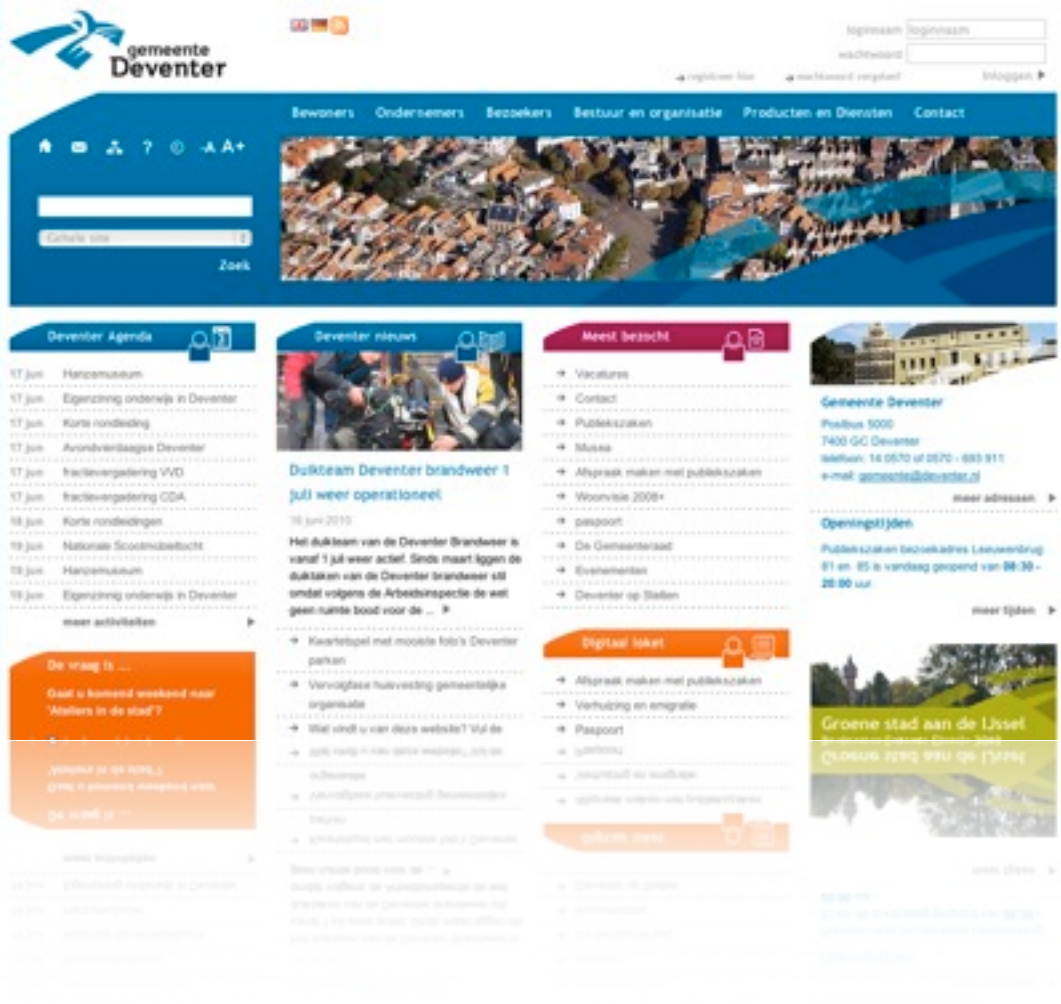
DEVCMS

TECHNICAL MANUAL



Change Log

DATE	BY	DESCRIPTION
2010-06-17	Reinier de Lange	Initial version
2010-07-15	Arthur Holstvoogd	Updated dependencies, added TOC, added gov. WS intergration
2010-11-04	Arthur Holstvoogd	Updated dependencies, added separation of GOV& CORE, many other updates. Changed CMS name to DevCMS
2010-12-31	Arthur Holstvoogd	Add chapter on Layout engine, correct other references, add new CT's
2011-01-20	Arthur Holstvoogd	Update to reflect new views/stylesheetsstructure, update dependencies

Table of contents

Introduction

This document is meant to give developers an understanding on how the DevCMS works. This includes installation information, an explanation of the architecture and guides for setting up new applications or extending existing functionality. Finally, this document describes possible future improvements and considerations of the core system.

System Dependencies

The DevCMS is a Rails 2 application that uses several gems and plugins. The plugins are packaged with the application and are installed using *piston* to allow updates from remote repositories.

Required Gems

NAME	VERSION	NOTES
shuber-sortable	1.0.6	Defines the order of nodes and content representations
ancestry	1.2.0	Enables the node tree structure
dsl_accessor	0.3.3	
feed-normalizer	1.5.2	
ferret	0.11.6	Windows users might need to install version 0.11.5 using the following command: `gem install ferret -v '= 0.11.5'`
httpclient	2.1.5.2	Dependency of soap4r.
libxml-ruby	1.1.3	
mocha	0.9.8	Only used in the *test* environment
pg	0.9.0	DevCMS uses database dependent queries for speed, therefore only postgresql servers are supported.
rails	2.3.5	Rails and all its dependencies.
rmagick	2.12.2	Uses *librmagick-ruby*, see *Required Libraries*. Windows might need to install version 2.12 instead.
rsolr	0.12.1	Used for searching using Apache SOLR (Luminis).
rubyntlm	0.1.1	Support for NTLM authentication, needed for sharepoint integration.
soap4r	1.5.8	For integration with SOAP services, DEPRECATED
tidy	1.1.2	
dynamic_attributes	1.1.3	
settler	1.2	Used for keeping project specific setting.
faster-csv		Used to generate permit csv's
acts-as-taggable-on	2.2.6	Tags are used as title altenatives
haml	3.0	

Optional Gems

NAME	VERSION	NOTES
faker	0.3.1	Used when generating test data using *rake db:populate*.
piston	2.0.8	Enables checking out plugins from remote repositories.
rcov	0.9.8	Used when measuring code coverage

Required Libraries

NAME	NOTES
imagemagick	Used by *librmagick*. This library might need `libjpeg`, `libpng3`, ... to be able to successfully transform images

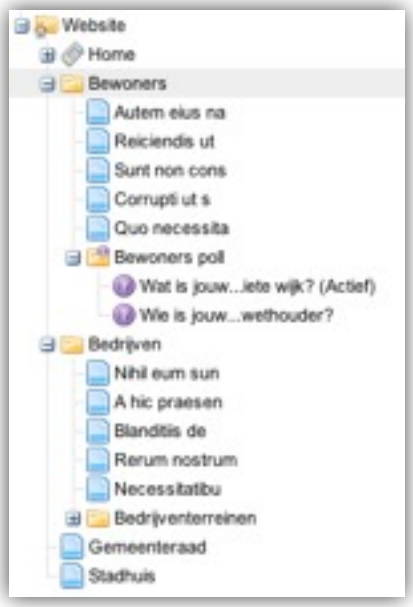
librmagick-ruby	Used by the rmagick gem to transform images.
	
	Installation:
	
	• Linux: `sudo apt-get install librmagick-ruby`
	• Windows:
	  • Download the RMagick package for Win32: http://rubyforge.org/frs/?group_id=12&release_id=20692.
	  • Unzip the file to a temporary location
	  • Follow the installation instructions in the README:
	  • Install ImageMagick by using the setup that can be found in the zip file.
	  • Be sure that you tick `Update executable search path` in the *Additional Tasks* part.
	  • Make sure RubyGems is up to date: `gem update -system`
	  • Install the RMagick gem by running this command from the temporary directory: `gem install rmagick -local`
	  • Install the Microsoft Visual C++ Redistributable Package
	  • It may be necessary to restart ruby services / running applications

postgresql82	The postgresql server. Should be version 8.1+.
postgresql82-dev	The postgresql development libraries.
libxml2	

Architecture

This section describes the global architecture of the system. First, the core principle behind the application, the tree node structure, is explained. The other subsections deal with the several aspects of content management that are available to the users of the system.

Node Tree Structure



The system has been built around the idea that every content type should be seen as a node of a tree structure. This can be justified by the fact that every website can be represented as a tree, as becomes clear when viewing a website's sitemap (see the example on the left).

Every content type in the system is associated with a node object. The node object contains information that should be available to every content type, of which the most important ones are:

- Node parent and position;
- Content and content-box style;
- Publication date;
- Visibility on site and in the site menu;
- URL alias;
- Approval status.

On the other hand, every content type consists of attributes specifically for that particular type. Consequently, every content type has its own routes, controllers and views. Most (but not all) models are a content type, as can be seen in the database diagram found in the appendices. In the diagram, all content types have a double line. The choice for not having made *everything* a content type was usually based on the fact that it wasn't logical for a particular model to be managed from the sitemap tree (e.g. newsviewer items, poll options) or that the amount of records that would be displayed in the tree would become too big (e.g. comments, forum threads/posts). Moreover, usually these models don't need a custom URL alias, as they are of lesser importance than the content nodes that are using them.

The tree structure has been implemented using polymorphic relations. Every node record belongs to exactly one content record. A future improvement would be to use Single Table Inheritance (STI) instead. This was not used initially, because the Rails STI implementation was very unstable when the core system was implemented.

Subsites and subdomains

It is possible to create 'subsites' using the Site content type. Site is a subclass of Section and can be handled as such with a few exceptions: It requires a domain to be specified, this should be a FQDN, and it can only be nested under the root node. It is not necessary that the set domain is a subdomain of the root sites domain.

In the front-end most content queries are scoped on the current_site, thus scoping a.o. the menu's, related content, abbreviations and synonyms, the site map, breadcrumbs, search etc.

Note that for routing purposes, the lookup *is* scoped, but the urls are not.

Authorization

A user in the system can have no roles at all, or is either an *administrator*, a *final editor* or an *editor* for a given subtree. This means that a user can have the *editor* role for one section of a site, but a *final editor* role for another. This has been implemented using the `RoleAssignment` model. If an administrator assigns the *editor* role to a user for a particular node in the system, that user will become an editor for that node and all of its descendant nodes.

Every role will grant a user to perform the following actions:

- **Administrator**:
  Full privileges: Creation, updates and destruction of all nodes, ability to assign user roles to nodes, approve content, change the global frontpage, manage users, manage permissions, add and delete synonyms/abbreviations, manage programs, manage weblog & forum comments.
- **Final Editor**
  Creation, updates and destruction of all nodes in the subtree, approve subtree nodes created/edited by editors, manage weblog & forum comments.
- **Editor**
  Creating and editing nodes in the subtree, which will be marked as *unapproved* and will therefore not be shown on the public site directly. It needs to be approved first by a final editor or an administrator.
- **Reader**
  A special role that allows no access to the backend, but is used to grant users access to 'hidden' sections in the frontend.

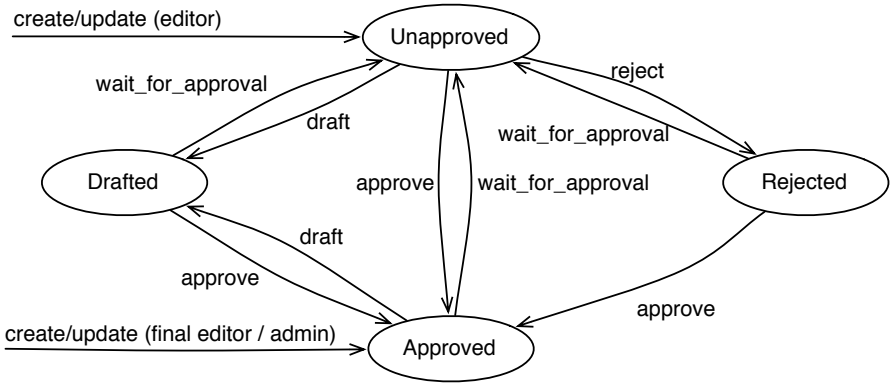
Publication Dates

Publication dates consist of a publication start and end date, which are defined on nodes. Every record of every content type will only be shown on the public site between those two dates. When no end date is set, it is assumed the content should be published indefinitely after the start date has passed. Remember to properly set the publication start date of an object when testing functionality.

Node Versioning & Approval

Nodes in the tree can be in several states, which is stored in the node's `status` attribute:

- `unapproved`: The node has been created or changed by an editor and is waiting for approval.
- `approved`: The node has been approved, created or updated by an administrator or final editor. A new version of the content node is recorded.
- `rejected`: An unapproved node has explicitly been rejected by an administrator or final editor. The editor responsible for the change will be notified when a node enters this state.
- `drafted`: The node (unapproved or approved) is drafted, meaning a user has not finished changing this node and should therefore not be shown on the website. The node will not be listed for approval.



When a content node is created by an administrator or final editor, it is automatically considered to be approved and therefore a new version of this node is recorded. There is a separate table to store these versions (the *versions* table), to which a serialized (yaml-ized) version of the content node is stored. The main idea is that the versions table always contains the last approved version of a content node. Conclusively, when an editor makes a change, the table of the content type will contain the new, unapproved version and the website should show the yaml-ized, approved version until the unapproved version has been approved by an administrator or final editor.

While the current method works, it has shown that it also introduces some disadvantages. First of all, it makes the query for finding a whole subtree of nodes that should be visible on the public site and its main menu very complex. For every node, it should check whether it is approved and it should get the last approved version from the versions table for unapproved nodes if an old, approved version exists. This is one of the main functions of the `Node#find_accessible` method.

For an approval system, it would be better to turn this around: unapproved versions of a record should be serialized and the old version should not be overwritten. This would greatly simplify the finding mechanism of approved content. However, there is another problem that should be taken into account. A big drawback of this mechanism is that it will only serialize the attributes of the content node, not the node itself. For example, a change in the publication date of the node by an editor will not be recorded: the node will become unapproved, but the old publication date will be lost.

Content Node Configuration

Each content type specifies a content type configuration and passes it to the acts_as_content_node method. Each configuration options has been listed in the table below:

NAME	DEFAULT	DESCRIPTION
enabled	TRUE	Whether users should be able to create this content node
allowed_child_content_types	[]	A list of content nodes that should be allowed as children. For example, a news archive can only contain news items.
allowed_roles_for_update	[admin, final_editor, editor]	Only allow node updates for users having a particular role on the current node (set explicitly or inherited)

NAME	DEFAULT	DESCRIPTION
allowed_roles_for_create	[admin, final_editor, editor]	Only allow node creation for users having a particular role on the current node (set explicitly or inherited)
allowed_roles_for_destroy	[admin, final_editor, editor]	Only allow node destruction for users having a particular role on the current node (set explicitly or inherited)
available_content_representations	[]	Array of available content representations, specify available representations as a string matching the names used in the layout configurations
has_related_content	FALSE	Some content types have related content, for these content types a custom representation is available to show this content in a content box
show_in_menu	TRUE	Whether this node should be enlisted in the main menu.
copyable	TRUE	Defines if a node can be copied, meaning it can be represented by a `ContentCopy` node. A content copy acts as a proxy to the copied node.
has_own_feed	FALSE	Defines if a RSS atom feed of this node is available.
children_can_be_sorted	TRUE	Defines if children of this node can be ordered by dragging.
tree_loader_name	nodes	Defines the controller name that loads the subtree of a node. Some content types show their children node in a way that is different from the default behavior defined in `Admin::NodesController#index`. For example, when a user lists the children of a `NewsArchive`, it will group all children by publication year and month.
has_edit_items has_sync has_importer	FALSE	Enable a menu item with the name (i.e. without has_) in the backend, that loads a action with that same name on the content type's controller. Can be reused for anything fitting. By default this is used by `ProductCatalogues` for sync options and `NewsViewers` for managing the related items.

The default configuration can be overwritten in the root application by editing the model class `dev_cms.rb`. The options specified in the hash returned by the method `DevCMS#content_types_configuration` are merged with the configuration passed to acts_as_content_node on demand. Example:

```
def content_types_configuration
  { 'ContactForm' => { :enabled => false } }
end
```

Administrator Interface

The administrator has completely been built using the ExtJS (v2.2.1) javascript framework. Although programming using this framework is a little bit harder than normal, it provides our users with a clean, dynamic and easy to understand interface.

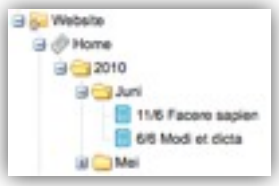
Custom ExtJS JavaScript Classes

Some DevCMS specific extJS javascript classes have been created, namespaced with 'Ext.dvtr'. Most of these classes are for browsing through the admin sitemap. A small description of every custom class:

- Nodes:
  - `Ext.dvtr.AsyncContentTreeNode`

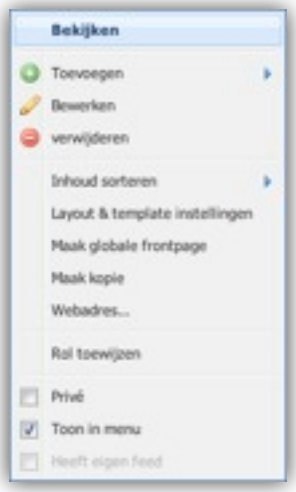
Represents a node in the sitemap. An `AsyncContentTreeNode` can have several attributes that define how the node is displayed and which actions a user can perform on the node. These attributes are loaded using JSON. All configuration attributes are defined in the `Node` model class by the method `Node#to_tree_node_for`.

  - `Ext.dvtr.AsyncVirtualTreeNode`



A virtual tree node is a container for content nodes instead of a representation of a real node. It is used to group big amounts of child content nodes together to prevent the tree from growing too big and to ensure that tree loading times stay small. There are several examples for which this technique has been applied. First of all, there are many content nodes that represent an archive of items that have a publication date, like for example news archives. When a news archive node is expanded, its attached news items are split up in virtual nodes that represent a year or a month of a year. The content nodes of a virtual node will only be loaded when the virtual node is expanded.

- Context Menus:



  - `Ext.dvtr.TreeNodeContextMenu`

This class represents the context menu of a node. It receives the node attributes by `Ext.dvtr.AsyncContentTree Node`, so that it knows which actions should be available to the current user.

  - `Ext.dvtr.VirtualTreeNodeContextMenu`

This class represents the context menu of a virtual node. Virtual nodes may have some actions as well, such as deletion of every content node contained by the associated virtual node.

  - `Ext.dvtr.MultipleTreeNodeContextMenu`

The sitemap also allows selection of multiple nodes at once. When this happens, the multiple tree node context menu is loaded. It provides general actions that are available to all nodes for the current user.

- Drag & Drop Classes
  - `Ext.dvtr.NodeDropField`

A node drop field is an input field in which a user can drop a node from the tree. On drop, the node ID will be appended to the form. This class is used for creating internal links or to set the start page of a section.

  - `Ext.dvtr.Sorter`

A Sorter is a panel in which nodes can be dropped. On drop, an `Ext.dvtr.Sortlet` will be created inside the Sorter. If a Sorter contains multiple Sortlets, the order of the Sortlets can be changed by dragging and dropping them in the right position. Sorters are used for newsletter editions and content carrousels, which create an object that combines multiple nodes.

  - `Ext.dvtr.Sortlet`

    A Sortlet is a child element of a Sorter. It is a panel that can be expanded to show its contents. It can be dragged to a different position within the Sorter panel.

- `Ext.dvtr.RemovableTextField`

A removable text field is an input field that can be added and removed from a form. It is used to set the several answers for a poll question.

- `Ext.dvtr.TreeLoader`

  The tree loader is responsible for loading nodes in the admin sitemap. It will load the root node and its children on load and it will add children to a node when a user expands that node.

- `Ext.dvtr.Panel`

  The `Ext.dvtr.Panel` does exactly the same as a standard panel, but it adds a 'beforeload' event to which listeners can be attached.

- `Ext.dvtr.ContentNodeFormPanel`

  The custom content node form panel does exactly the same as a standard form panel, except that it contains a function for inserting validation errors into the form.

Views, forms and partials

Since most of the forms and views follow the same CRU pattern for all content types, generic view templates are used that load content-type specific views. The generic templates are located in app/views/admin/shared, the content specific types are in the controller based view directories. I.e. in app/views/admin/*content_type_controller_name*. Although the controllers could probably be generalized to some extend as well, this had not yet been implemented.

Engines

DevCMS is using Rails Engines (http://rails-engines.org/) to integrate with Rails applications. This means core functionality can be overridden by placing classes in your application that have the same name as the target class found in DevCMS.

DevCMS consists of two engines: devcms-core and devcms-gov, the former containing all core and basic functionality, the latter containing all functionality geared toward use by municipalities.

It is encouraged to further extend the number of submodules and create smaller blocks of functionality.

Layout and templating system

DevCMS has a build in layout and templating system to allow the easy creation and management of multiple layouts and variants to be used in the CMS. These layout reside in app/layouts and override any the default views provided by the CMS engines.

In the following paragraphs the different aspects of views, layouts and templates are explained.

Content representations

Each layout and variants thereof provide the CMS user with different targets to place existing content in. Content placed in these targets is shown in the front-end using a representation specific to that target. Which targets exists, how they are laid out and what representation it requires are all specified in the layout configuration file.

Aside from content representations directly linked to existing content, dynamic representations, or custom representations, can be implemented. By default this functionality is used to create menu's and related content boxes. Custom representations can either render a partial or call a rendering helper that should output the wanted markup.

Layouts and variants

To provide extensible and flexible templating, DevCMS supports layouts and variants thereof. In this scope a layout provides the general makeup and styling of the website and a default layout of the content within. A variant provides a way to create alternative layouts of the content, but should not significantly change the styling of the website. I.e. the layout ensure the general look and feel of the website, variants determine the positioning of content target areas.

Aside from using layout variants to vary the layout of pages, it is also possible to extend layouts in another layout. This allows you to reuse most of the layout and its variants, while changing, for instance, the general styling of the website. This prevents you from having to copy paste entire variants when only the header and footer of a layout differ.

Extending cannot be used recursively.

Layouts reside in the app/layouts directory and consist of the following files and directories

FILE / DIRECTORY	DESCRIPTION
config.yml	Configuration of the layout and its variants, explained further in the next paragraph
settings.html.haml	Form for any layout specific settings that should be specified when this layout is used.
targets.html.haml	Simple representation of the layout, containing the content targets. Used in the CMS backend to render the content drop targets.
views	Any partials of overrides specific for this layout
variant 1, *variant 2*, *variant n*	Any variants of this layout, can also contain a targets.html.haml and view directory specific to this layout.

Layout configuration

The layout configuration file (config.yml) contains specifications of the layout and it's variants. All available options are described below.

OPTION	DESCRIPTION
name	A descriptive name for the layout
extends	A reference to another layout which this layout is based upon. Referenced by directory name. (Not path!)
custom_representations	Configuration of the available custom representations in this layout. Each custom representation available should be specified as a key with some configuration nested. Per representation the following options are available:
custom_representation	key identifying the custom representation.
name	A descriptive name for the custom representation.
content_partial	Content partial to render, should be available for the current content type. If no content_partial is given, a helper method is assumed to exist using the custom representations key prefixed with render_.
representation	The type of representation the content should be rendered in. This representation will only be allowed in targets that support the type specified.
targets_defaults	Target defaults, to prevent duplication between variants. The following options can be specified:
target	key identifying target.
max_items	Maximum allowed content representations to be placed in this target. Empty or 0 => no maximum.
representation	Allowed representations, only content that supports this representation is allowed to be placed in this target.
main_content	Specifies wether this is the primary content area of this layout(variant), corresponds to the default yield in the actual layout. Defaults to false.
height/width	height and width used in the backend representation of the target.
variant	Key identifying the variant. Specifying a default is required.
name	A descriptive name for the variant.
inheritable	Specifies wether this variant is inherited by children nodes, or wether they should fallback to the default variant. Defaults to true.
target	Targets need to be specified as a key, configuration defaults may be overridden here. See target_defaults for options.

Views load order

Views are loaded from different locations in a fixed order, this allows you to override certain layout elements as needed. The table below outlines these locations and which views are, or should be, placed there. The table is in order of precedence, higher item override lower items.

LOCATION	CONTENTS
app/layouts/*layout*/*variant*/views	Overrides specific to the current layout variant
app/layouts/*extended_layout*/*variant*/views	Overrides specific to the extended layout variant
app/layouts/*layout*/views	Overrides specific to the current layout
app/layouts/*extended_layout*/views	Overrides specific to the extended layout
app/views	Generic layout of application specific functionality
vendor/plugins/*engine*/app/views	Generic layout of CMS functionality

Styling

The default templates and partials are all styled using SASS, the default styles are included in the app/stylesheets directories of the engines. An initializer 'loads' these stylesheets into the application, it is recommend these stylesheet are then imported in an application specific application sass file that also imports all application specific styles and templating styles needed.

Guides

Setting up a new website

1. Install all system dependencies
2. Generate a new rails application using the DevCMS application template:

   ```
   rails application -m devcms-template.rb
   ```
3. Setup your database by modifying your `database.yml`
4. Create, migrate and seed the database: `rake db:create db:migrate db:seed`
5. Run the web server and browse to the root page. You should be up and running! The seed data sets you up with a root node and an admin user (admin:admin).
6. For development, there are rake tasks available to populate the database with random data:
   - `db:populate:users`

     Populate the database with 1000 random users that don't have any role
   - `db:populate:privileged_users`

     Creates an administrator, a final editor and an editor with the following login information:

LOGIN	ROLE	PASSWORD
webmaster	administrator	admin
eindredacteur	final_editor	final_editor
redacteur	editor	editor
   - `db:populate:nodes`

     Creates a small node structure of several content types for use during development.
   - `db:populate:all`

     Invokes all of the above tasks.

Adding a new content type

1. First, implement the model:
   1. Create or generate a new ActiveRecord model: `script/generate model <model>`
   2. Add validations and logic as usual
   3. Declare the model as being a content node by adding the line: `acts_as_content_node`
   4. If the new content node should be approvable (i.e. should be set to unapproved when created/updated by editors), add the following line: `needs_editor_approval`
   5. Add node configuration options to `DevCMS.rb`
   6. Some content node methods can be overridden if desired:

METHOD	DEFAULT	DESCRIPTION
`content_title`	Model `title` attribute, or `ModelName#id`	Should return the title or a substitute title for the record
`tree_text`	`content_title`	The text to show in the admin sitemap.

METHOD	DEFAULT	DESCRIPTION

`tree_icon_class`	Underscored model class name + *'_icon'*	The CSS class that this node in the admin sitemap should get. The css class should set an icon as *background-image* property:
		
		`.page_icon {`
		`  background-image: url('/images/icons/page.png');`
		`}`

`content_tokens`	nil	Should return a string with tokens (separated by spaces) that will be used for indexing.
`icon_filename`	Underscored model class name + *'.png'*	Returns the filename of the icon that will be used to represent the content node on the public website. This icon should be stored in */public/images/icons/*.
`self.owms_type` *(class method)*	I18n.t('owms.web_page')	Should return the OWMS type for the model. Every OWMS type has been added to the I18n YAML files, scoped under `'owms'`.

2. Create an admin controller for the model.
   1. Place the controller in the */apps/controllers/admin* directory. Prepend the `Admin` namespace to the class name and make it a subclass of `Admin::AdminController`.
      Example: `class Admin::PagesController < Admin::AdminController`
   2. Implement all resource controller methods (`new`, `create`, `edit`, `update`), except the `index` and `destroy` methods. These last two will be handled by the `Admin::NodesController`.
   3. Restrict access to all actions by calling the `require_role` method with an array of roles that are allowed to manage the resource: `require_role [ 'admin', 'final_editor', 'editor'].`
   4. Prepend the following filter for the `new` and `create` methods. This will ensure that the authorization is based on the permissions that the current user has on the parent of the new node.
      `prepend_before_filter :find_parent_node, :only => [:new, :create ]`
   5. Only when the new content node is approvable:
      1. Use the method `Model#save_for_user(user)` instead of the normal `save` method to save the model. This will ensure that the node state will be properly set.
      2. Besides the normal resource controller actions, add a method named `previous`, which should behave the same as the `show` method, but uses the old version of the model instead. It will be used to show a diff between the new and old version of the node to an administrator or final editor
         Example:
         ```
         def previous
           @page = @page.previous_version
           show
         end
         ```
   6. Create views for all resource actions. Please take a look at views of other types for examples. It is important for vies of approvable types that there is a `show` partial (`_show.html.erb`) with a local variable called `record` (containing the content node), because this partial will be used by the approval screen.
   7. Create a new route for this controller:
      Normal content types:
      ```
      map.namespace(:admin) do |admin|
        admin.resources :pages, :except => [ :index, :destroy ]
      end
      ```
      Approvable content types:
      ```
      map.namespace(:admin) do |admin|
        admin.resources :pages, :except => [ :index, :destroy ], :member => { :previous => :get }
      ```

```
end
```

3. Create the controller for the public part of the website
    1. Place the controller in the */apps/controllers/*directory and make it a subclass of `ApplicationController`. Example: `class PagesController < ApplicationController`
    2. Only implement the `show` action in this case. The `ApplicationController` will automatically load the requested node in the `@node` variable, so you can easily access the last approved version of the content node by calling `@node.approved_content` (will work for normal and approvable content types).
    3. Create a `show` view
    4. Add a new route for your controller. Example: `map.resources :pages, :only => :show`.

Overriding core functionality

Note that the behavior of Engines is different for models and controllers. Functionality of an overridden controller will be merged with the target plugin controller, while an overridden model will completely be overridden. So, if you want to override a model class, you will need to include the original model or override it completely. When you want to extend functionality, you could choose to use Engines again and move your code to an Engines plugin instead of adding it to the root application.

While you can add new migrations to the root application you cannot override the migrations copied from the plugin, these are updated on migrate.

Setting up a search engine

DevCMS uses the Strategy software pattern to switch between search engines. Currently, there is support for Ferret.

The `Searcher` class is responsible for deciding what to do with a search query. Two settings can be used to modify its behavior:

SETTING	DEFAULT	DESCRIPTION
`DEFAULT_SEARCH_ENGINE`	:ferret	The search engine to use when no engine name is explicitly being passed.
`ENABLED_SEARCH_ENGINES`	[:ferret, :other]	Defines which engines are allowed to be used.

The `Searcher` expects a search query and an `options` hash, to which you can pass options that are specific for every engine. It should return a `PagingEnumerator` object (for pagination support) containing `Searcher::SEARCH_RESULT_STRUCT` structs, which consists of the following attributes:

- `:title` - The title of a found resource
- `:tstamp` - The creation/modification date of the resource
- `:content` - The content or an excerpt of the content
- `:url` - The unique url to the found resource
- `:node` - Optionally, the DevCMS node of the resource.
- `:score` - Optionally, the result (relevance) score

Example: Searching using Ferret

- Call the Searcher:
    - `Searcher(:ferret).search('my_query') / Searcher.new(:ferret)).search('my_query')`
    - `Searcher.new.search('my_query') # Uses the default search engine`
    - `Searcher(:ferret).search('my_query', :for => current_user, :page => 2, :page_size => 25)`
- Example response:

- `Searcher(:ferret).search('my_query').to_a`
  `=> [#<struct Searcher::SEARCH_RESULT_STRUCT title="Home", tstamp=Thu Jun 17 14:56:59 +0200 2010, content="Rerum consequatur amet dicta aliquam sit quia.", url=nil, node=#<Node id: 2>, score=nil>]`

Creating a new search engine

1. First, create a new class in */app/models/search* that implements the following method:

```
def self.search(query, page, page_size, user, top_node, options)
    # Perform the search

    # Create an array of Searcher::SEARCH_RESULT_STRUCT objects

    # Encapsulate and return the results in a paging enumerator:
    PagingEnumerator.new(page_size, numFound, false, page, 1){ search_results }
end
```

The `Searcher` class will ensure that all arguments passed to the `search` method will be set. It should be noted that `user` might be `nil` and that `top_node` will always be the root node of the website if a top node is not explicitly set.

2. Expand the `Searcher` class
   - Add a symbol for your new engine to the `Searcher::ENGINES` constant
   - Expand the initializer to load your new class when the engine symbol is passed. For example, considering you would like to add a Sphinx engine, you would have to add '`when :sphinx then Search::SphinxSearch`' to the case statement.
3. Change the `ENABLED_SEARCH_ENGINES` and optionally the `DEFAULT_SEARCH_ENGINE` constant to be able to use your new engine.
4. For your engine, you might need to extend the `Node` class to receive updates for the search index. When nodes are created or updated, the `Node` class will look for the following methods and execute those if present:

METHOD	DESCRIPTION
`self.without_reindex(&block)` *(class method)*	The block passed should be executed without updating the search index.
`without_reindex(&block)`	The block passed should be executed without updating the search index. The block applies specifically to the node instance.
`update_index`	Should trigger the engine to update the index for the current node instance. Will be called after a node update.
`add_to_index`	Called after a node has been created to add the node to the search index.
`disable_reindex_until_saved`	Should defer a reindex call until the node is saved.

Please take a look at the `Search::Modules::Ferret::FerretNodeExtension` module found in */app/models/search/modules/ferret* class for an example.

Notes & Remarks

This section contains several important notes on functionality and conventions used in DevCMS.

Content archives

A common case is the usage of an 'archive' as an encompassing concept for a set of content of a certain type, e.g. the NewsArchive containing NewsItems. The relation between the two is supported both in the model, using the has_children/has_parent marcos, and in the controllers using the acts_as_archive marco.

More information can be found in the documentation of these methods in the documentation of `ActsAsContentNode` and `ActsAsArchive`.

Settings

DevCMS provides functionality to store settings using Settler. Defaults and available settings can be configured through settler.yml in /config. Editable options can be changed in the backend.

Localization

DevCMS has used localized strings since the start. Due to the absence of Rails I18n, the initial version used the *simple_localization* plugin, which is similar to Rails I18n. After the inclusion of Rails I18n, the code has been altered to use that functionality instead.

JavaScripts have been localized in i18n(_admin).js on a per-project basis. Usage is similar to the usage of Rails I18n.

Opus product catalogue integration

Product catalogues are currently only capable of importing products from the Opus product catalogue. Once setup products for both the municipality and the SamenwerkendeCatalogie can be updated with a cron job.

To setup a OpusPlusImporter has to be created from the console and initialized with the customer ID, the supplier ID and the product catalogue to sync to.

Integration with governmental webservices

There are currently two integrations with the governmental webservices of Overheid.nl available. One for pushing announcements (Bekendmakingen) to Overheid.nl and one for pulling legislation (Regelgeving) from Overheid.nl.

Announcements

Announcements are basically changes in a permits status, however they are stored as a new Permit in the database. New permits are pushed to the Overheid.nl webservice once a week with a cronjob (Permit.publish).

Legislation

Legislation is pulled from Overheid.nl with a daily cronjob and stored in the LegislationArchive.

Caching

Caching has been done using the built in Rails caching mechanisms (`ActionController::Caching`). By default, DevCMS will use the file_store as its cache store. Caches are expired by using sweepers. Cached fragments include:

- Main menu
- Feeds
- Sitemap
- Selections on products

Should I override, create an engine or update DevCMS Core?

The choice for one of the three options depends on the functionality that is implemented. When certain functionality will only be used in one application, it should be implemented in the core application. Functionality that may be used in other projects (but not all) should be implemented as an engine. Finally, functionality (and of course bug fixes) that will be useful for all DevCMS implementations should be added to the core.

Future Improvements & Considerations

Node versioning, document management

In the current system, versioning is used to show the difference between an approved and an unapproved node. It would be nice to have multiple versions of nodes as well. Moreover, these versions should be able to have different statuses in order to create a document workflow, as is common in Document Management Systems (DMS's). This would also enable collaborative creation of website content.

The best way to support document management would be to have nodes that have a common parent node and can be in several states. However, this would need to apply to the node as well as its content node. Therefore, the first step towards this goal would be to switch to Single Table Inheritance (STI) and getting rid of the one-to-one association between nodes and content nodes.

Approvals

DevCMS uses versioning for approvals, meaning that a version of a content node will be versioned when it is saved. This also happens when an editor saves a content node. The consequence is that the query for retrieving approved nodes becomes much more complex, as the previous version of a content node must be loaded when the current version is unapproved. A good optimization of the system is to turn this mechanism around: when an editor saves a content node, it shouldn't update the current node, but it should serialize the unapproved content instead. This way it is known that the current content node is always an approved version, thus eliminating the need to retrieve a specific version.

Modularity

The DevCMS Core was originally a normal Rails application instead of an engine. Although it is modular in the sense that it has been built using the MVC pattern, there are quite a lot models that could be extracted to a separate engine as they are not common for a general purpose CMS.

This might provide some improvement, but possibly also drawbacks, in updating and maintaining versions of the CMS.

Content accessibility management

One of the main performance hits is caused by the complex nature of access rights and publication date inheritance. By removing these inheritance properties the complexity of finding accessible content can be reduced significantly. This should not only improve performance and complexity (and thus maintainability), but also make it possible to do more caching, which would improve response times and scalability.

Rails3 Upgrade

Several changes need to be made to make DevCMS Rails 3 compatible. However, most notably the new routing capabilities and the Rails XSS plugin would probably make the code a lot cleaner. Tasks involve:

- Reimplement node routing
- Remove `h()` helper calls and make strings `html_safe` where needed (*rails_xss*)

- Rewrite `find` calls to *Arel* queries
- Rewrite named scopes to *Arel* scopes
- Revise / update Rails 2 plugins and gems
- Rewrite ActionMailer classes
- Change the way the database is seeded

Most notable improvement would be the ability to gemify the engines an increase maintainability.

Memcached

The use of a different caching mechanism like memcached should be considered. The performance gain of switching to a mechanism different form *file_store* is unknown, but a big advantage of memcached is that it can automatically expire cached fragments, eliminating the need for cache sweepers.

Testing

- DevCMS has been extensively tested using unit tests and functional tests. The use of new technologies for testing functionality, like behavior driven development (BDD) solutions as Cucumber or RSpec should be considered.
- To speed up testing, it would be a good idea to use factories instead of fixtures. The current test suite heavily relies on fixtures, which is one of the main reasons that running this test suite takes a while.

Appendices

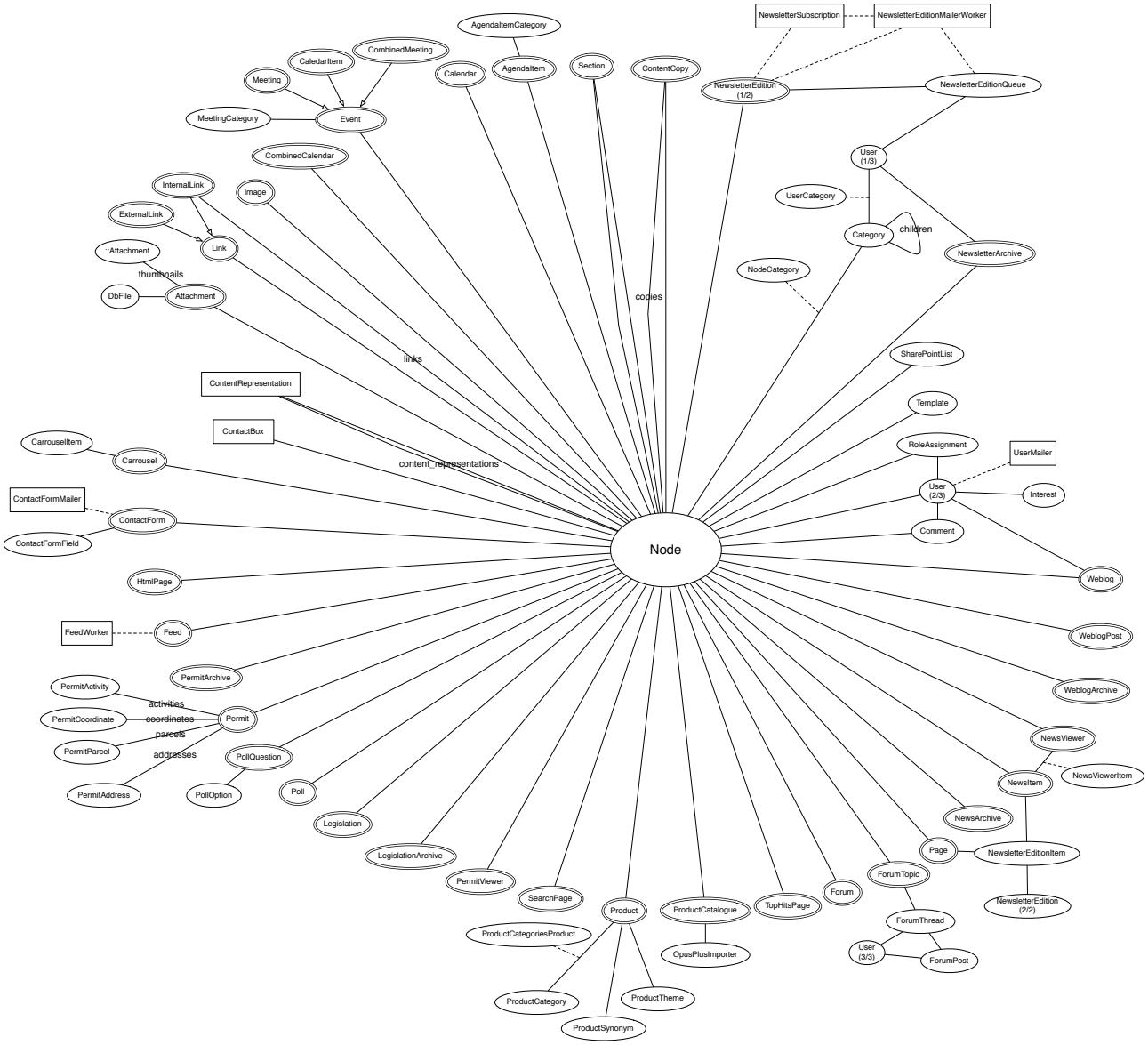
Background Tasks

For all of these tasks, it is assumed that the current directory is the location of the root application, e.g. */home/deploy/production/current*

All tasks are managed using a schedule.rb in the config directory.

TASK	FREQUENCY	DESCRIPTION
vendor/plugins/DevCMS_core/script/sharepoint_updater (calls script/runner 'SharePointList.import_all')	`Every 5 minutes`	Imports SharePoint lists.
script/runner 'FeedWorker.new.update_feeds'	`Every 10 minutes`	Updates RSS Feeds.
script/runner 'Node.reduce_hit_count'	`Weekly`	Reduces all node hit counters by a given factor (default 0.9).
script/runner 'Permit.publish!'	`Hourly`	Publishes approved permits of which the publication date has passed to overheid.nl.
script/runner 'NewsletterEditionMailerWorker.new.send_newsletter_editions'	`Daily`	Delivers new newsletter editions to subscribed users.
script/runner 'LegislationArchive.all.each { |la| la.import_legislations }'	`Daily`	Imports changed CVDR legislations.
script/runner 'Node.find(:all).each { |n| n.update_index }'	`Daily`	Updates the search index of all nodes.
script/runner 'OpusPlusImporter.find(:all).each { |opi| opi.update_all }'	`Daily`	Updates all OpusPlus products and product categories.
rake db:remove_unverified_users	`Daily`	Removes all users that haven't verified their account in a week.

Database Diagram



Approvable Content Types