



# Developing Achievo Extensions

## Part 2 - Advanced

Ivo Jansch <ivo@achievo.org>  
© 2002

Last modified: August 28, 2002

## Contents

<b>1 Relationships</b> .....	<b>4</b>
1.1 Many-to-one relation .....	4
1.1.1 Description .....	4
1.1.2 Preparations .....	4
1.1.3 Implementing the relation.....	6
1.2 One-to-many relation.....	7
1.2.1 Description.....	7
1.2.2 Preparations .....	7
1.2.3 Implementing the relation.....	8
1.3 Many-to-many relation .....	9
1.3.1 Description.....	9
1.3.2 Preparations .....	10
1.3.3 Implementing the relation.....	12
1.4 Conclusion .....	13
<b>2 Triggers</b> .....	<b>15</b>
2.1 Types of triggers .....	15
2.1.1 Post-triggers .....	15
2.1.2 Pre-triggers .....	15
2.2 Implementing a trigger .....	15
2.3 Keeping track of record changes.....	17
<b>3 Improving user-friendliness</b> .....	<b>19</b>
3.1 Setting default values .....	19
3.2 Improving the menu.....	19
3.3 Language files .....	20
3.4 Tweaking the user-interface .....	21
<b>4 Conclusion</b> .....	<b>23</b>
<b>Appendix A – Completed source</b> .....	<b>25</b>

## Introduction

So, you've read part 1 of the guide, perhaps even created some modules for Achievo, and you are ready to read about the more powerful features of the software.

In part 1, you've seen how you can build nodes, which represent management of information. In this guide, I will show you how these nodes can have relationships with other nodes. After that, I will explain 'triggers', functions that you can implement that are executed at certain events (for example after the insertion of a new record). Finally, I will explain some smaller topics that can make your application more user-friendly, like how to make your module support multiple languages. These three topics aren't really related, but it's better for you general understanding of the inner workings of Achievo, to discuss them in this order.

Before we start, I would like to thank Dave Nuttall, Lineke Willems and Michiel Schalkx for proofreading this guide and helping me debug it, and Tom Schenkenberg for numerous PDF conversions (I promise this is the final version Tom!).

### Prerequisites

Before starting to work with this guide, you should have at least read part 1. If you have a fair understanding of the things discussed there, and have toyed around with the example code, this guide should not be too hard to understand.

Some knowledge about relations in a database context (referential keys, etc.) is practical, but not absolutely necessary, as I will try to explain the relationships in very basic terms.

I will continue to use the example of the Pizza management application, so you should use the code from part 1 as a starting point for the examples in this guide. If you haven't implemented the classes from the previous guide, check out its appendix, which contains the complete example code.

# 1 Relationships

Entities in a data model (tables) are seldom ‘stand alone’. Most of the time, they have some kind of relationship with surrounding entities.

For example in Achievo, a project is related to a customer, a coordinator and to phases. A phase is related to activities, etc.

Achievo’s backend contains features to quickly implement these relations in a user-friendly manner.

In the previous guide, you have seen that nodes contain attributes, that represent fields in the database. Just like attributes, a node can contain relations to other nodes. The addition of a relationship is just as easy as adding an attribute. There are several types of relations, each of which will be discussed in this chapter. The difference between the types of relations is made based on their ‘cardinality’ (for those of you unfamiliar with that term: cardinality indicates how many items can be involved in a relationship).

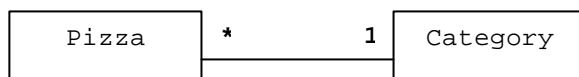
## 1.1 Many-to-one relation

### 1.1.1 Description

Of the three basic relation types, the many-to-one relation is probably the hardest to explain, but the easiest to implement, so I’ll start with this type of relation.

Many-to-one means, as seen from our node, that many records of this type have a relationship with one record of some other node. As an example to explain this mumbo jumbo, we’re going to introduce ‘pizza categories’ and create a relationship between pizzas and categories.

Suppose we have a set of categories, (for example ‘small’, ‘medium’ and ‘large’). Suppose each pizza falls into one category. In one category, there can be many pizzas. So, from the pizza point of view, you might say there is a many-to-one (many-pizzas-in-one-category) relation with categories.



Such a many-to-one relation is represented in the pizza edit-screen by a dropdown box from which the user can select in which category the pizza belongs.

Ok, let’s implement this.

### 1.1.2 Preparations

First, we need a place to store categories. A category will only have a name, and of course, a primary key. This will lead to a simple table definition. For MySQL, executing the following SQL statement on the Achievo database, should create the table for storing the categories:

```
CREATE TABLE pizza_categories (  
  id int(10) NOT NULL,  
  name varchar(50) ,  
  PRIMARY KEY(id)  
);
```

Then, we need to create a node in the pizza module. I'm not going to explain how to do this, as you already know everything you need to know to create such a node. But, just in case, here's the basic implementation:

```
<?php

class pizza_category extends atkNode
{
    function pizza_category()
    {
        $this->atkNode("pizza_category");
        $this->add(new atkAttribute("id", AF_AUTOKEY));
        $this->add(new atkAttribute("name", AF_OBLIGATORY |
                                AF_UNIQUE |
                                AF_SEARCHABLE));

        $this->setTable("pizza_categories");
        $this->setOrder("name");
    }
}

?>
```

This should all look familiar to you. The only new thing you might have noticed is the use of the flag `AF_AUTOKEY` for the id attribute. In the previous guide, we used the flags `AF_PRIMARY|AF_HIDE|AF_AUTO_INCREMENT`. Actually, `AF_AUTOKEY` is a kind of 'shortcut flag', which does exactly the same as specifying the three flags separately. Saves you some typing on those primary key fields, which usually have these flags.

Save this code as `class.pizza_category.inc` in the pizza module directory. Next, we must add a menu-item to the Achievo menu, and setup access rights for the new node, by adding two lines to the appropriate functions in our `module.inc`:

In the `getMenuItems()` function, add this line:

```
menuItem("pizza_categories", dispatch_url("pizzaman.pizza_category",
                                         "admin"));
```

In the `getNodes()` function, add this line:

```
registerNode("pizzaman.pizza_category", array("admin", "add",
                                              "edit", "delete"));
```

If these lines are not clear to you, read over the parts that deal with these issues in part 1 of this guide.

If you browse the Achievo installation now, you should be able to add some categories to the database. Add a few ('Small', 'Medium', 'Large' for example), we'll use them later on.

There's one preparation left to do, before we can implement the relation. In the pizza table, we need a new field to store the chosen category. Let's call this field 'category'. People might be in favor of calling such a referential key 'category\_id', but I prefer to just name the field after the 'role' it performs. Use whatever you're used to.

The referential key must be of the same field type as the key it refers to (the id field in the `pizza_categories` table), so the following statement should alter the pizza table to add the correct field:

```
ALTER TABLE pizza ADD category int(10) NOT NULL DEFAULT '0';
```

Execute this statement on the Achievo database. We're done with the preparations.

### 1.1.3 Implementing the relation

Implementing the relation is actually very simple. Paragraph 1.1.1 was a bit lengthy, but that was because we first had to create something to create a relation to.

So we planned to implement a many-to-one relation in the pizza node, for which we added a 'category' field to the pizza table.

Relations are actually a special kind of attribute. They're added to a node in exactly the same way as 'regular' attributes.

Open the class.pizza.inc file, and add the following line to the top of the file.

```
useattrib("atktextattribute");
useattrib("atknumberattribute");
useattrib("atkdateattribute");
userelation("atkmanytoonerelation");
```

Just like attributes, we have to specify this line so ATK can include the correct files. (Like in the previous guide, this line is only necessary if you run Achievo 0.9 or higher.)

Then, right after the attribute we added for the 'entrydate' field, we add a line to add the relation to the node:

```
$this->add(new atkNumberAttribute("price"));
$this->add(new atkDateAttribute("entrydate"));
$this->add(new atkManyToOneRelation("category",
                                   "pizzaman.pizza_category",
                                   AF_OBLIGATORY|AF_SEARCHABLE));
```

The first parameter we pass to the `atkManyToOneRelation` is the name of the field that the relation is stored in. In our case, this is the 'category' field. The second parameter is the *destination* of the relation. We specify this as *modulename.nodename*, since a node with the same name might already exist in another module. Also, because this way you could create relations to nodes in other modules. (To create a relation to an existing Achievo base node, like 'employee', you don't need to specify a module name, but we'll talk about this later.)

As third parameter, we specify that 'category' is an obligatory field (we don't want pizzas that don't fall into any of the categories), and that the field is searchable (so we can quickly view all pizzas from a certain category).

Let's test what we have so far.

If you now edit a pizza in the pizza admin screen, you can see the dropdown-box with the categories. But you will notice immediately that the dropdown box only contains numbers! These are the id's of the `pizza_categories` table. ATK can not guess which values it should use to display in the dropdown-box, so we have to tell it. We do this by adding a small function to the `pizza_category` class:

```
    $this->setOrder("name");
}

function descriptor_def()
{
    return "[name]";
}
```

```
}  
}
```

The system automatically calls this function to determine what to display in the drop-down box when this node is used in a relation. What the function returns is actually a *template*. It's a string that contains fieldnames between brackets. In this case, we use the 'name' field for the dropdown box. Had this been a 'person' node for example, we might have returned "[lastname], [firstname]". The fieldnames in brackets get replaced at runtime by fields from the table.

After adding this little function, re-open the pizza edit screen. Instead of a dropdown with numbers, you should now see a list of category names.

This completes the example of the many-to-one relation. We had to take some preparations at first, because we didn't have any nodes to link to yet, but once we had created the node, we linked them using only very few lines of code.

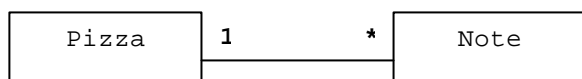
## 1.2 One-to-many relation

### 1.2.1 Description

The 'opposite' of a many-to-one relation is the one-to-many relation. This relation is used when we speak of one pizza having a relation with many records from another node. (This type of relation is sometimes also called a 'master-detail' relationship.)

At first, I wanted to use 'ingredients' as a one-to-many example, but that isn't a one-to-many, but many-to-many relation (as you will see in paragraph 1.3).

We should take something simple as an example. So let's add 'notes' to a pizza. A way for the pizza bakers to add hints for their colleagues, like 'when you put tuna fish on this pizza, leave out the spinach; it doesn't combine well'. A pizza can have many notes, and each note belongs to one specific pizza, so from the pizza point of view, there's a one-to-many relation to notes.



In the user-interface, a one-to-many relation is represented by a list of records in the edit screen of the master record, so in this case, a list of notes in the pizza edit screen.

### 1.2.2 Preparations

First, we create a table for storing the notes. For the purpose of this example, a note consists only of a text field (you might want to add an author, and a date later on). Of course we also need to add a primary key (id), and we need a field to store to which pizza the note belongs, which I would call `pizza_id`. (Note that in paragraph 1.1.1, I chose the name `category_id` over `category_id`. I did that because I wanted to name the field after the role it plays. In this case however, the field does not have any special meaning, it's just a way of linking notes to pizzas, and we will not even display the field.)

```
CREATE TABLE pizza_notes (  
  id int(10) NOT NULL,  
  pizza_id int(10) NOT NULL default '0',  
  note text,
```

```
    PRIMARY KEY(id)
);
```

After executing this query on the database, we also need to create a node file for notes. There are a few things to notice about this node, which I will explain in a minute. Here's the implementation of the `pizza_note` node, which you should save as `class.pizza_note.inc` in the `pizza` module directory:

```
<?php

// next two lines are only needed when using Achievo 0.9 or higher:
useattrib("atktextattribute");
userelation("atkmanytoonerelation");

class pizza_note extends atkNode
{
    function pizza_note()
    {
        $this->atkNode("pizza_note");
        $this->add(new atkAttribute("id", AF_AUTOKEY));
        $this->add(new atkTextAttribute("note"));
        $this->add(new atkManyToOneRelation("pizza_id", "pizzaman.pizza",
                                          AF_HIDE));

        $this->setTable("pizza_notes");
        $this->setSecurityAlias("pizzaman.pizza");
    }
}

?>
```

The first thing to notice here is that the note contains a many-to-one relation. The current implementation of the one-to-many relation is such that it requires that there is a many-to-one relation on the other side. So, if we want to create a one-to-many from a pizza to notes, there should be a many-to-one relation from notes to a pizza. We pass the `AF_HIDE` flag to this relation though, because we don't want to display the pizza, as we will only see the notes in the context of the pizza to which they belong.

The second thing to notice is the line that follows the `setTable` statement. Remember that for every node we implemented until now, we added a `menuItem()` call and a `registerNode()` call to the `module.inc` file? Well, notes will not have a menu-item, since they will be edited from inside the pizza edit screen. The `registerNode()` call was used so we could grant rights to users to manage the node. You could add a `registerNode()` call for this node, so you could grant the right to edit notes to people. But in my opinion, if you are allowed to edit a pizza, you are also allowed to edit its notes. That's where this last line, with the `setSecurityAlias()` function call, comes in. It makes the `pizza_note` node equal to the `pizza` node, from a security point of view. If someone wants to edit a note, the system will now check if this person has the right to edit a pizza. This helps to keep the number of checkboxes in the security profile screen of Achievo smaller.

### 1.2.3 Implementing the relation

As with the previous relation, the preparations are again more work than the actual implementation of the relation, but we must first create something to create a relation to. (I could've used existing Achievo classes to relate to, but then you would've missed some important explanations I did in the previous paragraph).

We will implement a one-to-many relation in the pizza node, so open up the class.pizza.inc file.

If you use Achievo 0.9 or higher, add the following line to the top of the file:

```
useattrib("atkdateattribute");
userelation("atkmanytoonerelation");
userelation("atkonetomanyrelation");
```

Right below the many-to-one relation ‘category’ that we added in paragraph 1.1, we add the one-to-many-relation:

```
$this->add(new atkDateAttribute("entrydate"));
$this->add(new atkManyToOneRelation("category",
    "pizzaman.pizza_category",
    AF_OBLIGATORY|AF_SEARCHABLE));

$this->add(new atkOneToManyRelation("notes",
    "pizzaman.pizza_note",
    "pizza_id",
    AF_HIDE_LIST|AF_CASCADE_DELETE));
```

When comparing this line to the previous line, the many-to-one relation, you will notice that there’s one extra parameter. The third parameter (“pizza\_id” in our case), is the field in the target class that links back to this node. In other words, this is the field that the system uses to load all notes that belong to this pizza (pizza\_id must be equal to the id of the current pizza). You might argue that the system should be able to determine this on its own, because of the many-to-one relation that we implemented in the pizza\_note class. In a more complex system however, there may be more than one relation between two nodes (consider for example two many-to-one relations between projects and employees. One relation might indicate the technical coordinator of the project, whereas the other relation might indicate the account manager), so it’s necessary to pass this third parameter, to link the correct field with this relation.

As fourth parameter to the one-to-many relation we pass two flags that you haven’t seen before. The AF\_HIDE\_LIST flag indicates that this field is not shown in the list of pizzas. In the pizza admin screen, we don’t want to see the notes from the pizzas (this would obfuscate the admin-screen), so we use this flag. The notes will become visible when we edit the pizza. The AF\_CASCADE\_DELETE flag makes sure that notes will be deleted from the database, if the pizza they belong to is deleted. This flag is often forgotten, leaving orphaned records in the database. Make sure you set the flag when needed.

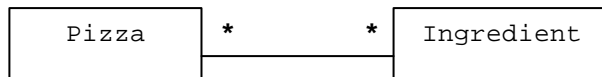
Ok, that’s it. You can now edit a pizza and add some notes to it, via the ‘pizza note add’ link in the edit screen (later on we will change the text to a more user-friendly string).

## 1.3 Many-to-many relation

### 1.3.1 Description

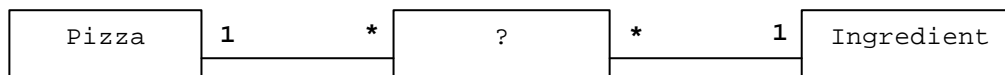
The last type of relation, the many-to-many relation might be difficult to grasp if you do not have much experience with database relations. I’ll try to explain it as easy as possible.

A perfect example of a many-to-many relation in pizza terms is: ‘ingredients’. A pizza has many ingredients, and the same ingredient can be used on many pizzas.



These kinds of relations cannot be stored like this in the database. We can't add a `pizza_id` field to the `ingredient` table, because that way each ingredient can only be linked to one pizza. We can't add an `ingredient_id` field to the `pizza` table either, because then each pizza can only contain one ingredient (ok, you could argue that a pizza with only tomato is a pizza, but I doubt you will sell many of them).

What we need to do here is 'normalize' the relation. We need a third table that holds the information about which ingredient goes on what pizza. If we draw a model of this, we get:



For each pizza in the `pizza` table, there are several records in the intermediary table, each describing an ingredient that goes on the pizza. Likewise, for each ingredient, there are several records, each describing a pizza on which the ingredient goes.

Look what we now have. Instead of a many-to-many relation, we now have a set of many-to-one and one-to-many relations! This is the way to handle all many-to-many relations. And we already discussed those in the previous paragraphs! So actually I'm not teaching you anything new. I will just be showing you what is the best approach to implementing this.

### 1.3.2 Preparations

First, we need two tables, one for ingredients, and one intermediary table.

The ingredients table can be simple. For this example, a name and a primary key (`id`) are sufficient. Here's the SQL statement to create this table:

```

CREATE TABLE ingredients (
  id int(10) NOT NULL,
  name varchar(50),
  PRIMARY KEY(id)
);
  
```

Now we need to define and create the intermediary table. It is a custom to name the intermediary table after the tables it links, for example `pizza_ingredients`. However, I prefer to give them a logical name, whenever possible (it can be hard to find a name for some relations).

A logical name here would be 'recipe', since a recipe describes exactly what ingredients go on a pizza.

The 'recipe' table contains two fields, a `pizza_id` and an `ingredient_id`. We don't need an extra primary key, because we can use both fields as primary key, because the combination of a `pizza_id` with an `ingredient_id` is unique (we don't want to add the same ingredient to the pizza twice).

We can also add some extra fields to intermediary tables, which might be considered 'properties of the relation'. A logical property in the case of recipes would be 'amount', indicating how many of the ingredient to use on the pizza.

We could use an integer for the amount field, but since each ingredient has a different unit of measurement (milligrams, liters, pieces) etc, that is not enough. Cooks tend to use ‘fuzzy’ terms when it comes to recipes (‘add one half teaspoon of this, and just a sniff of that’), so it might be better to just use a string. (This means we could never use the field to do calculations, or to use it in stock management, so we would have to revise this later, should we ever want to implement such a feature. But for the purpose of this example, the string approach is sufficient.)

This leads to the following SQL statement for the recipe table:

```
CREATE TABLE recipe (  
    pizza_id int(10) NOT NULL,  
    ingredient_id int(10) NOT NULL,  
    amount varchar(100),  
    PRIMARY KEY (pizza_id, ingredient_id)  
);
```

Now we must create a node for the ingredients table. Since we have an intermediary table, we should also create a node for this table.

We must first make some decisions. We’re going to add ingredients to a pizza. Not the other way round. For some relations you could edit the relation both ways, for example, in the case of user/group relationships, you could both edit a user and add her/him to several usergroups, or you could edit a usergroup and add several users to it. In our pizza case however, it does not make much sense to add pizzas to an ingredient. This means, we will only have to implement the one-to-many relation from the pizza point of view.

This leads to a very basic implementation of the ingredient node:

```
<?php  
  
class pizza_ingredient extends atkNode  
{  
    function pizza_ingredient()  
    {  
        $this->atkNode("pizza_ingredient");  
        $this->add(new atkAttribute("id", AF_AUTOKEY));  
        $this->add(new atkAttribute("name", AF_UNIQUE |  
                                   AF_OBLIGATORY |  
                                   AF_SEARCHABLE));  
  
        $this->setTable("ingredients");  
        $this->setOrder("name");  
    }  
  
    function descriptor_def()  
    {  
        } return "[name]";  
    }  
}
```

?>

We store this code as *class.pizza\_ingredient.inc* in the pizza module directory. Nothing hard to explain here, such a node should now be familiar to you. We already added a *descriptor\_def* function, so an ingredient is displayed with its name, when it is used in relations. In fact, it is good practice to always provide a *descriptor\_def()* function in your node class, as you will never know what nodes might get related to your class later on.

Now we must edit the module.inc file, to add a menu-item where ingredients can be managed:

```
function getMenuItems()
{
    menuitem("pizza", dispatch_url("pizzaman.pizza", "admin"));
    menuitem("pizza_categories", dispatch_url("pizzaman.pizza_category",
                                             "admin"));
    menuitem("pizza_ingredients", dispatch_url("pizzaman.pizza_ingredient",
                                             "admin"));
}
```

Finally, we need to add a registerNode() call to the getNodes() function in the module.inc file, so we can grant the right to manage ingredients to people. We could do the same as in paragraph 1.2.3, where we made notes equal to pizzas from a security point of view, but adding new ingredients is more of a setup issue, and could be done by another person than the one who adds the pizzas.

So, this is what the getNodes function should look like by now:

```
function getNodes()
{
    registerNode("pizzaman.pizza", array("admin", "add",
                                         "edit", "delete"));
    registerNode("pizzaman.pizza_category", array("admin", "add",
                                                  "edit", "delete"));
    registerNode("pizzaman.pizza_ingredient", array("admin", "add",
                                                    "edit", "delete"));
}
```

You can test the ingredient node now. Open Achievo, and via the ingredient menu item, add some ingredients (for example 'cheese', 'tomato', and whatever you will be putting on your pizza).

Now we will implement the recipe node, and add the relation to the pizza node.

### 1.3.3 Implementing the relation

As you recall from paragraph 1.2.2, every one-to-many relation needs a many-to-one relation on the other side. So if we implement a one-to-many relation from the pizza node to the recipe node, the recipe node should have a many-to-one relation to the pizza node (on the pizza\_id field). This field may be hidden, since we are already in the edit-screen of a particular pizza, so we don't need to see for each recipe record to which pizza it belongs.

When we add a recipe record to a pizza, we would like to see a dropdown of all available ingredients. Following the model from 1.3.1, there's a many-to-one relation between the recipe node and ingredient, and a many-to-one relation is automatically represented by a dropdown box.

This leads to the following implementation of the recipe node (class.pizza\_recipe.inc):

```
<?php

    userrelation("atkmanytoonerelation"); // only needed for Achievo 0.9 or
                                         // higher

    class pizza_recipe extends atkNode
    {
        function pizza_recipe()
```

```

    {
      $this->atkNode("pizza_recipe");
      $this->add(new atkManyToOneRelation("pizza_id",
                                        "pizzaman.pizza",
                                        AF_PRIMARY|AF_HIDE));
      $this->add(new atkManyToOneRelation("ingredient_id",
                                        "pizzaman.pizza_ingredient",
                                        AF_PRIMARY));
      $this->add(new atkAttribute("amount"));
      $this->setTable("recipe");
      $this->setSecurityAlias("pizzaman.pizza");
    }
  }
}
?>

```

Notice that both `pizza_id` and `ingredient_id` have the `AF_PRIMARY` flag, as they together form the primary key of the table. I also use the `setSecurityAlias` function again, as I think that people who can edit pizzas should also be able to edit the recipe.

Now, one line left to add, is the one-to-many relation between the pizza node and the recipe node. In `class.pizza.inc`, we add the relation:

```

$this->add(new atkManyToOneRelation("category",
                                   "pizzaman.pizza_category",
                                   AF_OBLIGATORY|AF_SEARCHABLE));

$this->add(new atkOneToManyRelation("notes",
                                   "pizzaman.pizza_note",
                                   "pizza_id",
                                   AF_HIDE_LIST|AF_CASCADE_DELETE));

$this->add(new atkOneToManyRelation("ingredients",
                                   "pizzaman.pizza_recipe",
                                   "pizza_id",
                                   AF_HIDE_LIST|AF_CASCADE_DELETE));

```

And that's it. You should now be able to add ingredients to a pizza. Notice how notes and ingredients behave the same in the user-interface. The fact that ingredients is a many-to-many relation is only visible through the dropdown box when you add an ingredient to the pizza. The most important thing to remember from this chapter is: a many-to-many relation should always be normalized into an intermediary node and a set of one-to-many and many-to-one relations.

## 1.4 Conclusion

Notice how, like in part 1 of this guide, we did not have to implement any query yet! We have created several types of relations between nodes, and the system takes care of every single query for inserting the correct record, reading the correct record etc.

Another strong point of Achievo's backend is the amount of code required to implement relationships. Once you have nodes, creating the actual relationships between them only takes a few lines of code: 1 line for a many-to-one relation, 2 lines for a one-to-many-relation (1 in the source and 1 line in the destination node) and about 10 lines for a many-to-many relation (1 line in the source node, and the implementation of an intermediary node).

You might have noticed, that when adding a new pizza, you can't immediately add ingredients or notes. This is because at that moment, there isn't a pizza record yet to which

these relations can link. This is a bit awkward, since you have to edit the pizza after adding it, to start filling it with ingredients. In paragraph 3.5, we will change some things in the user-interface to remedy this.

## 2 Triggers

Now that we've covered the relationships between nodes, we're going to discuss a completely different subject. Triggers are functions that are called when a record is manipulated. In this chapter, you will see what triggers there are, and how to implement them.

### 2.1 Types of triggers

A record can be added, edited or deleted. For each of these actions triggers can be defined. There are two kinds of triggers: Post-triggers and Pre-triggers.

#### 2.1.1 Post-triggers

Post-triggers are called right after a record has been manipulated. The data is already been changed in the database, so you can't do any processing on the data. This type of trigger is useful for example for sending out mail notices. For example when a pizza is added, you might want to mail the pizza bakers that there is a new pizza available. Or when a pizza is deleted, you might need to remove it from your cash registers.

Because the trigger is called after the data is updated in the database, the data can not be modified in a post-trigger.

#### 2.1.2 Pre-triggers

Pre-triggers are called right before the database operations will be performed. After (!) the user has clicked the save-button, but before the record is validated and before the record is modified in the database, the trigger is called. This is useful for performing some last-minute changes to the record.

Because this type of trigger is called before the data is updated in the database, it is possible to modify the record from inside the trigger.

Note: pre-triggers are only available in Achievo 0.9.1 or higher.

## 2.2 Implementing a trigger

A trigger is implemented by adding a function to your node class. The following function names are available:

postAdd:	Executed right after a new record has been inserted into the database.
postUpdate:	Executed after a record has been modified and updated in the database.
postDel:	Executed after a record has been deleted.
preAdd:	Executed before a new record is inserted into the database.
preUpdate:	Executed before a record will be updated in the database.
preCopy:	Executed before a record is copied.

One parameter is always passed to every trigger: the current record. This is an associative array containing all values of all the fields. There's a small difference between pre- and post-triggers though. The record is passed 'by value' to the post-trigger. This means, as I wrote in paragraph 2.1.1, that you can't modify the record that is passed to you (which wouldn't make sense, since the data has already been saved).

This is what a post-trigger might look like in your code:

```
function postUpdate($record)
{
    // Perform some things here.
}
```

As described in paragraph 2.1.2, in pre-triggers the record can be modified. Therefore, the record is passed 'by reference' to the pre-trigger. Here is an example of a pre-trigger:

```
function preUpdate(&$record)
{
    // Perform some things here, modify $record, etc.
}
```

As an example we're going to add a postAdd trigger to the pizza class, in which we're going to send a mail notification to someone.

Following the above example, we're going to edit class.pizza.inc and add the trigger function:

```
$this->setTable("pizza");
}

function postAdd($record)
{
    $subject = "New pizza: ".$record["name"];

    $body = "A new pizza has been added:\n";
    $body.= "Name: ".$record["name"]."\n";
    $body.= "Description: \n";
    $body.= $record["description"]."\n";

    usermail("youraddress@yourdomain.tld", $subject, $body);
}
}
```

In the above function, we use the name and description fields from a pizza. They are both present in the record.

The function *usermail()* is an Achievo wrapper around PHP's standard *mail()* function. It puts a standard prefix ('[achievo notice]') in front of the subject, and uses 'achievo' as sender name, so mails that users receive from Achievo are easily recognized. This function is located in the file *achievotools.inc*, which is not included by default. Therefore, before this function can be called, we add the following line to the top of the *module.inc* file:

```
<?php
require_once("achievotools.inc");

class mod_pizzaman extends atkModule
{
```

(Note: it would be more optimal to include the file only in the file we need it (*class.pizza.inc* in this case), but since we'll be using the *usermail()* function again later on, it's easier for now to put the include in the *module.inc* file.)

After implementing this function, you should receive mail whenever someone adds a pizza to the database.

## 2.3 Keeping track of record changes

Sometimes in the `postUpdate` trigger, you want to base your actions on the fact that a record has changed. The `postUpdate` function is the only trigger where this makes sense and where this feature is implemented.

By default, only the current record is passed to the trigger, so you don't know what has changed in the record. We can change this behavior by using a 'node flag': Just like we use flags to influence the behavior of attributes (the `AF_` parameters), we can use flags to influence the behavior of the entire node.

Remember how every node has a line like the following in its constructor:

```
$this->atkNode("nodename");
```

The `atkNode` function actually accepts a second parameter, where we can pass node flags. Whereas attribute flags are prefixed with `AF_`, node flags are prefixed with `NF_`.

The flag we should use here is `NF_TRACK_CHANGES`. If this flag is set, the system passes both the original and the modified record to the trigger function. (We will see other node flags later in this guide. For an overview of all node flags, see the API documentation.)

So the first line of the constructor of the pizza node now becomes:

```
$this->atkNode("nodename", NF_TRACK_CHANGES);
```

For backward-compatibility reasons, and because we shouldn't change a function definition based on a flag that could be on or off at runtime, the trigger still receives only one parameter. The original record is passed as a special part of the `$record` parameter, namely `$record["atkorgrec"]`. This original record has the same structure as the new record, so you can easily verify if there are any changes.

Let's demonstrate this with an example. Suppose we want to receive a mail whenever the name of an ingredient changes. This sounds logical, since renaming an ingredient could be a dangerous practice: if someone renames 'tomato' to 'garlic', then most of the pizzas will end up with so much garlic you'll have to distribute gasmasks along with them.

First we activate the feature by setting the correct flag in the first line of the constructor in `class.pizza_ingredient.inc`:

```
$this->atkNode("pizza_ingredient", NF_TRACK_CHANGES);
```

The following function in `class.pizza_ingredient.inc` will send you a mail whenever the name of an ingredient is changed:

```
    return "[name]";
}

function postUpdate($record)
{
    $orgname = $record["atkorgrec"]["name"];
    $newname = $record["name"];
    if ($newname!=$orgname)
    {
        $subject="WARNING: ingredient $orgname was renamed to $newname!";
        $body = "Take caution, this might corrupt your recipes.\n";
    }
}
```

```
        usermail("youraddress@yourdomain.tld", $subject, $body);
    }
}
```

This code is pretty straightforward, so I don't think it needs any additional explanation. After adding this function, you can try it by renaming an ingredient. If you edit an ingredient and save it without changing the name, you should not get any mail.

## 3 Improving user-friendliness

By now, we've added a lot of features to our pizza module. There are some areas though where we can improve the user-friendliness of the application. Just as in part 1 of the guide, I've left these issues until the end of the guide.

### 3.1 Setting default values

Sometimes you want a field in an add-screen to be filled with a default value. Let's assume we want to enter a default price of 6.50 when we add a new pizza. Also the entrydate should automatically be set to today's date. We can do this by creating a new function in the pizza node in *class.pizza.inc*:

```
    $this->setTable("pizza");
}

function initial_values()
{
    return array("price"=>"6.50",
                "entrydate"=>date("Y-m-d"));
}

function postAdd($record)
```

If the `initial_values()` function exists in your node, the system calls it to determine initial values for the attributes in your node. It doesn't have any parameters. It is supposed to return an associative array, with for each attribute name, a default value. In this case, we set a default value of "6.50" for the price attribute. (Note the quotes around the number. You could leave them out, but then the value in the box would be rounded to '6.5'.)

A thing I should tell you, is that the format of the value you specify in the `initial_values` function depends on the type of the attribute. For example for `entrydate` (an `atkDateAttribute`), I specify the date in format `date("Y-m-d")`. It would also accept an associative array with a year, month and day element. Every attribute can have its own way of specifying default values. Most just accept a 'literal' value (a value that you could also have inserted directly in the database).

### 3.2 Improving the menu

In this guide, we added 2 menu options to the Achievo menu, so we now have three pizza-related menu-items. It would be nice to create a separate submenu for the pizza manager. We need to make a little adjustment to the `getMenuItems()` function that we implemented in the *module.inc* file:

```
function getMenuItems()
{
    menuitem("pizzaman");
    menuitem("pizza", dispatch_url("pizzaman.pizza", "admin"), "pizzaman");
    menuitem("pizza_categories", dispatch_url("pizzaman.pizza_category",
                                             "admin"), "pizzaman");
    menuitem("pizza_ingredients", dispatch_url("pizzaman.pizza_ingredient",
                                              "admin"), "pizzaman");
}
```

The first new line creates an empty menu-item. This is used to create a submenu. In the three existing *menuItem()* calls, there's now a third parameter. This parameter indicates in what submenu the menu-items appear (default is the main menu, if this parameter is not specified). When you browse Achievo now, you should see only one pizza item in the main menu, which opens a submenu when clicked.

Another improvement we can do is related to security. We have implemented security such, that we can grant users the right to manage pizza's. By default, the pizza menu item in the main menu is always displayed. The *menuItem()* function accepts a fourth parameter that indicates what rights a person needs to see the menu-item:

```
function getMenuItems()  
{  
    menuItem("pizzaman", "", "main", array("pizzaman.pizza", "admin"));  
}
```

Note: if we want to specify the fourth parameter, we can't skip the second and the third. Therefore, you can now see that we put the pizza manager menu-item in the main menu.

The fourth parameter can be three things:

- True: menu item is always displayed
- False: menu-item is never displayed.
- An array with node/action pairs. The first element of the array must be a node (identified by *modulename.nodename* notation), the second element is an action. The third can be a node again, the fourth an action, and so on. If the user has the right to perform one of these actions, he will see the menu-item. In this case, the menu is displayed if the user has the 'admin' right on the pizza node.

Note: In our case, a user could of course have the right to manage ingredients, but not pizzas, in which case the menu-item would not be displayed. But for the sake of brevity, we assume that users will at least have the right to manage pizzas, if they need to access any of the other menu-items.

### 3.3 Language files

One thing that must have annoyed you throughout guide 1 and 2 is the fact that the menu-items are called 'menu pizzaman'. Let's remedy that now.

In the pizza module directory, create a subdirectory called 'languages'. In that directory, you can put language files for your module. There's a language file for each language your module supports. The filename should be equal to the filename used for that language in the Achievo languages directory. In this guide, I'll implement an *english.lng* file, assuming that we're using the English language in Achievo's *config.inc.php* file.

A language-file contains a set of variables, to which a text is assigned. Any text you see in the user-interface (menu-items, links, fieldnames) can be translated. Usually, the name of the string in the language file is equal to the string you see on screen, but with *\$txt\_* prepended to it, and with underscores instead of spaces. To translate the 'menu\_pizzaman' string, we would put the following in the *english.lng* file:

```
<?php  
  
    $txt_menu_pizzaman = "Pizza Management";  
  
?>
```

When we save this file, and reload the Achievo menu, the menu item should now have a correct label.

Field labels can also be put in the language file. Let's change the label of the 'entrydate' field in the pizza node to 'Date of entry'. This can be done in several ways.

The simplest way is to add `$txt_entrydate` to the language file. However, it is feasible that other nodes have an entrydate field as well, but need another description. For this reason, the language file also accepts a string called `$txt_pizza_entrydate`. (In the future, we might also add the module name to the accepted languages strings, in case multiple modules have a node with the same name and the same field.)

Another field we should translate is the field `ingredient_id` in the recipe node, since it shows up in the edit screen of a pizza. (If you want to support multiple languages, you should of course put every fieldname in the language file, but here I just give a few examples.)

It is also wise to put the module-name, and the individual node names of our module in the language file, as these show up in Achievo in some pages (for example the profile editor, a future about-page etc).

When we add the translations for these fields, the module name, the node names and the menu-items of the pizza-manager submenu, we get the following language file:

```
<?php

// Menu items
$txt_menu_pizzaman = "Pizza Management";
$txt_menu_pizza = "Pizzas";
$txt_menu_pizza_categories = "Categories";
$txt_menu_pizza_ingredients = "Ingredients";

// Some field translations
$txt_pizza_entrydate = "Date of entry";
$txt_pizza_recipe_ingredient_id = "Ingredient";

// The module and node names
$txt_pizzaman = "Pizza Manager";
$txt_pizza = "Pizza";
$txt_pizza_ingredient = "Ingredient";
$txt_pizza_recipe = "Recipe";
$txt_pizza_category = "Category";

?>
```

### 3.4 Tweaking the user-interface

One final point of annoyance: as we've seen in paragraph 1.4, we can't add the recipe and notes, before we added the pizza. We have to add a pizza first, then edit it.

There's a method to make this more user-friendly. This involves making the add-screen as small as possible, and then automatically display the edit screen after the save button was pressed. You might have seen this behavior in several places in Achievo. For example when adding a project, you just select a name and a template, after which the record is added and the edit screen is presented. The same is implemented for employees, and several other nodes.

Fields that are obligatory must be present in the add screen, because we can't save a pizza without giving a value for these fields. The other fields can be hidden in the add-screen. This is done by specifying the `AF_HIDE_ADD` flag. The fields that we can use this flag with in this case are the price and entrydate fields. For the sake of userfriendliness, we also decide to change the description field to `AF_HIDE_ADD`, instead of `AF_OBLIGATORY`, just to loose the big textbox on the add screen (note that in normal situations, you have to think carefully about which fields are obligatory and which are not).

Here are the modified lines from `class.pizza.inc`:

```
$this->add(new atkTextAttribute("description", 15,
                                AF_HIDE_ADD|AF_SEARCHABLE));
$this->add(new atkNumberAttribute("price", AF_HIDE_ADD));
$this->add(new atkDateAttribute("entrydate", "", "", 0, 0, AF_HIDE_ADD));
```

Note that for the `atkDateAttribute`, the flags are the sixth parameter. We need to specify the default values for the other parameters to be able to specify a sixth parameter (you can lookup the meaning of these other parameters in the API documentation).

Now, we only need to specify that the edit-screen should appear right after the add-screen. This is done with a node-flag. As we've seen in paragraph 2.3, node flags are passed as second parameter to the `$this->atkNode()` call.

The flag we use here is `NF_EDITAFTERADD`. When we add it, this is the resulting code in `class.pizza.inc`:

```
function pizza()
{
    $this->atkNode("pizza", NF_EDITAFTERADD);
}
```

If you test the pizza manager now, you should see the desired behavior when adding a pizza.

## 4 Conclusion

We've seen a lot in this part of the guide.

By now, you should be able to build quite useful modules for Achievo. Again I would like to point out how relatively few code you actually need to write to accomplish something. It was one of our goals when we developed the backend for Achievo, and we still strive to keep it as easy as possible to develop extensions, in the hope that a lot of them will be contributed to the Achievo project.

If this part of the guide went smooth for you, you might want to read part 3 as well. It describes the most advanced features of the backend. One important part of part 3 is modifying the behavior and functionality of existing Achievo nodes, without touching the base code. This is extremely useful when adapting Achievo to the needs of your organization, while still remaining upgradeable.

## Index

AF_AUTOKEY.....	5	NF_EDITAFTERADD.....	22
AF_CASCADE_DELETE .....	9	NF_TRACK_CHANGES .....	17
AF_HIDE_ADD.....	22	node flag.....	17
AF_HIDE_LIST.....	9	normalize.....	10
atkManyToOneRelation .....	6	One-to-many relation .....	7
atkOneToManyRelation .....	9	postAdd .....	15
atkorgrec.....	17	postDel .....	15
Conclusion.....	23	Post-triggers.....	15
Contents .....	2	postUpdate .....	15
default values .....	19	preAdd .....	15
descriptor_def.....	6	preCopy.....	15
getMenuItems.....	19	Prerequisites .....	3
initial_values .....	19	Pre-triggers .....	15
intermediary table.....	10	preUpdate .....	15
Introduction.....	3	Relationships.....	4
Language files .....	20	setSecurityAlias.....	8
mail.....	16	submenu.....	19
Many-to-many relation .....	9	Tracking changes.....	17
Many-to-one relation .....	4	Trigger .....	15
master-detail.....	7	userelation.....	6
menuItem.....	19	usermail.....	16

## Appendix A – Completed source

In part 1, I put the entire code of the pizza module in the appendix. As there are quite a few files now, I've put the completed source code up for download on the Achievo site: [http://www.achievo.org/achievo\\_guide/pizzaguide\\_part2\\_module.tar.gz](http://www.achievo.org/achievo_guide/pizzaguide_part2_module.tar.gz)