METADOT

Metadot Portal Server
Developers' Guide

Version 5.6

April 18, 2003

Table of Contents

The **Metadot Portal Server** is an open-source point-and-click website builder that allows non-technical users to build powerful websites and portals in just a few minutes. It provides many content management and collaboration features, including pluggable applications called Gizmos, as well as a dashboard feature like “My Yahoo”. The Metadot Portal Server is fully customizable by the site administrator. Logo, colors, content and components can be edited using a standard web browser.

The Metadot Portal Server is based on Apache, Perl and MySql, and runs on Unix-based operating systems like Linux, Solaris, and MacOS X, as well as Windows. Developers can create new Gizmos using the open gizmo architecture and, if they like, contribute them to the open source project. The Metadot Portal Server has been deployed in many global corporations and leading universities. It is reliable and scalable, and can support large communities of users.

The purpose of this document is to provide more information about the Metadot Portal Server’s development interface, for developers who wish to extend the system’s functionality, or modify the system’s default look or layout. For information about system installation and configuration, or general usage, see the **Users**’ and **Administrators**’ guides included in this distribution. We highly recommend that you read the Users’ and Admin guides first before reading this document, since they provide a good overview of the site features. We also recommend that you install and get hands-on experience with using and adding content to the site before you begin development.

1. Overview

In this section, we provide an overview, from a developer’s perspective, of some of the Metadot Portal Server’s functionality and components. For some of the Perl modules described in this guide, Perl POD documentation is also available to the developer. The POD documentation can provide additional information about some of the specific methods referenced.

1.1 Dynamic Generation of Site Content

Pages delivered by the the Metadot Portal Server software are all dynamically generated from content stored in a relational database. The content may include static HTML fragments, but all of the delivered pages are built dynamically.

There are two basic types of pages. The first type of page can be labeled, in Metadot terminology, a “Category” page. These are pages for which users, if they have the appropriate permissions, can create new content, such as sub-pages (sub-categories), discussions, polls, news items, etc. The second type of page provides a customized set of information *channels* for each user. These are the “My Page” site pages. For their “My Page”, a user does not create content directly, but builds his/her page from pre-existing content channels.

A Metadot site supports a community of registered users. Once a user has registered and created an account, they may log on. A logged-in user can typically view more of a site than a non-logged in user, and can add and modify site content. Metadot supports a variety of models for authenticating and registering a user, including use of LDAP directories; and for browsing information about the user base.

In conjunction with the code that generates site content, the Metadot Portal Platform includes "backend" functionality, which runs asynchronously to its portal requests. The backend scripts gather information from the internet, process it, manage user subscriptions (described below) and perform various site cleaning tasks.

1.1.1 Gizmos and Category pages

Most of the pages generated by a Metadot Portal Server are populated by an information class called a “Gizmo”. *Categories* are a subclass of Gizmo, and other content objects such as Items, Discussions, NewsItems, Tables, etc., are Gizmos as well. Categories are the primary organizational/container metaphor for this type of page: a category can have any number of child gizmos added to it, including sub-categories; and this creates a ‘tree’ of gizmo objects. Site pages are then dynamically built from this content tree, with a category object’s children contributing to the rendered content of the category as a web page. For example, if a user creates a child Discussion for a Category page, then a ‘summary’ view of the Discussion info will typically be included when the Category page is rendered.

A gizmo may be a child of only one parent, but *shortcuts* (essentially, symbolic links) can be created to allow a gizmo to appear to be associated with more than one parent.

Gizmos are subject to the Metadot Portal Server’s *access control* scheme (as described in Section 3); their visibility and editability can be controlled with respect to each user. This means that a page may ‘render’ differently for different users, depending upon what page content the user has permission to view or modify.

When a user account is created, a set of Category pages is created for the new user, over which they have editorial control. This area is called the user’s “My Website” pages. On these pages, they can create new content, including child pages, and set the permissions of the content to determine who can edit or view it. There may be other parts of the site, in addition to the “My Website” area, for which others have given a user edit permissions as well.

Some types of Gizmos can export their information as *channels* (see below).

1.1.2 MyPage pages and channels

The ‘MyPage’ pages built for each user of the portal, are different than the rest of the portal. Conceptually, each *MyPage* displays the information from a set of information *channels*, where the channels can be selected and arranged on the page according to the user’s specifications. A user can also subscribe to change notifications on the channels.

Nearly all of the information pushed to the MyPage channels is locally warehoused (cached) so that it is available quickly on demand. Behind the scenes, the information delivered to a *MyPage* is of four different types:

- External channels: information from external RSS channels, locally parsed and cached.
- Internal channels: information exported from Category News and Discussion Gizmos. (Not all Gizmo types can currently export their contents as a channel).
- “Gizmo Tools”: Display information from non-RSS external sources, such as information about whether a web server is up.
  All Gizmo Tools are subclassed from `GizmoTool`.
- “Basic Tools”: Basic Tools are HTML fragments inserted into a *MyPage*. Typically they implement external search tools of some type (e.g. web search or map generation), though there is no restriction on what the HTML can contain. By their nature, “Basic Tools” have no locally cached content.

A user’s MyPage is built by accessing the user's page preferences (selected channels and page layout), in conjunction with the cached channel content. The same channel content may be accessed and shared by many users. Internally-derived channels respect the permissions settings of their original information sources and thus not all channels may be viewable by all site users.

In addition to channels, users may also select to display on their MyPage other information sources such as weather indicators and webserver monitors.

A portal 'backend' script runs periodically to perform channel management tasks. It fetches the external channels, then converts all the information sources, internal and external, to an html channel format, and stores the html channel content in the database (thus locally warehousing or caching the content so that it is available quickly on demand). The refresh interval for the cached information is configurable by the portal administrator. In addition, the script checks user subscriptions and sends out *change notifications* on the channel content as appropriate. The change notifications can be delivered through various media such as email and pager.

1.2 Levels of Administrative Access

A Metadot site has two levels of administrative access: *Site Managers*, and *Admins*. Typically there is one Admin per site, and multiple Site Managers. Site Managers have a subset of the capabilities of the Admin. They can edit any site content, and manage users and groups. They can add and edit “My Page” content, and change the site look and feel. However, the Admin has access to a number of site configuration parameters that the Site Managers can not access.

1.3 Backend portal management

A portal site runs two server-side scripts periodically. The first performs channel management tasks, as described in the section above. The tasks are scheduled, with run times stored in the database. Some tasks run just once a day, others run every hour, etc. The second script performs various cleanup tasks.

If *virtual servers* are in use, as described in the Administrator’s guide, then this means that several different content databases are set up, each accessed by a different virtual server. In this case, the scripts run through their set of scheduled tasks for each database in turn.

1.4 Layout and Configuration

Much of the “look and feel” of a Metadot site is configurable through the browser interface. High-level layout (the way the content of each page is rendered) is determined by selecting one of several basic layout modes. It is possible to switch a site back and forth between styles dynamically.

Two of the layout modes provide a relatively fixed, non-template-based layout style, and two use HTML *templates*. Embedded in the templates are Metadot tags called GizmoTags. The GizmoTags allow various content to be inserted into the HTML page, resulting in more control over what goes where and how the page looks, than is possible with the fixed layout styles. One of the template-based modes is the *Themes* mode, in which a fixed layout template is used for each page type, and the site manager has the ability to change logos and colors for the pages[1]. The second template mode is Metadot’s “Style 4”, in which a site manager has the ability to tailor the set of templates used by the site to their own specifications. This provides more control, but also typi-

1. Currently, there is no browser-based interface for modifying the *Themes* layout.

cally requires a higher initial outlay of effort in setting up the sites, than is required with the fixed layout styles. Section 8 describes template and gizmo tag use in more detail.

It is also possible to configure, through a browser interface, many of the table cell colors used on the site (in order to render object frames, etc.), and to configure the style rules used for the site pages. More info is provided in the Administrator's guide.

In the remainder of this guide, we provide more detail about the aspects of the Metadot Portal Server system that a developer must be familiar with in order to generate new Gizmos and gizmo-based "applications".

2. Database Tables

While you may not need to directly access the database tables that underlay the system, it is useful to have a model of how they are employed. The database serves as a persistent store for all of the objects served up to the dynamically-generated Metadot Portal Server pages. The system does very little out-of-database caching, and thus performs a set of database queries for each page that it builds.

Here we list some of the more important information objects in the system and the database tables they map to. You may wish to revisit this section after you've read the sections below.

Gizmos: all types of Gizmos (Categories, Discussions, Tables, Items, etc.) use the `instance` table to store their data. In addition, some gizmos utilize secondary tables as well. The Discussion gizmo keeps its messages in the `message` table. The Poll gizmo uses `poll_response` and `poll_vote`.

GizmoTools: The GizmoTool parameter info is in the `gizmoitemparam` table as attribute/value information; and `gizmotoolitem` is used to associate sets of gizmotool parameters with channels.

Basic Tools: The description of a basic tool (e.g. the HTML used to render it) is stored in the `instance` table.

User information is stored in the `user` and `extended_user` tables. Required "core" fields are in the `user` table. Then, subclasses of the default user class map their specific fields to the `extended_user` table, allowing different Metadot sites to be configured to support different user profile information (see Section 4).

Groups: information about groups and the users in each group is stored in the tables `grp` and `grpmembers.`

Channels: channel information is stored in `channel` and `channelitem` tables.

The `mypage` table holds information about the 'my page' channel selection and layout for each user.

The `permissions` table holds information about the permissions for every Auditable object (see Section 3).

The `session` table holds user session data.

The `uploads` table holds the file “attachments” associated with portal objects (where these objects are typically Gizmos, but need not be).

The `subscription` table holds the *channels* to which each user has subscribed (for example, an external news channel), and info about the form in which they wish to receive the change notifications (e.g., daily digests).

The `params` table holds all of the system parameters; all of the site’s configuration information. This includes layout and ‘look’ information as well as the variables that determine site operation (for example, the mail server address).

The `event_log` table is used to log information about nearly every site request. This information can then be displayed in the browser via an administrative interface. The event table is cleaned periodically by one of the portal’s “backend” scripts. See the Administrator’s guide for more information. (Note: not available in open-source release).

3. Access Control

3.1 Overview

Metadot's Access Control (AC) system is the infrastructure that allows the system to check whether a given user can perform a given operation over an object. Most of the access control information is contained in a Permissions hash associated with each Gizmo object. The Permissions information is consulted when determining whether or not an operation is allowed.

The permissions information for an object consists of operation *bundles,* which associate groups of operations on the object with a permissions level for those operations; and an access control list, where users and groups on the list are given 'owner' permissions for specified operations.

A bundle specification is defined for each gizmo class. For a given gizmo (such as Item), the developer may define a *view* bundle, and associate it with the operations `show`, `show_as_bullet`, and so on. Then, the developer may define an *edit* bundle, and associate it with a set of editing operations such as `modify`, and so on. The Operations class supports modifications of the bundle structures.

For each operations bundle defined for a class, the designer must define a *default access level* for the bundle, and a *minimum access level* for the bundle (described in more detail below). The *default* access level is the level set for the object when it is created. The *minimum* access level defines the lowest (most permissive) access level to which the bundle can be set. The "access level" corresponds to the status of the user attempting to access the object. There are four possible access levels: *public access, logged in, owner, site manager,* and *admin*. As is further described below, when an object is created, its default access settings are used in conjunction with the access levels of the object's parent, to instantiate the new object's Permissions info.

A user's status is matched against the permissions info, as is further described below, to check that the operation requested by the user is allowed. So, for example, if an object's *view* bundle permission was set to 'logged in', then a user of the site can not perform the 'view' operations on the object (essentially, can not see it), until they log in. As another example, the Discussion gizmo has a *Moderate* bundle, for which both the default and minimum access levels are set to "`owner`", so that non-owners can not moderate the discussion.

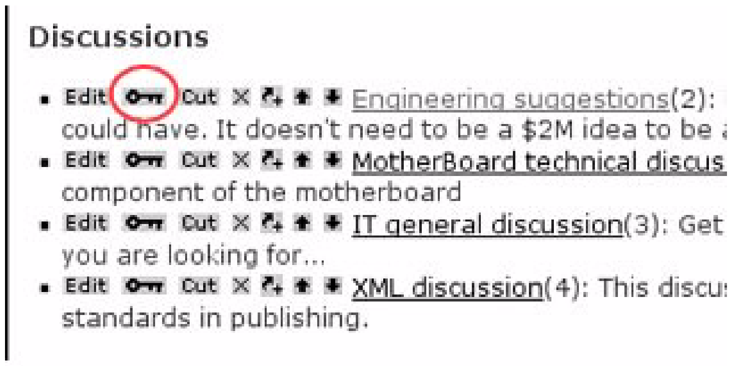
In addition, the Permissions hash for a given object may include an access control list, where the users and groups on the list are given 'owner' permissions for specified operations.

The logic of the permissions framework is contained primarily in the `Auditable` abstract class, from which most of the portal objects are subclassed. Auditable uses an object's Permissions information, in conjunction with the User info, to determine whether the User is allowed to perform a given operation on the object. Certain methods in the Auditable interface need to be implemented by the subclass, and for Gizmos, this implementation is facilitated by using the GizmoBuilder class. It is recommended that any new gizmo classes inherit from GizmoBuilder (see Section 5 for more information on building gizmos using GizmoBuilder).

3.1.1 Permissions-editing interface

All objects that inherit from Auditable, including GizmoBuilder and its subclasses, can support a permissions-editing interface. One of the GizmoBuilder's operations bundles is the *edit permissions* bundle, which is used to determine whether a given user can edit the permissions for the objects. By default, only the object's owner (and site managers and admins), can edit an object's permissions.

When edit mode is enabled on the site, each object will display a 'key' icon next to its 'edit' icon, if the user is privileged to edit the object's permissions. When this key is clicked, a form comes up which allows the user to modify the access levels for each of the objects bundles. In the form, there will be one pulldown menu for each of the object's operations bundles.

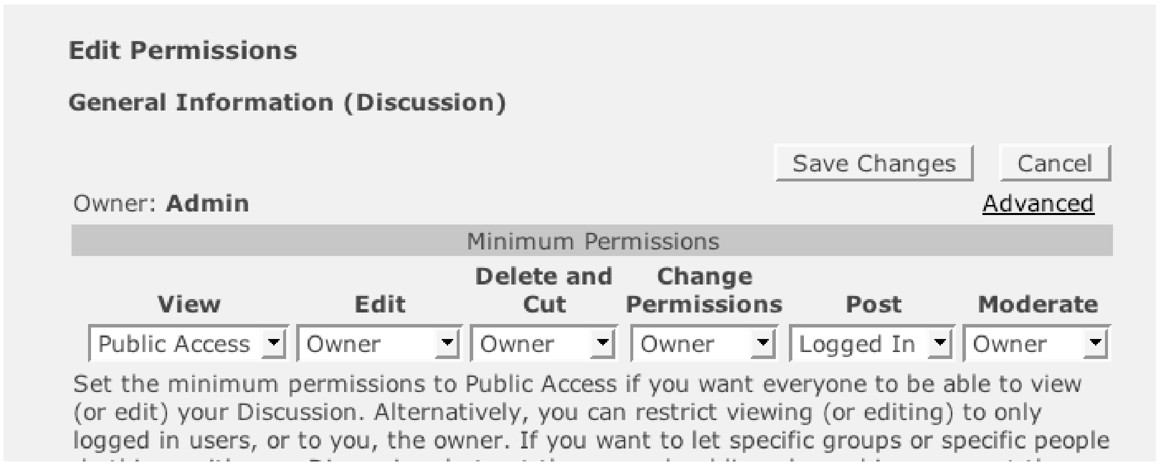


Discussions

- Edit Cut Engineering suggestions(2): could have. It doesn't need to be a $2M idea to be
- Edit Cut MotherBoard technical discus component of the motherboard
- Edit Cut IT general discussion(3): Get you are looking for...
- Edit Cut XML discussion(4): This discu standards in publishing.

FIGURE 1. The key icon brings up the permissions-editing form for a content object.

For example, the top of the permissions form for editing a Discussion object looks like this:



Edit Permissions

General Information (Discussion)

Owner: **Admin** Advanced

Minimum Permissions

View	Edit	Delete and Cut	Change Permissions	Post	Moderate
Public Access	Owner	Owner	Owner	Logged In	Owner

Set the minimum permissions to Public Access if you want everyone to be able to view (or edit) your Discussion. Alternatively, you can restrict viewing (or editing) to only logged in users, or to you, the owner. If you want to let specific groups or specific people

This form is generated because the developer of the Discussion gizmo has defined bundles for each of the listed classes of operations (*View, Edit,... Post, Moderate*), and has specified which Discussion methods are associated with each bundle. Further down on the form (not shown), is the interface for adding users and groups to the object's *access control list*. The information in this form essentially describes the contents of the object's permissions hash.

3.2 Specifying Access Control for a GizmoBuilder subclass

This section describes in more detail the required interface for specifying a GizmoBuilder subclass's permissions information. Briefly,

- A set of operations bundles must be defined for the class.
- The class must implement a get_default_permissions method which returns this operations object.

3.2.1 Numeric Access Levels and Bundle Permissions

Access levels are represented numerically in Metadot. They are defined in Auditable.pm, and are as follows:

- No Access: 11
- Admin: 10
- Site Manager: 9
- Owner: 8
- Logged In: 2
- Public Access: 0

An `Operations` object stores information about allowed access levels for an object's operations. This is done by defining permissions *bundles*, each of which is associated with names of one or more of the object's *operations.*

Each permissions bundle is defined with a bundle name, a "print name", and the default and minimum permissions levels for that bundle, by using the Operations `add_bundle` method. For example, where `$ac` is an Operations object,

```
    $ac->add_bundle('DISP', 'View', '0', '0');
```

defines a bundle with name 'DISP', which will be shown as 'View' in the permissions editing form. Both default and minimum permissions levels for 'View' operations are 0, which corresponds to 'public access'.

Then, a set of operations is associated with the bundle, by using the operations `add_op` method. For example,

```
    $ac->add_op('DISP','show');
```

sets the 'show' operation to be part of the 'View' bundle. This means that in the permissions editing form for the corresponding object, the permissions settings for 'View' will be used whenever the object gets a request to perform the 'show' operation

The bundle operations effectively describe the external (browser-based) API for the gizmo: these are the operations that a user can request on the object via a CGI query. Each operation in the bundles must be implemented by an object method. **The method name is not the same as the operation name, but rather must be derived from it by prepending 'www_'**. The mapping from operation name to method name is implemented in `Metadotclass`.pm, from which all Gizmos are inherited. In `Metadotclass`, the `handle` method prepends 'www_' to any op passed as a cgi parameter[1]. In practical terms, this means that **when defining an operations bundle for a**

1. This scheme allows future extensions of the display logic in which methods for different display media, in addition to www , can be later developed and mapped to object operations.

gizmo class, there must be a 'www_<op_name>' method in your class for each <op_name> in the bundle.

Figure 2 shows the bundle definitions for GizmoBuilder.pm.

```
sub get_default_permissions {
    my $self = shift;
    my $ac = Operations->new;
    #modify the ac
    $ac->add_bundle('DISP', 'View', '0', '0');
    $ac->modify_bundle('DISP', 'DEFAULTS_TO', 'on' );
    $ac->add_op('DISP','show');
    $ac->add_op('DISP','download');

    $ac->add_bundle('MOD', 'Edit', '8', '2');
    $ac->add_op('MOD', 'create');
    $ac->add_op('MOD', 'modify');
    $ac->add_op('MOD', 'save');
    $ac->add_op('MOD', 'up');
    $ac->add_op('MOD', 'down');
    $ac->add_op('MOD', 'paste');
    $ac->add_op('MOD', 'delfile');
    $ac->add_op('MOD', 'delfileok');
    if ($self->is_gizmo_shortcut_available()) {
        $ac->add_op('DISP', 'create_shortcut');
    }
    $ac->add_bundle('DEL', 'Delete and Cut', '8', '8');
    $ac->add_op('DEL', 'delete');
    $ac->add_op('DEL', 'delete_ok');
    $ac->add_op('DEL', 'cut');
    $ac->add_op_parent_dependency('DEL', 'cut', 'MOD');

    $ac->add_bundle('EDITP', 'Change<br>Permissions', '8', '8');
    $ac->add_op('EDITP', 'edit_permissions');
    $ac->add_op('EDITP', 'set_permissions');
    $ac->add_op('EDITP', 'change_owner');
    return $ac;
}
```

FIGURE 2. An example of the syntax used to define operations bundles for an object. The example shown is from `GizmoBuilder.pm`, which constructs and returns the set of bundles shown above.

3.2.2 Implement `get_default_permissions`

Each GizmoBuilder subclass must implement a `get_default_permissions` class method, which must return the default Operations object to use for that class. Typically, the default Operations object is stored as a class variable. Usually, it will make sense for a given gizmo to build its bundle permissions by starting from the bundles defined by its parent class, and using the Operations methods to add, delete, and modify entries from those bundles. Note that care must be taken to *copy*, or clone, the parent's Operation object before modifications if necessary.

In the case of GizmoBuilder, its get_default_permissions method returns a new default Operations object. Thus, its subclasses may use this operations object as a starting point for modifications without needing to clone it. Figure 3 shows how the Discussion class modifies the Gizmo-Builder's default operations bundle, and then defines its own get_default_permissions method to return the result. [In addition to the 'add' methods shown in this example, the `Operations` class also provides `remove_bundle` and `remove_op` methods, which can be used to edit the parent class' bundle settings as well].

3.2.3 Object permissions instantiation and checking

As described above, each gizmo must implement a `get_default_permissions` method. When a new gizmo is created, it is instantiated with a set of initial bundle permissions. It will inherit bundle permission settings from its parent as appropriate, and uses the defined permissions defaults, via `get_default_permissions`, where it can not inherit from the parent. An object's permissions are represented by an object of the `Permissions` class[1]. These instantiated permissions, for each object, are stored in the `permissions` table in the db.

The **`is_allowed_to_do`** method in `Auditable` is the method which actually implements the permissions checking logic on an instantiated object. This method will obtain the permissions structure for the object, and will check, given a `User` object and an operation, whether or not the user is allowed to do that operation. All Gizmos inherit from Auditable.

All external requests for an object to perform an operation (e.g., a request from a client browser), go through the `index.pl` handler. `index.pl` accesses the referenced object's `is_allowed_to_do` method to decide whether or not to allow the request for the operation.

The following code sample is from `index.pl`. Given an incoming CGI request like this one:

```
http://<your.domain.com>/metadot/index.pl?iid=1118&isa=Discussion&op=show
```

index.pl will make the following check:

```
if ($gizmo_object){
    unless($gizmo_object->is_allowed_to_do($op, $USER, 1)) {
    $object->print_click_back( "Error", "index.pl:<BR>Sorry, you are not allowed to do
operation:<BR> -$op-<BR>$0");
    exit(0);
    }
}
```

If the current user does not have permissions to 'show' the given Discussion, an error page is returned.

1. In the database, an object s permissions information is represented as a frozen hash record in the permissions table. In a future version of Metadot, we plan to pull the permissions info out of the frozen hash representation so that it is more directly queryable.

```
my $default_permissions = Discussion->SUPER::get_default_permissions;

$default_permissions->add_bundle('POST', 'Post', '2','2');
$default_permissions->add_bundle('MODERATE', 'Moderate', '8','8');

$default_permissions->add_op('MOD','modify_message');
$default_permissions->add_op('MOD','save_message');
$default_permissions->add_op('MOD','create_shortcut');
$default_permissions->add_op('MOD','invite');

$default_permissions->add_op('POST','reply');
$default_permissions->add_op('POST','send');
$default_permissions->add_op('POST','compose');

$default_permissions->add_op('MODERATE','moderate');
$default_permissions->add_op('MODERATE','delete_message');
$default_permissions->add_op('MODERATE','delete_message_ok');
$default_permissions->add_op('MODERATE','approve');


sub get_default_permissions {
    return $default_permissions;
}
```

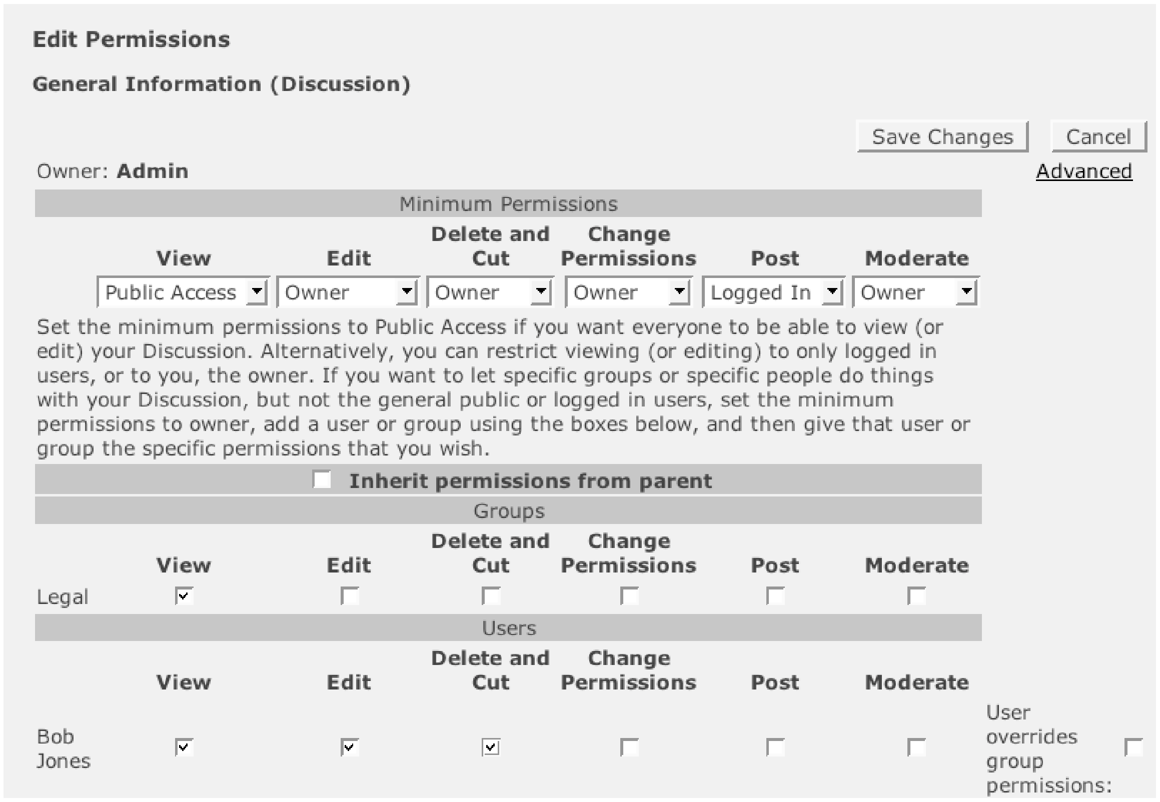
FIGURE 3. Modify a (copy of) the parent class Operations object by adding additional bundles and bundle ops. Then override the parent s `get_default_permissions` method to return the modified info. The example shown is from `Discussion.pm`. GizmoBuilder builds a fresh Operations structure with each call to get_default_permissions. If, however, the parent class were to cache its permissions object as a class variable, then the subclass would need to first *clone* the object before modifying it.

3.3 Groups, Permissions, and Access Control Lists

Via the permissions-editing interface for an object, it is possible to add both groups and individual users to an *access control list* for the object, whereby the users on the ACL can be treated as "owner" for a specified subset of the object's permissions bundles. An object's access control list is stored as part of its permissions hash. The ACL is used by Auditable's `is_allowed_to_do` method to identify if the given user should be considered as "owner" for a given operation.

This interface is supported for all classes that implement the Auditable interface, and thus is supported for all GizmoBuilder subclasses.

In Figure 4 below, the "Legal" group has been added to the ACL but has not yet been given any special permissions, in addition to "View", for the Discussion object. The user "Bob Jones" has been added to the ACL, and can access the ops in the "Edit" and "Delete and Cut" bundles as if he were the owner. However, he can *not* access, e.g. the "Post" permissions as owner.



Edit Permissions

General Information (Discussion)

Save Changes | Cancel

Owner: **Admin** | Advanced

Minimum Permissions

View	Edit	Delete and Cut	Change Permissions	Post	Moderate
Public Access	Owner	Owner	Owner	Logged In	Owner

Set the minimum permissions to Public Access if you want everyone to be able to view (or edit) your Discussion. Alternatively, you can restrict viewing (or editing) to only logged in users, or to you, the owner. If you want to let specific groups or specific people do things with your Discussion, but not the general public or logged in users, set the minimum permissions to owner, add a user or group using the boxes below, and then give that user or group the specific permissions that you wish.

☐ **Inherit permissions from parent**

Groups

	View	Edit	Delete and Cut	Change Permissions	Post	Moderate
Legal	☑	☐	☐	☐	☐	☐

Users

	View	Edit	Delete and Cut	Change Permissions	Post	Moderate	
Bob Jones	☑	☑	☑	☐	☐	☐	User overrides group permissions: ☐

FIGURE 4. Example of adding a group and a user to an object s access control list. Each entity has permissions to perform their checked set of ops as owner .

If the "User overrides group permissions" box is checked, then this means that the specified individual permissions will be applied even if the user is in the group and the group permissions are less constraining.

3.3.1 Recursive Permissions Propagation and Application

Child objects created for a given object will inherit the parent's ACL for the bundles that they have in common, with the child's default bundle permissions used for the remainder. The bundle settings may be further modified manually after creation.

Note that the child permissions are not reapplied retroactively. That is, changing the permissions for a parent after a child has been created will *not* automatically affect the child's permissions. However, there are two ways to apply a parent's permissions to its children, one dynamic and one static. See the "Propagating Permissions Changes" section from the Admin's Guide, at `<metadot>doc/md_guides/install/recursive_perms.html`, for more information.

3.4 The Access Broker

In some cases, you may need finer-grained access control logic than at the operations level. This can happen, e.g., if you want to change part of the implementation of a method based on the current user.

A Metadot::AccessBroker::Default subclass can be defined to support this finer-grained access control. The AccessBroker class can contain the logic for a specific set of access decisions related to a given app or context, and this class can be consulted as necessary. See the code for examples of AccessBroker usage.

3.5 Steps required to build a class which supports AC

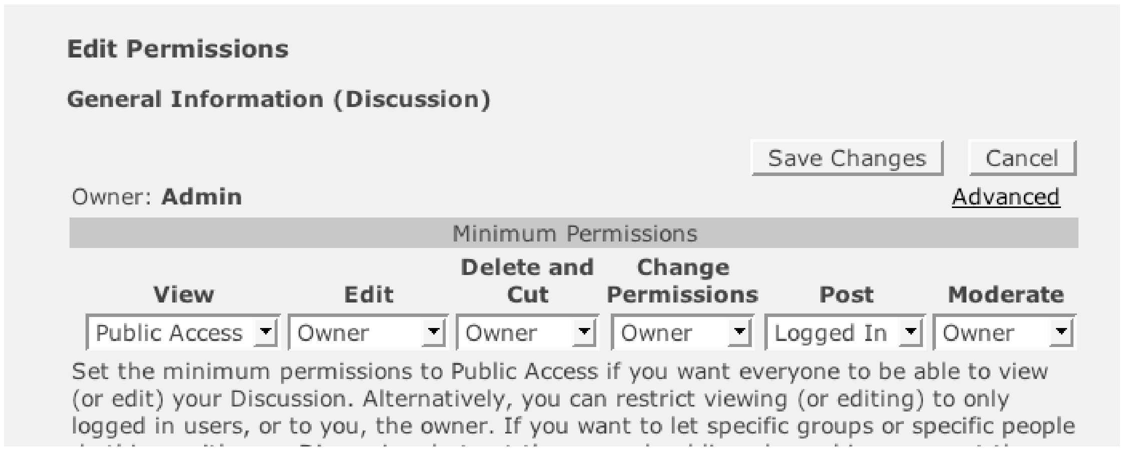
For completeness, this section describes in more detail the steps required to build a class which uses Auditable to support access control, for any case where you are not subclassing GizmoBuilder. In most cases, you will not need to use this information. The preferred means of building new gizmos is by subclassing the **GizmoBuilder** class. In that case, these methods are already implemented, and you need only to follow the instructions in the section above.

For a non-GizmoBuilder class, the steps are as follows:

1. Inherit from the Auditable class by including Auditable in the @ISA array.
2. Implement the following methods:

   ```
   is_a
   name
   id
   save
   uid
   parent_id
   ```

   See Section 3.6 for details on each of them.
3. Generate the Access Control (AC) information for that object. The preferred way to do this is via the Operations object. See the Operations.pm documentation for more information, and see the GizmoBuilder class for an example of its use. [Other Metadot classes still use a 'deprecated' approach and construct a AC hash— essentially the Operations object —by hand (see Gizmo::Category for an example)]. For any new classes, use the Operations object.
4. Build a method called `get_AC` which returns the `Operations` object (the AC hash). The `GizmoBuilder` class implements a method called `get_default_permissions`, which is used to deliver the `Operations` object. If you are subclassing `GizmoBuilder` to build a gizmo, then you can override `get_default_permissions` to add to or modify the `Operations` information for your subclass. See `Gizmo::Discussion` for an example. As discussed in the previous sections, the `Operations` bundles define the set of 'permissions' put-downs that the user will see in the 'edit permissions' form. This is illustrated in Figure 5.


Edit Permissions

General Information (Discussion)

Save Changes | Cancel

Owner: **Admin** | Advanced

Minimum Permissions

View	Edit	Delete and Cut	Change Permissions	Post	Moderate
Public Access	Owner	Owner	Owner	Logged In	Owner

Set the minimum permissions to Public Access if you want everyone to be able to view (or edit) your Discussion. Alternatively, you can restrict viewing (or editing) to only logged in users, or to you, the owner. If you want to let specific groups or specific people

FIGURE 5. The permissions pull-down menus in an object s edit permissions form are determined by its Operations permissions bundles. This set of pull-down menus for a Discussion object matches the bundles shown defined in Figure 2 and Figure 3.

5. Override method `&get_parent` if class objects can have parents. This method must return a parent object which must also belong to Auditable.
6. Call `$self->init_permissions()` at the end of the object's constructor method, after methods in step 2. above are implemented. See the Gizmo class for an example.
7. Call `save_permissions()` from the object's `save()` routine. See the Gizmo class for an example.
8. Call `save_permissions()` from the object's `add()` method (or from whichever method commits a new object of the class to the database).
9. Call `delete_permissions()` from the class destructor.
10. Do permissions enforcement by way of the `is_allowed_to_do()` method (see index.pl and Category.pm for examples of this). One must be particularly careful in sheltering all Access Controlled code sections around `is_allowed_to_do()` conditionals.
11. Implement `www_edit_permissions` and call `edit_permissions_form()` from it (see `Gizmo::www_edit_permissions` for example).
12. Override `&go_back_to_parent_after_editing_permissions` to return false for classes that require browsers to return to the object itself whose permissions are being edited, such as Category.

3.6 The Auditable Interface

This section describes the Auditable interface. This interface is implemented by GizmoBuilder, and all new gizmos should inherit from GizmoBuilder.

3.6.1 Must-Provide Methods

`&get_AC:` Should return the AC hash of the class.
There should be an AC hash for every non-abstract class in the system.

`&is_a:` Should return the class name, as a string.

`&name:` Should return the name of this particular instance of Auditable, which is used to generate the Edit Permissions page.

`&id:` Should return a unique ID associated with this instance.
If only one instance is expected to ever exist for this class this method should return 1.

`&save:` The only requirement for this method is to call &save_permissions.
Normally this method will also save non-Auditable object data to disk (e.g. Gizmo::save).

`&uid:` Should return the User ID for the owner of a particular instance of Auditable.

`&parent_id:` Should return the ID of the parent of a particular instance of Auditable.
If class instances can’t have parents the ID must return 0.

3.6.2 OVERRIDEABLE Methods

`&get_parent:` Should return the Auditable object representing the parent of a particular instance of Auditable. If class objects can’t have parents then it shouldn’t be overridden.

`&go_back_to_parent_after_editing_permissions:` Should return false on classes that require browsers to return to the object itself whose permissions are being edited, such as Category. Otherwise it shouldn’t be overridden.
Should be called from `www_edit_permissions`
(See `Gizmo::www_edit_permissions` for example of its use).

3.6.3 NON-OVERRIDEABLE Methods

`&is_allowed_to_do:` Should be called from all places in the class code where Access Control enforcement is needed.
See `Category.pm`, `index.pl` and `MyPage.pm` for examples of this).

`&edit_permissions_form:` Should be called from `www_edit_permissions` and will return the HTML of the Edit Permissions form, which should be sent to client browsers.

`&init_permissions:` Should be called at the end of the class constructor, right after the “must-provide” methods of Section 3.6.1 are guaranteed to return true data.

`&delete_permissions:` Should be called from the class destructor, but before the “must-provide” methods of Section 3.6.1 cease to work.

`&save_permissions:` Should be called from `save()`
and from any method that commits this objects data to disk for later restoring.

`&get_bundle_level &set_bundle_level &get_bundle_of_op`:
These are used to modify/query minimum permissions associated with
a particular instance of `Auditable`.

4. Customizing Metadot's User/Registration and Authentication Functionality

Metadot's user management classes make it easy to customize Authentication and Registration behavior by subclassing. This means that you, as a developer, can modify Metadot's behavior with respect to Authentication and Registration without needing to modify the existing base classes.

Customization requires subclassing the appropriate User, Register, Authenticator and Ses-sion-Handler classes, and overriding the methods necessary to achieve the behavior you want. Then, information must be placed in the site's `metadot.conf` file to tell the relevant Factory classes which subclasses to return.

Section 4.1 provides a tutorial on customizing User and Register classes. Section 4.2 describes how to use and subclass Authenticator and SessionHandler classes.

4.1 User and Registration Customization: Tutorial

This section provides a tutorial on the process of customizing user information and registration functionality. The user- and registration-related classes are as follows:

`Metadot::User::Default, and its subclasses in Metadot/User/*:`
Metadot::User::Default is Metadot's main user management class. Its main function is to provide a front-end to data storage of info about a user. Its various methods allow adding and deleting users, and editing of users data.

The Metadot::User::FlexUser subclass provides a more flexible definition of the fields in a user's profile. The FlexUser class provides a mechanism for mapping registration/profile fields to fields in an `extended_user` database table. By subclassing FlexUser to change the default mapping, arbitrary registration/profile fields can be defined.

FlexUser is the default class used when the regular login/password authentication method is used for a site—this is the class that will be enabled with an out-of-the-box Metadot install.

Metadot::User::Default is used when LDAP authentication is enabled.

`Metadot::Register, and its subclasses in Metadot/Register/* :`
Metadot::Register handles the registration process. It provides methods to generate and process the HTML form for user registration data. It will invoke Methods of the User class to create users after validating registration form input. The Metadot::Register::FlexRegister subclass is used in the default Metadot install.

Metadot::UserFactory
Metadot::RegisterFactory
These Factory classes return the correct User or Register subclasses based on the configuration information in the site's `metadot.conf` file.

In this tutorial, we will go through the exercise of subclassing Metadot's Default User and Registration classes in order to provide new functionality. The end result of the tutorial exercise will be the 'Metadot::User::FlexUser' and 'Metadot::Register::FlexRegister' classes in the code distribution.

4.1.1 Customization Steps

To build FlexUser and FlexRegister as subclasses of the default User and Register classes, we will need to take the following steps:

STEP 1. Define the configuration data structure.

Subclass `User::Default` to make it store and retrieve fields according to configuration. We will need to extend the storage subsystem to be able to store and retrieve custom user fields.

STEP 2. Override methods for User Registration Form generation and processing.

Subclass `Register::Default` to make it render and process registration forms according to configuration.

STEP 3. Override methods for user data saving and retrieving.

Make sure we override all methods in `User::Default` and `Register::Default` that rely on knowledge of storage details, since we are going to extend the storage subsystem to allow for customizable parameters.

We will call the new User and Registration classes `FlexUser` and `FlexRegister`. These classes are already part of the Metadot main distribution and we are going to refer to their methods constantly during this explanation. Please be sure to have them by your side when studying this tutorial (either by printing them or viewing them in your editor).

Our new User and Registration classes are easily pluggable into the Metadot system, because the user and registration classes to be used are selected at runtime via the system's `etc/metadot.conf` configuration file. Thus, if we want `FlexUser` and `FlexRegister` to be used instead of the Default classes, we do not need to make any configuration changes to the Perl code, but only need to add the following lines to our metadot.conf file:

```
user_type = FlexUser registration_type = FlexRegister
```

Let's move forward with the tutorial.

STEP 1. Defining the configuration data structure.

In order for the registration form to be custom generated, we need to provide a specification of what fields are need in the generated form, and what their names, lengths, and other parameters are. We use a plain perl data structure for this. We have named this structure `$FIELDS_CONFIG` and it currently lives in file `FlexUser.pm`. Please take a look at it. Most of the fields of the fields structure are self-explanatory. Some may not be as clear, so here are some comments about them:

The `DISPLAY_SETS` parameter is used for form generation and is mainly there for cosmetic purposes. Each field in the form will belong to one of the defined display sets. Form fields will then be rendered in a separately-framed set for each defined display set. Fields are assigned to display sets using the "display_set" parameter, which is defined as an index number (first defined set will be 0 and so on). Any number of sets can be defined.

Form fields are defined as an array of hashes under the `FIELDS` key of the data structure. The order in which they are listed in the array will be the order used to display them within the display set where they belong.

The storage type for each field defines where it is going to be stored It can take a value of "primary" and "secondary". The former will be stored in Metadot's primary (legacy) 'user' table, and the latter will be stored according to a mapping in the new "extended_user" table, that we have defined for this new class. The extended_user table (please take a look at its code at the bottom of `FlexUser.pm`) has room enough for a number of 255 character strings and some Text fields of 64K characters. Each "secondary" field needs to be assigned to a slot in this table, and we use the "store_at_column" parameter to define this mapping.

Note that some fields in the structure have a "field_type" set to "textarea". These will be rendered as HTML textareas instead of the default "text" boxes. If no "field_type" is included, the default will be "text". Currently, only text and textarea form fields are supported.

In order for this configuration to be available to consumer classes (i.e., the `FlexRegister` class, which will need it for form generation and processing) we create the `get_fields_config()` method. Please take a look at it. You will note that this method does some validation to make sure that some of the fields are always present in the config data structure. We require these to preserve backward compatibility with the old `Default` class and because those fields are required by many of the system modules.

STEP 2. Override methods for User Registration Form generation and processing.

These methods live in the `Register::Default` class. The methods involved are `Register::Default::www_register_form` (generates the registration form) and `Register::Default::www_register` (will process registration form).

We define methods to override both of these, in a new Register::FlexRegister class. The `www_register_form` method will generate the form according to the fields defined in the `$FIELDS_CONFIG` structure in FlexUser. `www_register` will process this form and will call `FlexUser::add` to have the new data saved. Note how these methods use the configuration data to generate the right HTML and execute the requested validation of form data.

The other two methods overridden in FlexRegister are `save_modify_settings_form` and `show_modify_settings_form`. These methods are in charge of displaying and processing the same user data forms but for the case of users or administrators modifying existing users data.

STEP 3. Override methods for user data saving and retrieving.

In order to allow for customizable fields, we need to implement a storage backend capable of storing data according to a custom-defined mapping. We do this by having our storage table consist of a number of blank columns not tied to any particular user field. What gets stored on these columns will depend on the mapping defined in the `$FIELDS_CONFIG` data structure.

Given that fields are customizable, we will not use hard-wired accessors/mutators for user instances. We instead define the methods `FlexUser::get_value` and `FlexUser::set_value` for this purpose.

We override method `User::Default::add`, which is the method in charge of inserting new user data to the database. We do this so that custom fields that are non-legacy (i.e, those marked as "secondary" in the config data structure) can be saved in the extended_user table. Similarly, we override `User::Default::save` so that fields can be stored in the extended_user table when data of existing users is modified.

The new configuration data structure introduces fields for first, initial, and last names. These are required and come as substitutes of the "fullname" legacy field. Thus, in order for FlexUser to remain backward compatible with clients of the old `User::Default` class, we need to keep supporting the `fullname` field. We do this by overriding the “fullname” accessor/mutator method and by keeping the fullname field of the old user table in sync with the contents of the new first, initial and last name fields.

We also override methods `www_show_users` and `show_user_list`. These methods are used to display the user management console. We override them to provide extended functionality for searching and sorting according to first and last names, to improve on the older “fullname”-based interface.

4.2 Customizing Authentication and Session Handling classes

Authentication is the mechanism through which user-entered credentials (usually a user name and password) are validated. Session handling is the mechanism for establishing users identities across HTTP requests, independently of whether the user is authenticated or not. Metadot allows for easy customization of these functions via its `Authenticator` and `SessionHandler` classes. In this section we will explain how to customize Authenticator functionality via subclassing and we will comment on how the `SessionHandler` class works.

The primary classes are:

`Metadot::Authenticator, and its subclasses in Metadot/Authenticator/*`
`Metadot::SessionHandler, and its subclasses in Metadot/SessionHandler/*`

`Metadot::AuthenticatorFactory`
`Metadot::SessionHandlerFactory`
These Factory classes return the correct Authenticator or SessionHandler subclasses based on the configuration information in the site’s `metadot.conf` file.

4.2.1 Authenticator class

The `Authenticator` class is in charge of authenticating users. It is invoked on every hit to establish the identity of users. It provides methods to generate and process forms for input of user credentials. What particular subclass of authenticator is used should be specified by adding a line to etc/metadot.conf as follows:

```
authenticator_type = UserPassAuthenticator
```

(if none is specified, `UserPassAuthenticator` is used)

The main method you should be aware of in `Authenticator` is `Authenticator::authenticate`. This method is not intended to be overridden. It acts as a template that will call some abstract methods (meant to be overridden in subclasses) to perform authentication. This method will return a User and a Session object to the framework, to be used during the life of the request.

It will be useful to refer to method `Authenticator::authenticate` in `Authenticator.pm` as we discuss it. Authentication happens as follows:

1. The method `Authenticator::determine_action` is called. This method will return a token that will be used for determining whether a new session must be created (if you review the concrete implementation of this method in `UserPassAuthenticator::determine_action` you’ll see that it prescribes session creation immediately after a login form has been submitted).
2. A `SessionHandler` object is created. This will be used to either create or restore a new session depending on the case.

3. If a new session needs to be created, it is so by calling the `create_session` method (to be overridden by concrete Authenticators) and asking the SessionHandler object to keep track of it by invoking its `persist_session` method.
4. If a session id (in the form of a request parameter "key") is available it means that a session already exists for this user, so we restore that by invoking the `restore_session` method (with the session id as parameter) in the session handle object.
5. If a session has neither been created nor restored, we first ask the session handler for one, and if it fails to do so we assume this is the first request of an anonymous user and consequently create a a new session for her, restore the profile of the anonymous user (a system default user) and return these.
6. At this point we certainly have a session object available and use it to restore the user object associated with it.
7. If the user is logged in then we refresh her session at this point. Refreshing means resetting the countdown timer for session expiration.
8. We return the User and Session objects of the authenticated user.

At this point you may be asking: But where are the username and password verified? This step should happen on session creation and should be implemented by concrete Authenticators in method `create_session`. For examples of this please look at the concrete implementations of `create_session` in classes `UserPassAuthenticator` and `LDAPAuthenticator`.

Other important methods in Authenticator are the ones used for generating and processing login forms. We include this function inside Authenticator because any given authenticator may take different input parameters as login credentials (e.g., a smartcard authenticator may only take a single one-time-valid authentication code). The relevant methods to be overridden for this purpose are `show_login_form`, `show_my_page_login_box` (a login box formatted as a MyPage element) and `process_login_form`. Again, please look at concrete implementations of these in `UserPassAuthenticator` and `LDAPAuthenticator` for reference.

A good example of customizing the Authenticator class can be found in the `LDAPAuthenticator` class included with the standard Metadot distribution. `LDAPAuthenticator` is a subclass of the standard `UserPassAuthenticator`. `LDAPAuthenticator` modifies the parent class to have authentication take place against an LDAP server, instead of the statically stored username and password of Metadot's default class.

4.2.2 SessionHandler class

The `SessionHandler` class implements a session handling strategy. That is, it provides the means for identifying users across HTTP requests. Currently we only provide a browser-cookie implementation for this. Other session tracking strategies (such as URL rewriting) should be implemented by subclassing `SessionHandler`. Note that other strategies would probably require making changes in other parts of the framework as well (URL rewriting, for instance, would require that session ids be encoded in all URLs generated in the system, something that is not currently doable from within the scope of `SessionHandler`).

Methods in `SessionHandler` that must be overridden by subclasses are `persist_session` (will start tracking a session across HTTP requests); `delete_session` (will stop tracking a session) and `restore_session` (will restore a Session object previously being tracked by the session handler). Please refer to `CookieSessionHandler.pm` for a concrete implementation of the `SessionHandler` class.

In order to substitute a different subclass of `SessionHandler`, the following line should be added to `<metadot>/etc/metadot.conf`:

```
session_handler_type = CookieSessionHandler
```

(if none is specified, `CookieSessionHandler` will be used)

5. Building Gizmos

As described in Section 1, the `Gizmo` class is the parent class for most of the “information objects” used in the Metadot Portal Server, in particular those objects which can be used to create content on “Category” pages. All Gizmos store their data in the `instance` table.

The **`GizmoBuilder`** class, which subclasses `Gizmo`, provides a set of interfaces and methods to facilitate Gizmo construction. In particular, the GizmoBuilder class will provide a default create/modify interface for the gizmo, so that new gizmos can be created with very little new code.

All new Gizmos should subclass `GizmoBuilder`.

All gizmos (that is, subclasses of GizmoBuilder) must be placed in the `<metadot>/metadot/Gizmo` subdirectory; this allows them to be automatically recognized by the system. All ‘visible’ gizmos in the Gizmo subdirectory will be listed in the “Add New..” pull-down menu that is displayed when site editing is enabled. There are two ways to indicate which gizmos are visible and should be listed in the Add menu. See Section 5.4 for more information. (See the Users’ and Administrator’s guides for more information about the existing gizmos).

This section describes how to build new gizmos by subclassing GizmoBuilder. In addition, it is of course possible to subclass existing gizmos.

5.1 The `instance` table

All gizmos “encapsulate” a record in the `instance` table. All but a set of reserved fields may be used to store data for a gizmo. The following fields in the `instance` table are reserved:

```
+------------------+--------------
| Field            | Type
+------------------+--------------
| UID              | int(11)
| IID              | int(11)
| ParentIID        | int(11)
| IsA              | varchar(20)
```

In addition to its required fields, the instance table has a number of free fields of different data types which may be used to store gizmo data. The free fields are the following. The first group of fields below, such as “name”, “description”, etc. should be used in a semantically consistent manner across gizmos; this allows default display methods to work consistently. See existing gizmos for more information. (However, these fields are not required to be used consistently; only the fields listed above are reserved). The second group of fields below are “extras” which may be used for any purpose.

```
----------------+--------------
Field           | Type
----------------+--------------
Name            | varchar(80)
Description     | text
Cool            | varchar(5)
URL             | varchar(222)
Keywords        | varchar(255)
showfrom        | datetime
file1           | varchar(255)
longdescription | text

t1              | text
t2              | text
t3              | text
```

```
    t4              | text
    t5              | text
    c1              | varchar(111)
    c2              | varchar(111)
    c3              | varchar(111)
    c4              | varchar(111)
    c5              | varchar(111)
    d1              | date
    d2              | date
    d3              | date
    d4              | date
    d5              | date
    i1              | int(11)
    i2              | int(11)
    i3              | int(11)
    i4              | int(11)
    i5              | int(11)
    t6              | text
    t7              | text
    t8              | text
    t9              | text
    t10             | text
```

Of course, it is always possible to create auxiliary tables for use with a new gizmo in addition to using the free instance fields.

5.2 The UploadsManager Utility

Older versions of Metadot allowed only one file “attachment” (or associated file) per gizmo. The location of this file was stored in the `file1` field of the instance record for the object.

In Metadot 5.0 and higher, file attachments are now managed by the UploadsManager class. The UploadsManager handles both *access-controlled* and ‘public’ files. Access-controlled uploads are put in a non-web-server-accessible directory, and accessed (downloaded) via a cgi query, which checks that the user has permission to view the download. Public files are put in a web-server-accessible directory. There can now be multiple associated files, of either type, per gizmo. The file information is now stored in the `uploads` database table. All new gizmos should use the UploadsManager interface for any associated files, as described in the example below.

5.3 The GizmoBuilder class

The `GizmoBuilder` class subclasses the Gizmo class. It provides an easy way to create Gizmos.

`GizmoBuilder` provides a set of methods to

- manage Gizmo data -- both reading from and saving the data to the database; and
- to generate the forms for creating and modifying new instances of the Gizmo.

The developer then just needs to:

- define the which fields of the instance table will be used by the gizmo, and
- write display methods for the Gizmo (that is, to define how a gizmo instance is rendered on a web page).

Using `GizmoBuilder`, a new gizmo can be created in a few steps, listed below. There are also some optional steps that can be taken to modify Gizmos if developers wish to change the default settings provided to them by `GizmoBuilder`.

5.3.1 Write a constructor and specify field mappings

Gizmos can use all the fields of the `instance` database table. So, in the constructor, specify which of those fields you will be using in your gizmo. At the same time, specify labels for those fields, and specify which fields are required. By doing so, you allow the Metadot framework to automatically generate a form for your gizmo that asks people to fill in the appropriate fields, and ensures that required fields are actually filled in.

5.3.1.1 Mapping instance fields to form fields

The following code from `GizmoBuilder` shows the default form type mapping used by `GizmoBuilder`, which associates fields in the `instance` table with a *"formtype"*. This formtype is translated, by the `GizmoBuilder` when constructing a gizmo **create** or **modify** form, into a form input element with specific characteristics. For example, a formtype of "smalltext" corresponds to an input element of type `text` and `maxlength` 10, and a formtype of "longtext" corresponds to an input element of type `textarea`. See the GizmoBuilder code for translations of all the formtypes.

```
sub _initialize {
    my $self = shift;
    $self->{_intro_text_string} = "";

    ## set the form type for each field
    $self->{fields}->{name}->{formtype}              = "text";
    $self->{fields}->{description}->{formtype}       = "mediumtext";
    $self->{fields}->{long_description}->{formtype}  = "longtext";
    $self->{fields}->{cool}->{formtype}              = "radio";
    $self->{fields}->{url}->{formtype}               = "text";
    $self->{fields}->{keywords}->{formtype}          = "mediumtext";
    $self->{fields}->{show_from}->{formtype}         = "smalltext";
    $self->{fields}->{t1}->{formtype}                = "longtext";
    $self->{fields}->{t2}->{formtype}                = "longtext";
etc...
```

When building a new gizmo, if the defined form types are sufficient, then it is only necessary to define the set of instance record fields that will be used for the gizmo (in addition to the required fields), and to associate a form label with each of them. This is accomplished by the `set_field_info` method. It takes arguments a set of triplets, where the first element in each triplet is the name of the field (e.g. 'name' or 'description'), the second element in each triplet is the label to be used in the gizmo's **create** and **modify** forms for that field, and the third is the number 0 or 1. A '1' means that the form field is to be treated as 'required', meaning that if the user must fill it in.

```
    $self->set_field_info (
        "name",        "<b>Calendar Name</b>",              1,
        "description", "<b>Description</b>",                0,
        "keywords",    "<b>Keywords and Synonyms</b>",      0,
        "c3",          "<b>Show Event Quick List</b>",      0,
        "c1",          "<b>Title row background color</b>", 0,
        "c2",          "<b>Footer background color</b>",    0,
        );
```

The gizmo developer may also want to redefine some of the preset form types for the gizmo fields. This can be done via the `set_field_type` method. For example, the code below redefines the 'c3' field to be of type 'checkbox' instead of the default 'smalltext'.

```
        $self->set_field_type('c3','checkbox');
```

The allowable formtypes are the following:

- `smalltext, hidden, text, mediumtext, longtext, radio, checkbox, select, file`, and **`other`**.

The 'other' formtype is a special case. No defaults are used; the form element HTML must be explicitly specified. 'Other' form fields are set using the `set_field_form_text` method. It takes two arguments: the field name, and the form text to use. The method sets formtype to 'other'. For example, the following two lines set the 'c1' and 'c2' fields to HTML generated by a special-purpose instance method for the example gizmo.

```
        $self->set_field_form_text('c1', $self->_get_form_text('c1'));
        $self->set_field_form_text('c2', $self->_get_form_text('c2'));
```

5.3.1.2 Example of `GizmoBuilder` subclass constructor

The constructor for a `GizmoBuilder` subclass should define the form field info and do some other bookkeeping. Here is an example constructor (the line numbers, of course, would not be used in actual code). This is in fact the *Item* constructor. The Item gizmo is a simple gizmo which allows the user to specify a name, abstract (short description), longer description, and and optional downloadable file attachment.

```
    1.  sub new {
    2.      my $proto = shift;
    3.      my $id = shift;
    4.
    5.      my $class = ref($proto) || $proto;
    6.      my $self;
    7.
    8.      if (defined ($id)){
    9.       $self  = $class->SUPER::new($id);
    10.     } else {
    11.       $self  = $class->SUPER::new();
    13.       $self->{cool}='No';
    14.     }
    15.
    16.     $self->{is_a} = __PACKAGE__;

    17.     $self->set_field_info (
    18.         "name",        "<b>Item Name</b>", 1,
    19.         "url",         "<b>URL</b> <i>e.g. http://www.yourcompany.com</i>", 0,
    20.         "description", "<b>Description</b>",   0,
    21.        "item_attachment_file",   "<b>Attachment file</b>",  0,
    22.         "keywords",    "<b>Keywords and Synonyms</b>", 0,
    23.         "cool",        "<b>Star this Item</b>",  0,
    24.         );

    25.     $self->set_field_type( 'item_attachment_file', 'user_selected_upload' );

    26.     bless ($self, $class);        # reconsecrate
    27.     return $self;
    28. }
```

Here is a line-by -line explanation of the code above.

1. This line names the subroutine. The new subroutine will be called by the metadot framework when it instantiates your gizmo

2. This line assigns the name of the class to the `$proto` variable

3. This line assigns the id (an optional parameter) to the `$id` variable

5. This line ensures that if the class name passed in was actually a reference to a class, we use the actual class name.

8-14. These lines are important: if an id is passed in, that means we are instantiating a gizmo which has previously been created and then stored into the database. If an id is not passed in, we are instead instantiating a new gizmo.

13. This line means that gizmos are not starred by default

16. This line takes the package that the gizmo is in (such as Item, or Table), and assigns it to the `is_a` instance variable

17-24. These lines specify which fields of the `instance` table are going to be populated for this particular gizmo class, and the form field labels to use with them. The first parameter is the name of the `instance` table field. The second parameter is the label, used when displaying forms when the item is created or edited. The third parameter is set to 1 if it is a *required* field. Otherwise, it is set to zero. (If a field is required, then the form submission will generate an error if it is left blank). The exception in this list is the "item_attachment_file" field. It does not map to an actual instance table field, but instead will be mapped to an entry in the `uploads` table. This is set via the code in line 25 (see below).

25. This line indicates that the 'item_attachment_file' field is of type 'user_selected_upload', which means that it will be handled by the `UploadsManager`. This example just shows one field of type 'user_selected_upload', but potentially a gizmo may support multiple uploads fields.

26. This line makes `$self` into a perl object.

5.3.2 Define display methods for the gizmo

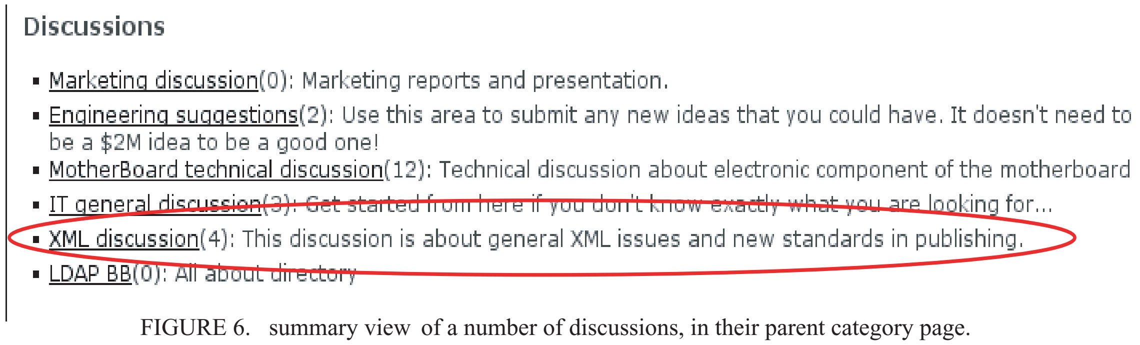
In addition to the constructor, a subclass of `GizmoBuilder` should define the following methods:

```
show_summary
show
```

These are the canonical display methods for a new Gizmo. If no operation on a gizmo is specified, 'show' is the default.

The simple example below shows 'hard-coded' generation of HTML. It is, of course, possible to use other approaches to generate the object's "display" HTML instead. For example, some Metadot gizmos use a Template-Toolkit template to build their HTML.

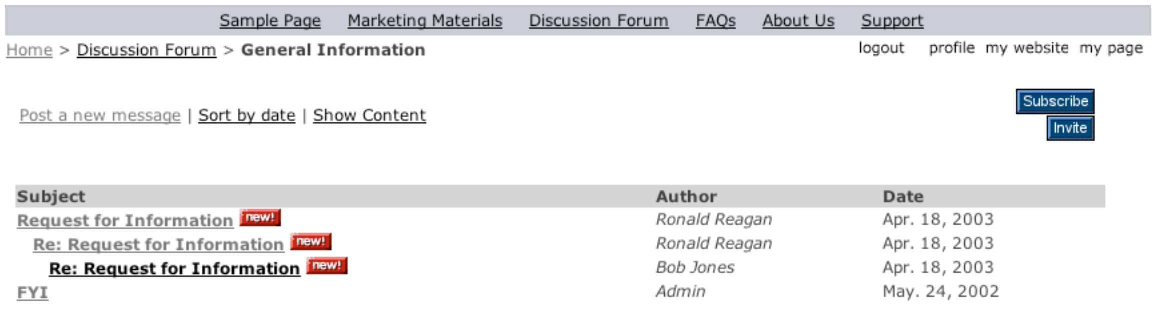
`show_summary` is used to render a 'list view' of the gizmo in its parent page, as in the screen shot below. Often, the 'summary view' for a gizmo will include a link that when clicked on invokes the gizmo's `show` method. See the `show_summary` method in `<metadot>/Gizmo/Discussion.pm` for a relatively complex example.


Discussions

- Marketing discussion(0): Marketing reports and presentation.
- Engineering suggestions(2): Use this area to submit any new ideas that you could have. It doesn't need to be a $2M idea to be a good one!
- MotherBoard technical discussion(12): Technical discussion about electronic component of the motherboard
- IT general discussion(3): Get started from here if you don't know exactly what you are looking for...
- XML discussion(4): This discussion is about general XML issues and new standards in publishing.
- LDAP BB(0): All about directory

FIGURE 6. summary view of a number of discussions, in their parent category page.

The **show** method is used to display the 'full' view for a Gizmo, as in the screen shot below. For example, for a Discussion gizmo, the 'show' method is called when a user clicks on one of the discussion links on a category page.


Sample Page Marketing Materials Discussion Forum FAQs About Us Support
Home > Discussion Forum > General Information
logout profile my website my page
Post a new message | Sort by date | Show Content
Subscribe
Invite

Subject	Author	Date
Request for Information new!	Ronald Reagan	Apr. 18, 2003
Re: Request for Information new!	Ronald Reagan	Apr. 18, 2003
Re: Request for Information new!	Bob Jones	Apr. 18, 2003
FYI	Admin	May. 24, 2002

FIGURE 7. full view of a discussion, rendered by the Discussion s show method.

In writing the display methods, you can use *getter* methods to retrieve the data for the gizmo, where the data is persisted as a field in the `instance` table, and is instantiated by the GizmoBuilder class. A list of all *getter* and *setter* methods is given in the SYNOPSIS of the POD documentation for GizmoBuilder.

Below is an example of a `show` method, again for the Item gizmo. This method displays the Item's name, and (if set), its abstract, description, and a download link to the gizmo's file attachment.

```
sub show {
my $self = shift;

my $name            = $self->get_name();
my $description     = $self->get_description();
my $url             = $self->get_url();
my $keywords        = $self->get_keywords();
my $file            = $self->get_file_1();
my ( $file, $file_icon ) = $self->get_upload_filename_icon_and_url( {
                          id => $self->id(),
                          isa => $self->is_a(),
                          field_name => 'item_attachment_file',
                        } );
10.    my $cool_icon       = $self->get_cool_icon();
11.    my $longdescription = $self->get_long_description();
12.    my $html            = '';
13.
14.    $html .= ($url) ?
    "<a href='$url'>$name</a>\n" : "<b>$name</b>\n";
15.    $html .= $cool_icon;
16.    $html .= "<BR>$description\n" if ($description);
17.    $html .= "<P>$longdescription\n" if ($longdescription);
18.    $html .= "<BR><BR>File attached: $file $file_icon\n" if ($file);
19.    $html .= "<BR><BR>Synonyms: $keywords\n" if ($keywords);
20.
21.    return $html;
22. }
```

Line-by-line explanation:

1. This line names the subroutine

2. This line assigns the object to the `$self` variable. `$self` is analogous to the 'this' object in Java.

4-11. These lines retrieve values from the object into variables. Note line 9. in particular. This line handles the special case of a file attachment associated with the object via its "item_attachment_file" field. Retrieval of the file attachment information is handled ultimately via the UploadsManager utility. If the object had been defined to have more than one uploaded file associated with it, then a call to `$self->get_upload_filename_icon_and_url` would be made for each such field. The icon that is returned indicates the file type (derived from the file suffix).

14-19. These lines use the variables previously retrieved to generate HTML that contains those variables. It is, of course, possible to use other approaches to generate the object's "display" HTML. For example, some gizmos use a Template Toolkit template to build their HTML.

In addition to the two methods above, other display methods may be written. *The name of any new method must start with the prefix 'www"*. For example, one could write a method called `www_show_details`, which would provide a way to show additional fields that you have chosen not to display in your `show` method. You can then do all of the processing in the `www_show_details` method, or it can call additional helper methods.

The 'www' prefix is required in order to utilize Metadot's 'rendering' framework. In `Metadotclass.pm`, from which all Gizmos are inherited, the `handle` method listed below prepends 'www' to any op passed as a cgi parameter. (In the future, the framework can be extended to support other types of rendering in addition to HTML).

```
sub handle {
    my $self = shift;
    my $op = shift;
    $self->debug_msg(3, "Portal::handle($op)");
    ## for HTML RENDERING
    my $handler = "www_$op";
    $self->$handler();
}
```

Therefore, the name of any new display methods, or other external interface methods, that you write for a new gizmo must prepend 'www' as well. Below is the `www_show` method for the Gizmo class, from which all gizmo subclasses are inherited. `www_show` calls the '`show`' method implemented for the given gizmo subclass.

```
sub www_show {
    my $self = shift;
    $self->debug_msg(3, "Gizmo::www_show()");

    my $iid = $self->defined_or_exit($self->id,"id is missing.");

    my $title = $self->name();
    my $content = $self->show();
    $self->print_portal($content);
  }
```

5.3.3 Gizmos and Access Control

When adding new `www_` methods, there is an additional issue that must be addressed. As described in Section 3, Metadot security is based on the idea of controlling access to the `www_` methods— essentially, the methods that define the gizmo's external, browser API. These methods are grouped into *bundles,* with one or more *operations* per bundle. The operations map to methods, and permissions are assigned to bundles. So, a the operation corresponding to a new method, e.g. the *show_details* operation corresponding to the `www_show_details method`, would need to be added to either a new or existing operations bundle. If you do not remember to add a new operation to one of the gizmo's bundles, then you will be blocked from calling that operation from the browser interface.

The Operations class stores the association between the operations (or subroutines) and bundles. To get an operations instance, call the get default permissions as a class method on the superclass of your gizmo (GizmoBuilder). That call looks like this:

```
  my $default_permissions = Discussion->SUPER::get_default_permissions;
```

Once you have the object, you can modify it as described in the documentation of the Operations class. See Gizmo::Discussion for an example of how the GizmoBuilder module's default permissions can be further modified.

Note that both the retrieval and the modification of the operations object take place as class operations.

Once you are finished modifying the Operations object, you need to make it available as a class method call named `get_default_permissions`, as below. Note that this class method will override the superclass' method of the same name:

```
  sub get_default_permissions {
      return $default_permissions;
  }
```

5.3.4 Overriding Existing GizmoBuilder Methods

In addition to writing your own new methods, such as `www_show_details`, you can override existing `www_` methods which are inherited from GizmoBuilder. Documentation on these methods is found in the documentation of the Gizmo and GizmoBuilder modules.

These methods are:

```
      www_save();
      www_delfile();
      www_delfileok ();
```

```
        www_show ;
        www_change_owner ;
        www_edit_permissions ;
        www_modify ([ $errormsg] );
        www_save ;
        www_delete ;
        www_delete_ok ;
        www_up ;
        www_down ;
        www_cut ;
        www_paste ;
        www_download ;
```

5.4 Making new Gizmos available to the system

All new Gizmos must be put in the Gizmo subdirectory; this will allow the system to automatically identify them. Any new gizmos will by default appear in the "Manage.." menu listing that appears for a user when editing is enable. If a gizmo of the new Gizmo type is created for a category, it will appear in the center "content" panel for that category (where Tables, Discussions, and Items also appear). The content is grouped by Gizmo type; that is, all *Items* are displayed together; all *Discussions* are displayed together, etc.

To have more extensive control over where gizmos are placed on a page, the use of *templates* and gizmotags is required. See Section 8 below for more information.

6. Basic Tools

Basic Tools can appear on a MyPage, and are essentially just HTML that implements a search or query interface of some sort. The information comprising a Basic Tool is stored in the `instance` table in the portal's database, not as a `channel` (since the Basic Tools have no cached information).

To modify and add Basic Tools, log in as Admin, then select Manage..->Toolbox from the Manage.. pulldown menu. From the resulting page, click on the subcategory "**Generic Elements for MyPage**". All of the Items on this page correspond to "Basic Tools" — the HTML in the description field for each Item is used to display that tool on the MyPage. From this page, edit the existing Basic Tools Items, or add a new Basic Tool by adding an Item to the page, and *inserting the HTML that makes up the Basic Tool* as the Item's description. Any new Items added to this page will then appear in the "Tools" list accessed from the MyPage Configuration form.

7. GizmoTools

All GizmoTools are specializations of the GizmoTool class, and must reside in the `GizmoTool` subdirectory for them to be identified as available for use on a *MyPage*. For example, the `GizmoTool` subdirectory currently contains two GizmoTools; WebMonitor and WeatherTool. The `doc` subdirectory of `GizmoTool` contains a sample GizmoTool called `SampleTool.pm`, which indicates the methods that need to be implemented by the GizmoTool subclass.

Each GizmoTool must implement a `refresh_item` method, as described in `SampleTool.pm`. This method is invoked when the "backend" process, metadotd.pl, runs, and populates channel information for the GizmoTool at that time. The GizmoTool must implement a display method, and the channel info plus the GizmoTool parameter settings and display method determines how the channel is rendered on a MyPage.

The GizmoTool configuration parameters are stored in the database in the `gizmoitemparam` table. A corresponding `channelitem` entry stores the result of running the `refresh_item` method.

8. GizmoTags and Templates

GizmoTags and *templates* provide a way to change the page layout and "look" of Metadot pages. GizmoTags are constructs embedded in the HTML templates, and are *expanded* during the page rendering process to insert content inline. The gizmo tags each correspond to a perl module, as described in more detail below, and it is the perl module which implements the generation of the inline content. The term 'gizmo tag' is a bit of a misnomer: the gizmo tag module may and typically does generate content pertaining to a gizmo object, but it may also generate arbitrary content as well (for example, one gizmo tag inserts a "fortune" into the page).

The gizmo tags correspond to modules defined in `<metadot>/metadot/GizmoTags`. When a display request is received, a gizmo tag-enabled template is "expanded" by executing the 'exec' method corresponding to each tag object, and inserting the resultant HTML in the template.

Templates, and the use of GizmoTags, are activated by choosing "Style 4" in the `Config->Style&Colors` page. It is suggested that you start with the default templates shown there and modify them to suit your needs. As is described below, the gizmo tags embedded in the default templates are what determine the page content rendered using the templates, and therefore, as you edit the default templates, make sure that you do not inadvertently delete any gizmo tags which provide content that you want to retain.

The first template, '`maintemplate`', is used for creating Metadot's front page. The second template, '`subtemplate`', is used for subcategory pages. The third template, the '`utility`' template, is used to display the 'detailed view' of all gizmos except Category. E.g., the utility template is the one used if you execute the '`show`' op for a Discussion object.

You can make your site look different by changing the places from where GizmoTags are called, by adding or removing GizmoTags, or by changing the HTML tables layout in the templates.

GizmoTags also provide a way for developers to distribute Metadot enhancements in simple and self-contained manner. Developers only need to put their code in a GizmoTag perl module and distribute it. Site administrators can then copy this module to their metadot install directories and add a GizmoTag to render it to their site templates. For more on how to program GizmoTags see the "Developing GizmoTags" section below.

8.1 Gizmo Tags in Page Description Fields (Advanced)

While not documented further, it is possible to use gizmo tags in page description   fields.

However, two important warnings must be noted:

Gizmotag processing does not currently guard against loops generated by using the "wrong" gizmo tags in a description field. For example, using a md_catdesc tag (see list below), which generates the page description, within a page description, will cause an infinite loop when you try to display the page.

Gizmo tags in a description field are NOT preserved if a page is edited using Windows IE. This is because in Windows IE, a WYSIWG HTML editor is used to edit the page's description field. (Thanks to **interactivetools.com**). This editor will remove any non-standard HTML. You will need to add and edit gizmo tags in a description field using a different browser.

Update: As of Metadot 5.5.2.1, it is now possible to use square brackets as well as angle brackets for specifying gizmo tags. This allows the gizmotags to be preserved in a page description field

when using the WYSIWYG HTML editor. For example, this construction will now work:
`[ gizmotag name="md_quota_meter"][ /gizmotag]`

To see GizmoTags in action, follow this simple tutorial:

8.2 Gizmotags Tutorial

The following set of steps will illustrate from a user standpoint how gizmotags work. Then, the subsections below describe in more detail how they are implemented.

1. From the main page, log in as “admin” and then Enable Editing

2. From the “Manage” menu go to “Style & Colors”.

3. Select “Style 4” on the Global Look selector. This will make the system generate its pages from the templates that follow.

4. There are three templates in Style & Colors: ‘maintemplate’, ‘subtemplate’, and ‘utilitytemplate’. The first one is used for generating the front page. Let’s work with that.

5. Cut and paste the Template for the Style & Colors page to the HTML or text editor of your choice. (This will make the template easier to edit than in the browser text area). .

6. Lets try moving the news column to the left side of the page. We assume that you are starting from the template that comes with Metadot’s default content.

7. In the template file in your editor, search for the GizmoTag for rendering the news column, or ‘newscol’. The name of the GizmoTag is **md_newscol** (search for the ‘md_newscol’ string). You’ll see start and end markers in HTML like this:

```
   <!-- GIZMOTAG : md_newscol : Begin -->
   <!-- GIZMOTAG : md_newscol : End -->
```

These are just comments placed there to let you easily find GizmoTags in your template. The only requirement for GizmoTags to work is that the GizmoTag tags themselves be present, like follows:

```
<GIZMOTAG NAME=”md_newscol”>
</GIZMOTAG>
```

The tag element is GIZMOTAG; the ‘name’ attribute is the name of the tag. This name must correspond to the name of some module in `<metadot>/metadot/GizmoTags` for the template to find and expand the tag.

Note that the template in the default metadot content also includes HTML between these two tags, similar to the content that will be rendered when the template is interpreted. This chunk of HTML between an open and close pair of gizmo tags is optional, and will be *removed* when the tag is “expanded” and the content from the expansion inserted. It can be useful to have this default HTML in the template in order to allow an HTML editor to render the template similarly to how it will look when all its tags are expanded. If the referenced gizmotag object (e.g. the “md_newscol” object in this example) does not actually exist, then the tag will *not* be expanded. In this case the default HTML will be retained and displayed.

8. For this example, suppose we want to move this gizmo tag so that it appears in the same place where the ‘`md_v_subnav`’ GizmoTag is. ‘`md_v_subnav`’ is the one in charge of rendering the vertical subcategories column. We need to create an html table around the ‘`md_v_subnav`’ GizmoTag

and move the 'md_newscol' GizmoTag to a row in it. The layout that you should put in the template where 'md_v_subnav' is should look like this:

```
<table>
    <tr>
        <td>
            <GIZMOTAG NAME="md_v_subnav">
            </GIZMOTAG>
        </td>
    </tr>
    <tr>
        <td>
            <GIZMOTAG NAME="md_newscol">
            </GIZMOTAG>
        </td>
    </tr>
</table>
```

For clarity we have left out the HTML comments and the default HTML within the gizmo tags.

After you do this change, copy the template back to the Colors & Styles page, save and return to the main page. That's it! You should now see that the newscolumn appears on the left side.

8.3 Changing Gizmotag Styles

Each GizmoTag is defined to have an associated CSS style class name. For the associated style class to have any effect when the gizmo tag is used, a style rule for the class must be defined in the HTML. With templates, the "Style Sheet" box in the Manage..->Styles & Colors config page is *not* used. Instead, any style rules used within a template must be defined in the template HTML itself. To do this, you must identify the style name for the GizmoTag you want to modify, and then define a style for it in your template (some gizmotag styles are already defined in the default templates). You will find GizmoTag style names listed in a column in the GizmoTag browser, accessed through the Gizmo Browser page mode. (See Section 11 for more information).

8.3.1 Example:

'`md_date`' is a GizmoTag that will display today's date and time when rendered. To change the font style and color for the '`md_date`' GizmoTag, go to `Manage->Style&Colors` and enter the following line inside the `<style></style>` section for the template you are modifying:

```
.tagDateClass {  font-family: Helvetica, Verdana,Arial,sans-serif;
font-size: 10pt; color: red;}
```

Make sure your Global Look style is set to 4, and save your template. You'll notice that the date now appears in red and with a smaller type.

8.4 Metadot Gizmo Tags

The following are the GizmoTags that come bundled with the Metadot distribution. These tags correspond to modules under the `<metadot>/metadot/GizmoTags` directory. Any of them may be embedded in a template via a GIZMOTAG as described in the examples above. To find other useful GizmoTags, please visit the Gizmo Gallery section at metadot.net.

Some of these gizmo tags take additional attribute “arguments” in addition to the required name attribute. (For example, see the `md_imagesrc` gizmo tag below).

All gizmotags may be passed the optional attribute/value pair **`no_comments=”1”`**. For example:

```
   <gizmotag name=”md_title” no_comments=”1”>...</gizmotag>
```

The ‘no_comments’ attribute suppresses the comment lines that are normally placed around each expanded gizmotag in the html. Sometimes the comments are detrimental, for example if you wish to place the gizmotag expansion results in a <title>..</title> HTML element. The ‘no_comments’ option works for any gizmotag, including ones that you might write yourself.

NOTE: All gizmo tags starting with ‘md_’ are considered reserved for tags that are distributed in the Metadot core (ie. your GizmoTag names shouldn’t start with ‘md_’ unless they are intended to be in the metadot core). See the Gizmo Gallery section in metadot.net for existing GizmoTags to make sure the name you choose is not already taken.

`md_adminbar` : Will display a bar with two combos for adding gizmos to a page and for going to different management sections. It will also display a button to enable/disable Edit mode. The two combos will only show up when in Edit mode.

`md_applicationlist`: display Applications, the list of all the applications within the category

`md_calendar_month`: displays the month view of a calendar for navigation.

`md_catlist`: display categories, the list of all the sub-categories within the category.

`md_catdesc` : Will display the description field of the current category.

`md_catname`: Will display the name field of the current category.

`md_catpath`: Will display the path of clickable names from home to current category.

`md_gizmopath`: Will display the path of clickable names from home to current gizmo. A more general version of the `md_catpath` gizmotag.

`md_clipboard`: displays the clipboard interface

`md_cat2col`: Will display the categories and subcategories under the current category as a wide two column list.

`md_date`: Will display today’s date and time (e.g.. Sun Oct 7 16:55:57 2001).

`md_discussionlist`: display Discussions, the list of all the discussions within the category

`md_editpanel`: Generates the ‘edit panel’ info for the current object — the edit, permissions, cut, etc. icons.

`md_fortune`: Display a random (sometimes funny) message.

`md_h_topcat`: Displays a horizontal list with the topmost categories.

`md_h_topnav`: Displays a horizontal list with subcategories of the current category.

`md_imagelist`: Display list of all ImageItem children of the category

md_imagesrc: Given image name passed as an ‘image’ attribute, generates full server path to image (as configured when the site was set up). You will want to use this gizmotag with the ‘no_comments’ attribute as described above to suppress the extra comment output. For example:

```
<gizmotag name=”md_imagesrc” image=”xyz.gif” no_comments=”1”>..</gizmotag>
```

md_itemlist: Display list of all Item children of the category

md_gizmolist: Display the list of all the gizmos within the category by gizmo type (excepting Categories, Polls and News Gizmos).

md_newslist: Displays a list with all newsitems in the current category.

md_newscol: Display newsitems under the current category in column format.

md_newsitemlist: Display list of all News Item children of the category

md_poll: Displays all polls under the current category in vertical column format.

md_pollist: Display list of all Poll children of the category

md_scrollviewlist: Display list of all ScrollView children of the category

md_tablelist: Display list of all Table children of the category

md_title: Generates the name (title) of the current object. If you are using this gizmotag to generate the title for an html template page, call it with the ‘no_comments’ attribute as described above to suppress the extra comment output.

md_username: Displays the name of the current logged in user.

md_quota_meter: Displays the remaining disk quota for file uploads of the current user.

md_v_subnav: Display a vertical navigation bar with subcategories of the current category.

md_v_subnav_flex: Display a parameterizable vertical navigation bar with subcategories of the current category. This tag supports the following parameters, passed as additional gizmotag attribute/value pairs:
- `show_parents`: tells the nav bar to show the parents of the current page
- `unconditional_expansion_depth`: tells the nav bar how many levels should be unconditionally expanded.

md_v_subnav2: Displays a vertical navigation bar with subcategories of the current category.

md_h_applist: Displays horizontally the applications list.

md_search: Displays the search box. Can take optional attribute `img_src` specifying an image to use for the search button. For example,
```
<gizmotag name=”md_search” img_src=”../images/search_button.gif”>...</gizmotag>
```
Of course, the image must be installed in the specified directory in the web server.

md_todays_events: displays the event listing panel for the current date

md_welcome: Displays the currently logged-in username and a link for signing in.

md_users_online: Indicate how many users are currently logged in, and have been active within the last 15 minutes.

`md_links_panel`: Displays a panel images linked to user functions. If the user is not logged in, the panel contains links for: ‘**register’, ‘login’, ‘my website’,** and **‘my page**’. When the user logs in, the panel contains links for: ‘**logout’, ‘profile’, ‘my website’,** and **‘my page**’.

`md_gizmorunner`: This tag allows displaying objects other than the current object inline. This is done by specifying the type (the ‘isa’) and iid of the object that you wish to display inline, as so:

```
  <gizmotag name=”md_gizmorunner” isa=”Table” iid=”1234”>...</gizmotag>
```

You may optionally include an “op” attribute also; if this value is not specified, then ‘show’ is used as the default. (To find the iid — the unique id — of the object, it is simplest to just display it in your browser and note the iid in the URL generated for the display operation).

8.5 Developing Gizmo Tags

Creating a GizmoTag requires only that you know where or how to get the content it will display when rendered. Typically, all infrastructure to generate and render the content is already part of the system functionality. You will usually retrieve content by calling methods in modules within Metadot or from your own extensions to Metadot. You could also get data by directly querying the database, but we don’t recommend this approach because the Metadot’s database structure is subject to change from version to version.

GimzoTags are perl modules that inherit from the GizmoTag class. Inheriting from the GizmoTag class and overriding a couple of methods of it is the only requirement for GizmoTags.

GizmoTag modules should be placed under the GizmoTag directory in your Metadot install directory. GizmoTags will be recognized and added to the system automatically when placed under this directory.

8.5.1 Two required GizmoTag methods

There are two GizmoTags methods that must be overridden (implemented) in any subclass. One of them is the ‘`new`’ method, which is called internally to create a new instance of this GizmoTag. The other one is the ‘`md_tag_exec`’ method, which is invoked by the system to render the GizmoTag.

The ‘new’ method is used mainly to describe what the GizmoTag does so that it can be placed in the “installed GizmoTags” browser that appears for site Admins when Edit mode is enabled.

The ‘`md_tag_exec`’ method is in charge of generating the HTML content of the gizmo tag. This method generates content data, often by accessing other Metadot methods, and then formats the content for display as HTML.

Let’s look at the complete code of a GizmoTag so you get the idea. The following GizmoTag will display today’s date when rendered.

```
package md_date;
use GizmoTag;
@ISA = (“GizmoTag”);
use strict;

sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new();
    $self->{name} = ‘md_date’;
    $self->{version} = ‘$Revision: 1.3 $’;
12:      $self->{authorname} = 'Mirko Scavazzin';
13:      $self->{email} = 'mirko@metadot.com';
14:      $self->{url} = 'http://www.metadot.net/';
15:      $self->{license} = 'GNU/GPL';
16:      $self->{styleclass} = 'tagDateClass';
17:      $self->{embeddedstyleclasses} = [];
18:      $self->{description} = 'return local time and date (i.e. Sun Oct 7 16:55:57
2001)';
19:      bless ($self, $class);
20:      return $self;
21:  }
22:
23:  sub md_tag_exec {
24:      my $self = shift;
25:
26:     return "<DIV><SPAN CLASS=" . $self->get_styleclass() . ">" . localtime(time) .
"</SPAN></DIV>";
27:  }
28:
29:  1;
```

Line 1 is the package name, and is also the name by which the gizmo tag will be invoked from templates. It is important that the name you use is not already taken by other GizmoTags. All gizmo tags starting with 'md_' are considered reserved for tags that are distributed in the Metadot core (i.e., your tags shouldn't start with 'md_' unless they are intended to be in the metadot core). See the Gizmo Gallery section in metadot.net for existing GizmoTags to make sure the name you choose is not already taken.

Lines 2 and 3 are for inheriting from class GizmoTag and should always be present.

On Line 6, subroutine 'new' is a standard object oriented perl constructor.

From lines 10 to 20 we fill in necessary data for the GizmoTag. Of these maybe the only fields not self-evident are 'url', 'styleclass' and 'embeddedstyleclasses'.

On line 14 'url' is the URL where you can find the latest version of this GizmoTag. We recommend that you upload your GizmoTags to the Gizmo Gallery in metadot.net if you want to make them available to the public as OpenSource.

On line 16 'styleclass' is the name of the CSS styleclass definition that has to be included in the template <styles> section to override the default styles in this GizmoTag. The name you put in this field should be used as the style name in the HTML generated in the 'md_tag_exec' method.

On line 17 'embeddedstyleclasses' is a field reserved for future functionality. In your GizmoTags you can make that point to an empty array reference ('[]') as well.

On line 23 we have the 'md_tag_exec' method. This is the one in charge of generating the tags's content. As you can see it is rather simple how we do it.

On line 26 we generate a return string with the HTML of the tag. Note that we use the SPAN tag to associate the CSS style name declared on line 16 to the HTML of the GizmoTag. Note that we retrieve the style name using the 'get_styleclass' method inherited from the parent class. The content, which in this case is simply today's date and time, comes from calling the standard perl localtime(time) functions.

That's it! For further knowledge you can look at other GizmoTags in the Metadot directory of your 4.0 install or download new ones from the GizmoTags section of the Gizmo Gallery at metadot.net.

9. Themes

Before reading this section, you may wish to read the "Themes" page in the admin documentation:
`<metadot>/doc/md_guides/install/themes.html`
so that you are familiar with the user interface and basic functionality. Selecting "Themes" as the site's Style provides an interface where page layout is fixed, but site colors, text colors, and header logos can be modified easily.

There is not currently a browser-based interface for modifying the template HTML for a given Theme, nor a browser-based interface for switching between Themes, particularly with respect to switching between alternate HTML layouts. (Other theme parameters can be modified via the Themes Styles & Colors interface). However, the underlying code and database structure for themes supports defining more than one theme (in terms of its color settings and HTML templates), and then using a specified theme as the Theme for the site. [An admin browser interface, to provide access to the Themes html, is planned for the future].

With some work, it is possible to create or modify a site's Themes without a browser-based interface, but for a given site, you may not have the need to do this. If you simply need to develop a new "look" for the site, it will probably be sufficient if you select Style 4 from the Styles and Colors configuration page, instead of themes; then modify the Style 4 template layout and gizmo tags as described in Section 8 .

If, however, you want to change the HTML templates for a Theme, this can be done without too much difficulty via your database client interface.

9.1 Examining the Themes Configuration

Look at the value of the `template_theme` parameter in the params table, by running the following query (from a mysql client interface):

```
select * from params where name = 'template_theme';
```

This query shows you the name of the current theme, e.g. `theme2` .

(For the current Metadot release, the '`theme2`' template is in use).

Then look at the themes definitions in the `themes` database table. E.g., enter the following query:

```
select theme, pname from theme;
```

You will see configuration information for one or more themes. For a given theme, the value fields for the following parameters hold template HTML: `theme_maintemplate, theme subtemplate`, and `theme_utilitytemplate`. In addition, the value fields for the following parameters hold page header HTML: `theme_adminheader`, and `theme_simpleheader`. Additional fields hold color and logo settings for the Theme. These fields can be viewed as the "configuration set" for a given Theme.

For a given theme, the three theme templates are roughly equivalent to the Style 4 Main Template, Sub-Template, and Utility Template defined in the Advanced Styles and Colors page, in that they both use GizmoTags to place content in specified locations on the page. However, there is one key difference: the Themes templates contain specialized gizmo tags which insert Template-Toolkit-derived HTML fragments based on the current theme's parameter settings.

The three specialized gizmotags are the modules:
 `<metadot>/metadot/GizmoTag/md_theme1*.pm`.
These sub-templates control the header and left-nav-bar layout for the theme, as well as the style rules for the theme. The Template-Toolkit templates used by these gizmotags are in `<metadot>/metadot/Metadot/templates`, and these files utilize information about the current theme's parameters, such as the current header logo used by the theme, in order to generate HTML fragments specific to that the current theme.

This implementation has two important implications.

First is that the specialized gizmotags, which construct HTML using the Theme's parameters (e.g., navbar color and text), are providing the "hooks" to the color and logo values set in the Styles & Colors page. If you want to replace the Themes HTML, you will need to retain these gizmotags (or build similar ones) in order to have the site pages change properly when the Styles & Colors settings change.

Second, since some Themes HTML content comes from the static Template-Toolkit files used by these specialized gizmotags, then this means that there are some aspects of Themes page layout that are encoded in static files, not in the database. An example is the left-nav-bar width generated by the `md_theme1_lefttable.pm` gizmotag. In a virtual server setup, with multiple Metadot servers running off the same code base, all virtual servers will share the same Template-Toolkit template files. You will need to make new gizmotags if you want to make a change to one of these Template-Toolkit files for just some of your virtual sites; e.g., if you want to use Themes mode for all your sites, but change the left-nav-bar width for the Themes layout in only one of your virtual sites. (This limitation will be addressed in a future release; however, note that if you want to change the layout or look for one virtual site, you can do this straightforwardly using Templates/Style 4 rather than Themes for that site, as described in Section 8).

9.2 Changing or Adding a Theme Configuration set

To change a Theme's HTML, you may need to modify either the HTML strings stored in the themes table for that theme (the `theme_maintemplate, theme_subtemplate`, or `theme_utilitytemplate` strings), or modify one of the Template-Toolkit template sub-files used by the special gizmotags, as described above. The location of the modification will depend upon what aspect of the page's layout you need to change. For example, if you want to change some aspect of the header HTML, or the Theme's CSS definitions, this will require changing a sub-template file. If you want to change the width of the center 'content panel', this is defined in the HTML strings stored in the database.

Using your MySql client interface, you can examine and modify the HTML strings for a given theme. The three HTML strings — `theme_maintemplate, theme_subtemplate`, and `theme_utilitytemplate` — are the templates used for the front page, other site pages, and non-page gizmos, respectively, when Themes mode is enabled.

The two header HTML parameters -- `theme_adminheader` and `theme_simpleheader`, are used for various form and admin pages. They contain "hooks" to utilize the Themes banner image settings. You probably will not need to modify them.

If you like, you can create a new theme configuration set by defining a new name for the theme, and adding its parameters to the `theme` table by inserting new records with the 'theme' field set to your new theme name. If you do this, it is best to use the configuration set of an existing theme as a starting point. When you have completed the new Theme, you can configure your site to use it by updating the params table:

```
update params set value = '<new theme name>' where name = 'template_theme'
```

10. SystemApps

The SystemApp provides a foundation for building stand-alone portal apps. A SystemApp is not associated with any object instantiation. It does not require use of any particular db tables or require the implementation of any particular interface methods. Therefore, it allows new 'application functionality' to be added to the system without needing to modify existing code. Metadot::SystemApp is an 'abstract' class; the actual apps should be subclassed from it. SystemApp inherits from Portal and Auditable.

A SystemApp subclass is invoked by giving the full path to the SystemApp module as the 'isa' parameter of the cgi query, as in this example:

```
http://127.0.0.1/metadot/index.pl?isa=Metadot::SystemApp::TestApp&op=show
```

The 'op' parameter will be used to call the www_<op> method of the system app ('www_show' in this example).

SystemApp, as an abstract class, provides a set of access control methods, and implements the method `www_show`. `www_show` calls the method '`main`', which is expected to be implemented by every SystemApp subclass. The subclass' implementation of '`main`' must return an html string, which will be wrapped in `SystemApp->www_show` by code to generate headers and footers, in order to generate a result page. Therefore, for the 'show' op, the SystemApp subclass need only generate its content html via the implementation of '`main`', not the entire result page. See `Metadot::SystemApp::TestApp` for an example.

A SystemApp subclass may implement any other op it wishes in addition to 'show'; however, it must generate the full result page in that case, including any desired headers, etc.

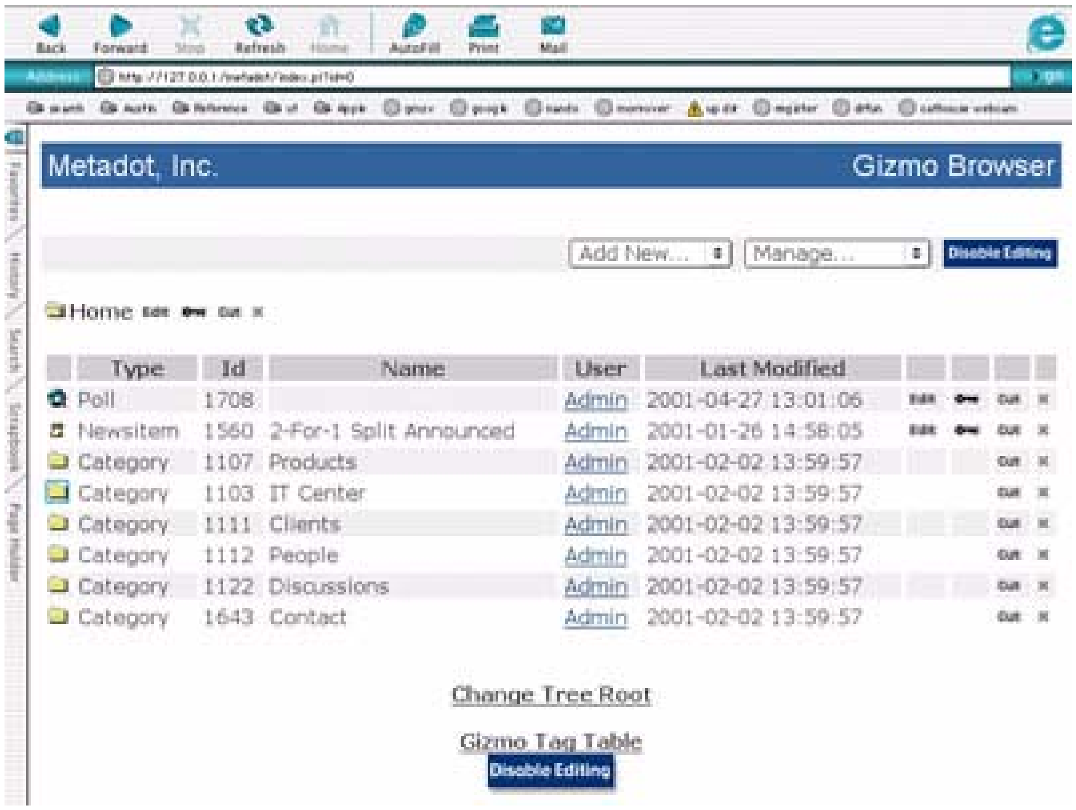
SystemApps inherit from Auditable to take advantage of Metadot's built-in access control mechanism. See Section 3 for more information about how to add access control to your classes. SystemApps will by default show the "change permissions" (key) icon when the show operation is invoked on them and editing is enabled. If your system app implements other operations you will have to invoke the inherited method called "show_portal_panel" to generate the html for the change permissions icon and add it to your page.

A system app has access to the system globals `$PARAMS`, `%FORM`, `$SESSION`, and `$USER`. It can be invoked with arbitrary parameters as well. E.g., below is an example of invoking a "FormDemo" system app and passing it a set of parameter values:

```
http://127.0.0.1/metadot/index.pl?isa=Metadot::SystemApp::FormDemo&op=show&mode=edit&f
orm_descr=formdata_demo&pkey=instance.iid,1642&pkey=message.mid,40
```

11. The Gizmo Browser

The GizmoBrowser is activated under the `Manage.. -> Enable GB` menu item for admin users (editing must be enabled to see this menu). It presents the admin user with table-based view of all the children of the current page, and allows them to be inspected and edited. This can be particularly helpful when the Template style (Style 4 on the `Manage..-> Styles & Colors` admin page) is in use for a site. This is because when templates are used, the GizmoTags embedded in the template determine the page content. If there is no embedded gizmo tag to address a given child object, it will not be exposed to the viewer. (See Section 8 for more information about GizmoTags).



	Type	Id	Name	User	Last Modified				
	Poll	1708		Admin	2001-04-27 13:01:06	Edit		Cut	×
	Newsitem	1560	2-For-1 Split Announced	Admin	2001-01-26 14:58:05	Edit		Cut	×
	Category	1107	Products	Admin	2001-02-02 13:59:57			Cut	×
	Category	1103	IT Center	Admin	2001-02-02 13:59:57			Cut	×
	Category	1111	Clients	Admin	2001-02-02 13:59:57			Cut	×
	Category	1112	People	Admin	2001-02-02 13:59:57			Cut	×
	Category	1122	Discussions	Admin	2001-02-02 13:59:57			Cut	×
	Category	1643	Contact	Admin	2001-02-02 13:59:57			Cut	×

Therefore, the Gizmo Browser allows the template builder to see all the underlying child content for a page, inspect the children, and confirm that the template includes the right gizmo tags to display desired content based on the list of children.

To disable the Gizmo Browser when you are finished with inspection mode, and return to the regular page view, choose `Manage.. -> Disable GB` from the admin's pull-down menu.