin resources in DPG. Following this will

be an investigation of possible technologies and solutions for these problems. Finally, an implementation based on the conclusions of the evaluations will be presented in the thesis.

2.4.1 The plugin architecture of DPG

The web is continuing its development toward information sharing and collaboration, as well as a user centered design, as defined by the *Web 2.0* standard [86]. Examples of this include social media, tag clouds, wikis and search suggestions, which all are based on user input. In DPG, this user input is handled by *plugins*.

The new, plugin oriented version of DPG has not matured much after Peder Skeidsvoll [94] and Tobias Olsen's [78] Master theses were presented. After DPG reached version 2.1, it has been very centered around the plugin architecture. It is therefore natural for the candidate to evaluate the current solution and improve it to facilitate for new and complex functionality.

2.4.2 Resource management in DPG's plugin architecture

To extend the practical uses of DPG, the development of DPG has tilted towards giving it effective support for large amounts of end-user data and migration of data with other systems. This new focus was sparked by discussions at the University of Bergen with OpenX Data [79], an *mHealth*-system [109] where large amounts of data is collected with the use of XForms [112]. OpenX Data collects this data via mobile phones, and is mainly used within the areas of health and education. The vision for JAFU and DPG is to support collection of this data and use the power of DPG to create innovative presentations. This will be especially useful with the current increase in smart phones, tablets and other portable electronic media devices. This functionality was partially realized in Morten Høiland [52] and Øystein Rolland's [91] Master theses, where they implemented XForms [112] support in DPG. Unfortunately, this is a very specific solution, which does not solve the general need for a better resource management for plugin data in DPG.

The DPG itself has gone through a lot of changes in the way it handles resources. The current persistence solution was mainly developed by Karianne Berg [17] in her Master thesis in 2008. Since then, some minor changes have taken place, such as the addition of a plugin resource API which is nothing more than a wrapper of the general resource API in DPG.

With the development of the plugin architecture arose a new challenge; how should

DPG handle plugin resources and persist them? Resource management for plugins was not a big priority when the plugin architecture was developed in DPG 2.1. The plugins of DPG all have access to the same resources, and there is no structure defined. There is also no functionality for data integrity and concurrency control, or performance functionality such as caching. Providing a generic and powerful solution for this functionality is a challenge for this thesis.

Plugin persistence API

Implementing a suitable plugin API for handling and persisting plugin resources will be very important for the future goals of DPG. It is therefore important that new possible solutions are thoroughly evaluated, by looking at the history of DPG and other similar CMS.

A choice must be made to either extend the current API, or replace it completely and sacrifice backwards compatibility. The candidate will evaluate the current solution for resource management for plugins in DPG. Alternatives for improvements and new solutions will be evaluated and presented.

Technologies for persisting large amounts of end-user data

Potential technologies for persisting large amounts of end-user data in DPG should be thoroughly evaluated. Criteria such as documentation, popularity, ease of use and functionality is important, since this will most likely be used by many different developers in the future. The query language, data structure, API and performance should all be included in the evaluation.

Guidelines for the new solution

The future development of DPG will likely revolve around plugin development. It is therefore important to provide guidelines for using the new plugin resource solution. These guidelines should be included in the thesis, to demonstrate the use of the new solution.

3

Evaluation of DPG's plugin architecture and entity list handling

This chapter will discuss and evaluate the plugin architecture of DPG 2.1. First the architecture and its evolution will be presented, followed by an evaluation of the plugin architecture.

This chapter contains work done in collaboration with Aleksander Waage [114], though it was written separately. Both Aleksander Waage and the candidate had a direct benefit of improving the plugin architecture, so it was natural to collaborate in the evaluation of the current plugin architecture.

3.1 The plugin architecture of DPG

The plugin architecture of DPG was first developed by Bjørn Ove Ingvaldsen during his Master thesis, with the goal of giving DPG multimedia support [60]. This architecture has since then been a focus of multiple Master students, being further developed in DPG 2.1 by Tobias Olsen [78] and Peder Skeidsvoll [94]. The architecture has then been used by Morten Høiland [52] and Øystein Rolland [91] in their development of support for XForms [112] in DPG.

The architecture is based on Martin Fowler's design pattern [42], *Plugin*, which in-

corporates principles such as the *Open-Closed Principle* [66]. Figure 3.1 shows how following this pattern means that plugins and the host application are connected through an implemented interface, which lets them provide each other with expected functionality. A plugin developer therefore does not need much knowledge of how DPG works, but only needs to relate to these interfaces. This means that following the Open-Closed Principle is important, which states that modifying the existing methods of an interface should always be avoided, as this can break plugin functionality. The more plugin functionality is implemented, the more important this will be, because a change in the interface will require a change in all the plugins implementing it.

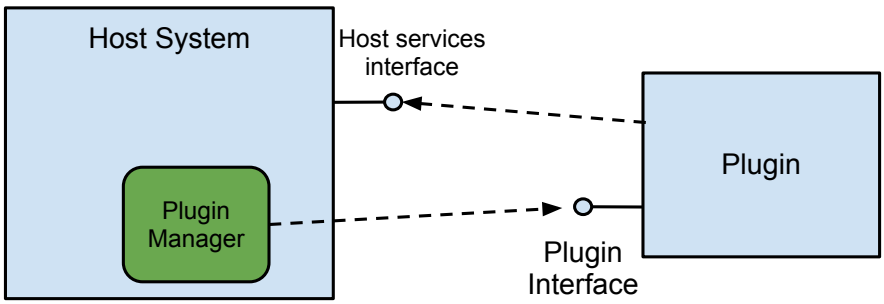

Figure 3.1: Example of an architecture following the Plugin pattern.

3.1.1 The Plugin Interface

To make a plugin in DPG 2.0, one of two interfaces had to be implemented; either `SingleFieldInputPlugin` or `EntityListInputPlugin`. This was changed in DPG 2.1, where the two interfaces were merged because of large overlaps in functionality, and resulted in the `FieldPlugin` interface. This also changed the possibility for a plugin to handle multiple fields at the same time. This change will be discussed further in section 3.2.

Given the increased role plugins play in DPG 2.1, changing or removing the interface will break large parts of DPG. This is discussed further in subsection 3.1.4.

The plugins are given functionality from DPG through an interface, with the goal of making them loosely coupled. The `FieldPlugin` interface includes these methods:

- `generateElement()`, which lets the plugin generate the output for the final presentation in HTML, returning it as a JDOM [5] `Element`.

- `getFormElement()`, which lets the plugin generate the form element to be presented in the PCE
- `getParameters()`, giving the plugin a list of parameters, specified in the configuration file `plugin-config.xml`.
- `setPluginResourceDao()`, giving the plugin functionality for saving and fetching resources
- `getXmlContent()`, which lets the plugin manipulate the input from the form element before it is persisted

As can be seen from the interface, the plugins get the responsibility to handle their own entity fields in both the PCE and the PV.

3.1.2 Making a plugin

Currently, to implement a plugin, there are multiple ways of defining it:

- Specify the name, plugin and parameters in DPG's configuration file `pluginConfig.xml`
- Specify the name of the plugin with the Java annotation `@PluginName` as shown in listing 3.1
- Just have the plugin implement the plugin interface, and let DPG derive the name of the plugin from DPG's class path

If no annotation is found, the name of the plugin will be the name of the class itself, given the class implements the `FieldPlugin` interface. Currently, the only way to define parameters for a plugin is through the configuration file `pluginConfig.xml`.

Listing 3.1: Plugins defined by implementing FieldPlugin and setting plugin name in an annotation.

```
1
2   @PluginName("xhtml")
3   public class XHTMLPlugin implements FieldPlugin {
4     ...
```

The plugins defined in any of these ways will then be loaded into the plugin manager using a `ClasspathPluginLoader` bean, which implements the `PluginLoader` interface. Plugins can also be loaded from JAR files from the folder `lib/plugins`, using the `JarPluginLoader` class which also implements the `PluginLoader` interface.

As previously mentioned, a plugin must implement the `FieldPlugin` interface, which includes many methods that most plugins probably will not use. Most plugin developers therefore choose to extend the `AbstractFieldPlugin` class, which provides a default implementation of all methods in a generic way, except for the `generateElement()` method. This makes plugin development a lot easier for most developers.

3.1.3 The Plugin Manager

The main functionality of the plugin manager is implemented in a method called `changeTreeIfPluginIsRequired()`, as shown in figure 3.2. In this method, the plugin manager handles each field in a content document, sending the content of the fields to each corresponding plugin through the `generateElement()` method. The input for the plugin manager is pure content generated by the PCE. The plugin manager recursively goes through each and every entity field in the content document, sending the field to the right plugin, which in turn transforms the content of the field to presentable content ready for a browser. The transformed document is then returned and sent back up to the PV, ready to be used in the XSLT transformations.

3.1.4 Entity Field Types

The changes from DPG 2.0 to 2.1 were mostly concerning how DPG handles plugins. In DPG 2.0, simple types like `string` were handled by DPG, and plugins were mostly used for multimedia. Listing 3.2 shows how the `type` parameter had to be set to the value `plugin` and refer to the plugin through the parameter `pluginConfig`.

Listing 3.2: Plugins in DPG 2.0 were explicitly referred to with the pluginConfig attribute.

```
1  ...
2    <field type="plugin" pluginConfig="singleVideo">video</field>
3  ...
```

In the development of DPG 2.1, it was decided that the entity fields should be handled in a more uniform way [78]. Plugins now play a much larger role in DPG, as every entity field node refers to a plugin specified by the `type` parameter. This change means that simple types like `string`, and even special types like `list`, are handled by a plugin with the corresponding name as shown in listing 3.3. Plugins now play such a sentral role in DPG, that if they are removed, DPG's core functionality would not work.

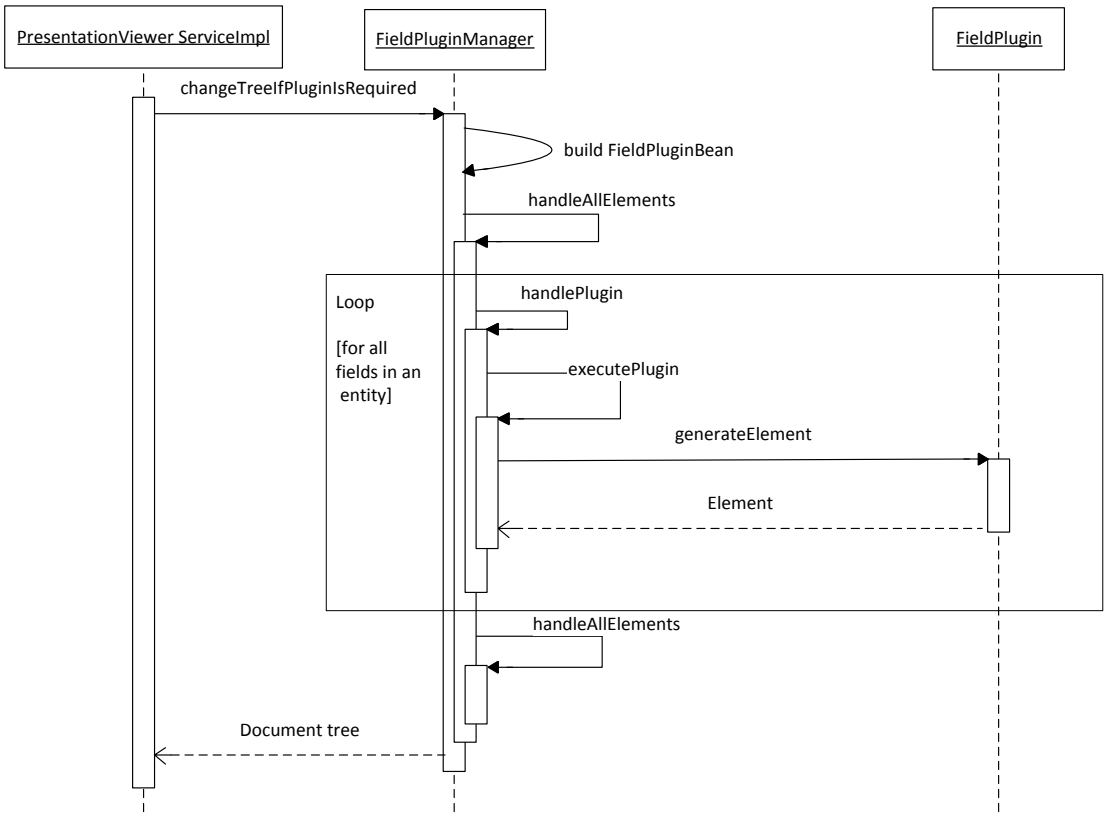

Figure 3.2: The plugin manager of DPG 2.1 uses plugins to build the presentation document one field element at a time.

Listing 3.3: The type attribute always refers to a plugin in DPG 2.1.

```
1  ...
2    <field type="string">address</field>
3    <field type="string">name</field>
4    <field type="list" entity-id="url">urls</field>
5  ...
```

3.2 Weaknesses of the current plugin architecture

To test the possibility of presenting dynamic geographic data using DPG, the candidates implemented a *Google Maps* [45] plugin. During the implementation, the candidates encountered some problems which led to the discovery of some clear weaknesses of the current plugin architecture. This also triggered a more thorough investigation of the limitations of the current plugin architecture and DPG.

The dynamic Google Map plugin will be used as an example to show some of the weaknesses presented in this section.

3.2.1 Multiple Field Input Plugins

As previously stated in subsection 3.1.1, the plugins in DPG 2.1 can no longer handle multiple fields at one time. An example of when this becomes a problem, is when a plugin wants to generate JavaScript [119] code, creating objects with multiple fields.

The candidates were making a *Google Map* plugin, called `DynamicMapPlugin`. The plugin featured a dynamic Google Map with markers loaded from the PCE, in contrast to the existing map plugin which featured a static google map with a position stored in the URL. The JavaScript code dynamically generated by this plugin would make *markers* and place them on a Google Map. The markers would contain multiple fields, including at a minimum:

- the name of the marker
- the latitude
- the longitude

The map itself would contain a list of these markers.

The goal of the plugin was for the pattern designer to know as little as possible about how the map plugin works. The problems arose when it was attempted to make the plugin generate all of the JavaScript code needed. This was desired to avoid the need for a pattern designer to prepare JavaScript code in the XSLT documents and make markers with the raw content from the PCE. This was not possible, because each plugin only gets access to one entity field at a time, through the `generateElement()` method in the `FieldPlugin` interface. A compromise would be to let the plugin persist the markers as resources belonging to the plugin, but then a publisher would not be able to add markers through the PCE.

The solution for this problem is discussed in subsection 4.1.4.

3.2.2 Handling multiple entity instances in one view

Entity-instances and views have a *one-to-many* relationship, meaning only one entity-instance can be mapped to a view. Views are defined in the DPG `pattern.xml` file, and are used to map the correct content to an XSLT file. Seeing as the plugin manager currently only handles one entity-instance at a time, this relationship is in a sense also true for entity-instances and plugins. In the DPG, each entity-instance has its content contained in a single document. This single document is the input for the plugin manager. The plugin manager will currently only be able to give a plugin the content from a single entity-instance. For most cases, this is fine, but to illustrate the issue, an example will be presented.

The previously mentioned dynamic Google Map plugin will be used as an example. The maps are represented in content as collections of markers. These maps will each be represented as their own entity instance. Figure 3.3 shows an example with one entity-instance called `Bars` and one called `Restaurants`, each storing different markers on different maps. Now lets say a pattern designer would like to also *combine* these two types of markers and present them on *one* map. This will not be possible in the current presentation pattern or system of DPG, because multiple entity-instances can not be mapped to a single view. The solution for this problem is discussed in section 4.2.

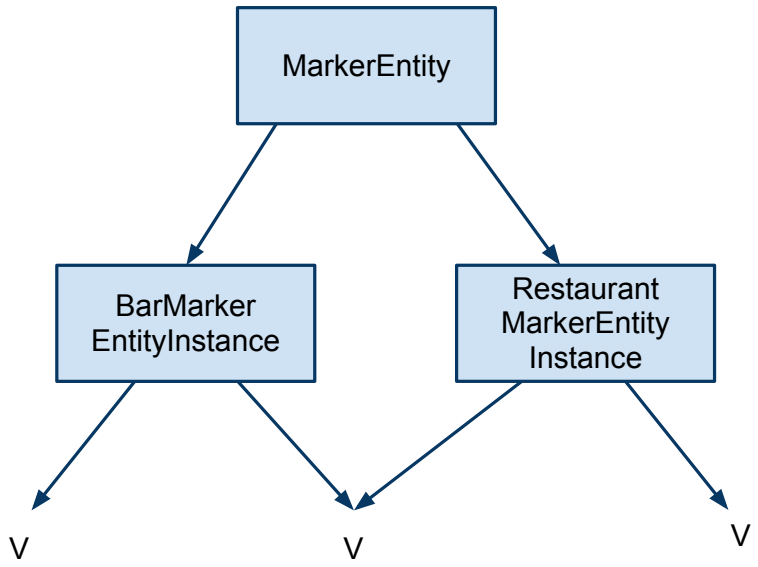

Figure 3.3: Multiple entity-instances mapped to one view.

3.2.3 Lists and subentities

While the candidates were developing DPG, it was very unclear how lists of entities were handled. Making a list of entities in the DPG can be done by setting the `type` attribute to `list` and referring to another entity through the `entity-id` attribute as shown in line 12 of listing 3.4.

Listing 3.4: Example of how a list is defined in the presentation pattern.

```
1
2 <entities>
3
4   <entity id="url">
5     <field type="string">address</field>
6     <field type="string">name</field>
7   </entity>
8
9   <entity id="softwareEntity">
10    <field type="string" required="true">title</field>
11    <field type="xhtml">description</field>
12    <field type="list" entity-id="url">urls</field>
13    <field type="entity" entity-id="phoneEntity">phone</field>
14    <field type="form2">form2</field>
15    <field type="savedComment2">savedComment2</field>
16  </entity>
17
18 </entities>
```

The plugin manager will see the reference to `list`, and send the underlying nodes to the `List` plugin. The `List` plugin will then just return the nodes back to the plugin manager. This was a bit confusing at first, as there was absolutely no documentation on how this actually worked, neither in the DPG Javadoc, wiki or the relevant Master theses.

Debugging and attempts of further development of DPG gradually revealed the mechanics. The handling of *lists* and *sub-entities* (which both behave and are treated in a similar way) is seen throughout multiple parts of DPG, including both the PCE, PM and PV. This very specific handling of lists and sub-entities in the different parts of DPG make it hard to change the plugin architecture or the plugins handling these cases. The benefit of a decoupled architecture is no longer there, when a change in a plugin can break other parts of the system. The process of DPG handling lists is presented below.

Figure 3.4 shows how an instance of the `ContentDocumentBuilder` class builds the content documents whenever a presentation is created. The method `generate()` first creates a root element for the document, then calls the `processEntityField()`

method, which recursively processes all entities and their fields in the document. The list nodes are *prepared*, but not populated by any child dummy nodes.

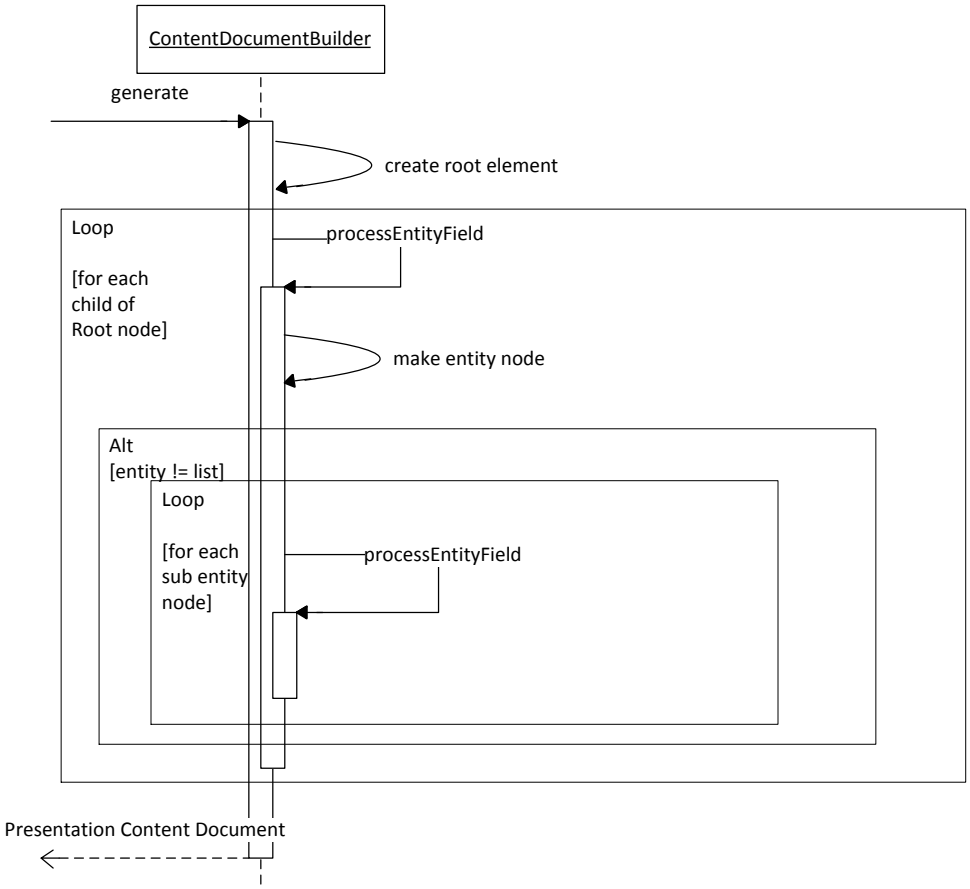

Figure 3.4: When a new presentation is made from a pattern, the `ContentDocumentBuilder` in DPG 2.1 builds the content documents for the presentation.

Figure 3.5 shows how a `FormBuilder` object builds the forms to be presented to the publisher in the PCE. The entities are resolved from the content document, and special cases are made for list entities because of their generated identifier. The resolving of entities involves checking the second to last element of an entity's `entityPath` parameter in case the entity is contained in a list.

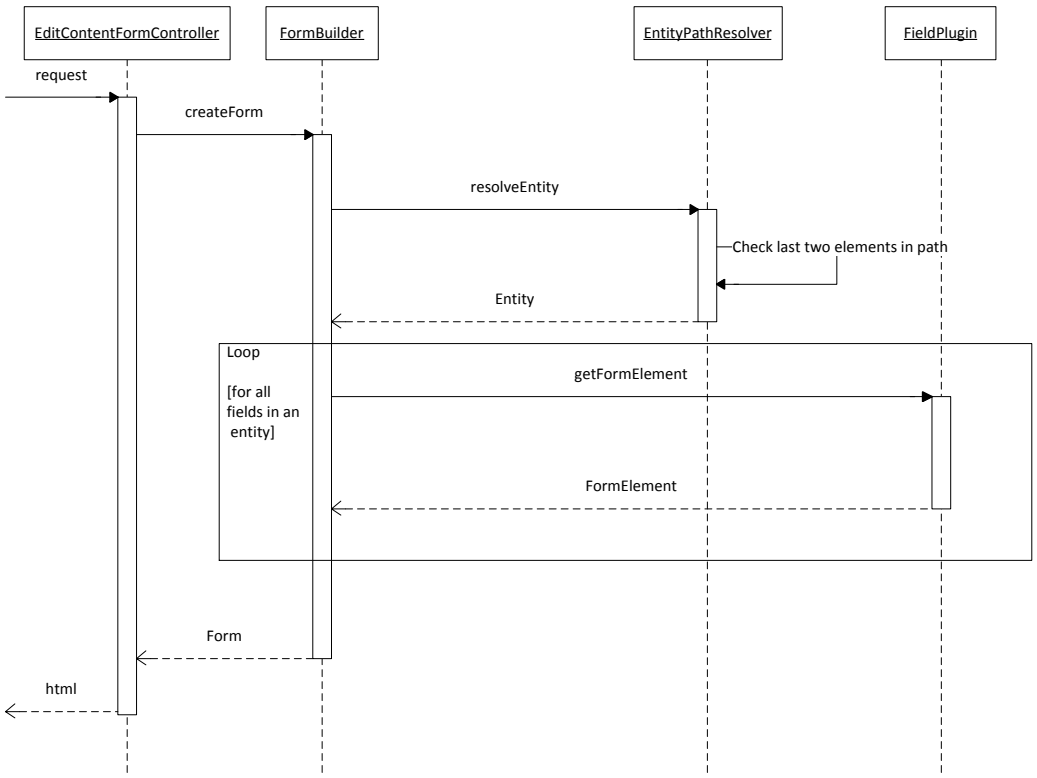

Figure 3.5: The `FormBuilder` class in DPG 2.1 builds the forms to be presented in the PCE.

Finally, a `FormProcessor` object checks if the publisher's action was to add or edit an entity, as shown in listing 3.5. Figure 3.6 shows how the action of *adding* an entity through the PCE will add empty lists to the corresponding content document.

Listing 3.5: The conditions of the add or edit entity action in the `FormProcessor`.

```
1
2   case EDIT:
3             documentEditor.updateEntityContent(doc, form);
4             break;
5   case ADD:
6             documentEditor.addSubEntity(doc, form);
7             break;
```

When it is time to present the content from these lists, the `FieldPluginManager` bean sends it to the loaded `ListPlugin` object. This plugin only returns the

element, letting the `FieldPluginManager` bean continue down the sub-tree, handling this as it would handle any other entity. This means that subtle changes to the `FieldPluginManager` class can easily break the list functionality, and it would be hard for anyone to tell why, because a lot of the list handling is not easily apparent throughout DPG. The conclusion is that lists and sub-entities are plugins in DPG, but are specifically referenced in the DPG core components, which goes against the Plugin design pattern. A modification of the `FieldPlugin` interface can therefore break core components of DPG.

The DPG would definitely benefit from a more general way of handling lists and sub-entities. This would help clean up the code for future development, and open for larger changes in the DPG, but is in itself a very large change which should be a topic for a new project.

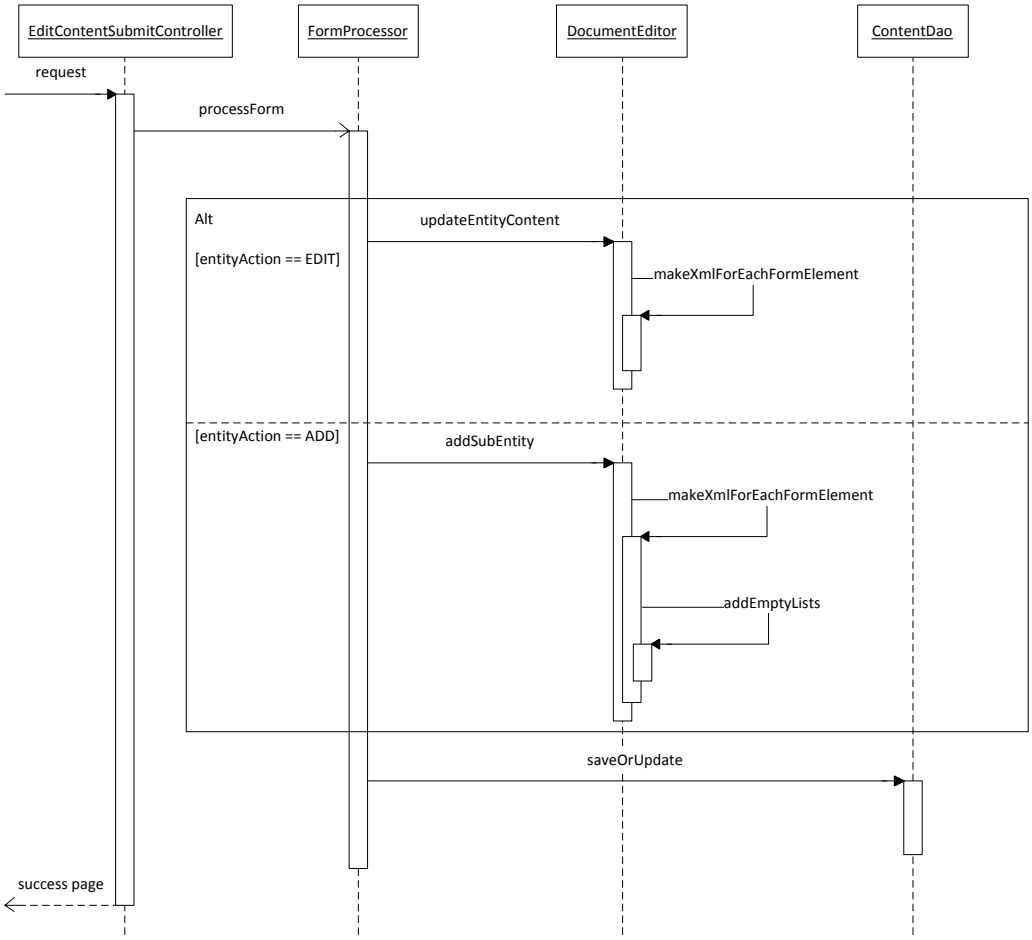

Figure 3.6: The `FormProcessor` class in DPG 2.1 updates the content documents based on input from the PCE.

4

Improvements of DPG's plugin architecture

This chapter presents work done in collaboration with Aleksander Waage [114], though it was written separately. Both Aleksander Waage and the candidate had a direct benefit of improving the plugin architecture, so it was natural to collaborate in the evaluation of the current plugin architecture.

This chapter will present some solutions to the weaknesses of the current plugin architecture described in section 3.2. The dynamic map plugin presented in section 3.2 will be used as an example for the solutions in this chapter. Pros and cons for each solution will be discussed, leading up to the final chosen solution and implementation.

An important part of DPG lies in the layers of abstraction between pattern developers, plugin developers, DPG developers and presentation administrators. The development of DPG has always been towards more general and simpler solutions, and the candidates would like to continue this practice.

Since DPG 2.1 multiple plugins have been developed, and in addition play a much larger role in DPG. A backwards compatible solution is therefore preferred, unless there is a very important change requiring otherwise.

The main goals of the solution will be:

- Abstraction
- Simplicity
- Generality
- Backwards compatibility

These goals should apply for all parts of DPG, meaning the solutions should be advantageous for all developers and users of DPG.

Martin Fowler's best practices [42] have been used throughout the previous development of DPG, and are therefore a guideline for the candidates throughout the solutions and implementations in this thesis.

4.1 Multiple field plugins

This section will discuss proposed solutions for the weakness presented in subsection 3.2.1.

4.1.1 Revert to a similar solution to DPG 2.0

As mentioned in subsection 3.1.1, in DPG 2.0, a plugin could implement one of two interfaces: `EntityListInputPlugin` or `SingleFieldInputPlugin`. The plugin manager would handle these two types of plugins in different steps. A variation of this solution is backwards compatible with DPG 2.1, as it is not necessary to change anything about the current plugin interface or how they work.

This solution was considered by Tobias Olsen [78] and Peder Skeidsvoll [94], but was dismissed because of the apparent similarities and overlapping functionalities of the two interfaces. Instead they merged the two interfaces into one, but unfortunately left out some functionality from the `EntityListInputPlugin` interface, as they did not see the need for it at the time.

A solution could be to use interface inheritance to let both plugin interfaces inherit the overlapping functionality. A problem with this solution is that it also means that a pattern designer would need to know the exact pattern structure the plugin requires. This goes against our goal of abstraction between the different kinds of DPG developers. The solution in DPG 2.0 was also specific for entity lists, which goes against the goal of a general solution.

4.1.2 Extend the existing `FieldPlugin` interface

A solution that was considered was to extend the existing `FieldPlugin` interface to allow for multiple fields. While investigating the possibility, the candidates found the current interface actually supports this. The `generateElement()` method has a JDOM `Element` object as input. This `Element` object can contain child elements, meaning it can support multiple fields. The change would have to be in how the plugin manager handles these elements in DPG, and how the fields refer to plugins. This solution is backwards compatible with plugins from DPG 2.1, because single field plugins could easily be treated similarly.

4.1.3 Refer to a plugin with a new attribute in the pattern

Another solution would be to have a reference to a plugin in the list and the sub-entity entity fields. By adding a `plugin-ref` attribute to these fields, the `FieldPluginManager` bean would be able to know when to send the underlying content to the referred plugin. This solution would be relatively simple to implement, but would unfortunately only give the plugin access to lists and sub-entities.

Listing 4.1 shows a more general solution; referring to a plugin in an *entity*. A plugin could then be given access to multiple fields of *any* type, including sub-entities and lists.

Listing 4.1: Example of how the pattern would look with a plugin reference in the entity.

```
1  ...
2      <entity id="map" plugin-ref="dynamicMapPlugin">
3        <field type="string" required="true">name</field>
4        ...
5      </entity>
6  ...
```

Unfortunately, both solutions do not solve the problem of the pattern designer requiring knowledge of how to structure these entities according to the plugin requirements.

4.1.4 Plugin defined pattern structure

A problem with plugins requiring multiple fields is that the entities and fields are defined in the file `pattern.xml` by the pattern designer. This means that the pattern designer had to know plugin specific requirements of the entity structure in the patterns, and the plugin developer is dependent on the pattern designer to

follow these requirements. This breaks the layer of abstraction between the plugin developer and the pattern designer. Listing 4.2 shows an example of how a pattern designer would need to structure the pattern according to the plugin's requirements.

Listing 4.2: Example of how a pattern designer would need to define the plugin required structure.

```
<specification>
  <entities>
    <entity id="marker">
      <field type="String" required="true">name</field>
      <field type="String" required="true">longitude</field>
      <field type="String" required="true">latitude</field>
    </entity>
    <entity id="dynamicMap">
      <field type="list">marker</field>
    </entity>
    ...
  </entities>
  ...
</specification>
```

A solution to this problem is to let plugins define their *own* entities in the pattern structure. The plugin would then be guaranteed to receive the correct entities. The pattern developer would also see the benefit of this, as it would no longer be necessary to have any knowledge of the inner workings of the plugin. Listing 4.3 shows an example of how the pattern would look. Pattern developers only need to refer to the plugin through a single field.

Listing 4.3: Example of how the pattern would look after the solution of a plugin defined structure.

```
  <entities>
    <entity id="dynamicMap">
      <field type="dynamicMapPlugin">map</field>
    </entity>
    ...
  </entities>
  ...
</specification>
```

4.1.5 Final solution and implementation

The final solution the candidates decided to implement lets a plugin define its own pattern structure, as explained in subsection 4.1.4.

The implementation is relatively complex compared to the other suggested solutions,

as this requires changes throughout many components of DPG. This is very time consuming, as it requires detailed knowledge of how these components work. There was little to no documentation, and no one available with the knowledge to help the candidates. Despite this, the solution was chosen based on its powerful functionality, general form and provided level of abstraction. Backwards compatibility is also a huge bonus. The many advantages of this solution made the extra effort well worth undertaking.

A goal of the implementation was to ensure that nested solutions also work. This means that a plugin can use another plugin, and even use another plugin that generates its own pattern structure.

Superficial changes to DPG

For a pattern designer, the pattern will look the same as in DPG 2.1. If a field refers to a plugin generating its own content, this will show up in the PCE and entity content document, but will be invisible to the pattern designer.

As seen in listing 4.3, the pattern designer does not need any knowledge of how the plugin works. The pattern designer only needs to make a single field referring to the plugin, and the plugin does the rest.

Figure 4.1 shows how, despite the pattern only defining one field, all the required fields of the plugin show up in the PCE. The content document is generated as it would be if these plugin defined entities existed in the pattern.

Finally, figure 4.2 shows how the entities are correctly presented. Everything is generated by the plugin; the pattern designer only selects the document node in the XSLT as it would with any other entity and entity-field.

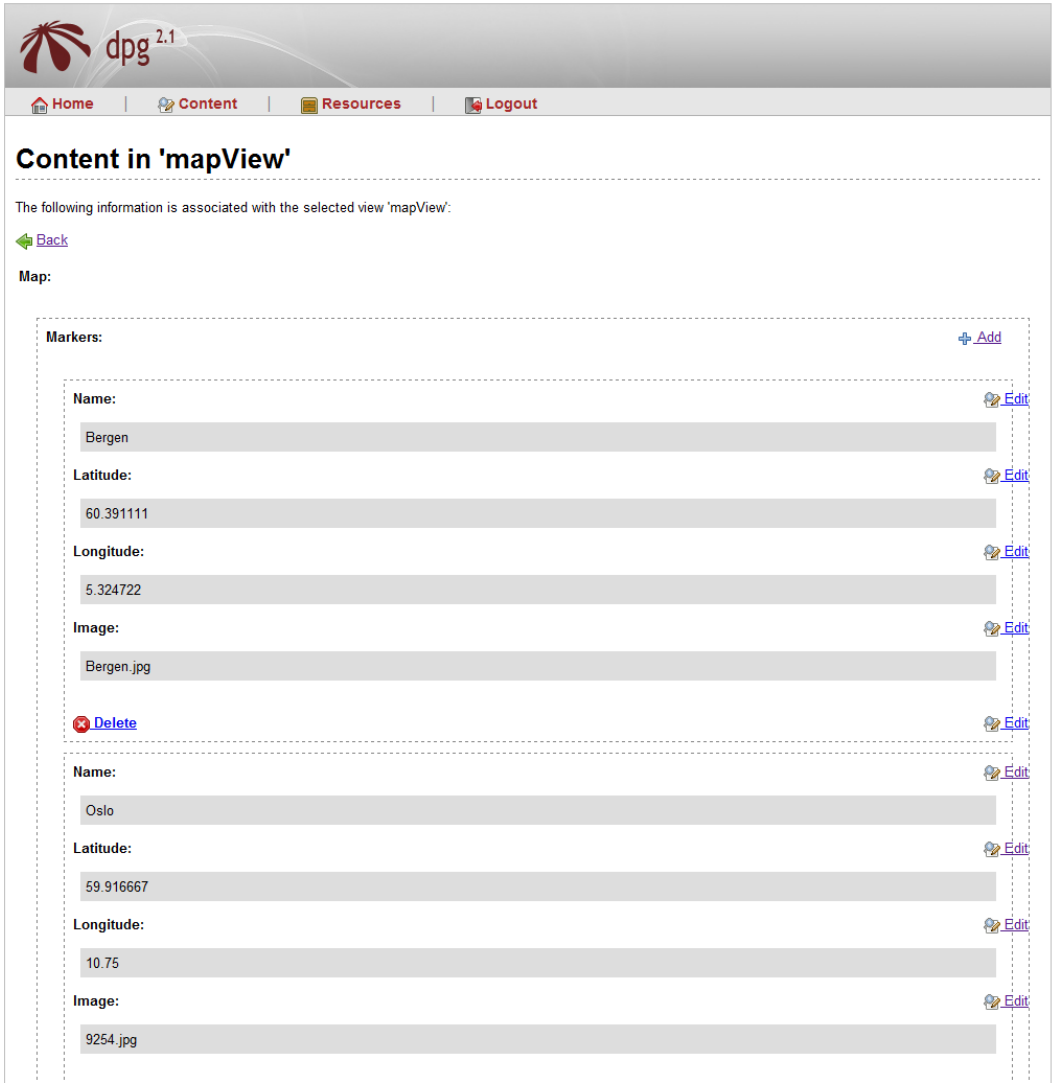

Figure 4.1: Entities generated by the plugin `DynamicMapPlugin` shown in the PCE.

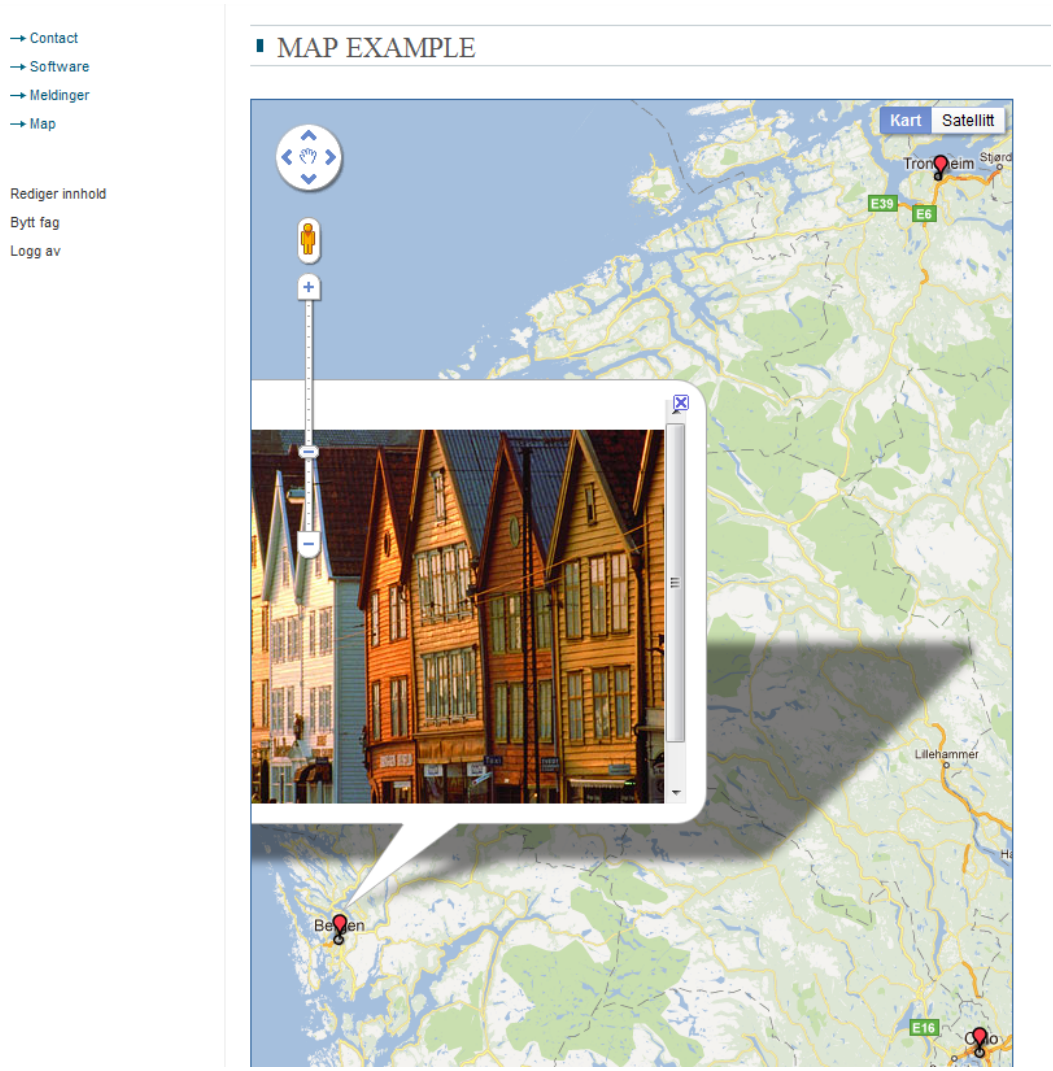

Figure 4.2: Map generated by the plugin `DynamicMapPlugin` with input from the PCE.

Changes to the plugin interface

To support backwards compatibility, and retain the uniform way of handling plugins and entity fields in DPG, it was decided to try a solution which only required an extension of the current interface, following the Open-Closed principle. The method `generatePatternStructure()` was added to the `FieldPlugin` interface as shown in listing 4.4. DPG now expects this method to generate a list of JDOM `Element` objects defining a pattern substructure. If this method returns `null`, the plugin and corresponding field will be handled as a DPG 2.1 `FieldPlugin`. The abstract class `AbstractFieldPlugin` that most plugins extend, automatically returns `null`, so that plugins that don't require multiple fields do not need to worry about this.

Listing 4.4: The generatePatternStructure() method is added to the `FieldPlugin` interface. The semantics of the other methods are explained in section 3.1.1.

```
1
2 interface FieldPlugin {
3
4     Element generateElement(FieldPluginBean, Map<String, Object>);
5
6     List<String> getParameters();
7
8     FormElement getFormElement(String, Field);
9
10    Content getXmlContent(FormElement, PluginContext);
11
12    void setPluginResourceDao(PluginResourceDao);
13
14    /**
15     * Generate the element structure this plugin requires.
16     * If it is a single field input plugin, return null.
17     * @return list of Elements defining the pattern structure for
18     * this plugin.
19     * The first element will be mapped to an entity instance */
20    Element[] generatePatternStructure();
21
22 }
```

Changes to the DPG architecture

It was decided by the candidates to utilize as much of the existing code and functionality in DPG as possible. Intercepting the actions regarding the pattern was tried as early as possible, meaning the first place the candidates looked was as low-level as the persistence API. The candidates then worked their way up, trying to find a solution.

It was not possible to change the pattern management for plugins at the DPG core and persistence level, because plugins are handled at a higher level. The end result required changes in both the PCE, PV and PM.

When a presentation is created, DPG first builds the content documents' basic structure, without any actual content. Most of the following actions actually check this content document's structure, instead of the actual pattern. The first change to the architecture was therefore in the `ContentDocumentBuilder` class in the PM, as shown in figure 4.3. The new `ContentDocumentBuilder` class checks the pattern to see how the content document should be built. The implemented change adds a condition when the fields of a pattern are resolved; it now also checks if the plugin from the field wants to generate its own pattern structure.

After the presentation is created, the rest of the changes in the content document are handled by the PCE. The main functionality here lies in the classes `FormBuilder` and `DocumentEditor` (called in the `FormProcessor` class). The changes required were only in the `FormBuilder`, as shown in figure 4.4. This builds the forms to be presented to the publisher in the PCE. It checks the entity-path of an entity in the content document, to resolve the actual entity from the pattern. The change required was only to intercept this, and add a condition to resolve the entity from the sub-pattern generated by a plugin if needed. The form is then built using these entities.

A challenge here was that there can be multiple instances of fields referring to the same plugin generating its own pattern structure. An example of this is if two maps generated by `DynamicMapPlugin` are made, the plugin would generate the same name for these entities. This would cause a collision when trying to resolve an entity. Because of this, there was a need for an extra identifier for the generated entities. It was decided to use the same solution as when the DPG generates sub-entities in lists, adding a unique identifier number.

Finally, the content is presented in the PV, using the `FieldPluginManager` bean to transform content to something presentable in HTML.
The `FieldPluginManager` bean recursively goes through the content document, sending the content of each entity-field to the correct plugin's `generateElement()` method. The changes required here included a condition to check if a plugin generates its own pattern through the `generatePatternStructure()` method. The plugin manager then continues down the tree, letting each field in the generated pattern be processed as normal by the corresponding plugins. When the recursive function returns to the original plugin, the underlying pre-processed sub-tree will be given to it through the `generateElement()` method. This ensures that each plugin, even the ones referred to in a pattern generated by another plugin, generates its own element to be presented as normal, avoiding unnecessary duplication of code.

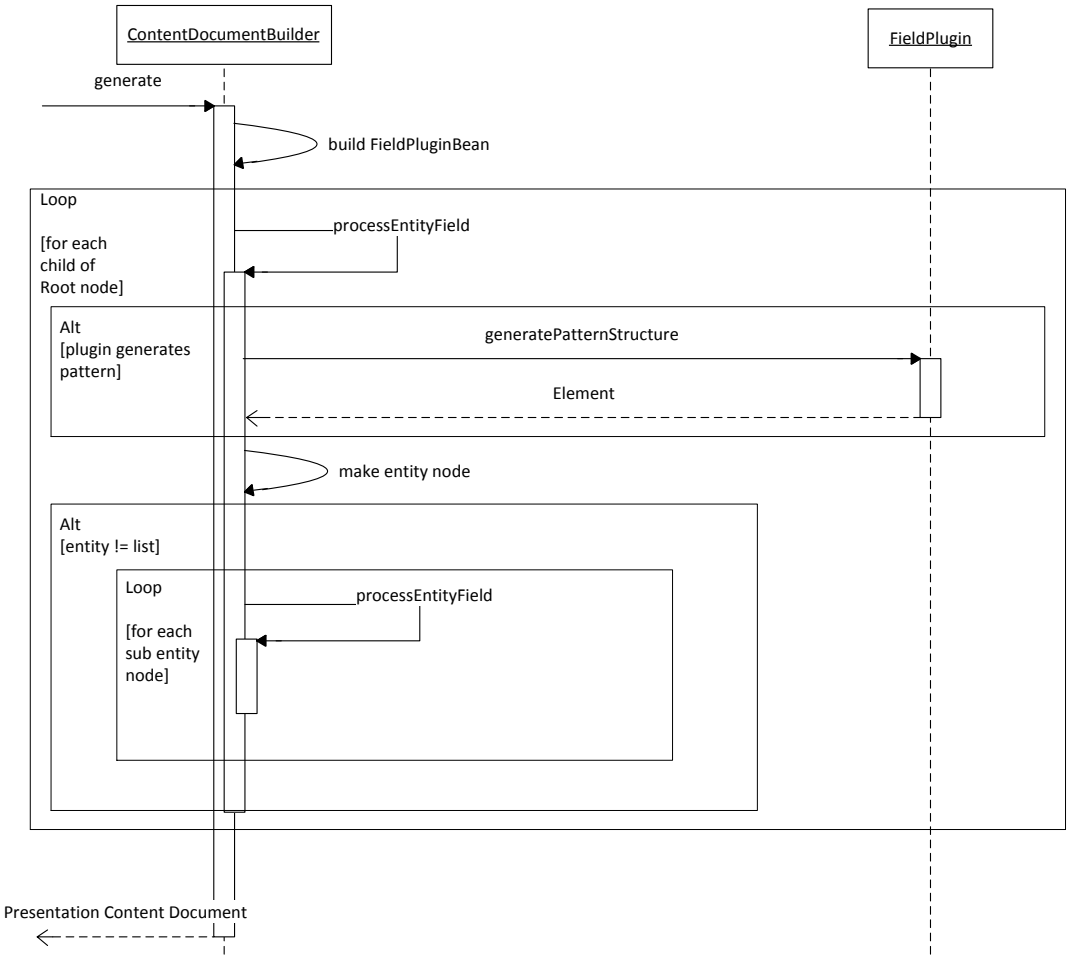

Figure 4.3: The new `ContentDocumentBuilder` class in DPG 2.1 builds the presentation content document from plugin generated patterns as well.

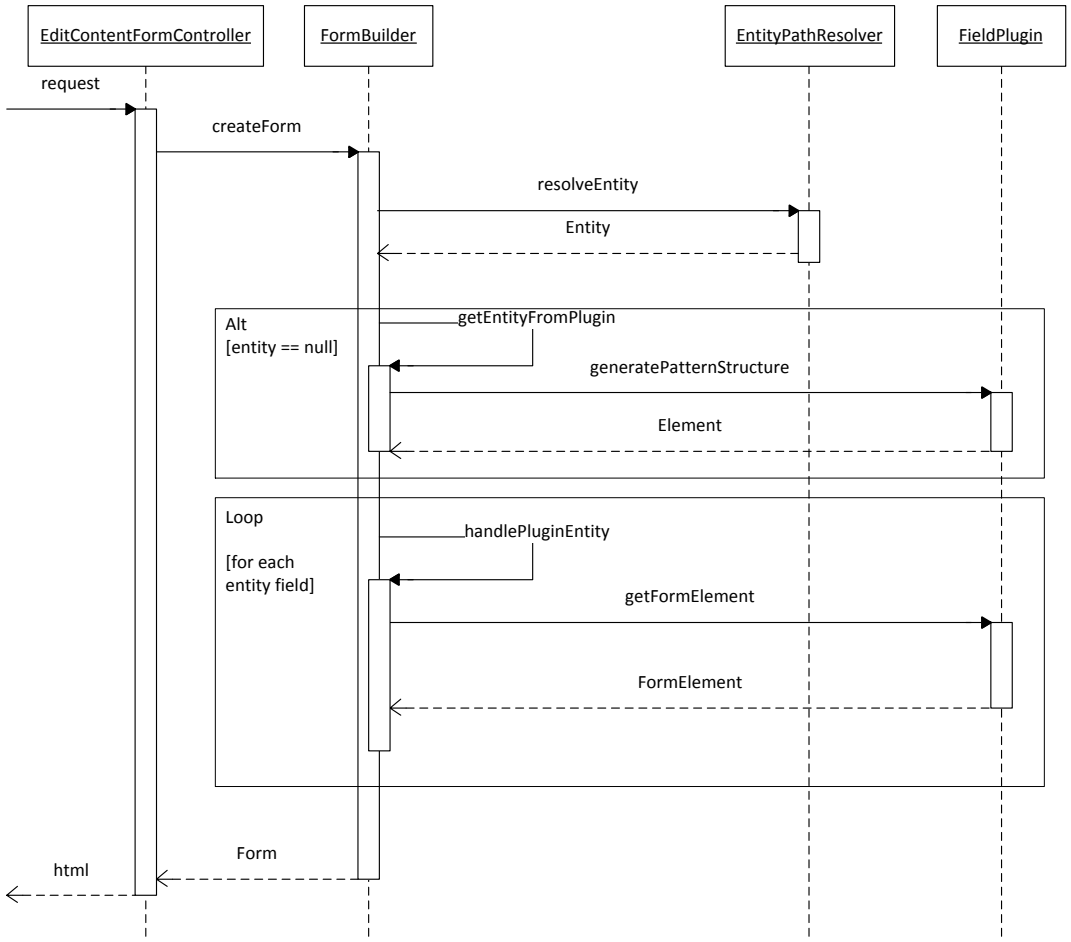

Figure 4.4: The new `FormBuilder` class in DPG 2.1 builds the forms to be presented in the PCE using plugin generated patterns as well.

4.2 Multiple entity instances in one view

This section will discuss proposed solutions for the weakness presented in subsection 3.2.2.

4.2.1 View composition plugin

A plugin could be made which composes multiple entity-instances into one entity-instance. DPG would then have to be altered to allow this special case, as is done with the `list` and `subentity` plugin. This solution goes against our goal of abstraction, and would be a complex implementation.

4.2.2 Single view list

A solution to the single view list problem, explained in subsection 3.2.2, would be to let a list of entity-instances be mapped to a single view. This would require a change in the presentation pattern specification, because of the current relationship between entity-instances and views as explained in subsection 3.2.2. One solution is to refer to other entity-instances in a new entity-instance, and map this new entity-instance to the view. This is similar to how a `list` field refers to other entities. Another solution is to directly map multiple entity-instances to a single view in the pattern.

Implementing this solution for views mapped to XSLTs should be relatively easy, as the PV can just compose these documents and send the new document to the correct view. The pattern designer could then treat this information as it likes. For this to work for plugins receiving multiple fields through the plugin manager, is a little more complex. Since the pattern designer has no control here, the plugins and the plugin manager would need to know which specific entities are meant to be presented together.

This solution should be easy to make backwards compatible with both plugins and the patterns of DPG 2.1. The current abstraction and generality of DPG 2.1 is also retained.

4.2.3 Final solution

The chosen solution for the single view list problem is to map multiple entity-instances to a single view in the pattern. This was a clear choice, as it is a backwards

compatible and general solution which retains the abstraction between the different components of DPG.

Changes to the presentation pattern specification

The presentation pattern specification is extended to let a list of entity-instances be mapped to a single view. This is done by mapping multiple entity-instances in the `entity-instance-ref` node of the view, separating each entity-instance by a semi-colon as shown in line 28 in listing 4.5.

Listing 4.5: Example of the new pattern with single view lists.

```
1
2   ...
3
4   <entity id="markerEntity">
5     <field type="string">name</field>
6     <field type="string" required="true">latitude</field>
7     <field type="string" required="true">longitude</field>
8   </entity>
9
10  <entity id="mapEntity">
11    <field type="list" entity-ref="markerEntity">markers</field>
12  </entity>
13
14  ...
15
16  <entity-instance id="barMarkersInstance">
17    <entity-ref>markersEntity</entity-ref>
18  </entity-instance>
19
20  <entity-instance id="restaurantMarkersInstance">
21    <entity-ref>markersEntity</entity-ref>
22  </entity-instance>
23
24  ...
25
26  <view id="markersView">
27    <description>Bars and Restaurants</description>
28    <entity-instance-ref>barMarkersInstance;restaurantMarkersInstance
          </entity-instance-ref>
29    <transformation>barAndRestaurantsTransformer</transformation>
30  </view>
```

Changes to the DPG architecture

The changes are mainly in the PV. The PCE should not be changed to let a publisher manipulate combined entity-instances, because this content should still be separated. The publisher will still have access to the entity-instances individually, but this means they currently each have to be mapped to a view. A solution here is to show each mapped entity-instance separately in the view in the PCE.

The change in the PV starts in the `createViewContent()` method of the `PresentationViewerServiceImpl` class. This is where the mapping of content from entity-instances to views is handled. This content is sent to the plugin manager and the document is finally transformed with XSLT. The composition of entity-instances happens here, before the content is transformed. This allows for multiple entity-instances to be transformed in a single view.

Some changes also had to be made to how views are loaded into `View` objects, because views can now contain a list of references to entity instances. The root element of the document sent to the XSLT transformation was the *entity instance* which was contained in a view. Since views can now contain multiple entity instances, the root element of the document is changed to the view. This makes more sense anyway, because the presentation documents are created for each view. This is an important change to note for pattern designers, as the XSLTs have to be slightly modified to use the view name as the root element of the content document.

5

Proposed solutions for plugin resource management in DPG

This chapter will discuss different architectural alternatives for improving the current plugin resource management in DPG. The conclusion will be presented at the end of the chapter. Figure 5.1 shows a general architectural overview of the plugin resource management in DPG. This chapter will discuss the plugin interaction with the DPG API and architecture, while the underlying persistence technology will be discussed in chapter 6.

The plugins play a large role in DPG 2.1, as shown in section 3.1. DPG can easily follow trends and implement new technologies by simply adding functionality through plugins. With the increasing importance of user interaction on the web, as defined by Web 2.0 [86], the plugins will be even more important to DPG. It is from user interaction the really large amounts of data are gathered, because there are many more readers than publishers. This makes improving the current plugin persistence solution a high priority in DPG.

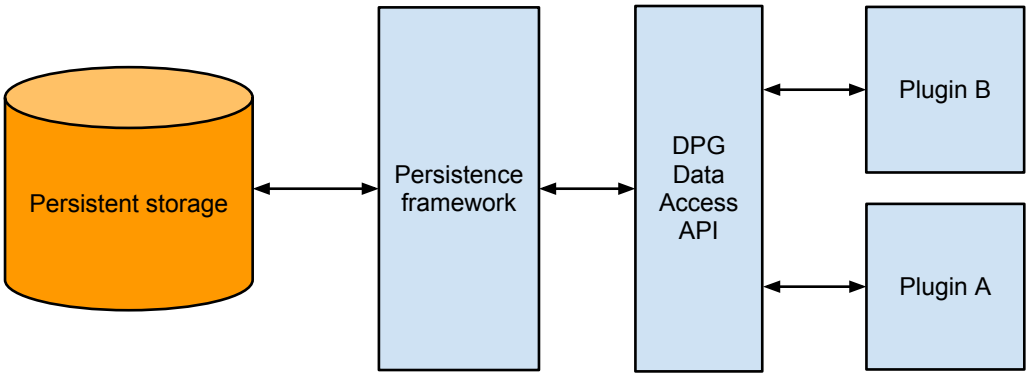

Figure 5.1: The overall architecture of the plugin resource management which is the topic of discussion in chapters 5 and 6.

5.1 Current solution

There are two types of content for plugins in DPG:

- *Presentation content*, which is given to the plugin through fields one at a time from DPG
- *Plugin resources*, which are fully controlled by the plugin and mostly used for storing content from readers interacting with the plugin

How DPG structures this content is shown in figure 5.2. This figure shows the logical organization of the content in DPG. The current use of a File System persistence implementation means the figure also shows the physical organization of the content.

The entity fields specified in a presentation pattern each refer to a plugin. The presentation, which is an instance of the presentation pattern, contains this content. This is the content that is managed by a publisher through the PCE, and given to the plugin through the `generateElement()` method of the `FieldPlugin` interface as shown in subsection 3.1.1.

To handle other types of data, such as data given through interactions with a reader, the plugins get direct access to their own resources. These resources are important for any type of reader interactions in DPG that require persistent data storage.

All content in DPG is accessed through a layer of Data Access Objects(DAOs), following the DAO design pattern [57]. DAOs provide an abstract interface for accessing a database or other persistent storage, without exposing details of the database. The

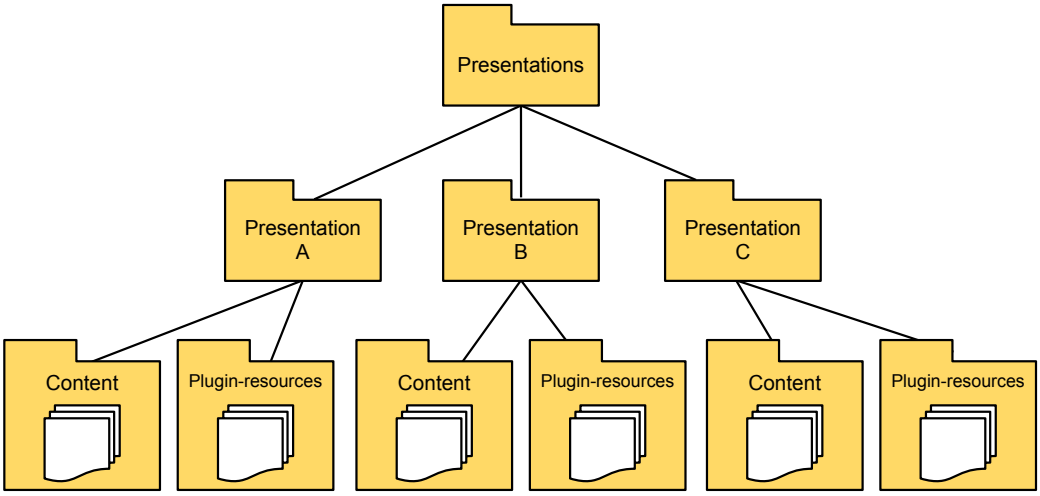

Figure 5.2: Tree showing the logical organization of the two types of content in DPG, PCE content and plugin resources, separated by folders.

implementation of the underlying persistence technology is not important as long as it is a fully functioning implementation of the DAO interface. This means that the persistence technology and implementation can be swapped without breaking the rest of the system. Currently, DPG has two DAO implementations: The File System DAO, and the JCR DAO, which are discussed further in chapter 6.1.

The current solution for plugin resource management in DPG is very simple. Each plugin gets access to an implementation of the `PluginResourceDao` interface through the `setPluginResourceDao()` method of the `FieldPlugin` interface. The `PluginResourceDao` interface contains four simple methods, shown in listing 5.1 below.

Listing 5.1: The current interface for plugin resources.

```
1
2   public interface PluginResourceDao {
3
4   /**
5    * Save resource
6    * @param presentationId the ID of the presentation to save the
         resource in
7    * @param filePath the path to the file, with root in the
         presentation folder
8    * @param mimeType the MIME type of the resource
9    * @param resource the resource to be saved
10   * @return URL to saved resource
11   */
12  String saveResource(String presentationId, String filePath, String
        mimeType, InputStream resource);
13
14  /**
15   * Get resource from a presentation
16   * @param presentationId the ID of the presentation to get the
         resource from
17   * @param resourceId the name of the resource
18   * @param getBinaryContent specifies if the content is binary, such
         as images
19   * @return URL to saved resource
20   */
21  ResourceFile getPresentationResourceById(String presentationId,
        String resourceId, boolean getBinaryContent);
22
23  /**
24   * Check if a resource exists in a presentation
25   * @param presentationId the ID of the presentation
26   * @param the name of the file/resource
27   * @return true if it exists
28   */
29  boolean resourceExists(String presentationId, String fileName);
30
31  /**
32   * Delete a resource of presentation
33   * @param presentationId the ID of the presentation
34   * @param the name of the file/resource to delete
35   * @return true if it was deleted
36   */
37  boolean deleteResource(String presentationId, String fileName);
38
39 }
```

These methods give a very basic CRUD [118] functionality; persisting, fetching and deleting resources. The plugin itself has to structure and manage the data; be it tabular, hierarchical or binary data. Because of this, plugin developers are forced

to do things like structuring data in simple text files with the data separated by white-spaces. This also means the plugin has to parse the *entire* text file each time it wants to retrieve some data. Figure 5.3 shows an example of how the poll plugin of DPG 2.1 structures its resources. When a user submits an answer to the poll, the plugin manipulates multiple documents: one containing the answer for the current user, and the rest containing the number of answers for each alternative in the poll.

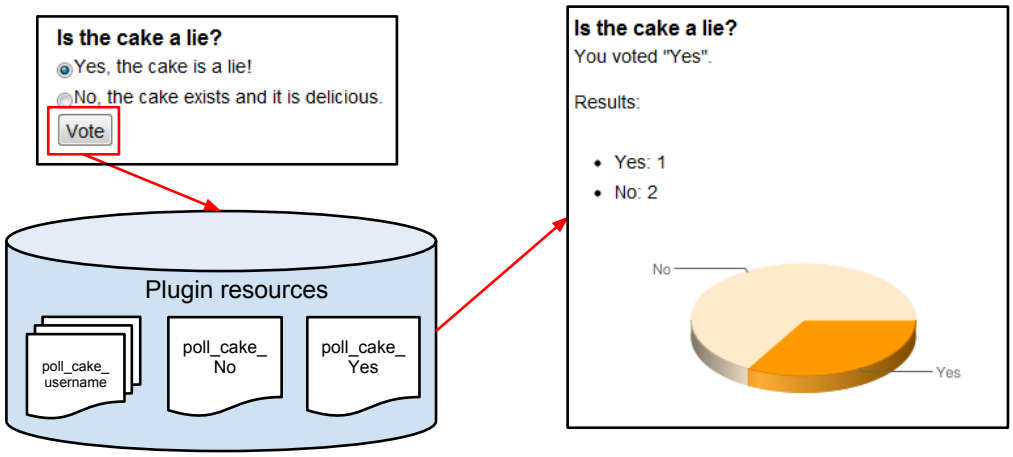

Figure 5.3: The poll plugin updates three different documents in a yes/no poll.

Depending on the DAO used, it may lack transaction support, so complications may arise if multiple instances of the same plugin try to manipulate the same resource concurrently. Other functionality such as caching is also not supported.

Figure 5.4 shows how every resource is tied to a presentation, but not a plugin. The implementation of the `PluginResourceDao` interface gives all plugins access to the same resources. This means that DPG does not do anything to separate the resources between plugins. Currently, all plugin resources are persisted to the same folder in the presentation. It is likely that multiple plugins will manipulate the same resource, because the name of the resource is the only thing that differentiates them.

The plugin gets access to the current user's name, view and page through a provided `FieldPluginBean` object. This means that the plugin can use DPG's authentication mechanism to separate content from users, or separate content by views and pages. The plugin itself would have to structure and separate this content.

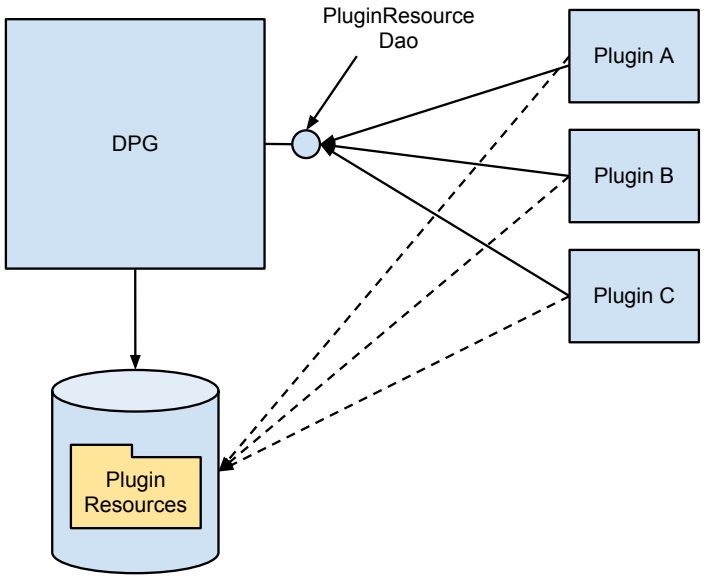

Figure 5.4: Plugins get access to resources through the `PluginResourceDao` interface. The stipled lines show which resources plugins have access to.

5.2 Goals for a new plugin resource solution

The solution has to meet certain criteria to support plugins requiring persistence of large amounts of structured data:

- Good performance for large amounts of data and concurrent users
- Facilitation of functionality such as caching and transaction support
- Resource authentication for plugins
- Persistence implementation abstraction (portability)

Fulfilling these criteria depends on the architecture and API of DPG, as well as the data model and technology for persistence. This chapter will focus on how the *architecture* of DPG can fulfill the criteria.

5.3 Proposed solutions

In this section, proposed solutions for the architecture and API for plugin resources will be presented. These alternatives will then be evaluated in section 5.4.

The danger of repeating others' mistakes makes it important to do extensive research of the history and solutions of other major CMS, such as Joomla [6], Moodle [10], Drupal [1] and WordPress [14]. This research will be the basis for inspiration of some of the proposed solutions.

5.3.1 Improvements in the current API

One solution would be to improve and extend the current API provided by the `PluginResourceDao` interface and change how the implementation structures and manages the content. This could be simple fixes such as having DPG give each plugin their own folder as shown in figure 5.5.

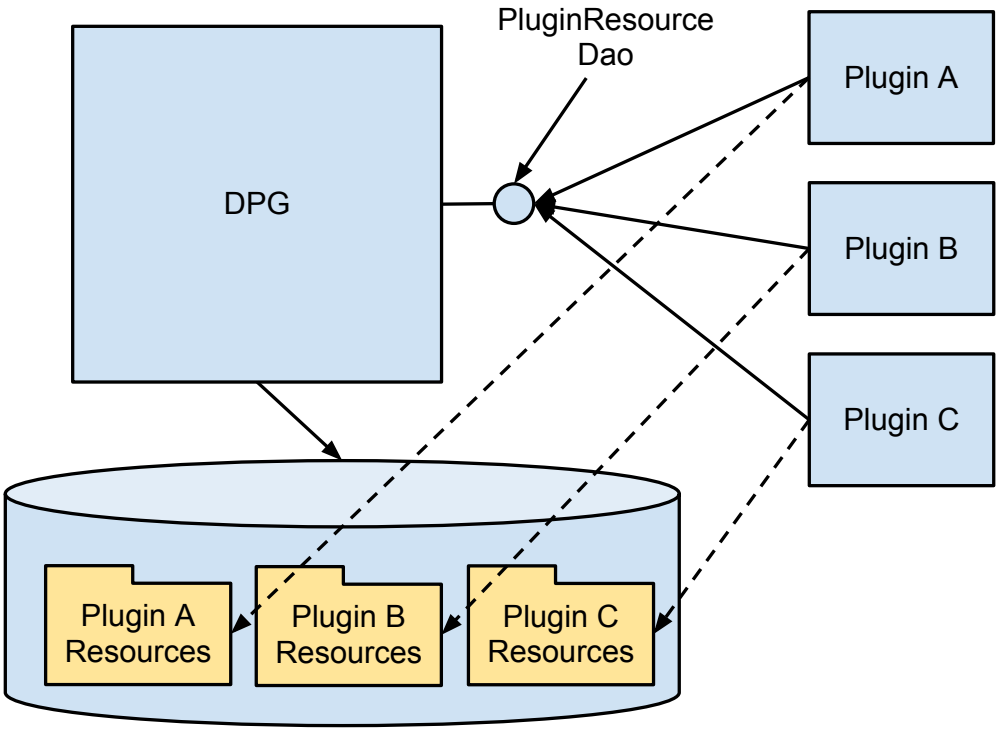

Figure 5.5: Plugins get access to resources through the `PluginResourceDao` interface. The stipled lines show the plugins' access to resources.

This solution would be completely backwards compatible with current plugins. The current API is a very general solution, meaning it is easy to implement nearly any persistence technology as its back-end, because it handles entire documents, not specific data values. It would also be the simplest solution to implement, and retains

a familiar structure of the resources.

The problem with this solution is it still does not facilitate for functionality such as structure, transaction and caching support on a low level. It also still restricts the plugin to some very simple methods of persisting and retrieving data, making it very inefficient.

5.3.2 Direct access through a standardized query language

Another solution would be to give the plugin direct access to the persistence technology. This could mean giving the plugin access to its own tables or folders through a query language such as XPath [110] or SQL [121], as shown in figure 5.6.

This solution would give full functionality of the underlying persistence technology, fulfilling the goals of performance and transaction support depending on the choice of persistence technology.

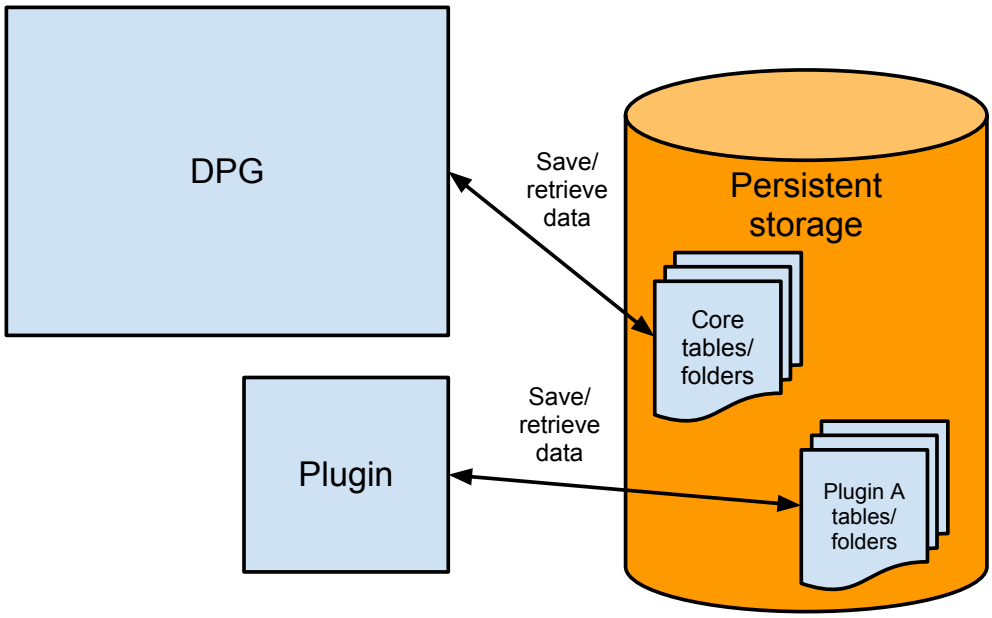

Figure 5.6: Plugins with direct access to the persistent storage.

Moodle [10] lets plugins define their own tables in the Moodle database. Moodle provides plugin developers with a tool called XMLDB to generate the XML configuration files that specify how Moodle should set up its database tables. Using this

tool makes it database-neutral [74]. This requires Moodle to present a long list of conventions for defining these tables and their content. The current plugin-centric architecture of DPG makes this a less valid solution for DPG, because the plugins would have to be trusted to follow these conventions, so they don't break core functionality or other plugins' resources.

Joomla is structured in a different way, with what it defines as modules and plugins for extending the system. Joomla's plugins are of little relevance to the discussion of plugin resources in DPG, because they do not manage any resources, but will be discussed further in subsection 8.3.1. Plugins responsibility in DPG can be described as a mix of that of a Joomla module and plugin. Joomla gives modules full access to the database, with nothing more than some warnings and tips of their use, given in tutorials on their website [61]. This is the same for WordPress, where plugins can create and manage their own database tables [124].

Another problem with this solution is that most *database management system* (DBMS) vendors have their own SQL dialect. This is often to support special features not supported in standard SQL. If DPG plugins are initially developed with one DBMS vendor in mind, it will be hard to change this later. This goes against the goal of portability. The solution would be to make a tool similar to Moodle's XMLDB, along with a similar list of conventions, but this would be time consuming and difficult to maintain.

DPG would also have to depend on the plugins for security, as this solution can make DPG vulnerable to attacks, such as a database injection attack, through the plugins.

5.3.3 Indirect access through stored procedures or an API

The last proposed solution would be to give the plugin indirect access to the database through stored procedures or an API. This is sort of a mix of the solutions presented in subsections 5.3.1 and 5.3.2. Figure 5.7 shows a solution where all queries to the persistent storage must go through an API controlled by DPG. Figure 5.8 shows a similar solution, but where queries go through a *persistence framework*, which will further process the queries before they are committed. These solutions provide a potential for a vendor-agnostic abstraction layer for accessing the database of DPG.

Drupal's extensions are called modules. Modules in Drupal use Drupal's database abstraction layer for accessing the database. This layer provides functionality for stored procedures, and "query builders", which further process queries before they are committed to the database. This solution has the extra benefit of security, as all queries are forcibly passed as prepared statement strings, preventing SQL injection attacks [87] from succeeding. Prepared statements can also increase performance,

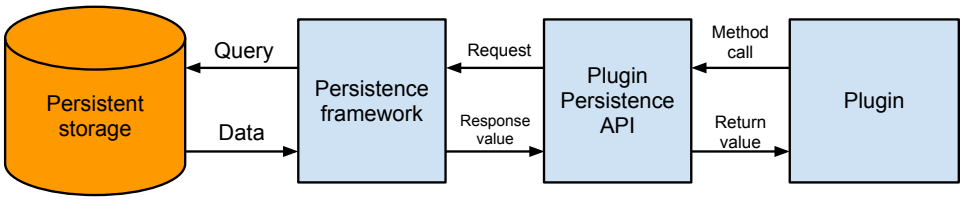

Figure 5.7: Plugins with indirect access to a persistence framework through an API.

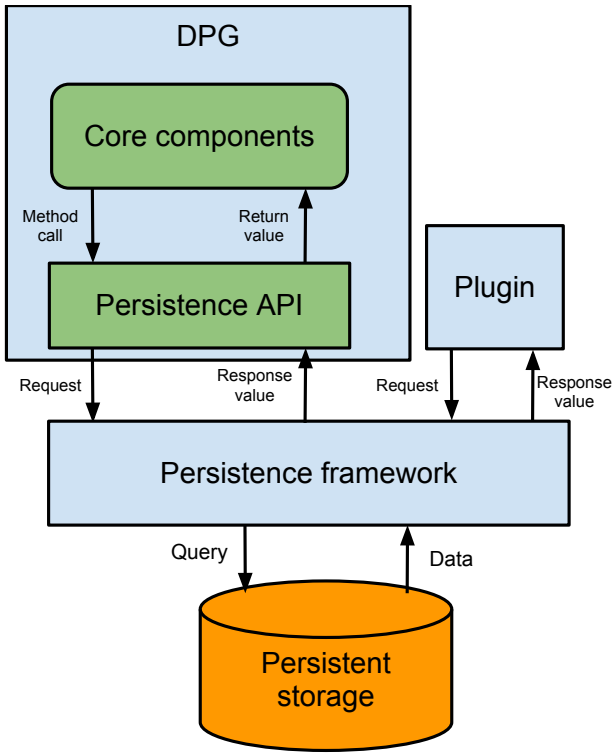

Figure 5.8: Plugins with indirect access to persistent storage through a framework.

since the database can execute them without compiling them first [85]. Stored procedures can give access control functionality and be cached in the database to increase performance. This is an example of how a similar solution would give DPG more control to increase overall performance and security. This is a win-win situation, as plugins do not need to focus on low-level performance and security issues, and DPG does not need to rely on plugins for this.

5.4 Evaluation of proposed solutions

Table 5.1 shows a comparison matrix of the persistence solutions for plugins in DPG. Making a very general API which is relatively neutral to a specific data model, would mean sacrificing functionality. There is a balance to this, where on one end there is rich functionality and the other end generality and abstraction of data model. This is because data values need to be stored and organized with some form of specific structure to be able to retrieve it efficiently. On the other hand, a direct access solution does not fulfill the goal of portability of persistence implementation or resource authentication for plugins. Many large CMS lock themselves to a specific database vendor, like MySQL, but DPG is designed to be portable across platforms and technologies. A middle ground would give the best solution fulfilling all goals while only sacrificing a slight amount of functionality, performance and portability. The conclusion is therefore to give the plugin indirect access to the persistent storage through an API or framework, as described in subsection 5.3.3.

Criteria	**Improve current solution**	**Direct access**	**Indirect access**
Performance	Poor	Very good	Good
Functionality	Poor	Very good	Good
Resource authentication	Yes	No	Yes
Portability	Very good	Bad	Good

Table 5.1: Comparison matrix of persistence solutions for plugins in DPG.

6

Evaluation of data models and persistence technologies for plugin resources in DPG

This chapter discusses and evaluates different data models and persistence technologies which can be used for managing plugin resources in DPG. This discussion is on a persistence implementation level, while the API and how the resources are made available to plugins is discussed in chapter 5.

6.1 Current data model

Currently, DPG uses a hierarchical data model for persistence. The data is accessed through DAOs, which function as a layer of abstraction between the persistence technology and the rest of DPG, as explained in section 5.1. This solution was developed in DPG 2.0 by Karianne Berg [17].

The first implementation of the DAO interface used JackRabbit [32], an implementation of Java Content Repository (JCR) [62], as shown in figure 6.1. JackRabbit offered many features, including transaction support, caching, XPath and SQL query support and being database agnostic. In practice, it proved to be difficult to administrate, possibly due to lack of documentation and maturity of the technology. The performance of the solution was also found to be less than satisfactory, as presented

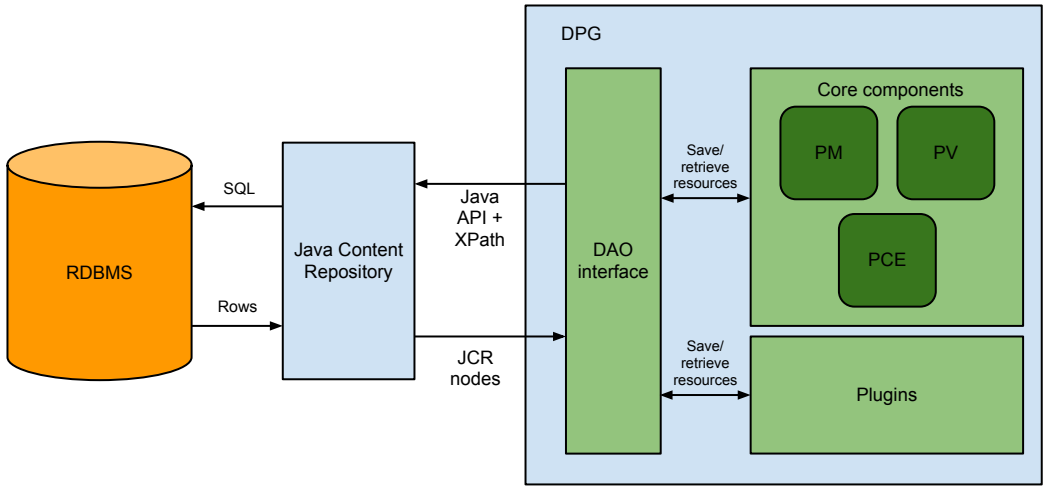

Figure 6.1: The current persistence implementation of DPG using JackRabbit.

in an INF-219 project [63] at the University of Bergen by Kelly Whiteley and Aleksander Waage. For this reason, a simple File System DAO was further developed and used in DPG 2.1 by Haakon Nilsen, the former system administrator and IT manager at the Department of Informatics at UiB, and the JackRabbit implementation has not seen any improvement since. JackRabbit and JCR will be discussed further in subsection 6.3.2.

The File System DAO which is currently being used in DPG 2.1, lacks transaction support and other functionality which makes it a viable solution for deployment of DPG on a larger scale. Currently, it has only been used in the JAFU project as a course management system for the distant learning of the courses INF-100F and INF-101F at the University of Bergen. Each course had its own presentation in DPG, based on a course pattern. There was only one active publisher and no reader interaction on each of these presentations, so there were no real problems with data concurrency or consistency. There were only 5-20 readers for each presentation, so performance was also never an issue.

6.2 Criteria for persistence technology

To properly compare and evaluate persistence technologies to be used in DPG, it is important to define some criteria. The criteria will be presented in this section. The discussions around the persistence technologies will be focused around these criteria.

6.2.1 Uniform solution

It is preferred to have DPG only use one persistence solution for all data. That is, storing plugin resources and content from the PCE in the same way. This means that depending on the chosen solution, the current DAOs for persistence in DPG may have to be changed as well. Regardless, it would still be extremely beneficial with an upgraded or completely new persistence implementation for DPG. The rest of DPG's content should therefore also be taken into consideration when choosing a solution for plugin resources. This will be discussed further in subsection 8.3.13.

6.2.2 Transaction support

One of the biggest problems with the current use of File System for persistence in DPG, is the lack of transaction support. Concurrent users writing to the same file may corrupt data. Some problems that may occur are:

- *Lost updates*: multiple processes read a value at approximately the same time, and then update the value based on the original value.
- *Dirty reads*: a process updates a value, and another process reads this before it is committed. An example of this is if an error occurs in a process after a value is changed, and another process reads the value before it is rolled back [80].

Other errors related to writing data to a persistent storage may also occur. For example an error such as a hard disk drive crash on the server can occur while data is being written to a file, corrupting the data in the file.

The solution to these problems is to use *transactions*. Transactions can be described using ACID [123] criteria:

- **Atomicity**: if one step fails, the whole transaction is rolled back
- **Consistency and Isolation**: transactions are isolated from each other, so the data inside a transaction is consistent
- **Durability**: even if a server or application crashes, the changes made by a successful transaction are permanent

A technology supporting transactions is therefore considered a very high priority in this evaluation of persistence technologies.

6.2.3 Performance

The downfall of the JackRabbit implementation was largely due to performance issues. The performance of both the JackRabbit and File System persistence implementation has been poor, and limits DPG's uses. Improving DPG's performance has not been a priority since DPG 2.0, since so far it has only been used on a very small scale. This makes performance a very high priority for the new persistence technology to be used in DPG.

6.2.4 Support for caching

It is safe to assume that under normal use, DPG will execute many more read operations than write operations. There are a lot more readers than publishers, and presentations will be presented much more often than they will be changed. To fetch this data from a persistent storage every single time a reader views a page in a presentation would be very inefficient. The presentation does not change, and the content rarely changes, so in most cases it would be better to hold the page or the content in memory. This is called *caching*, and will greatly increase performance of the application. The persistence technology of DPG should therefore support caching.

6.2.5 Maturity and documentation

It is important that the solution is mature and well documented. This makes it easier and more safe to commit to. The maturity and documentation usually reflects the popularity of the technology as well. This should be considered a very high priority.

6.2.6 Portability

To avoid vendor lock-in, the framework for accessing the persistent storage or database should abstract away from vendor-specific details. A criteria is that the framework supports at least two popular open-source databases.

6.2.7 Character encoding of data

In Karianne Berg's Master thesis [17], it was specified that all data in DPG should use the UTF-8 character encoding for Unicode [59]. This means that the persistence

technology has to support this.

6.2.8 Spring integration

DPG uses the Spring framework, which can integrate with persistence solutions to give extra functionality such as transaction management. To utilize Spring functionality to its fullest, the persistence technology should integrate well with the Spring framework. Many technologies provide a Spring integration module to make this easier.

6.2.9 Follows a standard

There are many benefits to a persistence technology which follows a standard. It assures that changes will affect many different implementations, usually leading to a more matured and carefully thought out solution. Other implementations of the standard can also be used for documentation on the technology. Completely relying on one persistence technology is risky, but using a standard usually means that the implementation can be changed relatively easily, further promoting portability.

6.2.10 Backwards Compatibility

Backwards compatibility with content from DPG 2.1 is a benefit. This is considered a *low* priority, because it should not be a deciding factor for the choice of persistence technology. This is because the only current use of DPG 2.1 is tied to the distant learning of the courses INF-100F and INF-101F, which are not running this semester, so there will not be much work transfering content to a new version of DPG.

6.2.11 Support for versioning of data

A popular feature which is supported by many persistence technologies today, is versioning of persistent data. This could allow a plugin or publisher to keep track of revisions of persistent data, and roll back if needed. This is considered *medium priority*, because it will likely have a performance impact because of the increased processing required for each transaction, having to also store and manage versions of data. The feature is practical, but not necessary. This idea will be discussed further in subsection 8.3.9.

6.3 Alternative persistence technologies

Some data is inherently tabular, while other data can be inherently hierarchical or even object relational. This section will discuss these different data models, along with some good frameworks supporting them, with regards to their use in DPG.

The following alternatives will be discussed:

- Relational data model
- Hierarchical data model
- Object-oriented data model

The focus of the evaluation is on the plugin resources and data from user interactions, but DPG's content is structured hierarchically, and this will be taken into consideration for the further development of a uniform persistence implementation in DPG.

6.3.1 Relational

The most common structure of large amounts of user input data is tabular as shown in table 6.1, and this data model by far has the most matured technologies and solutions. The performance is very high, due to the maturity and the very rigid schema defining the data structure. The rigid schema of the data is also one of its largest drawbacks. Complex structures, or data which doesn't follow a specific order, may be represented in an unnatural way, forcing a distribution of data over many tables. This reduces performance and makes the data difficult to manage. Performance may also suffer if schemas of data change often, because this would require a restructuring of the database.

Name	**Age**	**Address**	**Country**
Kelly	24	123 Fake St.	USA
Øystein	25	Torgallmenningen 8	Norway
Aleksander	25	Valkendorfsgaten 6	Norway

Table 6.1: Example of tabular data.

Relational databases have a standard query language, SQL [121], for retrieving and manipulating tabular data. SQL has existed since the 1970's and is a familiar language to most software developers. This means that most plugin developers would not need to learn anything radically new if it were to be used in DPG.

For a uniform persistence solution in DPG, the problem of persisting hierarhical data in a relational database must be discussed. DPG's presentation patterns are structured in an XML tree, so applying a relational database solution would require some form of mapping from XML to relational structure. The presentation patterns support nested entities, making it possible for very complex structures which do not translate well to a relational structure. There are many good open source frameworks for mapping between relational and Java objects, and XML and Java objects, but functionality for mapping XML to relational are lacking and usually specific for a database vendor. Another solution would be to store the XML data in a *Large Object* (LOB) or a *Character LOB* (CLOB), but this would still force DPG to process the entire XML tree each time a value is needed. The ideal solution would be a relational database with support for an XML data type, which can be queried.

Plugins in DPG are responsible for handling end-user data; that is, data from a reader. Most plugins' data from readers is tabular, meaning the relational data model would in most cases suit the plugins' needs very well, and would provide with a high performance solution. Examples of some existing plugins with tabular data are:

- `Pollplugin`: answers, such as "yes" or "no", tied to usernames
- `DynamicMapPlugin`: places with names, latitudes and longitudes

Technologies and solutions supporting the relational model are many, and it is widely documented. A very low-level and common solution for direct database access through Java is with the JDBC API [83]. The different SQL dialects of database vendors means a higher level solution which abstracts the underlying database is preferred, so as to not lock DPG in to one vendor. It is important that the frameworks are database vendor agnostic, but most open-source frameworks at least support the two most popular open-source DBMS; MySQL and PostgreSQL.

Most of the biggest similar CMS like Joomla [6], Drupal [1] and Moodle [10], all use a relational model for persistence of data. This is a tried and true model, and would therefore be a safe choice for DPG. On the other hand, DPG's hierarchical data model may also be one of the key features that separates it from these large competitors.

MyBatis

MyBatis is a framework for querying a *relational database management system* (RDBMS) using SQL, and is in use by large systems such as MySpace. In terms of use, MyBatis

lies somewhere between JDBC and an Object-Relational mapping tool (ORM) (further explained in section 6.3.3). The biggest advantage of MyBatis is in its simplicity, but it also gives great control over the content of the database itself. It minimizes boiler plate code from JDBC, and allows for mapping of SQL statements.

MyBatis couples Java objects with stored procedures or SQL statements using an XML descriptor [11]. A recent addition is support for mapping through Java annotations as well, as shown in listing 6.1. Depending on the database, it supports transaction management and caching.

Listing 6.1: Mapping comments in a relational database to a Java object with annotations.

```
1
2   public interface CommentMapper {
3     @Select("SELECT * FROM comments WHERE name = #{name}")
4     Comments selectComments(String name);
5   }
```

As shown in listing 6.2, MyBatis can make it very simple to persist and retrieve data with Java objects, once the data is mapped. It is important to provide a simple way for plugins in DPG to manipulate persistent data, but at the same time control it. One solution is to let plugins map the objects through an API provided by DPG. DPG can then process this before it is mapped, like splitting the resources by plugins.

Listing 6.2: Retrieving comments made by Kelly using MyBatis.

```
1   ..
2
3   SqlSession session = sqlSessionFactory.openSession();
4   try {
5     CommentMapper mapper = session.getMapper(CommentMapper.class);
6     Comments comments = mapper.selectComments("Kelly");
7   } finally {
8     session.close();
9   }
10
11  ..
```

There is little processing and overhead tied to using MyBatis. The performance is mostly limited to JDBC and the underlying database [115]. This makes for very good performance. MyBatis also integrates well with the Spring framework, and there exists Spring modules, tutorials and documentation for this.

6.3.2 Hierarchical

A hierarchical data model is usually represented by a tree of nodes with one root node, and data as leaf nodes or node properties. This offers features such as complex structures and a flexible schema, while still being very intuitive. An example of data which benefits from this model is comments on the web, as shown in figure 6.2. There can often be comments on comments, and there is no way of knowing how much this can be nested.

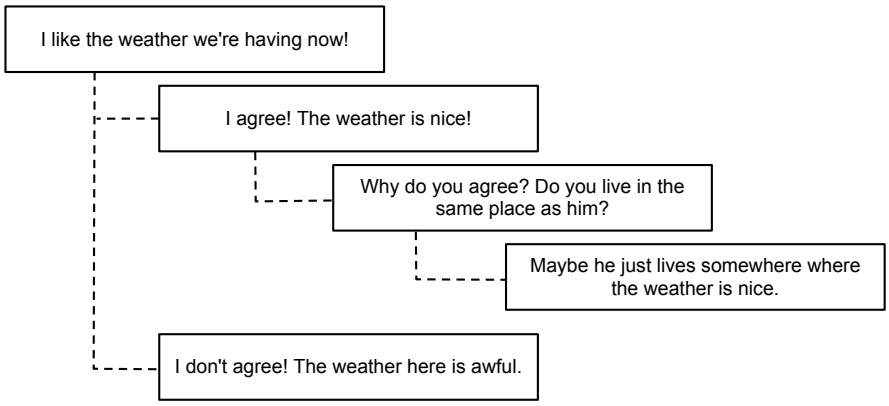

Figure 6.2: Example showing the hierarchical structure of comments.

A common way to transport data today is with a hierarchical data model such as XML. This data model is very flexible, but performance suffers from the amount of parsing required. It is beneficial if schemas of the data changes often, or are very complex.

Native XML databases, such as eXist [31], are currently very immature and there are few alternatives to choose from. It is therefore better to look at alternatives for frameworks mapping hierarchical data to a persistent storage such as an RDBMS.

A content repository is a hierarchical content store for both unstructured and structured data. JCR is a Java standard for accessing this hierarchical content in a uniform manner. JCR version 1 and 2 were specified in JSR 170 [62] and JSR 283 [29] respectively.

Alfresco [16] is an example of an open source CMS which provides with a JCR API to their content repository, but this is not its focus. The only real open source implementation of JCR is JackRabbit, which will be further discussed in this subsection.

The DPG currently uses a hierarchical structure for its content, defined in the presentation pattern. For a simple and uniform way of handling data in DPG, it would be

beneficial to continue with the hierarchical structuring of data. Support for XPath is deprecated in JCR 2.0, in favor of JCR-SQL2 and JCR-JQOM [24]. This means that DPG should move away from its use of XPath in the persistence layer and implementation, in order to stay updated with the newest JCR implementations.

The candidate has some experience with JCR and JackRabbit from previous projects tied to courses at the University of Bergen and the Bergen University College. This should be useful for a persistence implementation in DPG using JCR.

JackRabbit

JackRabbit is a fully conforming implementation of the previously presented JCR API. JackRabbit offers many features such as versioning, full text search and transactions [32].

JackRabbit structures content in trees of *nodes* with associated *properties*. Data is stored within these properties. An example of this is shown in figure 6.3, which shows a hotel reservation system made by the candidate using JackRabbit in a previous course project [64]. This model translates well to DPG's XML structure of presentation patterns.

Since the initial implementation of JackRabbit 1.6 in DPG, JackRabbit has gone through a lot of changes. With the new version 2.0 of JackRabbit, JCR 2.0 was implemented. This brought new features such as transactional versioning and hot backup. JackRabbit is currently on version 2.2.8, and is a more mature and better performing implementation.

In the long term, the goal is for JackRabbit to support JDBC over JCR, which would open up more possibilities for plugin resources while at the same time using JackRabbit as a persistence implementation in DPG.

Many systems are now using JackRabbit in some form: Hippo CMS [4], Day Communique [56], Alfresco [16], Nuxeo [12] and Magnolia CMS [9]. Despite this, there are still problems with maturity and documentation of the technology. This is because many of the large systems using JackRabbit have tweaked it for their own use, and document their solution internally.

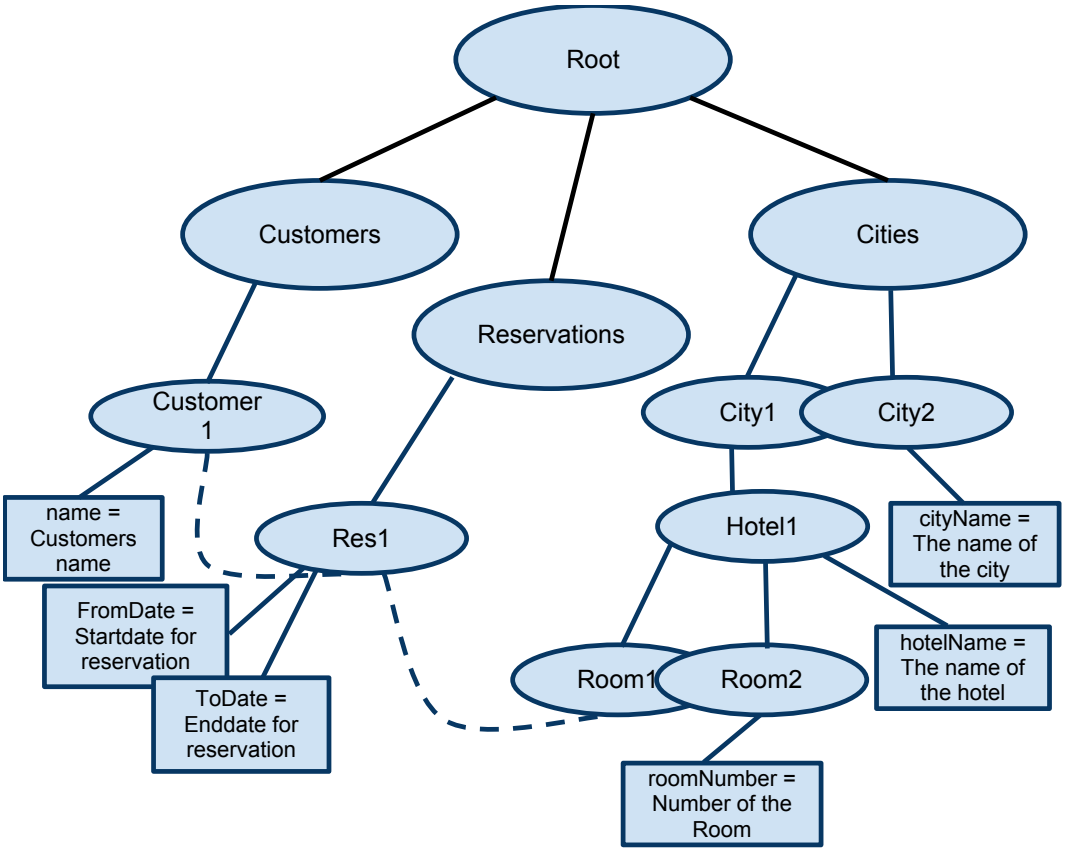

Figure 6.3: Example of a JCR node tree for a hotel reservation system. The circles are nodes, the rectangles are properties containing the data, and the stipled lines show references between nodes.

6.3.3 Object Oriented

Another way of representing information is in the form of objects, similar to how objects are used in object-oriented programming. This brings with it features and characteristics such as inheritence, polymorphism, encapsulation of objects, classes and complex objects. This model solves the impedance mismatch of Java objects and persistent storage.

The feature of an inheritance hierarchy and object references also translates into a better mapping of hierarchical data, than a relational model would be able to provide. A mismatch will occur while mapping circular references from an object model to hierarchical, because this would result in hierarchical nodes being nested an infinite number of times. This should not affect the mapping of patterns and content in DPG, as this will be mapped the opposite way. That is, a pattern defines the hierarchical structure of the content, which can be easily mapped to an object model. This makes an object-oriented data model good for the further development of a uniform persistence solution in DPG.

There exists free and open-source native object databases supporting Java, such as Db4o [108], but the technology is not very mature and there are few alternative vendors to choose from. Though there exists query languages for native object databases, such as *JDOQL* [98], none have been widely accepted and implemented by vendors. This makes it harder to find a portable solution using native object databases.

Another solution is using an Object-relational mapping (ORM) tool. As the name implies, an ORM maps objects to a relational database for persistent storage and retrieval. There are many mature ORMs and relational databases, so this solution will be discussed further.

The current trend of Java frameworks is to move away from XML configuration files towards more pure Java solutions using Java annotations. This desire for pure Java solutions is reflected also in persistent storage. Tools for mapping Java objects to persistent storage are becoming increasingly popular, and specifically mapping Java objects to relational databases. The Java Persistence API (JPA) is currently the most widely accepted standard for mapping Java objects to relational databases. JPA is specified in JSR 220 [106], and the new version 2.0 is specified in JSR 317 [107]. There are multiple implementations of JPA, provided by vendors such as Hibernate [3], EclipseLink [40] and OpenJPA [34]. This makes JPA more attractive, since there exists multiple mature implementations which DPG can swap between in the future. JPA also abstracts the underlying database away from the Java developer, as shown in figure 6.4, promoting separation of concerns [120].

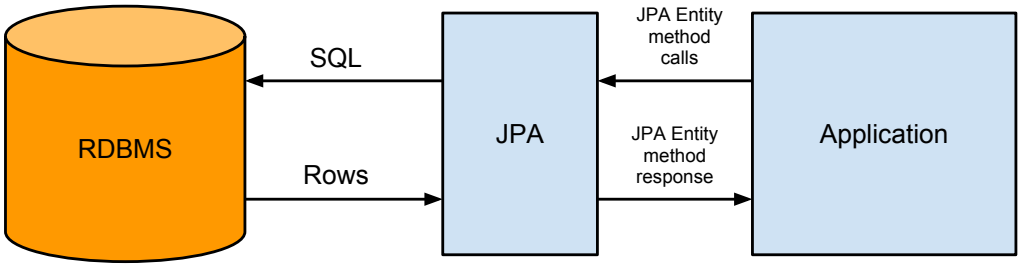

Figure 6.4: Communication between a RDBMS and the Java application through JPA.

Listing 6.3 shows a simple example JPA entity; the `Customer` class. The `Customer` class is mapped to a relational database table using JPA, by defining it as a JPA entity using the `@Entity` annotation. Simple fields are automatically mapped to columns in the table, while some fields such as the unique identifiers of the entities and relationships with other entities need to be specified using JPA annotations. The application can then simply call the object's setter and getter methods for storing and retrieving persistent data.

Listing 6.3: Example of a JPA entity mapping for customers with reservations.

```
1
2   @Entity
3   public class Customer {
4
5     private int id;
6     private String name
7     private Collection<Reservation> reservations;
8
9     @Id
10    public int getId() {
11      return id;
12    }
13
14    public void setId(int id){
15      this.id = id;
16    }
17
18    public String getName() {
19          return name;
20      }
21
22      public void setName(String name) {
23          this.name = name;
24      }
25
```

```
26        @OneToMany(cascade=ALL, mappedBy="customer")
27        public Collection<Reservation> getReservations() {
28            return reservations;
29        }
30
31        public void setReservations(Collection<Reservation>
              newReservations) {
32            this.reservations = newReservations;
33        }
34    }
```

The candidate has some basic experience with JPA through the course MOD-250 at the Bergen University College.

Hibernate

Hibernate is a persistence provider for JPA. It implements the JPA specification, while at the same time offering more features not provided by JPA. Hibernate and its extra features have been the inspiration for the expanded functionality of JPA 2.0, such as the Criteria API, second level caching and unidirectional one-to-many mapping of entities [117]. This shows how influential Hibernate is for ORM tools in Java. Hibernate offers advanced mapping features such as inheritance, polymorphism, support for the Java Collections framework and transitive persistence. It also has support for multiple query languages, including native SQL queries, *Java Persistence Query Language* (JPQL) and *Hibernate Query Language* (HQL).

Hibernate is a very popular framework, because it scales well and has high performance. There is some performance overhead related to Hibernate's mapping and use of reflection, but good strategies for fetching, initialization and locking means that Hibernate can often offer better performance than pure JDBC coding. Hibernate is also very mature and well documented.

It supports a wide variety of databases as its back-end, including Oracle, MSSQL, and the open source PostgreSQL and MySQL databases [49]. This makes it a very portable solution when it comes to persistent storage.

Hibernate integrates well with the Spring framework, and supports both transactions and caching of data. It also supports full text searching, bean validation and auditing/versioning of persistent classes.

6.4 Evaluation of proposed technologies and conclusion

Each framework supports a RDBMS as a persistent storage using JDBC. This is practical, because the IT-department at the Department of Informatics, UiB, can provide PostgreSQL servers for deployment. PostgreSQL also supports UTF-8, which can be defined through the JDBC connection with each of the frameworks, as shown in listing 6.4.

Listing 6.4: Setting the character encoding for persistent storage through JDBC.

```
1
2   jdbc:postgresql://localhost/exampledb?characterEncoding=UTF-8
```

DPG's persistence abstraction layer was designed with a hierarchical structure in mind, but the interface is almost completely neutral to which data model is used in the back-end. DPG handles entire XML documents, not specific values. Entire patterns and presentation content documents are retrieved/stored through the DAO layer. These documents can be stored in an RDBMS as LOBs or CLOBs, or in XML databases either as files or in their natural structure. The PV, PM and PCE also puts the content into Java objects such as `Pattern` and `Entity`, meaning an object model for persistence would work very well with the existing implementation. A good uniform solution is therefore very achievable with each of the proposed persistence technologies.

Table 6.2 presents a summary of the proposed technologies and the criteria specified in section 6.2, in the form of a comparison matrix.

Criteria	**MyBatis**	**JackRabbit**	**Hibernate**
Performance	Very good	Medium	Good
Maturity	Good	Medium	Very good
Documentation	Good	Poor	Very good
Spring integration	Good	Good	Good
Open Source	Yes	Yes	Yes
Transaction support	Yes	Yes	Yes
Support for caching	Yes	Yes	Yes
Standard	Only SQL	JCR	JPA
Provides pure Java API	No	Yes	Yes
Portability	Medium	Good	Very good
Versioning of persistent data	No	Yes	Yes

Table 6.2: Comparison matrix of persistence technologies.

MyBatis is a very high performance solution which works well, but the framework does not implement any standard, and the solution is very low level, which doesn't make it very portable. Mapping the SQL calls to Java objects needs to be done by DPG or the plugins themselves, which will likely tie the implementation to a specific RDBMS vendor. It does not provide with a pure Java API.

JackRabbit has a lot of useful functionality, provides an intuitive structure of data, is portable and looks very good on paper. The choice of JackRabbit in DPG 2.0 was optimistic to the fact that it would mature over time, but unfortunately it hasn't matured enough. The documentation is still poor, and it still suffers from some performance problems, which was why the implementation was abandoned in DPG in the first place.

Hibernate combines good performance with good functionality. It is a very mature and popular technology, which is simple yet powerful to use. It fulfills all the criteria for a new persistence technology to be used in DPG. After this evaluation, Hibernate came out as a very good choice. Hibernate will therefore be implemented as the new persistence technology in DPG.

7

Implementing new plugin resource management in DPG

This chapter will present the devised solution for the new plugin resource solution in DPG. Challenges of the implementation and how these were overcome, will also be discussed throughout the chapter.

7.1 Goals and challenges

The implementation must meet certain criteria, which are defined as goals. The implementation must:

- Use standard JPA, making it a portable solution, both when it comes to RDBMS vendor and JPA implementation.
- Integrate with the Spring framework and use Spring features whenever possible.
- Be easy to use, but still expose powerful features to plugin developers.
- Minimize configuration, both for setting up DPG and for plugin developers.
- Support caching, transactions and locking. This should be automatic unless explicitly specified.

- Be integration tested.
- Separate resources for each plugin.

The main challenge for the implementation was to make a feature rich and general solution, while still having DPG maintain some control over the persistence operations. The solution should also be easy and intuitive for the plugin developers to use. Typically, JPA is set up with a known domain model, and persistent objects are often defined in configuration files. The DAOs are usually also tailored towards specific domain models. Unfortunately, there is no way to know the domain model for the plugins' resources beforehand in the DPG, so the *data access layer* (DAL) must be very general. This makes features such as caching and transaction management a challenge; especially using the method-level solutions provided by Spring.

Even though JPA itself functions as a good and general DAL, giving plugins direct access to this would mean sacrificing DPG's control over the plugin resources. DPG would have to rely on the plugins to avoid using vendor specific functionality or breaking configurations. It would also be harder to use, and require more advanced knowledge of JPA, caching and transaction management.

Though persistence using JPA is mostly intuitive, many errors are not discovered until runtime. It is therefore important to provide guidelines of the specific use of this implementation. These guidelines are directed at plugin developers and are presented in appendix A.

7.2 The Hibernate and JPA persistence context

An overview of Hibernate and JPA was given in subsection 6.3.3. To understand some of the implementation decisions in this chapter, some additional concepts need to be explained.

Figure 7.1 shows the concept of a *persistence context* in JPA and Hibernate. Persistent Hibernate or JPA entities have two states: *managed* and *unmanaged*. When the entity object is in a *managed* state, it means that it is tied to a persistence context, and thereby managed by JPA. JPA then recognizes the changes to the object, and synchronizes this with the persistent storage whenever a transaction ends or a `flush()` operation is called.

An *unmanaged* entity object can either be a new entity, or one that is detached from the persistence context, that is an entity in a *transient state*. Changes to the unmanaged entity are not recognized by JPA, and will not be synchronized with the persistent storage, unless they are merged or persisted into the persistence context.

Unfortunately, the `merge()` operation does not make the passed object managed, but instead makes a copy of it managed. This can lead to concurrency issues and unexpected behavior, because multiple processes can work on what seems to be the same entity, using the same unique identifier, but which in reality are two different objects [21]. The solution to this problem will be discussed further in section 7.6.

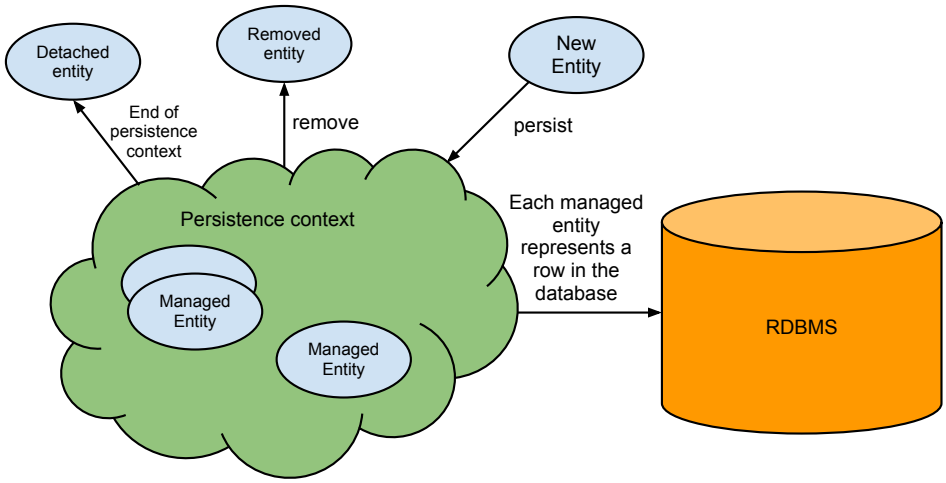

Figure 7.1: Entity objects managed by the persistence context are synchronized with their respective rows in the database.

7.3 Using Hibernate/JPA and the Spring framework

7.3.1 Native Hibernate vs standard JPA implementation

To maintain control over JPA and the plugin data access, a new DAO interface will be developed, called `PluginResourceJpaDao`. The plugin interface is expanded with a `setPluginResourceJpaDao()` method, which the plugin manager will use to give each plugin access to the DAO. This DAO and its methods will be presented in section 7.6.

The implementation itself was evaluated to be either using native Hibernate, or JPA using Hibernate as a persistence vendor. A native Hibernate implementation makes it possible to persist and retrieve JPA entities, so the DAO could seemingly be made to just expose JPA functionality. A native Hibernate solution combined with Spring is very feature rich and powerful, making it very appealing compared to a standard JPA implementation.

Both JPA and Hibernate provide type-safe queries in the form of *Criteria* objects (further explained in section 7.5), though Hibernate can provide *detached* Hibernate `Criteria` objects. This means the DPG plugins could make queries using Criteria API directly. This is not possible with standard JPA, where query building using Criteria must be *attached* to a persistence context.

The Spring framework also integrates very well with Hibernate. Spring can provide a feature rich tool, called `HibernateTemplate` [53], which provides multiple ways to save and retrieve Hibernate (and JPA) entities. These operations were also much more intuitive and predictable to use than their standard JPA counterparts.

The problems arise when trying to use more advanced features such as caching and querying. Native Hibernate uses *Hibernate Query Language* (HQL) for queries, and translates its own proprietary Criteria objects directly to an SQL query. The Hibernate Criteria objects are not an implementation of JPA Criteria, so a plugin resource interface using Hibernate queries or Criteria objects would lock DPG to Hibernate.

Currently, the extra functionality of a native Hibernate implementation in Spring, as opposed to using standard JPA, consists of mostly convenience features. This includes features such as the `HibernateTemplate.findByExample()` method or the `Session.saveOrUpdate()` method provided by Hibernate, which automatically figures out if an entity is new or already has been saved before, and persists or updates the entity based on this. With JPA, this has to be done manually, using the JPA `persist()` and `merge()` operations, along with careful use and configuration of the persistence context, as explained in section 7.6.

A pure JPA implementation provides a lot less features, but is much more portable. The importance of a fully portable solution outweighs the benefits of the convenient functionality provided by Hibernate. This implementation will therefore directly use JPA, with Hibernate as a persistence provider, as shown in figure 7.2.

7.3.2 The JPA implementation

Spring provides many convenient features, such as inversion of control (IoC) mechanisms [41], which will be used to the fullest in this implementation. Spring also provides a tool for JPA operations, called `JpaTemplate`, but this does not provide with any special features, other than automatically converting persistence exceptions to Spring exceptions [54]. This can also be achieved by using the Spring `@Repository` stereotype annotation [90] in the DAO class, which will be used in this implementation. The `@Repository` annotation also tells Spring that this is a repository module, which allows it to be further used in Spring module management and eliminates the need for XML configuration of the DAO bean.

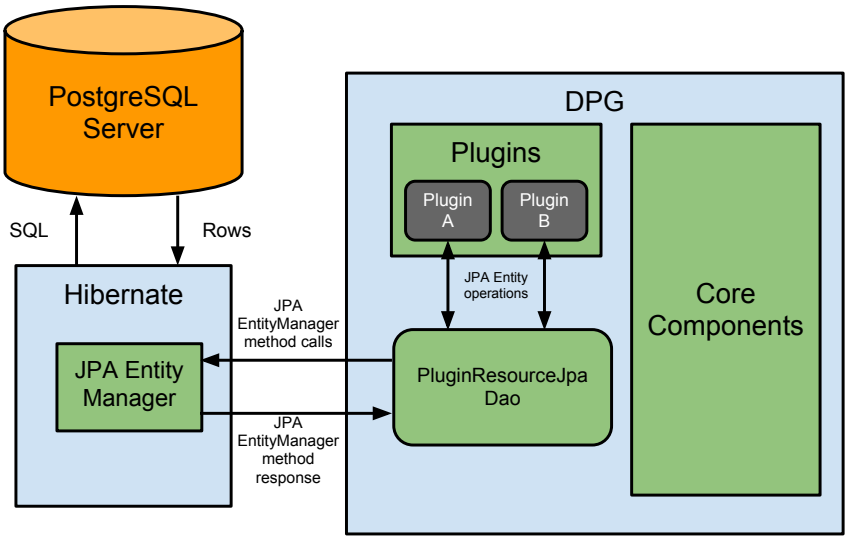

Figure 7.2: The new plugin resources architecture of DPG.

Support for JPA 2.0 in Hibernate was only recently added, so Hibernate's current release, version 3.6 [43], will be used. Spring and Hibernate/JPA currently provide solutions for JPA 2.0 that are backwards compatible with JPA 1.0. Since this implementation will only use JPA 2.0, it is therefore important to remember to use only the JPA 2.0 API in the implementation.

Throughout the implementation, JPA configuration using Java annotations is preferred over configuration in XML files. This continues DPG's trend of moving away from XML configurations, and further minimizes the amount of configuration for setting up DPG. Spring, Hibernate and JPA is constantly evolving towards making configurations more automatic and annotation-based. This also helps a lot for plugins, because entities are automatically scanned in the classpath, and features such as transactions and caching for entities can be made automatic unless otherwise specified.

The back-end RDBMS for deployment will be a PostgreSQL server [13]. This is because it is an advanced open source SQL database, and it is the server provided by the IT-services of the Department of Informatics, University of Bergen. Listing 7.1 shows a part of the `applicationContext-persistence-hibernate.xml` configuration file, where the data source is specified. This can easily be changed to a number of other popular RDBMS vendors [49], by simply changing the data source and the specified SQL dialect to be used by Hibernate. Another JPA implementation than Hibernate, such as OpenJPA [35], can also be used by simply changing some

lines in the JPA `persistence.xml` configuration file.

Listing 7.1: The current data source configuration using a local PostgreSQL server.

```
1   ...
2  <bean id="postgreDataSource"
3  class="org.apache.commons.dbcp.BasicDataSource" destroy-method="
       close">
4    <property name="driverClassName" value="org.postgresql.Driver"/>
5    <property name="url"
6    value="jdbc:postgresql://localhost:5432?useUnicode=true&amp;
       characterEncoding=UTF-8"/>
7    <property name="username" value="postgres"/>
8    <property name="password" value="postgres"/>
9  </bean>
10 ...
```

7.4 Entity management

There are multiple ways to access a JPA implementation, and they all differ on how the application is given access to the JPA DAO class, `EntityManager`. Spring provides multiple ways to access an `EntityManager` object in a Spring environment. An `EntityManagerFactory` object can be obtained through JNDI [82], or through a Spring bean such as the `LocalContainerEntityManagerFactoryBean` bean class, which will be used in this implementation. This is the only solution which provides with full JPA capabilities in a Spring environment, including full and flexible control over configuration within the application, and support for both local and global transactions [99].

The entity manager factory can be injected directly in a DAO, with the Spring `@PersistenceUnit` annotation. A problem with this solution, however, is that when the DAO creates a new instance of `EntityManager`, the persistence context is reset, and all entity objects will become detached (unmanaged), as explained in section 7.2. In the new plugin resource implementation, this sort of behavior should be avoided, because it is very error-prone unless the plugin developers have very good control over their entities and how they are managed. One way to solve this, is to use the Spring provided shared `EntityManager` object to be injected instead of the factory. A shared `EntityManager` object is therefore injected into the `PluginResourceJpaDao` using the `@PersistenceContext` annotation. An `EntityManagerFactory` instance still works behind the scenes to provide performance benefits of automatically preserving `EntityManager` resources in a connection pool [96].

By default, the Spring injected `EntityManager` bean uses a transaction-scoped

persistence context, which means a persistence context only exists inside of a transaction and all entities are detached after a transaction is committed. This works against the benefits of a shared `EntityManager` in this implementation, because the general DAO forces nearly every action to be in its own transaction, thereby detaching all entities from the persistence context immediately. This also means that entities in a collection cannot be lazily loaded in a plugin. To remedy this, the implementation will use an *extended* persistence context, which ties the persistence context to the scope of the `EntityManager` object [50]. This solution uses more memory, since the persistence context will contain more entities, but will solve these problems and make for a much more intuitive solution for plugin developers.

The JPA implementation of the `PluginResourceJpaDao` interface is a class called `PluginResourceJpaDaoImpl`. The Spring injected DAO bean referring to this class is called `pluginResourceJpaDao`. The extended persistence context is not thread-safe with the use of a singleton bean in Spring, which is default for Spring beans. The attribute `scope` of the `pluginResourceJpaDao` bean is therefore set to the value `prototype`, which means a new bean is created each time a request for that bean is made [48]. This also gives control over the life of the bean, and thereby the persistence context. Since plugin operations only get called through a view, a new persistence context is set for each view, through the plugin manager, so the increased memory usage should be minimal.

7.5 Query language

Hibernate provides multiple ways to retrieve data:

- Using HQL queries.
- Using JPA Criteria objects.
- Using Hibernate Criteria objects.
- By entity example, class etc.
- Using JPQL queries.
- Using native SQL queries.

The only real pure JPA queries are with JPQL [84] and JPA Criteria API [89], which will be the only ones used in this implementation, to maintain portability.

JPQL queries are easy to read and understand, as shown in listing 7.2, but suffer from some issues. The queries need to be parsed every time, and they are not type-safe,

meaning typos will likely not be noticed immediately, because the query string is not checked until runtime. Pure JPQL string queries are vulnerable to injection attacks, similar to SQL injection [87]. A solution could be to include a list of parameter objects with the string query, implementing `javax.persistence.Parameter` [81] together with an argument. There is unfortunately no simple way to let plugins make their own JPQL queries, and at the same time *force* them to make queries parameterized. For this reason, JPQL queries will not support parameters, and should only be used for very simple queries which do not directly include user input values. Criteria queries should therefore be used for secure, type-safe and high-performance queries, while JPQL can be used for simple static queries.

Listing 7.2: A simple JPQL query fetching all User entities with ages over 30.

```
1 String jpqlQuery = "SELECT u From User u where u.age > 30";
```

JPA 2.0 provides with an API to define queries dynamically with objects. This API is referred to as the JPA Criteria API. These queries are checked at compile time, prohibiting syntactically incorrect queries, and they force parameterized queries, making them less error prone and more secure [89]. Some simple examples of both JPQL and Criteria queries are shown in the guidelines in appendix A.

JPA also supports *named queries* [116], which can be mapped with the `@NamedQuery` annotation. Unfortunately, this can be either a JPQL query or a native SQL query, and naming collisions can easily occur in this implementation. Named queries also give the plugin developer too much responsibility over configurations, such as caching. The benefits of having named queries are small compared to the increasing risk of an unportable solution.

7.6 New plugin resource interface

This section will present the new plugin resource interface. The Java code and Javadoc for this interface is shown in listing 7.3. How the plugin developers should use these methods will be presented as guidelines in appendix A.

JPA entities are defined using the `@Entity` on a class, meaning they are determined at runtime. This provides a challenge for the implementation of the DAO, as not all methods can be made generic, because there is no way to know if the objects are JPA entities at compile time. Some methods, such as the ones using type safe JPA `Criteria` queries, can be generically typed, while others require casting of objects. This means that the plugins executing a JPQL string query need to know the expected type of the response. This should not be a problem, because the return type is defined in the query itself.

Listing 7.3: The new plugin resource interface

```
1
2     public interface PluginResourceJpaDao {
3
4   /**
5    * Persist a JPA entity if it's a new entity, otherwise synchronize
         persistence context,
6    *  and thereby the entity, with database.
7    *
8    * @param entity JPA Entity to save
9    */
10  public void saveOrUpdateEntity(Object entity);
11
12  /**
13   * Save or Update JPA entities inside a single transaction.
14   * For each entity, persist if it's a new entity,
15   * otherwise synchronize persistence context
16   * with database.
17   *
18   * @param entities JPA Entities to save
19   */
20  public void saveOrUpdateEntities(Object... entities);
21
22  /**
23   * Find all JPA entities with the provided name.
24   *
25   * @param name
26   * @return List of JPA entities loaded from the database
27   */
28  public List<?> findAllEntitiesByName(String name);
29
30  /**
31   * Find all entities by class.
32   *
33   * @param <T> The specific class marked with @Entity annotation
34   * @param clazz a class
35   * @return List of JPA entities loaded from the database
36   */
37  public <T> List<T> findAllEntitiesByClass(Class<T> clazz);
38
39  /**
40   * Find all entities by class and unique id (primary key).
41   *
42   * @param <T> The specific class marked with @Entity annotation
43   * @param clazz a class
44   * @param id unique id of the JPA entity
45   * @return JPA Entity loaded from the database
46   */
47  public <T> T findEntityByClassAndId(Class<T> clazz, Serializable id)
        ;
48
```

```
49   /**
50    * Get a JPA Criteria builder which can be used to build criteria
          queries.
51    * These queries can then be used in the method findByCriteriaQuery.
52    * @return CriteriaBuilder
53    */
54   public CriteriaBuilder getCriteriaBuilder();
55
56   /**
57    * Execute a find query based on the JPA Criteria API. This method
          does not allow
58    * DELETE or UPDATE queries.
59    *
60    * @param <T> The specific class marked with @Entity annotation
61    * @param query JPA Criteria query
62    * @return List of entity results from the query
63    */
64   public <T> List<T> findByCriteriaQuery(CriteriaQuery<T> query);
65
66   /**
67    * Execute a find query based on the Java Persistence Query Language
          .
68    * This method does not allow DELETE or UPDATE queries.
69    *
70    * @param query JPQL query
71    * @return List of results from the query
72    */
73   public List<?> findByQuery(String query);
74
75   /**
76    * Remove a JPA entity from the database.
77    *
78    * @param entity JPA entity to remove.
79    */
80   public void removeEntity(Object entity);
81
82   /**
83    * Remove multiple JPA entities from the database inside a single
          transaction.
84    *
85    * @param entities JPA entities to remove.
86    */
87   public void removeEntities(Object... entities);
88
89 }
```

A more detailed explanation of some of these methods will be given in the following paragraphs.

saveOrUpdate(): This method persists a JPA entity if it is a new entity, or synchronizes the persistence context, and thereby the entity, with the database. It was one of the hardest methods to implement, because JPA does not support a convenient way to automatically save or update an entity, such as a Hibernate `Session.saveOrUpdate()` operation. A popular belief is that the JPA `merge()` operation is JPA's answer to this problem, but it suffers from a fundemental flaw that is easy to overlook. The difference between JPA's `merge()` operation and Hibernate's `update()` operation, is that the `update()` method *attaches* the passed entity to the persistence context, while the `merge()` method *copies* the passed object into the persistence context, and then returns the copy. This means that the `merge()` method does *not* persist the passed entity object, or any of the referenced entities, into the persistence context [21].

It will not be possible to automatically point all eventual old references in the plugin to the new managed object. This makes it very easy for plugin developers to mistake a detached entity object with one that lives in the persistence context, and will very likely lead to concurrency issues, and unexpected behavior such as breaking bidirectional associations [88]. The solution is therefore to avoid any use of the `merge()` operation, and rather rely on the extended persistence context. This means changes to the managed entities from outside of transactions are buffered in the persistence context. Synchronizing with the database then requires a `flush()` operation to be called on the `EntityManager` inside the transactional method `saveOrUpdateEntity()`.

getCriteriaBuilder(): This method provides a JPA `CriteriaBuilder` object which can be used to build criteria queries. It is not possible to simply send a `CriteriaQuery` object to the plugin, because a `CriteriaBuilder` object is needed to add certain criteria to the criteria query. Unlike Hibernate, JPA does not currently support *detached* criteria query building, so a JPA `CriteriaBuilder` object is sent to the plugin through this method. Figure 7.3 shows how a plugin can use the `PluginResourceJpaDao` bean to retrieve a `CriteriaBuilder` object, build the criteria query and then execute the query using the `findByCriteriaQuery()` method.

findByCriteriaQuery() and findByQuery(): These methods support executing queries for retrieving data, but not for updating or deleting. A JPQL injection

attack can therefore never be used to update or delete anything. Updating and deleting should be done through the `saveOrUpdate()` and `removeEntity()` methods respectively. They both use the `Query.getResultList()` method inside a transaction, which Hibernate can guarantee will never return stale data [51].

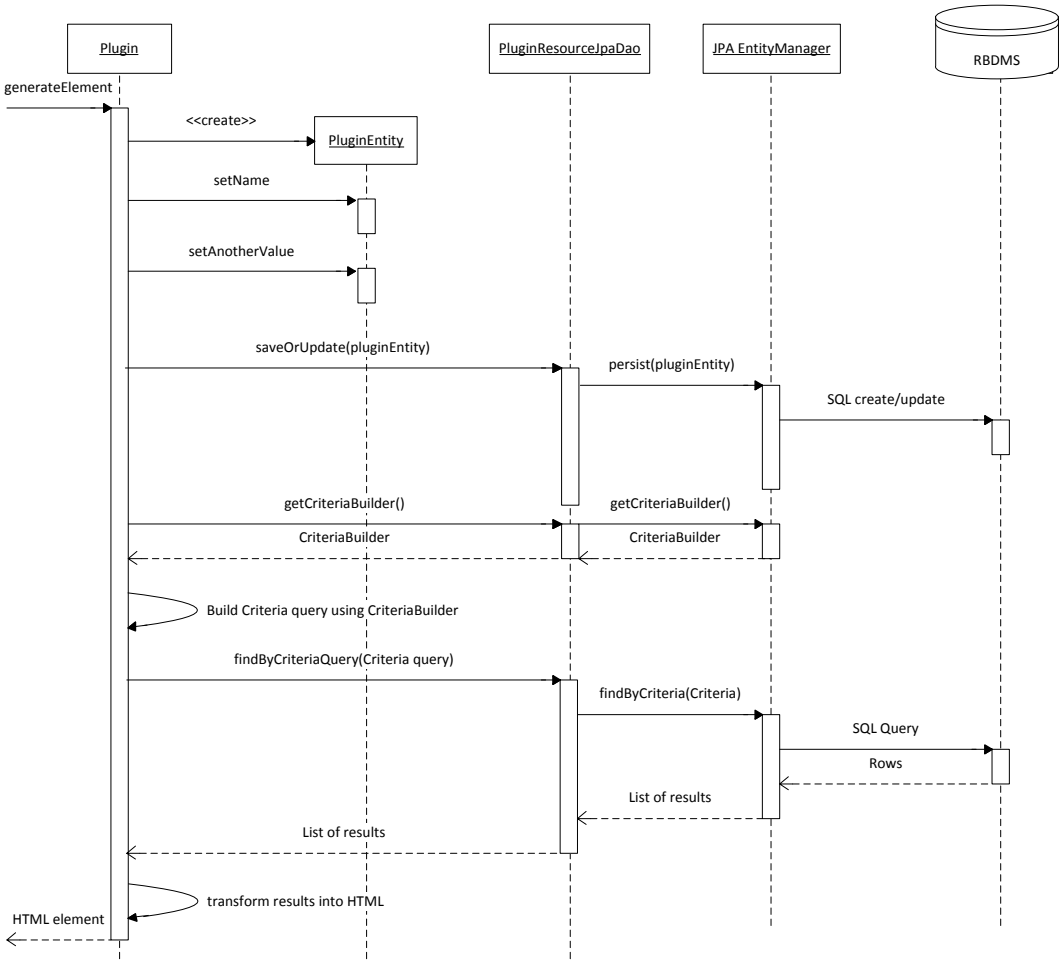

Figure 7.3: A plugin persisting and accessing resources using persistent objects and JPA `Criteria` queries.

7.7 Transaction management and locking

The extended persistence context solution of this implementation reduces the chance of concurrency issues, as explained in section 7.4. A transaction management covering lower level issues is still required. Plugin developers need not bother with the transaction management of the implementation. To the plugin developers, entities will be updated in the database at the same time as the methods are invoked, since each method is run in its own transaction.

The implementation uses Spring's annotation driven transaction management for transaction configuration [102]. As mentioned in section 7.4, both global and local transactions are supported. Default for the Spring provided transaction manager is that propagation of transactional methods is required. This means that all methods mapped with the `@Transactional` annotation are given a logical transaction scope, but are mapped to the same physical transaction if one exists. If one of these logical transactions trigger a rollback, the outer transaction scope containing them will cast an `UnexpectedRollbackException` exception. A rollback will be triggered if an exception occurs inside the transaction, unless the exception is explicitly defined to be ignored. This can be caught by the plugin, which can then choose to retry the process. Spring also defines six other propagation behaviors, but these are for more special cases, like throwing an exception if there is an existing transaction. An isolation level for transactions can also be defined. Its goal is to prevent issues such as lost updates and dirty reads, as explained in section 6.2.2.

Locking is important to avoid collisions of concurrent updates on detached entities, because this happens outside of the transaction and persistence context scope. There are two main types of transactional locking supported by JPA; pessimistic locking and optimistic locking. *Pessimistic locking* locks the resource from the time it is first accessed until the transaction is complete, disallowing concurrent access to the resource. *Optimistic locking* saves the state of the resource at the time it is accessed, but does not lock it, allowing concurrent access to the resource. When the resource is then *updated*, its state is compared to when it was first accessed, and if the two states differ, a conflict is detected and the transaction is rolled back. JPA 1.0 only supports optimistic locking, while both optimistic locking and pessimistic locking of persistent objects are supported by JPA 2.0 [69]. A JPA entity can be optimistically locked if it contains a lock value field annotated with `@Version`. Pessimistic locks are rarely needed, and can cause bottlenecks and deadlocks, so it will not be implemented. This implementation will automatically check if the entity is versionable, and optimistically lock it if it is. If a process tries to update an entity with a version lower or equal to the current persisted version, an `OptimisticLockException` will be thrown, giving the process a chance to try again.

7.8 Caching

EhCache [58] will be used as the cache provider for this implementation. This is because it integrates very well with both Spring, Hibernate and JPA, and was even previously bundled with Hibernate.

There are multiple ways to cache JPA entities with JPA, Hibernate and Spring. Spring provides with DAO method level caching, but this would make little sense with a generic DAO. This is because it will not be possible to find a reasonable time to flush or invalidate the cache based on the generic methods of the DAO. Caching query results and entities will still work very well. The implementation will therefore be made to support caching of both query results and entities.

The persistence context itself works as a cache for entities, which is defined as the level 1 cache. To support caching of entities across the entire application, a level 2 cache needs to be used. This cache is tied to the Spring managed `EntityManagerFactory` bean. The advantages of a level 2 cache is that database access for already loaded entities will be avoided, and read queries can be a lot faster. This is because when a query is executed, Hibernate first checks the level 2 cache for the entity [70].

Though Hibernate's caching strategies are more configurable in the annotations, the JPA caching is easier to use and standardized. This implementation will therefore use standard JPA entity caching using the `@Cacheable` annotation instead of Hibernate's proprietary `@Cache` annotation. All entities will be automatically cached unless explicitly specified otherwise with the `@Cacheable(false)` annotation. To enable this, the persistence unit property `<shared-cache-mode>DISABLE_SELECTIVE</shared-cache-mode>` was set. Query results provided by the DAO are also automatically cached, which will dramatically increase performance.

7.9 Separation of plugin data

A problem with Hibernate and JPA is that entities with the same names are not allowed. The same problem applies for the naming of database tables to be generated by a JPA entity. A custom naming strategy can be implemented using the `NamingStrategy` interface of Hibernate, to intercept and automatically change the names of tables tied to entities. This is the only safe way to automatically change the names without causing confusion when making queries, because Hibernate automatically converts the entity names in the queries as well. The problem is that the newest version of Hibernate only passes a *string* containing the simple class name

for the entity, not the canonical name, so there is no way to split the entities by package name. This can also not be solved using reflection, as the class or object is never passed through the `NamingStrategy` interface. This would also be a solution which would likely lock DPG to Hibernate, and would not solve the problem for queries.

To solve this, plugins must follow a naming convention for naming their entities and tables. Plugins should always name their entities and tables with the prefix `pluginName_`, as explained further in the appendix section A.1.

It is not possible with JPA to completely prevent plugins from accessing other plugins' data, because they can always find the entity classes in DPG if they specifically look for them. The new plugin resource solution still prevents accidental use of the same resources, which was the biggest concern.

Plugins get full control to separate their own resources by page, presentation, pattern or user. This was possible before aswell, but harder since the resources were physically split automatically by presentation, while now they are split by plugins. This gives plugins the ability to easily reuse resources across presentations, such as data migrated from other systems, avoiding unnecessary duplication of resources.

7.10 Integration testing

As specified by Karianne Berg [17], the application should be unit tested with a minimum of 75% test coverage. Test driven development [67] was used throughout the implementation, to force a good test coverage and reliable solution. This was seen as extra important for a portable solution such as this JPA implementation, since the tests can be used to see if an eventual new implementation works as it should and as expected by the plugins.

The testing of the implementation uses JUnit 4 [7] and the Spring TestContext Framework [101]. This was chosen because it lets the tests easily specify different application contexts to use, along with support for autowiring dependencies, changing transaction management and caching configurations. It also provides with templates for simple JDBC querying to confirm states and annotations for timed tests, repeating tests and more.

A new JPA persistence unit and Spring application context was made for testing. This allows the tests to use a different database and configuration, so as to not corrupt data in the database used for deployment. The database used for testing is an in-memory database, called HSQLDB [55], which will only live inside a single test

run. Test data is reset for each unit test, to ensure they do not affect each other.

A problem with the testing is that JPA's automatic entity scanning scans from the parent of the classpath META-INF/ folder, where the file `persistence.xml` is loaded from. This could be remedied with the implementation of a custom Spring `PersistenceUnitPostProcessor` class, or by defining a new `persistence.xml` file with the `persistence-xml-location` Spring property. The first option does not give enough benefits compared to the time it would take to develop. The second option does not work, as Hibernate still scans entities from the classpath. Future testing of plugins and an eventual full JPA persistence implementation will require entities from the main package anyway, so it is better to use an entity from the main package for testing.

EclEmma [20], a Java code coverage tool in the form of an Eclipse [39] plugin, was also used in the development of the unit tests. EclEmma checks Java test code coverage, by analysing methods and conditions which are visited by the JUnit unit tests. It does not give a full picture of everything that should be tested, but at least provides a method and instruction level view of what code the tests cover. Figure 7.4 shows how it also presents an intuitive overview of both code and packages when tests are run. This overview marks code with red, yellow or green depending on the coverage, and shows the test coverage percent of each package and class. This makes it easy to remember testing all methods and conditions. Cobertura [25] has been used at JAFU to track DPG's Java code test coverage in a similar way in deployment. EclEmma shows the current test coverage of the implementation as over 90%, which is a good margin above the previously specified 75% minimum.

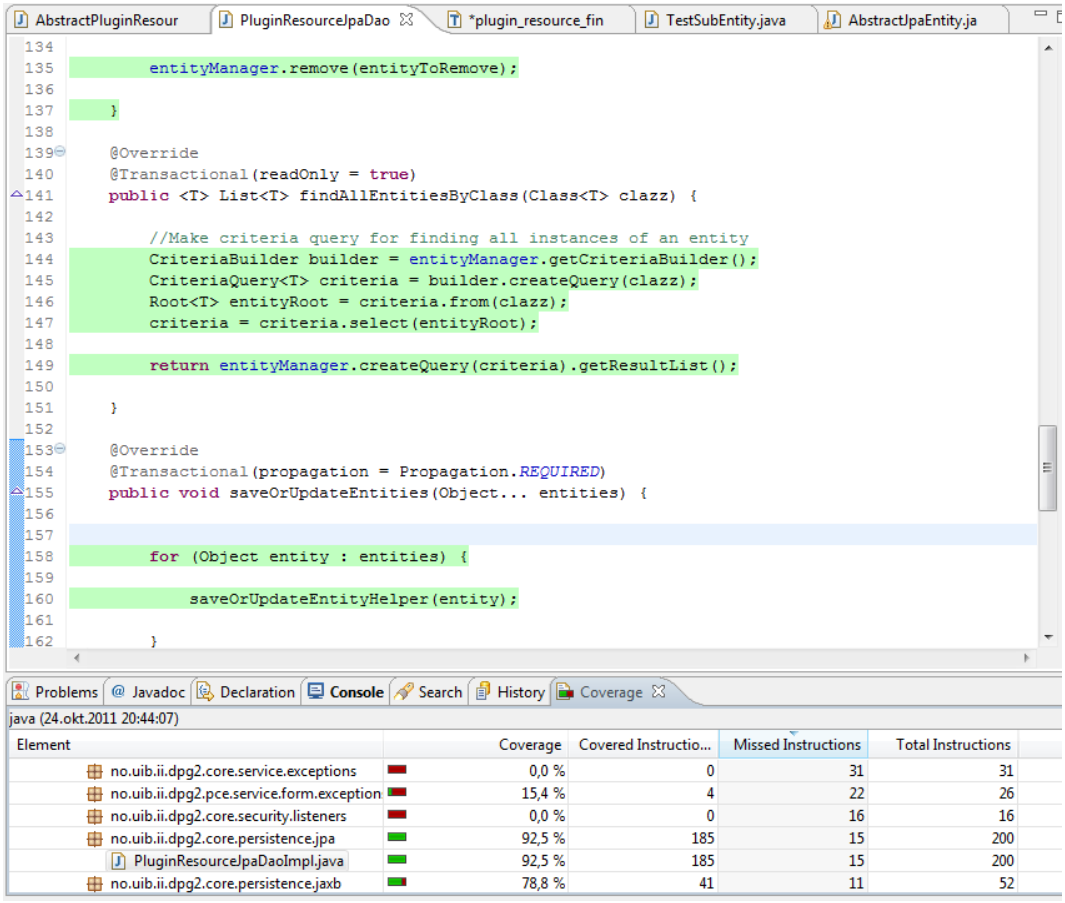

Figure 7.4: EclEmma eclipse plugin showing unit test code coverage. Code marked in green means it is covered by a test.

7.11 Evaluation of implementation

This implementation added the `PluginResourceJpaDao` interface and the `PluginResourceJpaDaoImpl` class. Changes were made in DPG's plugins, `FieldPluginManager` class and `FieldPlugin` interface. A new Spring application context file called `applicationContext-persistence-hibernate.xml` was made for configuration of Hibernate and JPA. The file `persistence.xml` is used for configuration of JPA persistence units, while the level 2 cache configurations are contained in the `ehcache.xml` file. Only a couple of lines regarding database configuration needs to be changed for deployment, as shown in listing 7.1.

To summarize, an evaluation of the implementation, the criteria and goals defined in section 7.1 will be discussed. This should be compared to the evaluation of the old resource plugin solution in section 5.1.

The new plugin resource solution in DPG is a portable solution, implementing a standard, JPA 2.0. This means that both the current RDBMS used, PostgreSQL, and the JPA implementation, Hibernate, can be easily replaced in the future, and plugins will not be affected. The testing of the DAO layer makes it easier to check if a new implementation works as intended.

The performance of Hibernate was evaluated to be satisfactory in chapter 6, and the performance has been further improved with the implementation of automatic caching of data.

The new plugin resource solution in DPG is easy and intuitive to use. Plugin developers can use Java objects for persistent storage of data, and use references and relationships between objects for complex structures. Plugins' entities are automatically scanned, and other than the annotation of the entities themselves, plugins do not have to configure anything. The DAO implementation provides with and handles transactions, while at the same time providing with locking support.

The solution integrates well with the Spring framework, utilizing its IoC functionality and JPA support. The current version of Spring (v.3.0.5) and Hibernate (v.3.6.3) used in DPG, makes configuration very easy and favors the use of Java annotations over XML configuration files.

Though it was not possible to fully separate plugin resources with this implementation, accidental corruption of data between plugins will no longer be likely.

Versioning of persistent data was considered very low priority and not implemented in this solution. A proposed solution for this will be presented in section 8.3.12.

The poll plugin, shown in section 5.1, can now store users and their answers in a JPA entity as shown in figure 7.5. The user's answer can be fetched with a simple JPQL `select` query, and the number of each answer can be fetched using the JPQL `count` operator. These are both very fast operations for a RDBMS.

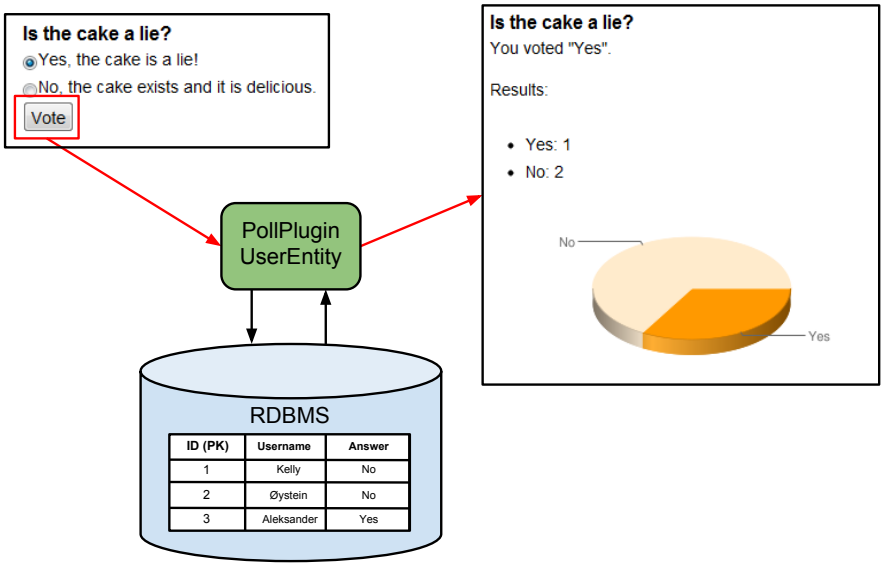

Figure 7.5: Poll plugin can use the new plugin resource solution to easily and efficiently fetch the required data through a JPA entity.

While relatively simple plugins, such as the previous examples, reap great benefits from this new solution, it really opens up possibilities for some complex and powerful plugins, such as plugins which migrate data with other systems than DPG. An example of this would be a plugin storing and presenting geographic data provided by OXD.

8

Evaluation, Experiences, further development and conclusion

8.1 Evaluation of goals

The overall goal of this thesis was to expand the plugin architecture and thereby the applications of DPG. The sub-goals to achieve this were:

1. Evaluate the current plugin architecture of DPG 2.1.
2. Propose and evaluate solutions for improvement of the plugin. architecture in DPG, and implement the best solutions.
3. Evaluate the current solution for persistent storage of DPG plugin data.
4. Propose and evaluate new solutions for plugin resources in DPG.
5. Implement an improved plugin resource solution for DPG.
6. Provide guidelines for using the new plugin resource solution.

Subgoal 1 was achieved in collaboration with Aleksander Waage [114] in chapter 3. The current solution was thoroughly documented as it was being evaluated, with the focus on making explanatory figures and diagrams. Major weaknesses were discovered while trying to make plugins for advanced maps using dynamic content.

This included entity list handling in DPG, support for multiple entity fields in plugins and support for multiple entity instances in a single view.

Followed by this, subgoal 2 was achieved by proposing direct solutions to the weaknesses which were discovered. This, along with the implementation improving the plugin architecture, was also achieved in collaboration with Aleksander Waage in chapter 4. Throughout the evaluation and the implementation, there was always a focus on generality, to make sure the solutions cover as many cases as possible. A plugin for generating dynamic maps with markers loaded from the PCE was made to test the implementation. The plugin was successfully made using the new plugin architecture. It will be extended with more functionality, such as user comments in map markers, by Aleksander Waage in his Master thesis [114].

The current solution for plugin resources in DPG 2.1 was evaluated in chapter 5. The evaluation concluded that it was a very bare-bone and simple solution, with no features for structure, performance, concurrency control or data integrity. This proved that a new plugin resource solution was very important, especially given the importance of plugins and their role in data collection and user interactions in DPG. This results in subgoal 3 being achieved.

Chapter 5 proposed solutions for a new and improved architecture for plugin resources in DPG. These solutions were evaluated based on maintaining portability and giving DPG some control over the plugin resource management, while at the same time providing good performance and functionality for plugins. The conclusion was to provide plugins with an indirect access to the persistence technology through a DAL controlled by DPG. In chapter 6, popular data models and technologies were discussed and evaluated. The criteria for the technologies were that they provide an intuitive solution requiring little configuration or special knowledge, while providing portability, concurrency control and good performance. The conclusion was to use Hibernate as a JPA implementation, as this is a very mature technology with benefits of both performance and ease of use. This means that chapter 5 and 6 achieve subgoal 4.

Chapter 7 presents the implementation of a new and improved solution for plugin resources in DPG. All decisions are thoroughly evaluated and described, with a focus on ease of use and portability while providing with powerful features for any types of DPG plugins. Plugin resources are now structured as Java objects using JPA. CRUD operations can be accessed through the new `PluginResourceJpaDao` DAO, which automatically handles things such as caching and transactions. The solution currently uses Hibernate as a JPA implementation, and PostgreSQL as a persistent storage back-end, but both of these can be easily changed in the future, due to focus of portability in the new plugin resource solution. More specific improvements over the old plugin resource solution is presented in section 7.11. This

implementation achieves subgoal 5.

Finally, appendix A provides guidelines for using the new plugin resource solution. This means subgoal 6 is achieved.

The fulfillment of all 6 subgoals results in a vastly improved plugin architecture. This means DPG is now a reasonable choice for advanced data collection and user interactions, making DPG more future proof. This has become increasingly important as the development of the web has veared towards user interaction and collaboration as defined by the Web 2.0 standard. It also opens up for possibilities for migration of data with other systems which can be persisted and presented using DPG. The main goal of the Master thesis is therefore achieved.

8.2 Experiences

8.2.1 Development process and methodology

Master students working on DPG are provided with office space at the JAFU office at the Department of Informatics. This work environment, along with the collaboration with other students and working together on a large, complex system, proved to be a great experience.

This thesis has included work on nearly every component of DPG. That includes development of plugins, changes to the persistence layer, pattern development, and general changes to both the PCE, PV and PM. It was challenging to work with such a large system and try to understand the code of a collaborative effort from many Master students. A reoccuring problem during this Master thesis, was that a lot of DPG's implementation details were very poorly documented. This consumed a lot of time for both the candidate and others working on DPG. To try to mitigate this problem for future developers of DPG, both the current state and architecture, and the new implementation, was thoroughly documented as the candidate went through it.

The candidate tried to follow the agile methods, as described in section 1.4. This was difficult at times, especially test driven development, but ultimately proved to give great benefits. For example, pair programming forces discussions about every implementation detail, and test driven development makes the code more focused and easier to debug.

Throughout the development, the candidate came in contact with a lot of different technologies used in DPG. While many of the technologies were made familiar

through courses taken previously during the Master degree, a lot was completely new. The entire Master thesis has been a learning process, and the candidate has matured his development skills greatly.

8.2.2 Technologies

The Spring framework was at first a scary concept, with a lot of apparent "behind-the-scenes magic". After learning about how it works, and using it during development, the candidate grew very fond of the many brilliant features and ease-of-use that Spring provides. The Spring framework saved a lot of development time.

Other technologies such as JDOM, XSLT and Velocity Templates also proved to be of great value. DPG uses Log4J [33] for advanced logging, and this provided great help in understanding the system and for debugging code.

At first, the implementation of some complex features using JPA and Hibernate produced some behavior that was very strange to the candidate. The implementation required a lot more research than the candidate had anticipated, but in the end was well worth it. The Spring source- [104] and Hibernate documentation and forums [3] helped a lot in understanding these technologies.

Experiences with other technologies, such as the source code and project management tools Subversion [36] and Trac [95], on a larger system has proved to be invaluable. They have made collaboration easier, and the version and ticket history can be used as documentation to better understand the system.

8.3 Further development

This section will present some possible improvements and further development of DPG. During the Master thesis work, the candidate gained a lot of experience with nearly every aspect of DPG, as well as its flaws. This sparked ideas for improvements in DPG.

8.3.1 Plugins reacting to events in DPG

The plugin architecture of Joomla! follows an observer design pattern [67], which means that plugins are observer classes that attach and react to a global event dispatcher object in the Joomla core. The plugins only get access to presentation content by handling the throughput of the content by implementing methods such

as `onBeforeContentSave()` and `onBeforeDisplayContent()` which respectively handle content before it is saved and before it is presented in HTML. This can inspire a similar pattern in DPG, where multiple plugins may react to the same event and manipulate the same content. An example is where one plugin fetches an image file path, while the other wraps it in an HTML image tag, splitting the responsibility of the `ImagePlugin` plugin in DPG into multiple plugins.

8.3.2 Communication between plugins

There is currently no functionality for plugins in DPG to communicate with each other. This proved to be a problem in the collaboration of Øystein Rolland and Morten Høiland during their Master theses, in their efforts to make plugins providing XForms support in DPG. One plugin was made to present XForms in DPG, while the other was made to design and persist them. This required a communication between the two plugins, which was not facilitated. The plugins ended up using a loophole, communicating through a configuration file in DPG. It would be useful with a good way for plugins to communicate with each other.

8.3.3 New functionality in PCE

In this Master thesis, support for plugins using multiple entity fields was developed. A plugin can also define its own form field to be presented in the PCE. However, this form field is currently only tied to *one* entity field. An idea would be to give a plugin the option to generate a form field in the PCE which can get multiple values. An example of this is using `DynamicMapPlugin` to let a publisher input markers by clicking on a map. The data for these markers (which contain name, latitude and longitude) is then given to the DPG, which persists and handles them like an *entity*.

During his Master thesis work, Øystein Lund Rolland developed a solution for designing XForms to be used in DPG. A part of this solution was a tool for administrating XForms, intented for publishers only. This tool was presented using the PV, but would be much more natural in the PCE along with all the other publisher tools. An idea for further development would be to give plugins the opportunity to present administration tools in the PCE.

8.3.4 Further abstract the plugins from the pattern designer

One of the goals for the solutions presented in this Master thesis, was to abstract as much as possible between the different users and developers of DPG. The pattern

designer much too often needs to know details of the plugins it uses. An example of this is that the pattern designer currently needs to define the JavaScript libraries that the plugins use, in the pattern. It would be an idea to let plugins define these libraries themselves and let DPG handle this further. There may also be other things that can improve abstraction between pattern designers and plugin developers, or even plugin developers and DPG. This should be further looked into.

8.3.5 Abstract plugins from the DPG core

The DPG core components are very tightly integrated with certain plugins, leaving some specific references to plugins for list and sub-entity functionality. This means that, currently, changes to the plugins or plugin interface can break some very core functionality in DPG. The plugins should be completely abstracted from the DPG core for a looser coupling and higher cohesion of DPG components [67].

8.3.6 Further development of PPDev

PPDev is a web-based tool for developing and validating presentation patterns in DPG. It was developed by Jostein Bjørge in his Master thesis [18].

The presentation pattern specification contains very specific rules to how a presentation pattern should be structured. It would therefore be reasonable to make a GUI for designing patterns. This could involve things such as using drop-down menus for field types (which contains a list of plugins). An example of a similar idea is the Field UI module for the Drupal CMS [27]. This functionality should be part of a further development of PPDev. Designing a pattern can be made a lot easier, and less error-prone, and should be taken into consideration for further development of DPG.

8.3.7 Upgrading plugins

After DPG 2.0, and especially 2.1, many plugins have been developed. It is also very likely that many more plugins will be developed in the future. The ever changing architecture and API of DPG, means that often plugins need to be rewritten for the new versions of DPG. This also means that upgrading a DPG in use, will likely mean that content for both DPG and plugins will be corrupted. Someone using an older version of DPG would therefore be more hesitant to upgrade to a newer version. An upgrade method for both DPG and plugins would make the transition more smooth. Examples of systems with solutions like this are Moodle [73] and Drupal [26].

8.3.8 Tighter PV and PCE integration

Because of the nature of a presentation pattern, a pure WYSIWYG (What You See Is What You Get) [122] solution for publishers would not be preferable. That is, a solution where the publisher can directly change the content in the presentation itself. This is because content of entity instances are usually presented in multiple views, so changing it one place will usually change it other places as well. However there could still be made some improvements to how the publisher edits and views the content. It is currently hard for a publisher to see where the content is presented. A better solution for this could involve something as simple as a direct link for publishers in the presentation to the actual content in the PCE.

8.3.9 Versioning of persistent data

A functionality that has become increasingly popular lately, is versioning of persistent data. Many persistence solutions support this, and it would be a good addition to DPG. This would include providing versioning of presentation content in DPG, and plugin resources. A publisher could then change to previous versions of the content whenever they want, through for example a drop-down menu in the PCE containing the different versions of entity instances.

Versioning is already supported with JackRabbit, but if the persistence layer of DPG is changed to use Hibernate, Envers [2] can be easily used to support this functionality. Envers implements the concept of *revisions* of JPA entities.

8.3.10 Support for multiple languages

Currently, support for multiple languages can be done at a pattern level, so different parameters can be sent to the plugin through the `pluginConfig.xml` configuration file. Each plugin then has to define a language parameter explicitly, and pattern designers must feed the language value to each plugin. A better solution would be to include the current language used in the `FieldPluginBean` object which is passed to each plugin, so no configuration is needed.

8.3.11 Extended support for simple entity field types

In DPG 2.1, the only simple field type is `string`, handled by the string plugin. Basic support for fields such as decimal numbers and booleans should also be provided. These can be easily implemented by making new field plugins for them.

8.3.12 Leveraging on future versions of Spring, JPA and Hibernate

Features such as Hibernates detached Criteria objects are very popular, but to the candidate's knowledge there is no official plan for this functionality in a new JPA specification. If the past JPA specifications are any indication, such as the Hibernate inspired addition of queries with Criteria objects, the popular features from other ORMs are high priorities. A specification request for JPA 2.1 is also looking to add support for translation between JPA Criteria objects and JPQL [30], and general improvements to both query languages.

Currently, JPA entities are loaded at the startup of the application. Plugins are loaded generically at runtime, so this should be done for plugin entities as well. Spring has support for dynamic scanning of packages for entities using Hibernate, but not JPA. This feature will be provided for JPA in Spring version 3.1 [100].

Spring 3.1 and Hibernate 4.0 are both in release candidate versions, which means their releases are right around the corner. They should definitely be looked at when released, to further develop the persistence layer of DPG.

8.3.13 Improving the persistence solution in DPG

The current solution for persistence in DPG was developed in DPG 2.0. The solution intented for use in practice was JackRabbit. The idea was to provide DPG with some much needed transaction support, and more functions such as versioning and caching of data. Unfortunately, this was abandoned in practical use, and in the transition to DPG 2.1, because the technology was immature, poorly documented, and proved to have some performance issues. Currently, a simple File System solution for persistence is used; a solution which was only intended for development and debugging, not for practical use.

DPG currently parses and fetches content from entire XML documents. There is no real solution for saving or reading specific data directly from a database or content repository. Examples of this are: If DPG is looking for a specific entity in a pattern, it will still load the entire `pattern.xml` configuration file, and if DPG needs to present specific content in a view, it will still load the entire entity-instance (which is currently an xml document containing the content). The solution doesn't fully use the benefits of the persistence implementation, which further degrades performance.

The lack of a proper persistence implementation is becoming one of DPG's biggest problems. The Hibernate/JPA implementation of plugin resources, presented in chapter 7, can be further extended to the rest of DPG. This should be relatively easy, as most of the evaluation and ground work is already done. DPG loads patterns

and presentations, as well as all their corresponding components, into Java objects when they are to be used. These objects can easily be represented as JPA entities and persisted/fetched directly. A new JPA persistence unit should be configured for DPG content, because the currently used persistence unit is configured specifically to handle plugin resources.

8.4 Conclusion

This Master thesis presents the evaluations and work done for improving the plugin architecture of DPG for better support of data collection and complex plugins. This greatly increases the potential applications for DPG in Web 2.0 and onwards.

It has been a very rewarding experience for the candidate, as this Master thesis required collaboration with other developers on a complex system which uses many of the popular frameworks and technologies of the industry today. It has been challenging to understand and work on such a large system, but the candidate has now learned which methods work for him. This includes methods such as making UML [47] diagrams, and debugging using the standard logging using Log4J [33] in DPG, to better understand the system. This, along with the experience of using the development methodologies and technologies, is likely to be very relevant for working in the software industry.

Since the candidate and fellow Master student Aleksander Waage had a common goal, a direct collaboration on parts of the thesis felt very natural. This has been a rewarding experience, teaching about practices such as pair programming. This thesis has been part of the JAFU project, which usually has approximately four active Master students at any time. Working in the same office on the same project means that the students have helped each other and collaborated on parts of the work. This has been a very good experience for the candidate, and made the development of good implementations extra motivating. Another motivation boost has been the fact that DPG is being used for distant learning at JAFU. The candidate has first hand experience with this from his work as a teaching assistant in INF-101F, managing the course web pages. This also helped to really see both the strengths and weaknesses of DPG 2.1 in a real world application.

The resulting implementations in this thesis have come from constantly thinking about making the solutions more general, and easier to use. A focus throughout the thesis has been to split the responsibilities of plugin developers, DPG developers, pattern developers and users of DPG, while making their work easier.

Plugin resources and persistent data are no longer an afterthought in DPG. Plugins

using the new solutions, provided from this thesis, will make applications of DPG much more varied and powerful. Plugins can now dynamically render presentations from multiple fields of content, and persist large amounts of user data in an intuitive, fast, secure and structured way.

The work on this large project has been a very positive experience. The candidate has never worked on such a large system, or written such a large report. The freedom to explore technologies and methodologies, and plan and execute such a large project are definitely good experiences to have.

Bibliography

[1] Drupal. http://drupal.org/. Accessed 2011.09.20.

[2] Envers. http://www.jboss.org/envers. Accessed 2011.11.04.

[3] Hibernate. http://www.hibernate.org/. Accessed 2011.10.29.

[4] Hippo CMS. http://www.onehippo.com/. Accessed 2011.10.29.

[5] Jdom. http://www.jdom.org/. Accessed 2011.10.20.

[6] Joomla! CMS. http://www.joomla.org/. Accessed 2011.09.20.

[7] JUnit. http://www.junit.org/. Accessed 2011.10.29.

[8] m2eclipse. http://eclipse.org/m2e/. Accessed 2011.10.29.

[9] Magnolia. http://www.magnolia-cms.com/. Accessed 2011.10.29.

[10] Moodle. http://moodle.org/. Accessed 2011.09.20.

[11] MyBatis. http://mybatis.org/. Accessed 2011.10.26.

[12] Nuxeo. http://www.nuxeo.com/en. Accessed 2011.10.29.

[13] PostgreSQL. http://www.postgresql.org/. Accessed 2011.10.29.

[14] Wordpress. http://wordpress.org/. Accessed 2011.09.20.

[15] Mert Can Akkan. JPA Criteria API by samples. http://www.altuure.com/2010/09/23/jpa-criteria-api-by-samples-part-i/. Accessed 2011.10.29.

[16] Alfresco. Alfresco Community Edition. http://www.alfresco.com/community/. Accessed 2011.07.10.

[17] Karianne Berg. Persistensproblematikk i Dynamic Presentation Generator. Master's thesis, Department of Informatics, University of Bergen, 2008.

[18] Jostein Bjørge. PPDev: Et nettbasert verktøy for utvikling og validering av presentasjonsmønstre i Dynamic Presentation Generator. Master's thesis, Department of Informatics, University of Bergen, 2010.

[19] Bert Bos. Cascading Style Sheets. http://www.w3.org/Style/CSS/. Accessed 2011.11.11.

[20] Mountainminds GmbH Co. EclEmma.
http://www.eclemma.org/. Accessed 2011.10.29.

[21] Stephen Connolly. How em.merge actually works.
http://javaadventure.blogspot.com/2006/06/how-emmerge-actually-works.html. Accessed 2011.10.29.

[22] Mort Bay Consulting. Jetty.
http://jetty.codehaus.org/jetty/. Accessed 2011.10.29.

[23] Kevin Cruickshanks. Verktøy for generering av XML-baserte presentasjonar: JPGen - Java presentasjons generator. Master's thesis, Department of Informatics, University of Bergen, 2004.

[24] Day. JCR v2.0 Specification: Query. http://www.day.com/specs/jcr/2.0/6_Query.html, 2009. Accessed 2011.11.11.

[25] Mark Doliner. Cobertura.
http://cobertura.sourceforge.net/. Accessed 2011.11.17.

[26] Drupal. Updating your modules.
http://drupal.org/update/modules. Accessed 2011.11.04.

[27] Drupal. Working with content types and fields (Drupal 7). http://drupal.org/documentation/modules/field-ui. Accessed 2011.10.29.

[28] Yngve Espelid. Dynamic Presentation Generator. Master's thesis, Department of Informatics, University of Bergen, 2004.

[29] David Nuescheler et al. JSR 283: Content Repository for Java Technology API 2.0 specification. *Java Specification Request*, 2009. Accessed 2011.10.29.

[30] Linda Demichiel et al. JSR 338: Java Persistence 2.1. *Java Specification Request*, 2011. Accessed 2011.11.04.

[31] eXist. eXist-db Open Source Native XML Database.
http://exist.sourceforge.net/. Accessed 2011.07.10.

[32] The Apache Software Foundation. JackRabbit.
http://jackrabbit.apache.org/. Accessed 2011.09.28.

[33] The Apache Software Foundation. log4j.
http://logging.apache.org/log4j/. Accessed 2011.11.11.

[34] The Apache Software Foundation. OpenJPA. http://openjpa.apache.org/. Accessed 2011.11.15.

[35] The Apache Software Foundation. OpenJPA.
http://openjpa.apache.org/. Accessed 2011.10.29.

[36] The Apache Software Foundation. Subversion.
http://subversion.apache.org/. Accessed 2011.10.29.

[37] The Apache Software Foundation. The Apache Velocity Project. http://velocity.apache.org. Accessed 2011.09.20.

[38] The Apache Software Foundation. Tomcat. http://tomcat.apache.org/. Accessed 2011.10.29.

[39] The Eclipse Foundation. Eclipse IDE. http://www.eclipse.org/. Accessed 2011.10.29.

[40] The Eclipse Foundation. EclipseLink. http://www.eclipse.org/eclipselink/. Accessed 2011.11.15.

[41] Martin Fowler. InversionOfControl. http://martinfowler.com/bliki/InversionOfControl.html. Accessed 2011.10.29.

[42] Martin Fowler. *Patterns of Enterprise Application Architecture*. Pearson Education Inc., 2003.

[43] Christian Bauer et al. Gavin King. Hibernate 3.6.3 reference documentation. http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/. Accessed 2011.10.29.

[44] Google. Chrome. http://www.google.com/chrome. Accessed 2011.10.29.

[45] Google. Google Maps API. http://code.google.com/intl/no-NO/apis/maps/index.html. Accessed 2011.10.29.

[46] Google. Online drawings in Google Docs. http://www.google.com/google-d-s/drawings/. Accessed 2011.10.29.

[47] Object Management Group. Unified Modeling Language (UML). http://www.omg.org/spec/UML/. Accessed 2011.11.17.

[48] Hibernate. Introduction to the Spring IoC container and beans. http://static.springsource.org/spring/docs/3.0.5.RELEASE/reference/beans.html. Accessed 2011.10.29.

[49] Hibernate. Supported databases. http://community.jboss.org/wiki/SupportedDatabases2. Accessed 2011.10.29.

[50] Hibernate. Transactions and Concurrency. http://docs.jboss.org/hibernate/entitymanager/3.6/reference/en/html/transactions.html. Accessed 2011.10.29.

[51] Hibernate. Working with objects. http://docs.jboss.org/hibernate/entitymanager/3.6/reference/en/html/objectstate.html. Accessed 2011.10.29.

[52] Morten Høiland. Datainnsamling med XForms i Dynamic Presentation Generator. Master's thesis, Department of Informatics, University of Bergen, 2010.

[53] Juergen Hoeller. HibernateTemplate API. http://static.springsource.

org/spring/docs/3.0.5.RELEASE/api/org/springframework/orm/hibernate3/HibernateTemplate.html. Accessed 2011.10.29.

[54] Juergen Hoeller. JpaTemplate API. http://static.springsource.org/spring/docs/3.0.5.RELEASE/api/org/springframework/orm/jpa/JpaTemplate.html. Accessed 2011.10.29.

[55] The hsql Development Group. HSQLDB - 100 http://hsqldb.org/. Accessed 2011.11.17.

[56] Adobe Systems Inc. Day communique. http://www.day.com/. Accessed 2011.10.29.

[57] Sun Microsystems Inc. Core J2EE Patterns - Data Access Objects. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html, 2007. Accessed 2011.10.20.

[58] Terracotta Inc. EhCache. http://ehcache.org/. Accessed 2011.10.29.

[59] Unicode Inc. About the unicode standard. http://unicode.org/standard/standard.html, 2011. Accessed 2011.09.15.

[60] Bjørn Ove Ingvaldsen. Multimedia i dynamisk presentasjons generator 2.0. Master's thesis, Department of Informatics, University of Bergen, 2008.

[61] Joomla. Basic hello world module. http://docs.joomla.org/Creating_a_simple_module. Accessed 2011.09.20.

[62] David Nuescheler et al. JSR 170 expert group. JSR 170: Content Repository API for Java Technology Specification. *Java Specification Request*, 2005. Accessed 2011.10.29.

[63] Aleksander Vatle Waage Kelly Alexander Teigland Whiteley. INF219 - Persistenstesting i DPG 2.0. *INF219 project report*, Department of Informatics, University of Bergen, 2010.

[64] Aleksander Vatle Waage Kelly Alexander Teigland Whiteley. Project Report - MOD250. *MOD250 project report*, Department of Informatics, University of Bergen, 2010.

[65] Kristian Skønberg Løvik. Webucator 3.0 - Brukerhåndtering og aksesskontroll for DPG 2.0. Master's thesis, Universitet i Bergen, 2008.

[66] Robert C. Martin. The Open-Closed Principle. *C++ Report*, 1996. Accessed 2011.11.06.

[67] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2002.

[68] Robert C. Martin. *Clean Code - A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2009.

[69] Carol McDonald. JPA 2.0 Concurrency and locking. http://blogs.oracle.com/carolmcdonald/entry/jpa_2_0_concurrency_and. Accessed 2011.10.29.

[70] Carol McDonald. JPA Caching. http://weblogs.java.net/blog/archive/2009/08/21/jpa-caching. Accessed 2011.10.29.

[71] Microsoft. Internet Explorer. http://windows.microsoft.com/en-US/internet-explorer/products/ie/home. Accessed 2011.10.29.

[72] Microsoft. Visio. http://office.microsoft.com/en-us/visio/. Accessed 2011.10.29.

[73] Moodle. Installing and upgrading plugin database tables. http://docs.moodle.org/dev/Installing_and_upgrading_plugin_database_tables. Accessed 2011.11.05.

[74] Moodle. Moodle architecture. http://docs.moodle.org/dev/Moodle_architecture#The_Moodle_database. Accessed 2011.09.20.

[75] Khalid A. Mughal. Presentation Patterns: Composing Web-based Presentations. Technical report, Department of Informatics, University of Bergen, 2003.

[76] ObjectDB. JPA 2 Annotations. http://www.objectdb.com/api/java/jpa/annotations. Accessed 2011.11.04.

[77] ObjectDB. JPA Queries (JPQL / Criteria). http://www.objectdb.com/java/jpa/query. Accessed 2011.11.04.

[78] Tobias Rusås Olsen. Interaksjon og søk i Dynamic Presentation Generator. Master's thesis, Department of Informatics, University of Bergen, 2010.

[79] openXdata. openXdata - Documentation. http://doc.openxdata.org/, 2011. Accessed 2011.10.29.

[80] Oracle. Data Concurrency and Consistency. http://download.oracle.com/docs/cd/B14117_01/server.101/b10743/consist.htm. Accessed 2011.10.26.

[81] Oracle. Interface Parameter<T>. http://download.oracle.com/javaee/6/api/javax/persistence/Parameter.html. Accessed 2011.10.29.

[82] Oracle. Java Naming and Directory Interface (JNDI). http://java.sun.com/javase/technologies/core/jndi/index.jsp. Accessed 2011.10.29.

[83] Oracle. Java SE Technologies - Database. http://www.oracle.com/technetwork/java/javase/jdbc/index.html. Accessed 2011.10.26.

[84] Oracle. JPQL Language Reference. http://download.oracle.com/docs/cd/E16764_01/apirefs.1111/e13046/ejb3_langref.html. Ac-

cessed 2011.10.29.

[85] Oracle. Using Prepared Statements. http://download.oracle.com/javase/tutorial/jdbc/basics/prepared.html. Accessed 2011.09.25.

[86] Tim O'Reilly. What is Web 2.0. http://oreilly.com/web2/archive/what-is-web-20.html, 2005. Accessed 2011.11.11.

[87] OWASP. SQL Injection. https://www.owasp.org/index.php/SQL_Injection. Accessed 2011.09.25.

[88] Vincent Partington. JPA implementation patterns: Saving (detached) entities. http://blog.xebia.com/2009/03/23/jpa-implementation-patterns-saving-detached-entities/. Accessed 2011.10.29.

[89] Pinaki Poddar. Dynamic, typesafe queries in JPA 2.0. http://www.ibm.com/developerworks/java/library/j-typesafejpa/. Accessed 2011.10.29.

[90] Seema Richard. Annotation based configuration in Spring. http://weblogs.java.net/blog/seemarich/archive/2007/11/annotation_base.html, 2007. Accessed 2011.10.29.

[91] Øystein Lund Rolland. Integrasjon av Orbeon Forms Designer i Dynamic Presenation Generator. Master's thesis, Department of Informatics, University of Bergen, 2010.

[92] Mohamed Sanualla. CamelCase Notation- Naming Convention for Programming Languages. http://blog.sanaulla.info/2008/06/25/camelcase-notation-naming-convention-for-programming-languages/. Accessed 2011.11.04.

[93] Bjørn Christian Sebak. Dynamic Presentation Generator 2.0 – Utvikling av ny dynamisk presentasjonsgenerator og presentasjonsmønsterspesifikasjon. Master's thesis, Department of Informatics, University of Bergen, 2008.

[94] Peder Lång Skeidsvoll. Støtte for rike klienter i DPG. Master's thesis, Department of Informatics, University of Bergen, 2010.

[95] Edgewall Software. Trac. http://trac.edgewall.org/. Accessed 2011.11.17.

[96] ObjectDB Software. Obtaining a JPA Database Connection. http://www.objectdb.com/java/jpa/start/connection. Accessed 2011.10.29.

[97] Opera Software. Opera. http://www.opera.com/. Accessed 2011.10.29.

[98] Apache software foundation. JDOQL. http://db.apache.org/jdo/jdoql.html. Accessed 2011.10.29.

[99] Spring source. Object Relational Mapping (ORM) Data Ac-

cess. http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/orm.html. Accessed 2011.10.29.

[100] Spring Source. Spring Framework 3.1 Reference Documentation. http://static.springsource.org/spring/docs/3.1.0.M1/spring-framework-reference/html/. Accessed 2011.11.04.

[101] Spring source. Testing. http://static.springsource.org/spring/docs/3.0.5.RELEASE/reference/testing.html#testcontext-framework. Accessed 2011.10.29.

[102] Spring source. Transaction management. http://static.springsource.org/spring/docs/3.0.5.RELEASE/reference/transaction.html. Accessed 2011.10.29.

[103] Spring. Spring - Web MVC framework. http://static.springsource.org/spring/docs/3.0.5.RELEASE/reference/mvc.html. Accessed 2011.11.11.

[104] Spring. Spring Framework. http://www.springsource.org/. Accessed 2011.09.20.

[105] Spring. Spring Security. http://static.springsource.org/spring-security/site/. Accessed 2011.11.11.

[106] EJB 3.0 Expert Group Sun Microsystems. JSR 220: Enterprise JavaBeans, version 3.0, Java Persistence API. *Java Specification Request*, 2006. Accessed 2011.10.10.

[107] Java Persistence 2.0 Expert Group Sun Microsystems. JSR 317: Java Persistence API, version 2.0. *Java Specification Request*, 2009. Accessed 2011.10.10.

[108] Versant. db4objects. http://www.db4o.com/. Accessed 2011.07.10.

[109] Vital Wave Consulting. mHealth for Development: The Opportunity of Mobile Technology for Healthcare in the Developing World. Technical report, UN Foundation-Vodafone Foundation Partnership, 2009.

[110] W3C. XML Path Language (XPath). http://www.w3.org/TR/xpath/, 1999. Accessed 2011.10.29.

[111] W3C. Extensible Markup Language (XML) 1.1. http://www.w3.org/TR/2006/REC-xml11-20060816/, 2006. Accessed 2011.10.29.

[112] W3C. XForms 1.1. http://www.w3.org/TR/xforms/, 2009. Accessed 2011.10.29.

[113] World Wide Web Consortium (W3C). XSL Transformations (XSLT) - Version 1.0. http://www.w3.org/TR/xslt, 1999. Accessed 2011.09.20.

[114] Aleksander Vatle Waage. Støtte for Geodata i Dynamic Presentation Generator. Master's thesis, Department of Informatics, University of Bergen, in preparation.

[115] Jim White. Sizing up open source java persistence. http://www.devx.com/java/Article/33768/0/page/4, 2007. Accessed 2011.10.29.

[116] Wikibooks. Java Persistence/Querying. http://en.wikibooks.org/wiki/Java_Persistence/Querying. Accessed 2011.10.29.

[117] WikiBooks. Java Persistence/What is new in JPA 2.0? http://en.wikibooks.org/wiki/Java_Persistence/What_is_new_in_JPA_2.0%3F. Accessed 2011.11.05.

[118] Wikipedia. Create, read, update and delete. http://en.wikipedia.org/wiki/Create,_read,_update_and_delete. Accessed 2011.09.25.

[119] Wikipedia. JavaScript. http://en.wikipedia.org/wiki/JavaScript. Accessed 2011.11.18.

[120] Wikipedia. Separation of concerns. http://en.wikipedia.org/wiki/Separation_of_concerns. Accessed 2011.11.18.

[121] Wikipedia. SQL. http://en.wikipedia.org/wiki/SQL. Accessed 2011.10.29.

[122] Wikipedia. WYSIWYG. http://no.wikipedia.org/wiki/WYSIWYG, 2011. Accessed 2011.10.20.

[123] Bobby Woolf. ACID Transactions. https://www.ibm.com/developerworks/mydeveloperworks/blogs/woolf/entry/acid_transactions?lang=en, 2011. Accessed 2011.10.11.

[124] WordPress. Plugin API. http://codex.wordpress.org/Plugin_API. Accessed 2011.09.20.

[125] Reinier Zwitserloot and Roel Spilker. Project Lombok. http://projectlombok.org/. Accessed 2011.11.04.

A

Guidelines for using the new plugin resource interface

This chapter presents some guidelines for the specific use of the plugin resource solution of DPG using JPA. It is very important that plugin developers read through this tutorial. The provided interface for plugin resources, `PluginResourceJpaDao`, is presented in listing 7.3. The relevant discussions and implementation of the `PluginResourceJpaDao` interface are presented in chapter 7.

A tutorial and overview of how the entity annotations and more are used, can be found at the ObjectDB documentation [76]. It is highly recommended to go through these tutorials to familiarize yourself with simple entity structures and setup. The Hibernate documentation [43] and JPA specification [107] can also be used.

A.1 Entities

JPA Entities are defined with the `@Entity` annotation on the class declaration. It is important that the `javax.persistence.Entity` is imported and used; *not* the Hibernate specific version. The names of entities, tables and columns can be explicitly defined by the name parameter. It is *important* that the names of the entities and tables contain a "`pluginName_`" prefix, for example

"`pollPlugin_TableOfUsers`". Entity names are implicitly derived from the class name, and table names are derived from the entity names, so if the entity name is not defined explicitly, the class name must use a prefix of `Pluginname`, like for example `PollPluginPollEntity`, using *UpperCamelCase* [92] instead of underscores. More about this can be read in section 7.9.

Entities can contain many types of simple fields, as well as references to other entities. Every entity must have a field marked with the `@Id` annotation, which specifies the primary key of the entity. It is highly recommended to let the id be automatically generated using the `@GeneratedValue` annotation, as shown at line 8 in listing A.1. Fields can be marked as `@Transient` if they should not be persisted. By default, objects in collections are fetched lazily, which will work fine since no entities are detached from the persistence context.

References to other entities must be mapped with the proper annotations for their relationship.

All persistent fields in an entity must have corresponding setter and getter methods. DPG uses Lombok [125], a tool which can be used to automatically generated setters and getters for the entire class, or specific fields. To do this, mark the class or field with the Lombok `@Setter` and `@Getter` annotations, as shown in line 4 in listing A.1. This removes boilerplate code and makes the entity classes very clean and easy to read.

If the plugin requires *optimistic locking* of an entity (as explained in section 7.7), the entity needs to define a field annotated with the `@Version` annotation, as shown in line 28 in listing A.1. The `PluginResourceJpaDao` bean will then enforce optimistic locking of the entity automatically.

Listing A.1: Two entities with bidirectional references

```
1
2 //An abstract entity superclass containing an automatically generated
      id
3 @MappedSuperclass
4 @Getter @Setter
5 public class AbstractJpaEntity {
6
7   @Id
8   @GeneratedValue(strategy = GenerationType.IDENTITY)
9   int id;
10
11 }
12
13 //An example entity class
14 @Entity
15 @Setter
16 @Getter
17 public class TestPluginTestEntity extends AbstractJpaEntity {
18
19
20   String name;
21
22   @OneToMany(mappedBy="referencingEntity",
23           cascade = CascadeType.PERSIST)
24   List<TestSubEntity> listOfReferencedEntities;
25
26   @Transient
27   String aTransientValue;
28
29   @Version
30   int version;
31
32
33 }
34
35 //The referenced example entity class
36
37 @Entity(name = "testPlugin_subEntity")
38 @Getter @Setter
39 public class TestSubEntity extends AbstractJpaEntity{
40
41   String name;
42
43   @ManyToOne
44   TestEntity referencingEntity;
45
46 }
```

A.2 Queries

The `PluginResourceJpaDao` interface provides support for both JPQL queries and JPA Criteria queries. It does not support saving, updating or deleting data through the queries. These operations should be done through the `saveOrUpdate()` and `remove()` methods.

A Query with JPQL is very similar to SQL, but working with objects and fields instead of tables, rows and columns. The JPQL string queries are very easy to read and understand, and should be used for simple, static queries. Due to the possibility of query injection attacks, it should not be used directly with inlined user controlled data, such as the example in listing A.2. These queries can be executed using the `findByQuery()` method of the `PluginResourceJpaDao` interface.

Listing A.2: A username can include a malicious JPQL code to gain access to additional information

```
1
2   String sql = "SELECT u FROM " + user + " u ";
```

It is recommended to use Criteria API for queries as much as possible, especially for dynamic and secure queries. These queries are also type-safe, while JPQL is not, and any errors will be recognized at compile time. Listing A.3 shows how a builder for Criteria query objects is provided by the `getCriteriaBuilder()` method of the `PluginResourceJpaDao` interface, and the query can be executed using the `findByCriteriaQuery()` method.

Listing A.3: A Criteria query executed using PluginResourceJpaDao

```
1
2   //Get the criteria builder
3   CriteriaBuilder builder = pluginResourceJpaDao.getCriteriaBuilder();
4
5   //Build the query
6   CriteriaQuery<TestPluginTestEntity> criteria =
7   builder.createQuery(TestPluginTestEntity.class);
8   Root<TestPluginTestEntity> testEntityRoot = criteria.from(
        TestPluginTestEntity.class);
9   criteria = criteria.select(testEntityRoot);
10  criteria.where(builder.equal(testEntityRoot.get("name"), "Kelly"));
11
12  //Execute query and get the list of results
13  List<TestPluginTestEntity> listOfEntities = pluginResourceJpaDao.
        findByCriteriaQuery(criteria);
```

More about how these queries can be structured can be found in the ObjectDB JPA query documentation [77], the Oracle JPQL language reference [84], Criteria tutorials

by Mert Can Akkan [15] and IBM [89], as well as the JPA specification [107].

The query implementation of the `PluginResourceJpaDao` interface is presented in section 7.5.

A.3 Saving, updating and removing entities

If the id of an entity is generated manually, the plugin needs to make sure that it does not make a new entity with the same id as a currently persisted one, because the `saveOrUpdate()` method does not merge these entities. The plugin should instead retrieve the entity object through the `PluginResourceJpaDao` bean, and update its values before calling the `saveOrUpdate()` method.

The basic operations on an entity, which are used by the DAO implementation are the JPA `persist()` and `remove()` operations. These operations can be configured to cascade to referenced entities as well. This configuration must be thought through, as lingering references can be set to `null`, if an operation was cascaded from a referencing entity. This is further explained in section 7.6.

A.4 Structuring resources

Plugins get access to the current page, view, pattern, presentation and username through the provided `FieldPluginBean` bean. This can be used by plugins to structure their data if needed.