Entity API <https://drupal.org/developing/api/entity/7>

An Introduction to Entities

**Entity types**

In earlier versions of Drupal, the field system was only used on content types. Now, thanks to the Entity API, we can add fields to other things, like comments. Fieldable entities make Drupal eminently flexible. An entity type is a useful abstraction to group together fields.

Below are the Entity types in Drupal core:

* Nodes (content)
* Comments
* Files
* Taxonomy terms
* Taxonomy vocabularies
* Users

You can also build new kinds of entity types where the options above don't suit your needs.

**Bundles**

Bundles are an implementation of an entity type to which fields can be attached. You can consider bundles as subtypes of an entity type. With content nodes (an entity type), for example, you can generate bundles (subtypes) like articles, blog posts, or products. Not all entity types have bundles, however. For example, users do not have separate bundles (subtypes).

**Fields**

A field is a reusable piece of content. In technical terms, each field is a primitive data type, with custom validators and widgets for editing and formatters for display.

What's important to know as it relates to Entities is that **Fields can be added to any of the bundles (or entity types)** to help organize their data.

**Entity**

An entity would be one instance of a particular entity type such as a comment, taxonomy term or user profile or of a bundle such as a blog post, article or product.

**You can use entity\_load to load any entity**. Note, however, that the core does not provide a save or delete function, but thanks to **Entity API** module the missing pieces are added (**entity\_create(), entity\_save(), entity\_delete(), entity\_view() and entity\_access()**).

**Putting this in Object-Oriented Design/Programming terms...**

* An **entity type** is a **base class**
* A **bundle** is an **extended class**
* A **field** is a **class member, property, variable or field instance** (depending on your naming preference)
* An **entity** is an **object or instance** of a base or extended class

**Entity API module**

The project Entity API extends the entity API of Drupal core in order to provide a unified way to deal with entities and their properties. Additionally, it provides an entity CRUD controller, which helps in simplifying the creation of new entity types.

Basic steps to add a new entity type:

---------------------------------------

* Describe your entities db table as usual in hook\_schema().
* Just use the "Entity" directly or extend it with your own class. To see how to provide a separate class.   
  Have a look at the "EntityClass" from the "entity\_test.module".
* Implement hook\_entity\_info() for your entity. At least specifiy the controller class (EntityAPIController, EntityAPIControllerExportable or your own), your db table and your entity's keys.   
  Again just look at "entity\_test.module"'s hook\_entity\_info() for guidance.
* If you want your entity to be fieldable just set 'fieldable' in hook\_entity\_info() to TRUE. The field API attachers are then called automatically in the entity CRUD functions.
* The entity API is able to deal with bundle objects too (e.g. the node type object). For that just specify another entity type for the bundle objects and set the 'bundle of' property for it.  
  Again just look at "entity\_test.module"'s hook\_entity\_info() for guidance.
* Schema fields marked as 'serialized' are automatically unserialized upon loading as well as serialized on saving. If the 'merge' attribute is also set to TRUE the unserialized data is automatically "merged" into the entity.

# Providing a new entity type

To create a new entity type

1. Depend on the entity module.
2. Describe your entities db table as usual in hook\_schema(). Add any columns specific to your entity. For some examples, see node\_schema() and user\_schema().
3. Implement hook\_entity\_info() for your entity. At a minimum, specify the controller class of this API, your db table and your object's primary key field. Optionally also set the 'entity class' to Entity, or your extended class.

A simple example hook\_entity\_info() implementation could look like the following:

1. /\*\*
2. \* Implements hook\_entity\_info().
3. \*/
4. function entity\_test\_entity\_info() {
5. return array(
6. 'entity\_test' => array(
7. 'label' => t('Test Entity'),
8. 'plural label' => t('Test Entities'),
9. 'entity class' => 'Entity',
10. 'controller class' => 'EntityAPIController',
11. 'base table' => 'entity\_test',
12. 'entity keys' => array(
13. 'id' => 'id',
14. ),
15. // Use the default label() and uri() functions
16. 'label callback' => 'entity\_class\_label',
17. 'uri callback' => 'entity\_class\_uri',
18. ),
19. );
20. }

Core defines the functions entity\_label() and entity\_uri() which return the label and the URI for an arbitrary entity type with the help of information provided in hook\_entity\_info(). In order to take advantage of this, extend the Entity class, override the defaultLabel() and defaultUri() methods, and specify entity\_class\_label() and entity\_class\_uri() callbacks in hook\_entity\_info().Then you can use $entity->label() or $entity->uri(), as well as the core API functions. Also, other modules will be able to override your behaviors by altering the hook\_entity\_info() callbacks, while $entity->label() and $entity->uri() reflect these changes just as entity\_label() and entity\_uri().

# Make an entity type exportable

By making your entity exportable, you make it possible for site builders to take an entity they've created through the UI and export it so it can be used elsewhere. Familiar examples of exportable configuration are image styles and views. In Drupal 7, you can create an image style through the UI and then export it to be used on another site or included in the code of a module.

To make your entity type exportable, specify EntityAPIControllerExportable as the controller class, set the exportable key in hook\_entity\_info() to TRUE

/\*\*

\* Implements hook\_entity\_info().

\*/

function entity\_test\_entity\_info() {

$return['entity\_test\_type'] = array(

'label' => t('Test entity type'),

'entity class' => 'Entity',

**'controller class' => 'EntityAPIControllerExportable',**

'base table' => 'entity\_test\_type',

'fieldable' => FALSE,

'bundle of' => 'entity\_test',

**'exportable' => TRUE,**

'entity keys' => array(

'id' => 'id',

'name' => 'name',

),

);

return $return;

}

# Making an entity revisionable

* The revision table for your entity type requires a **revision column set to serial** type as of entity 7.x-1.0.
* Any additional properties that will be saved in the **revision table can be set on the entity object**.
* is\_new\_revision property must be set on the entity object for a new record to be written to the revision table. Otherwise it will update the row corresponding to the current revision. Example:  
  **$entity->is\_new\_revision = TRUE;**
* default\_revision property must be set on the entity object for the revision being saved to be set as the default revision. Example:   
  **$entity->default\_revision = TRUE;**
* Don't forget to tell the entity info that your entity has a revision table and a revision field.

If you don't, entities with "is\_new" will not create a new revision. Example:

...

function hook\_entity\_info() {

...

'entity keys' => array(

'id' => 'id',

'bundle' => 'type',

'base table' => 'my\_entity', // table that stores your entity info

**'revision table' => 'my\_entity\_revision', // revision table name, can be any name you choose, but convention would be [base table]\_revision**

**'revision'** **=>** **'vid', // Revision column name, node uses 'vid', you can name it what ever you want.**

...

),

...

# Enabling Revisions on a Table With Existing Data

Enabling revisions on an entity after the fact requires 3 basic steps:

* Your Entity Info must have the same info (revision table, entity keys->revision) as is required for enabling revisions on a new entity.
* You must insert a record into the revision table for every record in your base table.
* You must update your the column containing the revision key on your base table to match the pkey of a valid value in the revision table (failing to add default revision entries will result in a "Page not Found" error when trying to view your content).

The following example is for a custom Entity named "dh\_properties", which has several meta-data columns. It comes from the project dH.

Example: inserting revision entries for custom entity "dh\_properties".

Entity Info:

$return['dh\_properties'] = array(

'label' => t('dH Properties'),

'entity class' => 'dHProperties',

'controller class' => 'dHPropertiesController',

'base table' => 'dh\_properties',

**'revision table' => 'dh\_properties\_revision',**

'fieldable' => TRUE,

'entity keys' => array (

'name' => 'pid',

'id' => 'pid',

'label' => 'pid',

'bundle' => 'bundle',

'revision' => 'vid',

'revision propvalue' => 'propvalue',

),

...

**Example Form:** Triggering a revision to be saved for every form submittal.

function dh\_properties\_form($form, &$form\_state, $dh\_properties, $op = 'edit') {

**$dh\_properties->is\_new\_revision = TRUE;**

...

# Viewing entities

The entity API assists in writing the code for viewing an entity by providing entity\_build\_content() and entity\_view() functions which take care of attaching any fields and works similar to the node\_view() function.

The best way to provide your own custom content is to override the buildContent() method of your entity's controller class. Then, as long as you use the functions above to display your entities the changes will be picked up automatically.

For theming the entity API uses the basic entity.tpl.php template by default and adds some useful template suggestions similar to what core does for nodes.

{ENTITY TYPE}.tpl.php

{ENTITY TYPE}\_\_{BUNDLE}.tpl.php

{ENTITY TYPE}\_\_{BUNDLE}\_\_{VIEW MODE}.tpl.php

Since the first suggestion is the name of the entity type you can simply define your custom template in your module using hook\_theme() and it will be used automatically when displaying your entities.

### Specify an 'admin ui' and other special settings in hook\_entity\_info()

Your implementation of hook\_entity\_info() must have an 'admin ui' property which takes an array of further properties. At the very least in this, you must specify a 'path' where the admin UI page will be located. You should also declare the file that will hold your add/edit/clone form and submit function (which is defined in step 4).

You must also set the 'access callback' (and define this function, in the next step) and set 'module' with your module name.

Here is an example from profile 2:

/\*\*

\* Implements hook\_entity\_info().

\*/

function profile2\_entity\_info() {

//...

$return['profile2\_type'] = array(

//... usual hook entity info implementation.

'access callback' => 'profile2\_type\_access',

'module' => 'profile2',

// Enable the entity API's admin UI.

'admin ui' => array(

**'path' => 'admin/structure/profiles',**

**'file' => 'profile2.admin.inc',**

// OPTIONAL- See "Customize the User Interface" below if you set.

'controller class' => 'Profile2TypeUIController',

),

);

return $return;

}

### Define an access callback

This is the function named in 'access callback' property in the hook\_entity\_info() implementation, above. It should return TRUE if the provided user account should have access to administer your entity, and FALSE if not.

Again, an example from profile2.

/\*\*

\* Access callback for the entity API.

\*

\* @param $op

\* The operation being performed. One of 'view', 'update', 'create', 'delete'

\* or just 'edit' (being the same as 'create' or 'update').

\* @param $profile

\* (optional) A profile to check access for. If nothing is given, access for

\* all profiles is determined.

\* @param $account

\* (optional) The user to check for. Leave it to NULL to check for the global user.

\*

\* @return boolean

\* Whether access is allowed or not.

\*/

function profile2\_type\_access($op, $profile = NULL, $account = NULL) {

return user\_access('administer profile types', $account);

}

### Implement the entity adding/editing form

The entity form must be defined by the function ENTITY\_TYPE\_form() so that it can be retrieved using entity\_ui\_get\_form(). You must also implement a submit function for this form. Both may live in the include file specified for the 'admin ui' via the optional 'file' key and could look like the following example form - a shortened version of the profile2 type form:

# Basic file skeleton + CRUD hooks

/\*\*

\* Acts on {ENTITIES} being loaded from the database.

\*

**\* This hook is invoked during {ENTITY} loading, which is handled by**

**\* entity\_load(), via the EntityCRUDController.**

\*

\* @param array {$ENTITIES}

\* An array of {ENTITY} entities being loaded, keyed by id.

\*

\* @see hook\_entity\_load()

\*/

function hook\_{ENTITY\_ID}\_load(array {$ENTITIES}) {

$result = db\_query('SELECT pid, foo FROM {mytable} WHERE pid IN(:ids)', array(':ids' => array\_keys({$ENTITIES})));

foreach ($result as $record) {

{$ENTITIES}[$record->pid]->foo = $record->foo;

}

}

/\*\*

\* Responds when a {ENTITY} is inserted.

\*

**\* This hook is invoked after the {ENTITY} is inserted into the database.**

\*

\* @param {ENTITY\_CLASS} {$ENTITY}

\* The {ENTITY} that is being inserted.

\*

\* @see **hook\_entity\_insert()**

\*/

function hook\_{ENTITY\_ID}\_insert({ENTITY\_CLASS} {$ENTITY}) {

db\_insert('mytable')

->fields(array(

'id' => entity\_id('{ENTITY\_ID}', {$ENTITY}),

'extra' => print\_r({$ENTITY}, TRUE),

))

->execute();

}

/\*\*

\* Acts on a {ENTITY} being inserted or updated.

\*

**\* This hook is invoked before the {ENTITY} is saved to the database.**

\*

\* @param {ENTITY\_CLASS} {$ENTITY}

\* The {ENTITY} that is being inserted or updated.

\*

\* @see **hook\_entity\_presave()**

\*/

function hook\_{ENTITY\_ID}\_presave({ENTITY\_CLASS} {$ENTITY}) {

{$ENTITY}->name = 'foo';

}

/\*\*

\* Responds to a {ENTITY} being updated.

\*

**\* This hook is invoked after the {ENTITY} has been updated in the database.**

\*

\* @param {ENTITY\_CLASS} {$ENTITY}

\* The {ENTITY} that is being updated.

\*

\* @see **hook\_entity\_update()**

\*/

function hook\_{ENTITY\_ID}\_update({ENTITY\_CLASS} {$ENTITY}) {

db\_update('mytable')

->fields(array('extra' => print\_r({$ENTITY}, TRUE)))

->condition('id', entity\_id('{ENTITY\_ID}', {$ENTITY}))

->execute();

}

/\*\*

\* Responds to {ENTITY} deletion.

\*

**\* This hook is invoked after the {ENTITY} has been removed from the database.**

\*

\* @param {ENTITY\_CLASS} {$ENTITY}

\* The {ENTITY} that is being deleted.

\*

\* @see **hook\_entity\_delete()**

\*/

function hook\_{ENTITY\_ID}\_delete({ENTITY\_CLASS} {$ENTITY}) {

db\_delete('mytable')

->condition('pid', entity\_id('{ENTITY\_ID}', {$ENTITY}))

->execute();

}

# View-related hooks.

If your module makes use of entity\_view(), add the following view-related hooks:

/\*\*

\* Act on a {ENTITY} that is being assembled before rendering.

\*

\* @param ${ENTITY}

\* The {ENTITY} entity.

\* @param $view\_mode

\* The view mode the {ENTITY} is rendered in.

\* @param $langcode

\* The language code used for rendering.

\*

**\* The module may add elements to ${ENTITY}->content prior to rendering. The**

**\* structure of ${ENTITY}->content is a renderable array as expected by**

**\* drupal\_render().**

\*

\* @see hook\_entity\_prepare\_view()

\* @see hook\_entity\_view()

\*/

function hook\_{ENTITY\_ID}\_view(${ENTITY}, $view\_mode, $langcode) {

${ENTITY}->content['my\_additional\_field'] = array(

'#markup' => $additional\_field,

'#weight' => 10,

'#theme' => 'mymodule\_my\_additional\_field',

);

}

/\*\*

\* Alter the results of entity\_view() for {ENTITY}s.

\*

\* @param $build

\* A renderable array representing the {ENTITY} content.

\*

**\* This hook is called after the content has been assembled in a structured**

**\* array and may be used for doing processing which requires that the complete**

**\* {ENTITY} content structure has been built.**

\*

\* If the module wishes to act on the rendered HTML of the {ENTITY} rather than

\* the structured content array, it may use this hook to add a #post\_render

\* callback. Alternatively, it could also implement hook\_preprocess\_{ENTITY}().

\* See drupal\_render() and theme() documentation respectively for details.

\*

\* @see hook\_entity\_view\_alter()

\*/

function hook\_{ENTITY\_ID}\_view\_alter($build) {

if ($build['#view\_mode'] == 'full' && isset($build['an\_additional\_field'])) {

// Change its weight.

$build['an\_additional\_field']['#weight'] = -10;

// Add a #post\_render callback to act on the rendered HTML of the entity.

$build['#post\_render'][] = 'my\_module\_post\_render';

}

}

If your module implements an entity form with the help of entity\_ui\_get\_form(), the following template helps you documenting the according hook\_form\_alter(). Note, that if your entity type has no bundles (or isn't fieldable), you need to remove the sentence concerning bundles.

/\*\*

\* Alter {ENTITY} forms.

\*

\* Modules may alter the {ENTITY} entity form by making use of this hook or

\* the entity bundle specific hook\_form\_{ENTITY\_ID}\_edit\_BUNDLE\_form\_alter().

\* #entity\_builders may be used in order to copy the values of added form

\* elements to the entity, just as documented for

\* entity\_form\_submit\_build\_entity().

\*

\* @param $form

\* Nested array of form elements that comprise the form.

\* @param $form\_state

\* A keyed array containing the current state of the form.

\*/

function hook\_form\_{ENTITY\_ID}\_form\_alter(&$form, &$form\_state) {

// Your alterations.

}

# Views integration

Entity API tries to generate suitable Views integration for the entity based upon:

* The schema information of its base table and the provided entity
* Property information.

For the integration to work, you need to:

* Make use of the Entity CRUD API as described here: http://drupal.org/node/878804. Most important is to provide the 'EntityAPIController' or descending class as the controller class for the entity type.
* Provide the name of your module in your hook\_entity\_info():

/\*\*

\* Implements hook\_entity\_info().

\*/

function entity\_test\_entity\_info() {

return array(

'entity\_test' => array(

'label' => t('Test Entity'),

// Entity API needs to know about your module for the views integration to work

'module' => 'entity\_test',

..

While Entity API tries to provide Views the needed information based on your schema, on some cases it may fail or not give the results you expected. To manually expose your data to Views as fields, filters or arguments, you'll have to configure this in hook\_entity\_property\_info() by making use of 'schema field'. This mapping overrides Entity API's 'guessing' and exposes those properties as Views fields, filters and arguments the way you need them to.

Note that if you use this, you need to define ALL fields since it completely overrides Entity API's implementation. If you just need to alter part of your fields, you can use hook\_entity\_property\_info\_alter() in similar manner.

/\*\*

\* Implements hook\_entity\_property\_info().

\*/

function entity\_test\_entity\_property\_info() {

$info = array();

$properties = &$info['entity\_test']['properties'];

$properties = array(

'some\_field' => array(

'type' => 'integer',

'label' => t('Some field'),

'description' => t('This is some field.'),

// Map here the field from hook\_schema().

'schema field' => 'some\_field',

),

// Other fields...

);

return $info;

}

Another way to alter this data is to extend EntityDefaultViewsController with your own implementation.

First let Entity API know you will use a custom Views Controller:

/\*\*

\* Implements hook\_entity\_info().

\*/

function entity\_test\_entity\_info() {

return array(

'entity\_test' => array(

'label' => t('Test Entity'),

// Entity API needs to know about your module for the views integration to work

'module' => 'entity\_test',

'views controller class' => 'EntityTestViewsController',

..

Then implement the class:

/\*\*

\* EntityTest Views Controller class.

\*/

class EntityTestViewsController extends EntityDefaultViewsController {

/\*\*

\* Edit or add extra fields to views\_data().

\*/

public function views\_data() {

$data = parent::views\_data();

// Add your custom data here

/\*

\* Example: change the handler of a field

\* if the 'created' field is a unix timestamp in the database,

\* Entity API will set the handler to views\_handler\_field\_numeric,

\* change this to the views date handler

\*/

$data['entity\_test']['created']['field']['handler'] = 'views\_handler\_field\_date';

return $data;

}

}

# Entity property information

To provide information about properties (including fields), you'll have to implement hook\_entity\_property\_info(). You may put your hook implementations in the include yourmodule.info.inc, however any callbacks have to reside in files that are always included.

Let's consider the node module integration as an example:

/\*\*

\* Implements hook\_entity\_property\_info() on top of node module.

\*

\* @see entity\_metadata\_entity\_property\_info()

\*/

function entity\_metadata\_node\_entity\_property\_info() {

$info = array();

// Add meta-data about the basic node properties.

$properties = &$info['node']['properties'];

$properties['nid'] = array(

'label' => t("Node ID"),

'type' => 'integer',

'description' => t("The unique ID of the node."),

'schema field' => 'nid',

);

$properties['language'] = array(

'label' => t("Language"),

'type' => 'token',

'description' => t("The language the node is written in."),

'setter callback' => 'entity\_metadata\_verbatim\_set',

'options list' => 'entity\_metadata\_language\_list',

'schema field' => 'language',

'setter permission' => 'administer nodes',

);

$properties['author'] = array(

'label' => t("Author"),

'type' => 'user',

'description' => t("The author of the node."),

'getter callback' => 'entity\_metadata\_node\_get\_properties',

'setter callback' => 'entity\_metadata\_node\_set\_properties',

'setter permission' => 'administer nodes',

'required' => TRUE,

'schema field' => 'uid',

);

// ...

return $info;

}

**Handling entity references**

In case of properties containing entity references like "uid" one should not expose an "uid" integer property, but a property that exposes the entity relationship. Just expose a property "user" or "author" of type "user", as seen in the example above.

Then, do not expose both the id and the entity reference, only expose the entity relationship. The id is accessible via the referenced entity (without having to load it) anyway.

Internally, an entity property is represented by its identifier, thus the setter callback has to accept the entity id as value. The getter callback may return the id or the full entity object though.

## Introduction

The contributed [Entity API](https://www.drupal.org/project/entity) module provides wrapper classes that make dealing with the values of an entity's properties and fields easier.

Wrappers make it easier to get and set the values of fields and properties as well as to programmatically retrieve additional information about these elements and iterate over lists of values in a consistent manner.

For example consider the following method of accessing the value of a field attached to a node. This is a pattern that we see used often when working with entities in Drupal. However there are a couple of things that make this less than ideal. For example what if the user had requested the page in a different language? Or what about the case where the ['value'] key just doesn't exist like with image and file fields?

$node->field\_custom\_field\_number[LANGUAGE\_NONE][0]['value'];

Using metadata wrappers from the entity module we can access this information like so:

$node\_wrapper = entity\_metadata\_wrapper('node', $node);

$node\_wrapper->field\_custom\_field\_number->value();

How about an example of making things consistent? All Drupal entities have a label of some sort. A string that can be treated as the canonical human readable name of an entity. All nodes have a title property and all user accounts have a name property. Given a standard Drupal entity it can be hard to know which property should be treated as the label. Metadata wrappers provide us with a consistent way of getting at this kind of information for any entity.

// Unified way of getting $node->title, $user->name, ...

$wrapper->label();

// Unified way of getting $node->nid, $user->uid, ...

$wrapper->getIdentifier();

// Unified way of getting $node->type, ...

$wrapper->getBundle();

## Examples

For making use of this information (metadata) the module provides some wrapper classes which ease getting and setting values.

For wrapping an entity object you can use the procedural function (as below):

$wrapper = entity\_metadata\_wrapper('node', $node);

Or since Entity API 7.x-1.6, you can also use Entity::wrapper():

$wrapper = $entity->wrapper();

This only works however if the $entity object uses the Entity class provided by the Entity API module.

The wrapper supports chained usage for retrieving wrappers of entity properties, e.g. To get a  
node author's mail address one could use:

$wrapper->author->mail->value();

To update the user's mail address one could use

$wrapper->author->mail->set('sepp@example.com');

or

$wrapper->author->mail = 'sepp@example.com';

The wrappers always return the data as described in the property  
information, which may be retrieved directly via entity\_get\_property\_info()  
or from the wrapper:

$mail\_info = $wrapper->author->mail->info();

In order to force getting a textual value sanitized for output one can use,  
e.g.

$wrapper->title->value(array('sanitize' => TRUE));

to get the sanitized node title. When a property is already returned  
sanitized by default, like the node body, one possibly wants to get the  
not-sanitized data as it would appear in a browser for other use-cases.  
To do so one can enable the 'decode' option, which ensures for any sanitized  
data the tags are stripped and HTML entities are decoded before the property  
is returned:

$wrapper->body->value->value(array('decode' => TRUE));

That way one always gets the data as shown to the user. However if you  
really want to get the raw, unprocessed value, even for sanitized textual  
data, you can do so via:

$wrapper->body->raw();

Many more examples can be found in the tests of the entity module.

### Example of using value(), set() and save()

<?php

$node\_wrapper = entity\_metadata\_wrapper('node', $node);

$var = $node\_wrapper->field\_number->value() + 1;

$node\_wrapper->field\_number->set($var);

$node\_wrapper->save();

?>

If you need to save a file into a field you must pass the file object or an array with a ``fid`` key:

<?php

$containing\_node = node\_load($nid);

$w\_containing\_node = entity\_metadata\_wrapper('node', $containing\_node);

// Load the file object in any way

$file\_obj = file\_load($fid);

$w\_containing\_node->field\_attachment\_content->file->set( $file\_obj );

// ..or pass an array with the fid

$w\_containing\_node->field\_attachment\_content->set( array('fid' => $fid) );

$w\_containing\_node->save();

// If an image field supports title or alt, you can set them in 2 ways.

// Set both file and title with an array.

$w\_containing\_node->field\_image->set(array('fid' => $fid, 'title' => $title));

// Set both file and title separately.

$image\_obj = file\_load($fid);

$w\_containing\_node->field\_image->file = $image\_obj;

$w\_containing\_node->field\_image->title = $title;

$w\_containing\_node->save();

?>

### Example of creating a node

You need to set the content type and user ID on a new node entity; and then that can be immediately passed in to entity\_metadata\_wrapper.

// Create an Entity.

$node = entity\_create('node', array('type' => 'image'));

// Specify the author.

$node->uid = $user->uid;

// Create a Entity Wrapper of that new Entity.

$emw\_node = entity\_metadata\_wrapper('node', $node);

// Set a title and some text field value.

$emw\_node->title = 'Test node';

$emw\_node->field\_text\_field = 'Field value text';

// And save it.

$emw\_node->save();

### Example using field collections

<?php

// Populate the fields.

$ewrapper = entity\_metadata\_wrapper('node', $node);

$ewrapper->field\_lead\_contact\_name->set($contact\_name);

$ewrapper->field\_lead\_contact\_phone->set($contact\_phone);

$ewrapper->field\_lead\_contact\_email->set($contact\_email);

// Create the collection entity and set it's "host".

$collection = entity\_create('field\_collection\_item', array('field\_name' => 'field\_facilities\_requested'));

$collection->setHostEntity('node', $node);

// Now define the collection parameters.

$cwrapper = entity\_metadata\_wrapper('field\_collection\_item', $collection);

$cwrapper->field\_facility->set(intval($offset));

$cwrapper->save();

// Save.

$ewrapper->save();

?>

## Working with lists

A list value that contains wrappers, such as a multi-valued reference field, can be iterated over thus:

$wrapper = entity\_metadata\_wrapper('node', $node);

foreach ($wrapper->field\_taxonomy\_terms->getIterator() as $delta => $term\_wrapper) {

// $term\_wrapper may now be accessed as a taxonomy term wrapper.

$label = $term\_wrapper->name->value();

}

Setting values or adding values to a list value, such as a multi-valued entity reference field, can be accomplished in several ways.

You can set an array as the whole value:

$containing\_node = node\_load($nid);

$w\_containing\_node = entity\_metadata\_wrapper('node', $containing\_node);

$nids\_to\_set = array(42, 23);

$w\_containing\_node->field\_entity\_reference\_field->set($nids\_to\_set);

The field wrapper can be appended to using square bracket syntax, the same as when working with a normal array:

<?php

// Setting or adding to a list using square bracket syntax

$containing\_node = node\_load($nid);

$w\_containing\_node = entity\_metadata\_wrapper('node', $containing\_node);

// This appends to what is already there, just like a normal array.

$w\_containing\_node->field\_entity\_reference\_field[] = 42;

$w\_containing\_node->field\_entity\_reference\_field[] = 23;

?>

Finally, you can get the array out of the field, manipulate it, and set it back:

<?php

// Add to a list using the whole array.

$containing\_node = node\_load($nid);

$w\_containing\_node = entity\_metadata\_wrapper('node', $containing\_node);

$curr\_list = $w\_containing\_node->field\_entity\_reference\_field->value();

if (!$curr\_list)

$curr\_list = array();

$curr\_list[] = $new\_nid;

$w\_containing\_node->field\_entity\_reference\_field->set($curr\_list);

$w\_containing\_node->save();

?>

## Deleting values

To delete values, there is no ->delete() method on the fields. You have to use this way to correctly delete a value:

// Using an empty ->set(NULL) removes the value - without NULL you'll get a PHP notice that set($value) requires 1 parameter.

$wrapper->field\_data->set(NULL);

// And handles correctly the deltas when using multiple values fields

$wrapper->field\_data->offsetUnset($delta);

## Get first value of multifield (multiple-value field)

Just set value index directly after field name:

$first\_name = $wrapper->field\_tags[0]->name->value();

## Handling Exceptions

It's recommended to wrap code in try...catch section when you work with entity\_metadata\_wrapper() to catch EntityMetadataWrapperException.  
You can add hints for you in watchdog for debugging errors. For example function name or line number. Example:

try {

$node\_wrapper = entity\_metadata\_wrapper('node', $node);

$price = $node\_wrapper->field\_product->field\_price->value();

}

catch (EntityMetadataWrapperException $exc) {

watchdog(

'MODULE\_NAME',

'EntityMetadataWrapper exception in %function() <pre>@trace</pre>',

array('%function' => \_\_FUNCTION\_\_, '@trace' => $exc->getTraceAsString()),

WATCHDOG\_ERROR

);

}

## Get Start-date and End-date values from Date fields

If you have Date fields (provided by [Date](https://drupal.org/project/date) module) you can get Start/End values from a field in this way:

$wrap\_node = entity\_metadata\_wrapper('node', $node);

$start\_date = $wrap\_node->field\_my\_data->value()['value'];

$end\_date = $wrap\_node->field\_my\_data->value()['value2'];

or alternatively:

$wrap\_node = entity\_metadata\_wrapper('node', $node);

$start\_date = $wrap\_node->field\_my\_data->value->value();

$end\_date = $wrap\_node->field\_my\_data->value2->value();

## Getting information about properties and fields

getPropertyInfo() returns information about available properties and fields, but depend on the $info parameter passed to the entity\_metadata\_wrapper().

// Wrapper for entity type 'node', but not specific bundle.

$wrapper = entity\_metadata\_wrapper('node');

// Info will contain information only about properties and not about fields.

$info = $wrapper->getPropertyInfo();

And now, when bundle provided:

// Wrapper for entity type 'node' with specific bundle 'page' specified.

$wrapper = entity\_metadata\_wrapper('node', NULL, array('bundle' => 'page'));

// Info will contain information about both properties and fields.

$info = $wrapper->getPropertyInfo();

## Getting Entity Translation based translations

For nodes with Entity Translation based translations, language() returns translated a node wrapper for specified language.

$langcode = 'ja';

$wrapper = entity\_metadata\_wrapper('node', $nid);

$wrapper->title->value(); // => source node title

$wrapper\_translated = $wrapper->language($langcode);

$wrapper\_translated->title->value(); // => translated node title

On the other hand, if you want to specify the language of the source node, you need to pass 3rd parameter to entity\_metadata\_wrapper().

$langcode = 'it';

$info = array('langcode' => $langcode);

$w = entity\_metadata\_wrapper('node', $nid, $info);

## Debugging or introspecting wrappers

To get a list of all available properties of a(n) (entity)wrapper, you can use the following snippet (requires the devel module to be installed and enabled):

dpm($wrapper->getPropertyInfo());

But this doesn't give you any of the property values.  
To be able to see the values you can write your own helper to debug wrapped objects:

<?php

function \_wrapper\_debug($w) {

$values = array();

foreach ($w->getPropertyInfo() as $key => $val) {

$values[$key] = $w->$key->value();

}

return $values;

}

?>

And then use it in your code:

<?php dpm(\_wrapper\_debug($some\_wrapped\_object)); ?>