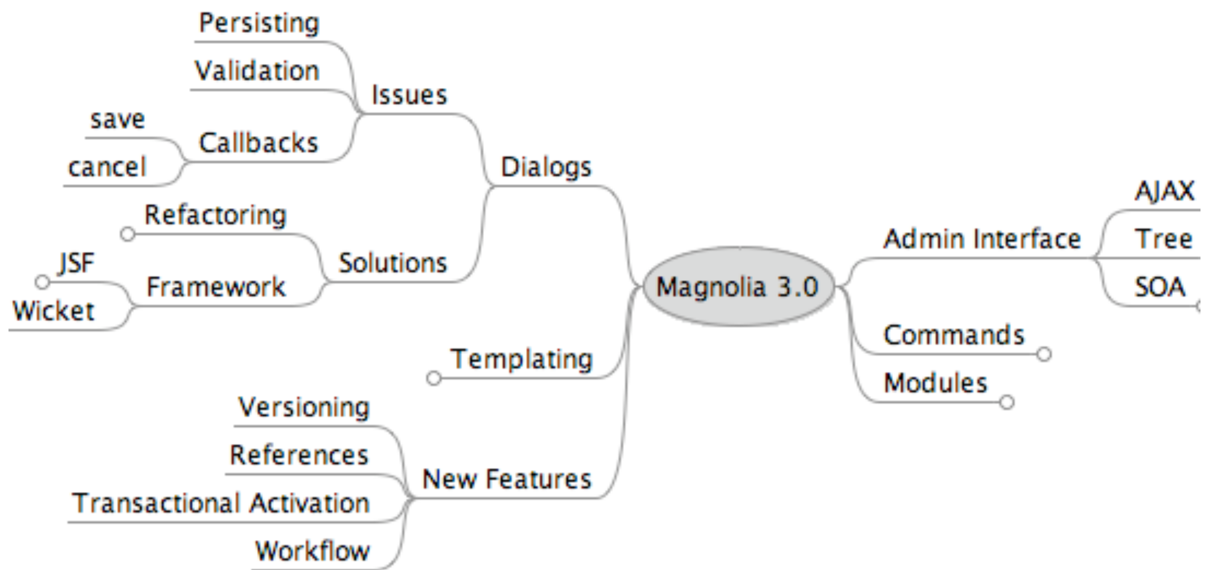
Magnolia 3.0

Author: Philipp Bracher

Date: 31. Oct. 2005

1. Overview 2
1.1. Goals 2
1.2. Tasks 3
2. Admin Interface 4
2.1. AJAX 4
2.2. SOA 5
2.3. Tree 5
2.4. Creating new content 5
2.5. Menu 6
2.6. Templating 6
3. Commands 7
4. Dialogs 9
4.1. Issues 9
4.2. Prerequests to the framework 9
4.3. Option 1: Refactoring with Velocity 10
4.4. Option 2: Wicket 10
4.5. Option 3: JSF with Facelets 10
4.6. AJAX in the context of Dialogs 10
5. Modules 12
6. New Features 13
6.1. Versioning 13
6.2. References 13
6.3. Improved Activation 13
6.4. Workflow 13
7. Templating 14
8. Process 15
8.1. Team 15
8.2. Voting 15

1. Overview



1.1. Goals

- Enterprise friendly architecture
- Customizable interface
- Easy module development
- Support for enterprice features: Versioning, Workflow

Doing this we must keep in mind the magnolia philiosphy:

- Simplicity
- Based on Standards

This means we will provide an easy to handle user interfaces for the end users and easy to understand API for the developers. The implementation must be so, that one can customize it for more complex use cases, where the standard implementation is following the principle of simplicity.

Simplicity: the degree to which a system or component has a design and implementation that is straightforward and easy to understand

Occam's Razor: “plurality should not be posited without necessity”. In other words: Given two equally predictive theories, choose the simpler.

Standard: a quality or measure which is established by authority, custom, or general consent

1.2. Tasks

If we separate the tasks in refactoring and new features, we have the following main tasks to do.

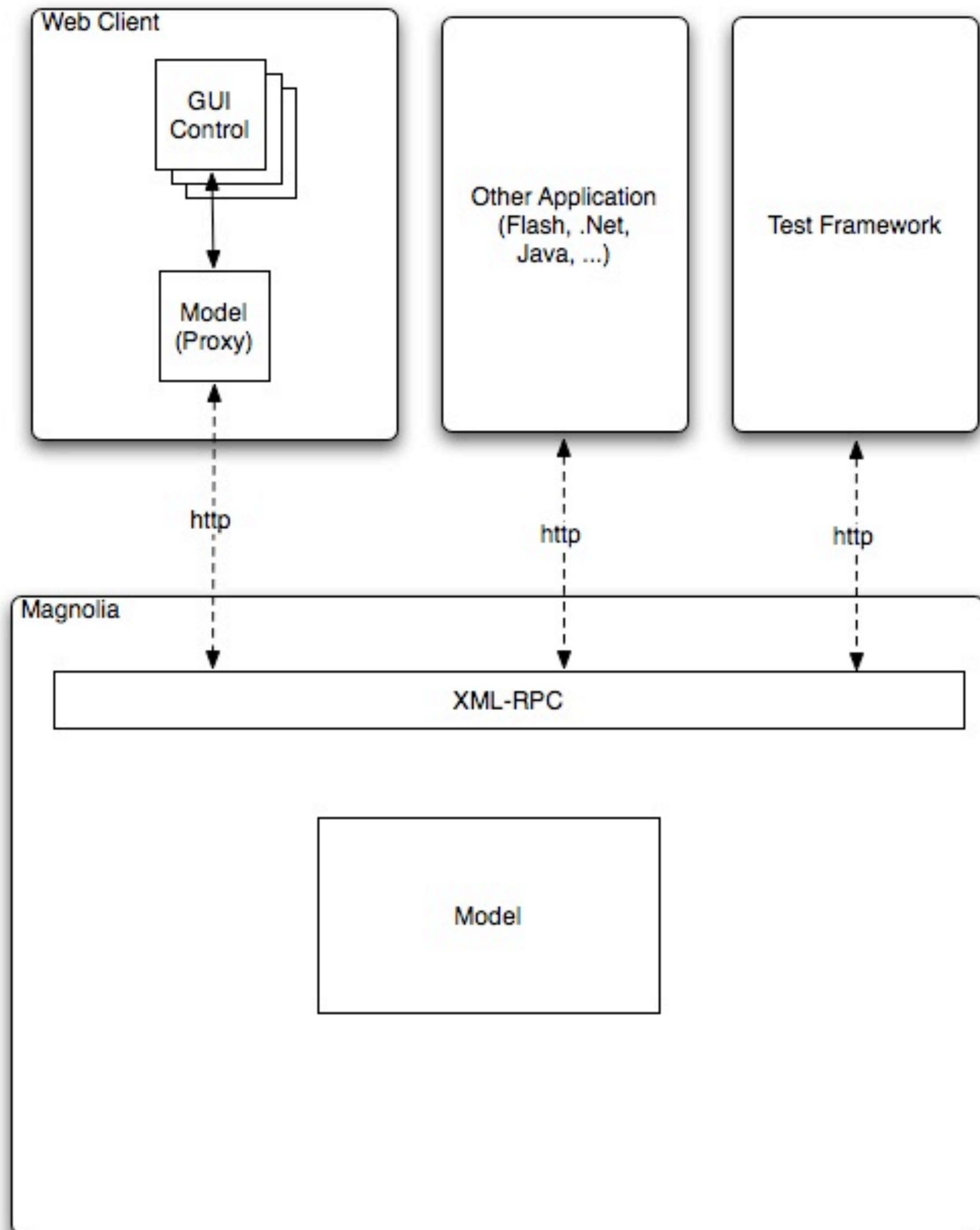
1.2.1. Refactoring:

Dialogs:	Validation, customize saving, framework
Admin Interface:	Use a standard AJAX implementation
Commands:	The commands / actions should be easily replaceable.

1.2.2. New Features

Versioning:	A new version is created if one activates content
References:	Keep references after activation. E.g. links should not brake after moving a page.
Activation:	Transactional, incremental
Workflow:	Integrate OpenWFE

2. Admin Interface



2.1. AJAX

We do already use AJAX, but we don't use a framework yet. The actual prefered solution is: use XML-RPC and Prototype.js on the browser side.

Here are some arguments

- XML-RPC is not only AJAX (web related)
- prototype.js is one js file providing everything you need
- DWR and JSON-RPC are too much Javascript related

2.1.1. Option 1: DWR

Creates javascribt stubs. Easy to use in GUI development.

http://getahead.ltd.uk/dwr/

2.1.2. Option 2: XML-RPC (using prototype.js)

XML-RPC is a well known standard. There exists support for a lot of other languages like Java, .Net, Flash. This is means that such an implementation is usable in a SOA Architecture.

For making the AJAX calls we will use prototype.js , which is a javascript library used by other frameworks (Dojo for example).

http://prototype.conio.net/

http://www.sergiopereira.com/articles/prototype.js.html

2.2. SOA

One of the benefit of a pure AJAX implementation is, that other Application can use the same services to get data from Magnolia and manipulate it.

The services (commands/actions) should be plugable. This means that e.g. one can replace the activation command with a workflow implementation.

2.3. Tree

The main work for a new admin interface will be refactoring the tree.

- no rendering at the server side
- client uses a tree control
- the tree control uses a client side model (proxy)
- the model gets its data with AJAX
- the tree control has now idea about the content of the tree. It is the model which decides which context menu should get displayed and how the chaged data should get saved.

2.4. Creating new content

If a user creates a new page or user it will open a coresponding dialog. The node is only created after a successful save. This will avoid the untiled notes.

2.5. Menu

A least a two level navigation. A graphical design already exists.

2.6. Templating

The templating itself will not change. The creation and moving of paragraphs in the admin interface will use AJAX to make sure that all functions are available for the SOA approach.

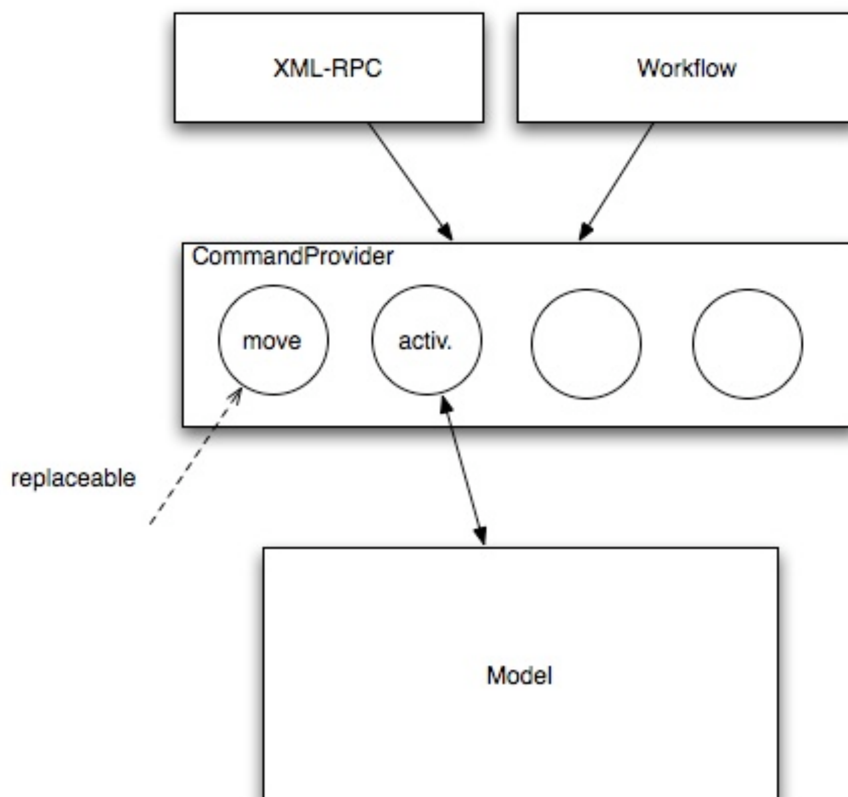
3. Commands

A command is a atom action like move, rename or activate. Those commands are provided due XML-RPC as services to other applications. The solution described below is yet a rough concept and must get worked out in details.

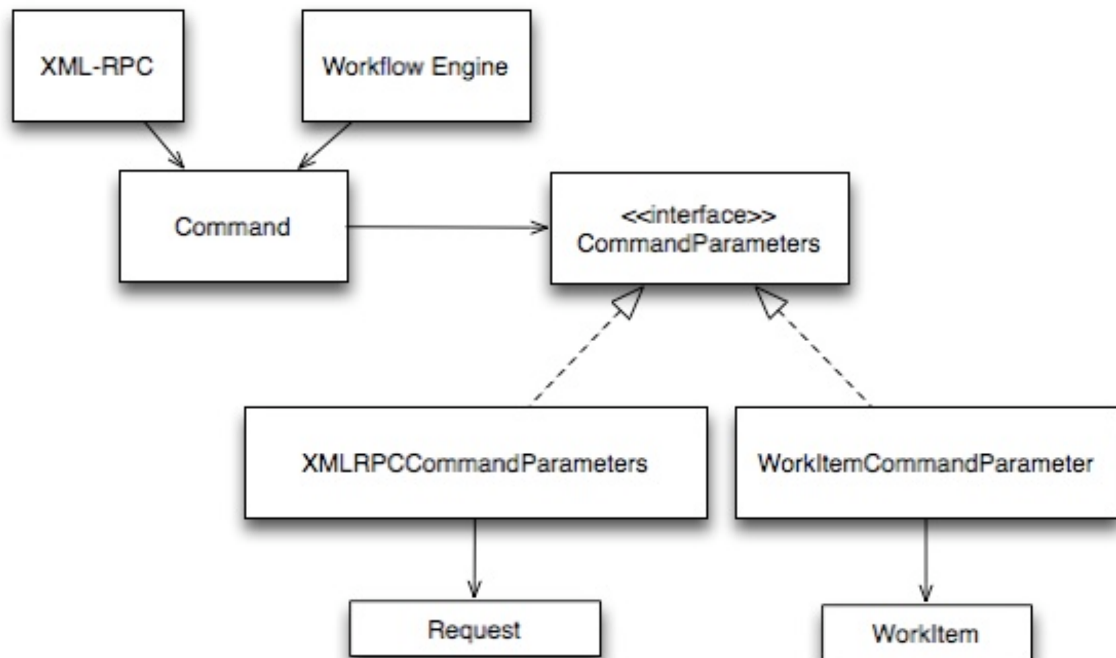
Requirement for the architecture:

- IoC (Inversion of Control) http://excalibur.apache.org/framework/guide-patterns-ioc.html
- Configured in the admin central
- Plugable / replaceable
- Useable in workflow
- Each tree (repository) has its own definition

The following graphics illustrates this. In this scenario we would use XML-RPC.



Only the commands will access the model directly and not the RPC-Layer or Workflow Engine.



XML-RPC and Workflow are both calling a command passing the parameters with a CommandParameters wrapper object. As a principle the command does not know anything about the the request or workitem.

4. Dialogs

Here we have mainly two possibilities. We can refactor what we have or we can use a standardized framework. If we like to use a framework we have to make a choice between: JSF with Facelets and Wicket. There are a lot of other well known and good frameworks not part of the final selection. But we must reduce the discussion to get a conclusion and these two were the most promising candidates.

4.1. Issues

Here is what we must add to the dialogs:

- Persisting: flexible customization of the persisting process. This means that one can change the 1:1 saving. For example it should be possible to store the data in a common DB.
- Validation
- Callbacks (one defines the javascript to be called after a successful saving, cancel, error). In this script one will reload the tree, alert messages, ...
- Programmatically access to the dialog/controls. This means access to the definition and the current objects.

We do not:

- We do not change the dialog definition. The dialogs are still defined in the configuration.

Optional things:

- Wizard
- Switchable Panels. This are panels used to switch detail informations based on an other user operation e. g. selecting a radio button.
- Defining new dialogs extending a existing configuration. Currently there are a lot of similar dialogs.

4.2. Prerequests to the framework

- Controls are template based
- The controls should be usable in common JSP too. The controls used in the templating (main bar for example) should not differ too much from the controls used in the dialogs.
- Dialogs can be build dynamically (based on the dialog definition in the repository and not based on tags)
- The bean to which the data is sent back is dynamic (we do not create a bean class for each dialog definition)

The current favourite option is option 3: JSF with Facelets

4.3. Option 1: Refactoring with Velocity

We would do:

- move the Save control to the admin interface package
- save delegates (configurable in the dialog definition)
- apache commons validators
- rename the super classes and build a better hierarchy
- Using a template engine to keep the html code out of the classes. This would be Velocity.

4.4. Option 2: Wicket

- simple
- not well documented

4.5. Option 3: JSF with Facelets

There is definitely a bigger amount of work to do if we choose JSF. But this is a supported standard used in many companies today. The usage of Facelets would eliminate the disadvantages of JSF.

The most attributes are

- Standard
- Controls
- Beans for saving
- Flexibel but complicated (Lifecycle)
- Documented

We will use Facelets as the ViewHandler. This replaces the JSP part and looks really slim and nice.

4.6. AJAX in the context of Dialogs

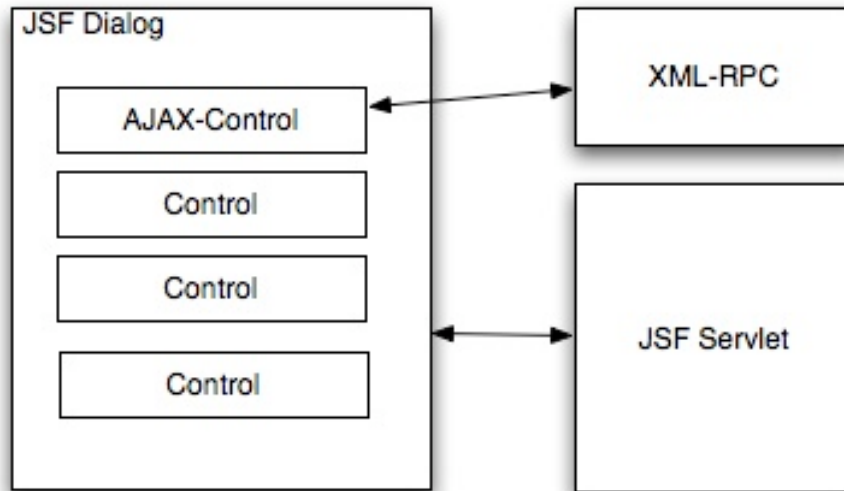
Unfortunately the most frameworks do not very well integrate with an AJAX solution. Still there are ways to solve the problem.

4.6.1. Wicket

There is an implementaion for Dojo. But we can use the same solution proposed for JSF. It would definitly be more consistent to use XML-RPC here too.

4.6.2. JSF

If we use JSF we must find a solution how to build AJAX components. Here we propose to use a separte controller for the AJAX aspects of the controls. In our case this would be XML-RPC. This controll can still access parts of the JSF framework. You will find a detailed explanation reding the strategy 2 on https://bpcatalog.dev.java.net/ajax/jsf-ajax


JSF Dialog
AJAX-Control
Control
Control
Control
XML-RPC
JSF Servlet

5. Modules

The current module 'framewokr' implementation is ok. It lacks a docuementation and an ready to copy example.

- One can copy a jar into the libs directory
- Based on the manifest the module is registered
- A base configuration is created automatically in the config repository
- Files can get extracted out of the jar. This is used for templates, paragraphs, ..
- Other configuration can get loaded based on properties files (perhaps we should use JCR import there too)

6. New Features

This are the most wanted ones:

6.1. Versioning

Like DMS.

- Activation creates a version
- List of versions
- Restore

Unfortunately the versioning using the JCR-Versioning is not straight forward. Since the pathes changes. It will not be easy to render a template based on a old version.

There are some possible sollutions:

- The content class wrapes this bevavior. getChild() returns the right freezed node. There is a method node.setToVersion().
- The subnodes (paragraphs) are versioned by copy (Check the specification). This will at least work for pages.
- Not using JCR Versioning (make a real copy of the nodes)

An other JCR-Versioning issue is that the import of version nodes is not straight forward.

6.2. References

- Keep references.
- Use SAX-Events to force UUIDs.

6.3. Improved Activation

- Transactional
- Synchronization: the author instance should be able to synchronize its state with the public instance.
- Incremental
- Activate through a Firewall. The author instance can be behind a firewall.
- Multiple Subscribers

6.4. Workflow

See separat project. Integration of OpenWFE.

7. Templating

WE DO NOT CHANGE THE TEMPLATING!

The only thing we change is how the creation, moving and deltions of paragraphs is handled in the backend (AJAX).

8. Process

To reach our goals we proceed as follow:

1. We start the final discussion on the list. Doing this we will use JIRA (Subtasks for specific topics)
2. If there is no consensus after one week we will vote.
3. We build a project team (who would participate, who takes responsibilities)
4. The team (maybe sub teams) makes a more detailed concept or prototype implementations.
5. We plan the work (priorities, milestones, resource assessment). We sill use centrics as the integral tool.

8.1. Team

- Up to 7 members
- Responsibilities. It should be clear to the team who is resonsible for which part of magnolia. A team member will consult a responsible person before he does work on an issue in his domain.
- The team members must be flexible and deal fairly with one the other. Every one must keep in mind that the work is done on a voluntary base.
- A team member supports the magnolia philosphy of simplicity and cares about usability.
- Obinary AG will cordinate the work

8.2. Voting

- The team members and contributors can vote
- A contributor is someone who would help in the implementation process
- The voters consult the community and consider their arguments
- Oly commit-votes are binding
- Obinary AG has a veto right