Field API <https://drupal.org/node/443536>

Field types

**Field types define the type of data that will be stored and any operations that should be performed before storing the data and after the data is loaded**. For example an Amazon ASIN field might store an ASIN number (which refers to a book) as a string. Before the ASIN number is stored it could check with amazon.com to verify the validity of the number. After the ASIN number is loaded it could query amazon.com for details about the book it references and pass that information to the field formatter.

Field storage

The actual low level work of saving/loading/deleting data is performed by field storage modules. **Drupal core provides a field SQL storage module which performs all the SQL statements necessary for handling field save/load/delete tasks for SQL database storage.** Field storage modules could be written to store field data in a variety of ways including HyperTable, Amazon S3 or CouchDB. **Per field a different storage engine can be chosen. Choosing the appropriate storage engine for your field can improve scalability.**

Field widgets

**Field widgets provide an interface for users to enter data.** This could be as simple as an HTML textarea, or as complex as a Farbtastic color picker like the one used by color module for theme settings. **Field widgets do not perform any validation on the actual data, they simply pass along whatever information has been entered for further processing.**

Field formatters

**Field formatters control how the data is displayed.** These are, practically, theme functions which also have a user readable name.

Field

**A field consists of a field type and specific settings which can not be changed once the field is created.** For example, you can create an **author** field which contains a text type and only one of that.

Fieldable objects

**A fieldable object is something that calls the field attach API for its CRUD (create, read, update, delete) operations.** In core, node, taxonomy term, comment, and user are fieldable objects.

Field bundles and instances

A bundle is a place to attach a field. Core defines a bundle for every node type, for comments on each node type, for taxonomy terms (for each vocabulary), and one for user, aptly named 'user'. Every bundle contains a number of field instances. A field instance is created from a field when that gets attached to the bundle. Also, the field instance has settings that are changeable any time, like a formatter and a widget.

# Adding and reusing a field

Let's begin with creating a last name field. We will add this to users and to the 'picture' node type so that it's possible to put names on them even if the author is not a user.

Our field type will be text. This will be stored in SQL (which is the default):

**$field** = array(

// It is strongly advised to prefix the field name with the name of the module

// that defines it, to avoid name clashes. Fields created through Field UI are

// prefixed with 'field\_'

**'field\_name' => 'mymodule\_lastname',**

**'type' => 'text',**

'cardinality' => 1, // Not required, as is default. Number of values field can hold.

);

**field\_create\_field($field);**

Users will use common HTML text input fields to enter the data -- this is called a [widget](http://drupal.org/node/443540#field_widgets). We will [format](http://drupal.org/node/443540#field_formatters) these as Plain text -- we do not need HTML tags in there.

**$instance** = array(

'field\_name' => 'mymodule\_lastname',

**'entity\_type' => 'user',**

**'bundle' => 'user',**

'label' => t('Last name'),

'description' => t('You can enter your last name here.'),

**'widget' => array(**

**'type' => 'text\_textfield',**

**'weight' => 10,**

**),**

);

**field\_create\_instance($instance);**

Now to attach the same field to another [bundle](http://drupal.org/node/443540#field_bundles) (here, node type 'picture'), that's very easy:

**$instance['entity\_type'] = 'node';**

**$instance['bundle'] = 'picture';**

**field\_create\_instance($instance);**

An example of **a select field:**

**$field** = array(

'field\_name' => 'name\_of\_field',

**'type' => 'list\_text',**

**'settings' => array(**

**'allowed\_values' => array(**

**'key\_1' => t('Value 1'),**

**'key\_2' => t('Value 2'),**

**'key\_3' => t('Value 3'),**

**),**

),

);

**field\_create\_field($field);**

**$instance** = array(

'field\_name' => 'name\_of\_field',

'label' => t('The label'),

**'entity\_type' => 'the\_entity\_type',**

**'bundle' => 'the\_bundle',**

**'widget' => array(**

**'type' => 'options\_select',**

**),**

);

**field\_create\_instance($instance);**

# Field Types and Settings

There are a defined number of field types in use, and each type has its own available settings (links below are to API HOOK\_field\_info() implementations). The settings array can appear in three places: the field definition, in instance definition, or within the instance definition's widget array.

List of field types:

* file (File)
* image (Image)
* taxonomy\_term\_reference (Term Reference)
* List fields
  + list\_boolean (Boolean)
  + list\_float (List (Float))
  + list\_integer (List (Integer))
  + list\_text (List (Text))
  + Field settings
    - allowed\_values (list\_integer, list\_float, list\_text only.)
    - allowed\_values[on] and allowed\_values[off] (list\_boolean)
  + Widget
    - settings[display\_label] (options\_onoff i.e. checkbox only)
* Number fields
  + number\_decimal (Decimal)
  + number\_float (Float)
  + number\_integer (Integer)
  + field settings
    - precision (number\_decimal only)
    - scale (number\_decimal only)
    - decimal\_separator (number\_decimal and number\_float only)
* Text fields
  + text (Text)
  + text\_long (Long Text)
  + text\_with\_summary (Long Text with Summary)
  + Field settings
    - max\_length. For unlimited textareas, set to ''
  + Instance settings
    - text\_processing (enables text filters)
  + Widget
    - settings[size] (textfield only)
    - settings[summary\_rows] (textarea\_with\_summary only)
    - settings[rows] (textarea)
* All fields
  + label
  + required
  + description (known as help text in the ui)
  + default\_value[][value]
  + settings[description] ('help' text)

To discover settings, search the Drupal API for implementations of hook\_field\_settings\_form. To discover widget options, search for implementations of hook\_field\_widget\_form. This is very difficult to document because the code is distributed through a handful of modules, each with multiple relevant hooks to examine.

# How to create a new entity type

Entities are a bit like objects in OOP in that they are collections of data that Drupal can operate on in consistent manner. It might be helpful to think of them as containers that you might like to attach fields to. For example nodes are entities, as are users, files and comments. If you're creating a new database table for a new type of content or object in your module, there is a good chance it should be made into an entity.

This page is going to explain how to make the simplest entity possible.

Your first step is to use Drupal's database schema API to create a database table that stores the base data for this new entity. The node and user tables are examples of such base tables.

Once you've created that table, you need to make Drupal aware of it by using hook\_entity\_info:

/\*\*

\* Implements hook\_entity\_info().

\*/

function yourmodule\_entity\_info() {

return array(

'name\_of\_entity' => array(

'label' => t('Name of entity'),

'base table' => 'your\_table',

'entity keys' => array(

'id' => 'your\_tables\_primary\_key',

),

),

);

}

What we are doing in the function above is telling Drupal the name of the table in the database that this entity is based on and its primary key. This is the minimum Drupal needs to know in order to load the data for this entity.

Clear your caches and then print out the result of [entity\_get\_info()](http://api.drupal.org/api/drupal/includes--common.inc/function/entity_get_info/7) and you should see the entity that you just defined listed in the array, along with entities that have been defined elsewhere such as node and user.

Lots of default values have been filled. The fieldable key should say FALSE at the moment because the default is to not allow an entity to have fields which makes that entity more lightweight. Also look under [drupal\_schema\_fields\_sql](http://api.drupal.org/api/drupal/includes--common.inc/function/drupal_schema_fields_sql/7) and then base table and you should see a list of the fields that are in the base table you defined.

Make sure your table has a few records in it and then run entity\_load('name\_of\_entity') and print out the result. You should see that all those records are presented in the array you've just printed.

If you look at the full API docs for [entity\_load()](http://api.drupal.org/api/drupal/includes--common.inc/function/entity_load/7) you will see it takes an array of $ids as an argument. As we used the function above without specifying a subset of records, it just returned everything but usually we would want to load only a subset of records.

You will also see the $reset parameter. Entities have a default caching implementation so requesting the same record multiple times per page load will not keep hitting the database. Setting $reset to TRUE forces that cache to clear so we get fresh data.

A design pattern has emerged for using [entity\_load](http://api.drupal.org/api/drupal/includes--common.inc/function/entity_load/7) in connection with a load function as used in a menu callback.

/\*\*

\* Load a single record.

\*

\* @param $id

\* The id representing the record we want to load.

\*/

function examplemodule\_load($id, $reset = FALSE) {

return examplemodule\_load\_multiple(array($id), array(), $reset);

}

/\*\*

\* Load multiple records.

\*/

function examplemodule\_load\_multiple($ids = array(), $conditions = array(), $reset = FALSE) {

return entity\_load('name\_of\_entity', $ids, $conditions, $reset);

}

Calling entity\_load() gives quite a lot of benefits over just grabbing the data straight from the table using the database API. As mentioned above, it provides per page caching. We also mentioned above that, if it is enabled, it will attach any fields associated with this entity type to the results it brings back. It also calls hook\_entity\_load() which modules can use to attach data to records from any entity type that is loaded and another hook which is specific to this entity. hook\_node\_load() is an example of this hook.

# Implementing entity revisioning

Entity revisioning is fairly simple - most of it is handled by field API, so you just need to implement a revision table, and appropriate load and save functions, and add an {example\_revision} table with the appropriate columns.

## hook\_schema implementation

/\*\*

\* Implements hook\_schema().

\*/

function example\_schema() {

**$schema['example'] = array(**

'description' => 'Keeps track of example entities.',

'fields' => array(

**'example\_id' => array(**

**'type' => 'serial',**

'unsigned' => TRUE,

'not null' => TRUE,

'description' => 'Unique example id.',

),

**'vid' => array(**

**'description' => 'The current {example\_revision}.vid version identifier.',**

**'type' => 'int',**

**'unsigned' => TRUE,**

**'not null' => TRUE,**

**'default' => 0,**

**),**

'created' => array(

'description' => 'The Unix timestamp when the example was created.',

'type' => 'int',

'not null' => TRUE,

'default' => 0,

),

'changed' => array(

'description' => 'The Unix timestamp when the example was most recently saved.',

'type' => 'int',

'not null' => TRUE,

'default' => 0,

),

// .. Any other columns you need in for your entity

),

'primary key' => array('example\_id'),

);

**$schema['example\_revision'] = array(**

'description' => 'Keeps track of example entities.',

'fields' => array(

**'example\_id' => array(**

**'type' => 'int',**

'unsigned' => TRUE,

'not null' => TRUE,

'default' => 0,

'description' => 'Unique example id (entity id).',

),

**'vid' => array(**

**'description' => 'The current {example\_revision}.vid version identifier.',**

**'type' => 'serial',**

**'unsigned' => TRUE,**

**'not null' => TRUE,**

**),**

'changed' => array(

'description' => 'The Unix timestamp when the example was most recently saved.',

'type' => 'int',

'not null' => TRUE,

'default' => 0,

),

// Any other columns that you have in the example entity table that you

// want to store in revisions.

),

**'primary key' => array('example\_id', 'vid'),**

);

}

## Add a \_save function

This assumes your entities being saved are stdClass objects, with an entity ID, a created date, and any other data that you want to push into the database. **New entities to be saved should also have a property called "is\_new" set to TRUE.**

function example\_save($example) {

if ($example->is\_new) {

// Adds the data to the entity table, and returns the new $example\_id:

$example\_id = db\_insert('example')

->useDefaults(array('example\_id'))

->fields(array(

'created' => REQUEST\_TIME,

'changed' => REQUEST\_TIME,

// Any other DB columns that you want to provide defaults for.

))

->execute();

// Add the new entity ID back into the entity object

$example->example\_id = $example\_id;

// Add a revision for all new nodes.

**$example->revision = TRUE;**

}

**if (!empty($example->revision)) {**

// Add the entity data to the revision table, and return the unique revision id.

$vid = db\_insert('example\_revision')

->useDefaults(array('vid'))

->fields(array(

'example\_id' => $example->example\_id,

// We don't need to store the creation time

'changed' => REQUEST\_TIME,

// Any other DB columns that you want to store, and for which you

// need revisioning.

))

->execute();

// Add the revision ID back to the entity

**$example->vid = $vid;**

**// Update the revision ID on the entity table.**

**db\_update('example')**

**->condition('example\_id', $example->example\_id)**

**->fields(array(**

**'vid' => $example->vid,**

**))**

**->execute();**

}

// let Field UI deal with the fields.

if ($example->is\_new) {

field\_attach\_insert('example', $example);

}

else {

field\_attach\_update('example', $example);

}

return $example->example\_id;

}

# Implementing the field management UI

The field\_ui module has a built in field management system for your entity bundles. Enabling this system for your entity is a relatively simple process.

This doc will assume that you want to use a similar format to the core modules, that is, stating entity types (bundles) are listed at admin/structure/example; entity types are added at admin/structure/example/add, and existing entity types are managed, edited, and fielded at admin/structure//manage/. If you want your pages somewhere else, just change the addresses appropriately. Obviously you need to replace 'example' with the name of your module, and "bundle\_key" can be anything you want - eg. for node.module, it is "content\_type".

**To implement this, you need to do five basic things:**

* **Make your entities fieldable.**
* **Implement the required menu structure.**
* **Add a listing page for your bundles.**
* **Add an edit page for your bundle settings.**
* **Add an edit page for your entities.**

### Make entities fieldable

In your hook\_entity\_info implementation you need to do two things: First, make your entity fieldable, then declare the management page for each bundle. Your hook\_entity\_info implementation should include at minimum this:

/\*\*

\* Implements hook\_entity\_info().

\*/

**function example\_entity\_info() {**

$entities['example'] = array(

'label' => t('Example label'),

'fieldable' => TRUE, // This bit is important

'entity keys' => array(

'id' => 'example\_id', // eg. 'nid' in node.module

'bundle' => 'bundle\_key', // eg. 'content\_type' in node.module

),

'bundle keys' => array(

'bundle' => 'bundle\_key', // as above.

),

'bundles' => array( // This is an array of all your bundles. You'll probably want to create it programatically.

// ... Bundles need, at minimum:

'bundle\_one' => array( // This is the bundle\_key.

'label' => 'Bundle One',

'admin' => array(

'path' => 'admin/structure/example/manage/%bundle\_key',

// This is unique to each bundle.

'real path' => 'admin/structure/example/manage/bundle\_one',

'bundle argument' => 4, // Arg 4 is %bundle\_key.

// What ever permission that you've assigned in hook\_permission.

'access arguments' => array('administer example bundles'),

),

// You might want to also include a bunch of other stuff here,

// depending on your entity.

),

),

);

return $entities;

}

### Implement the required menu structure.

You need it implement hook\_menu implementations for each of the following forms. Here's the basics of what you'll probably want in your hook\_menu implementation:

/\*\*

\* Implements hook\_menu().

\*/

function example\_menu() {

$items['admin/structure/example'] = array(

//The main list that displays your bundles, and has links to edit pages.

'title' => 'Example',

'access arguments' => array('administer example'), // Defined in hook\_permission

'page callback' => 'example\_list', // We define this function below.

'file' => 'example.admin.inc', // Not neccesary if defined in the same file

);

$items['admin/structure/example/list'] = array( // As above.

'title' => 'List',

'type' => MENU\_DEFAULT\_LOCAL\_TASK,

'weight' => -10,

);

$items['admin/structure/example/add'] = array(

'title' => 'Add new Example bundle',

'access arguments' => array('administer example'),

'page callback' => 'drupal\_get\_form',

'page arguments' => array('bundle\_key\_form'), // we define this form hook below.

'type' => MENU\_LOCAL\_ACTION,

'file' => 'example.admin.inc',

);

$items['admin/structure/example/manage/%bundle\_key'] = array(

'title' => 'Edit example type',

'access arguments' => array('administer example'),

'page callback' => 'drupal\_get\_form',

'page arguments' => array('bundle\_key\_form', 4), //same hook as above, also passing %bundle\_key

'type' => MENU\_CALLBACK,

'file' => 'example.admin.inc',

);

$items['admin/structure/types/manage/%bundle\_key/edit'] = array( // As above.

'title' => 'Edit',

'type' => MENU\_DEFAULT\_LOCAL\_TASK, // >>> TODO: This tab doesn't display <<<

'weight' => -10,

);

$items['example/%example\_id'] = array( // Page to display your entities.

'title callback' => 'example\_page\_title',

'title arguments' => array(1),

'access arguments' => array('access content'),

'page callback' => 'example\_page', // We define this function below.

'page arguments' => array(1),

);

$items['example/%example\_id/view'] = array( // As above.

'title' => 'View',

'type' => MENU\_DEFAULT\_LOCAL\_TASK,

'weight' => -10,

);

$items['example/%example\_id/edit'] = array( //page to edit your entities.

'title' => 'Edit',

'access arguments' => array('access content'),

'page callback' => 'example\_edit', // We define this function below.

'page arguments' => array(1),

'type' => MENU\_LOCAL\_TASK,

);

return $items;

}

### Add a listing page for your bundles.

Define your

/\*\*

\* List all example predicates.

\*/

function example\_list() {

// Call your hook\_entity\_info implementation, 'cause it includes all your bundles.

$example = example\_entity\_info();

$headers = array(t('Name'), t('Actions')); // Table Headers

$rows = array();

foreach ($example['example']['bundles'] as $name => $bundle) {

// Get the bundle edit link, add it to a table row.

$link = l(t('edit'), 'admin/structure/example/manage/' . $name . '/edit');

$rows[] = array($bundle['label'], $link);

}

// Format the table array. This is a pretty simple table, you could include

// links to the field management pages for each bundle too.

$output = array(

'#theme' => 'table',

'#header' => $headers,

'#rows' => $rows,

);

return $output;

}

### Add an edit page for your bundle settings.

Add the admin/structure/types/manage/%bundle\_key/edit page:

/\*\*

\* Relation example type bundle settings form. This is used for adding and

\* editing bundles.

\*/

function example\_type\_form($form, &$form\_state, $bundle = array(), $op = 'edit') {

// This function contains all the standard information that your entity

// bundles need.

}

The above form also needs a example\_type\_form\_submit() function.

### Add an edit page for your entities.

Add edit and display pages for your entities.

/\*\*

\* Example bundle edit page.

\*/

function example\_edit($example) {

$build = array();

// Add any form items you need for you standard entity information in the

// $build array (eg. node module has stuff here about whether node is

// published, author details, that kind of thing.

// Somewhere in your function, call field\_attach\_form to add the field forms

field\_attach\_form('example', $example, $build, $form\_state = array());

// And return your form

return $build;

}

/\*\*

\* Example entity display page.

\*/

function example\_page($example) {

$build = array();

// Again, load and add all your standard entity data, then call

// field\_atttach\_prepare\_view somewhere to add the field\_display

// >>>> TODO: THIS IS WRONG <<<<

field\_attach\_prepare\_view('example', array($example->example\_id => $example), 'full');

return $build;

}

That's it! If you now go to admin/structure, you'll see your "Example" sub-menu item. If you click there, you'll see a list of bundles, and edit links. You'll also see an "Add new Example bundle" button above the list. If you edit one of the bundles, you'll see the edit form, and two tabs, one for managing fields, one for managing the field display.

# Language support for entity fields

Entity fields natively implement multilingual support. The language associated with a field depends on a number of factors including if translation is enabled and the chosen translation method (e.g. for node entities, either node translation or field translation).

Before looking at the specific field language support, it is important to know the different possible entity configurations. For node entities, content types can be configured as follows:

* Locale module is disabled
  + No language assignment
  + No translation
* Locale module enabled
  + Language assignment
  + No translation
* Locale and Content translation modules enabled
  + Language assignment
  + Translation via node translation method (multiple nodes)
* Locale and Entity translation modules enabled
  + Language assignment
  + Translation via field translation method (one node)

To learn more about the two translation methods, check out the documentation on translating content into different languages. Note that the Content translation module only handles node entities, so other core entities must use the Entity translation module for translation.

As we will see, the entity configuration will affect what languages can be assigned to fields and also what the ubiquitous LANGUAGE\_NONE (und) means for each case.

**Handling of multilingual fields**

Fields implement multilingual support by holding multiple values in the field array. All fields use the following structure:

**$entity->{$field\_name}[$langcode][$delta][$column\_name]**

Here are some examples:

$entity->body[und][0][value];

$entity->image[und][0][value];

$entity->description[en][0][value];

$entity->description[fr][0][value];

$entity->history[en][0][value];

$entity->history[fr][0][value];

When dealing with entities, it is important to understand what entity languages and field languages are appropriate for each of the entity configurations. For node entities, the following figure provides a high-level overview of the possible language values and the different uses of **LANGUAGE\_NONE (und).**

### Locale module is disabled

By default a single-language site in English without the Locale module enabled uses LANGUAGE\_NONE as the $langcode, and there is always only one language variant for each field.

**$entity->body[und][0][value];**

**$langcode = entity\_language('entity\_type', $entity);**

**$langcode == 'und'**

### Locale and Content translation modules enabled

With the core Locale and Content translation modules enabled, fields will not have their own language information, and LANGUAGE\_NONE is used to signify that we don't have a field language specified. The whole entity with all its properties and fields is assumed to have the entity language code assigned (conceptually).

**$entity->body[und][0][value];**

**$langcode = entity\_language('entity\_type', $entity);**

**$langcode == 'en'**

### Locale and Entity translation modules enabled

Field translation is pluggable, but the only known implementation is the contributed [Entity translation](https://www.drupal.org/project/entity_translation) module that provides a user interface to enable field translation for entities and to change the translatability on fields.

#### Shared fields

When we enable field translation, a field that does not have translation enabled will always use LANGUAGE\_NONE, and no other language values will be present. The meaning of this is different from the above. It means that the field will have the same value across all translations. This is also called a shared field. The $entity->langcode value will persist to represent the language of the properties of the entity, such as the author and status, as well as signify the original language that the entity was saved with. That information can be useful for permission and workflow purposes.

$entity->body[und][0][value]; // not translatable (shared)

$langcode = entity\_language('entity\_type', $entity);

$langcode == 'en' // applies to properties of the entity

#### Translatable fields

**For a field with translation enabled, multiple variants of the value may be available in any of the enabled languages.**

**$entity->body[en][0][value];**

**$entity->body[de][0][value];**

**$langcode = entity\_language('entity\_type', $entity);**

**$langcode == 'en' // applies to properties of the entity**

## Multilingual field API

**The available** languages for a particular field are returned by **[field\_available\_languages()](http://api.drupal.org/api/drupal/modules%21field%21field.multilingual.inc/function/field_available_languages/7)**, which abstracts the above logic.

**Whether a field is translatable** can be determined by calling**[field\_is\_translatable()](http://api.drupal.org/api/drupal/modules%21field%21field.multilingual.inc/function/field_is_translatable/7) directly.**

If you are interested in looking up whether an **entity type has field translation support implemented** by one of the enabled modules (in the most common use case that entity\_translation module is enabled and this entity type is configured with it to support field translation), use**[field\_has\_translation\_handler()](http://api.drupal.org/api/drupal/modules%21field%21field.multilingual.inc/function/field_has_translation_handler/7)** on the entity type.

Given that the entity structure contains all the different language versions of fields when field translation is used, we need to pick from them for display. The **[field\_language()](http://api.drupal.org/api/drupal/modules%21field%21field.multilingual.inc/function/field_language/7)** function can be used to determine the **display language** for the fields attached to the given entity. That handles the fallback logic defined by handlers to pick the right language of each field for display. The [field\_get\_items()](http://api.drupal.org/api/drupal/modules!field!field.module/function/field_get_items/7) function can be used to return field values in that language.

When acting on fields, by default, [\_field\_invoke()](http://api.drupal.org/api/drupal/modules%21field%21field.attach.inc/function/_field_invoke/7) and [\_field\_invoke\_multiple()](http://api.drupal.org/api/drupal/modules%21field%21field.attach.inc/function/_field_invoke_multiple/7) process fields in all available languages. They allow to limit the operation to a specific language or a set of languages.

The [entity\_language()](http://api.drupal.org/api/drupal/includes!common.inc/function/entity_language/7) function can be used to retrieve an entity's language in a generic fashion. The value returned by entity\_language() is the recommended value to be passed as $langcode parameter of the [field\_attach\_form()](http://api.drupal.org/api/drupal/modules!field!field.attach.inc/function/field_attach_form/7) function, when the logic being implemented does not explicitly dictate a different one.