ATK - A Business Framework

The text from this article was originally published in the September 2009 issue of php|architect under the same name, and is reprinted here with Ivo's permission.

Introduction

A few months ago php|architect featured a series of articles called "A Refactoring Diary". They were written by Bart McLeod, and discussed his efforts to implement a CMS using various frameworks. One of the frameworks Bart looked at was the ATK framework, and he compared it to symfony and the Zend Framework. I was a bit surprised, since symfony and the Zend Framework are usually positioned in a wholly different league, but I was happy with the exposure it generated. This month, php|architect allowed me to dive a little deeper into the ATK framework.

By the end of this article, you'll know what a business framework is, what ATK has to offer you and in which situations you'll want to make use of a framework like ATK - and equally important, when you won't.

A Business Framework?

There are different types of framework. The most common ones are component frameworks, which are more or less libraries of components that you can use to build an application. Zend Framework, PEAR and eZComponents are examples of this. Another type of framework is the "full stack" framework, which is a framework that builds the complete application for you. CakePHP and symfony are examples of these. Full stack frameworks also prescribe how to code an application; they are more strict than component frameworks.

ATK is a full stack framework, as it builds a complete application for you. However, it is a framework with a specific target audience. Essentially, it tries to solve a problem in a particular niche. Judging from the type of projects that are built with ATK, I can say that this framework's niche is that of the business application.

So what makes ATK a business framework? It's the fact that ATK has features that make it very suitable for building business applications. Business applications, such as HRM, CRM and management applications, are usually all about data manipulation, and this is one of the core features of ATK. You might call it a CRUD framework (a framework to generate create/read/update/delete screens), but it doesn't stop there. ATK also has features such as CSV import and export of data built in, OpenOffice document generation and data search capability. You can build features like that with any framework, but the point is that with ATK you don't have to build it at all. The features are built in and come as part of the framework, meaning they're available to you virtually without coding. I'll demonstrate this with some examples later in this article.

With the functionality it offers, typical applications built with the ATK framework are found in business process management, order management, project management, CMS systems occasionally, backend software and accounting. ATK is also suitable for applications that revolve around a lot of business logic.

If you take a sneak peak at the image near the end of this article, you'll have a view of a typical ATK application.

What Makes ATK Better Than $framework?

It isn't. Being better is not even the point. It's not easy to compare ATK to the Zend Framework, symfony and so on, simply because there is a different focus. Sure, the Zend Framework has much better documentation, and symfony has a much cleaner API, but on the other hand, none of them allow you to build CRUD screens as fast as ATK.

I'm a big believer in using the best tool for the job. This means that for example that at Ibuildings, for each project we first select the framework that suits the job best. For a regular website we'd choose something like CodeIgniter or the Zend Framework, and for business applications we'd choose ATK. In many projects we use a combination: one of the popular frameworks for the frontend, and ATK for the backend.

Say you have to build an application to keep track of a farming company's livestock. You could take a framework and start to build views and controllers for every screen: the screen that lists all the animals the company has, the screens to edit that data, the controllers to save data to the database, the validation logic and user management...

ATK's strength is that it recognizes that none of the above is unique to the application you are building. The base functionality of managing cows is exactly the same as that of managing customers or orders. The only real difference is in the meaning of the data that you're managing. A cow has wildly different characteristics to a customer (ok, I could think of a few similarities), but the functionality of the application is the same. So ATK only makes you write the business logic and code that make your application unique. The rest, you don't need to code; it's provided as built-in functionality of ATK.

Some frameworks have chosen to do scaffolding; they also recognize that many features are similar across applications, and they will generate code for you. ATK distinguishes itself from this type of frameworks by not generating code. If it's the same, there's no need to generate any code. And if it's different, you just add a bit of code to cope with that difference.

If this seems a little abstract, don't worry. We will build an actual application during the course of this article.

Philosophy

Before we get our hands dirty, a few words on some of the philosophies behind the ATK framework.

The most important one is "*The less code, the better*". Any line of code, whether it be generated or manually written, has a maintenance cost. Therefore, we built ATK in such a way that you get a high functionality-to-code ratio. In particular, features that are common throughout data management applications should either be there by default or require a very minimal amount of code on your part. As you'll see in the example later on, some functionality can be implemented by adding a simple flag to a class, or by implementing just a single line of code.

A second philosophy of the framework is "*Don't Repeat Yourself*" (DRY). This is one that most frameworks promote, so it's logical enough that ATK should follow this philosophy. DRY basically means that whatever you write you write only once, so code re-use is important. The framework caters for this philosophy using a range of techniques, from the most elementary re-use of classes within an application, to the re-use of complete modules between various applications. This means that if you have implemented functionality for user management, group management or whatever, you can treat that as a module and plug it into any additional application that you write.

The third philosophy is "*Don't re-invent the wheel*". This is tightly linked to DRY, but it means that not only should you not repeat yourself, you should also not repeat what others have already built. It's easy to use external libraries or frameworks in an ATK application, but even the framework itself follows this principle nowadays: there are over fifteen third party packages bundled with ATK. These range from basic libraries such as the Smarty template engine or the SimpleTest unit test framework, to client side technologies such as Prototype, script.aculo.us and Rico. All of these are used by ATK itself, and can be used by application developers. The reason they are bundled is that this makes it easier for ATK users to get started.

Architecture

The ATK framework has a certain architecture. It's not just a set of APIs; there's a particular line of thought behind the framework that makes it do things the way it does. Before you start developing an application in ATK, it's useful to know about this.

The core element of the framework is the *node*. A node in ATK is like a business object, or a data entity. In its most basic form, it represents a database table. If you had an application dealing with employees and departments, you would have an **employee** node and a **department** node, for example. But a node is more than just a table wrapper. It contains the business logic, and it tells the framework how this particular entity should be treated within the application.

This is one of the architectural principles that give ATK its powerful CRUD capabilities. Many Object-Relational Mapping systems or automated CRUD screen generators create applications that are too close to the database. It's almost as if the system gave its users access to phpMyAdmin to edit their data. In ATK, you code the node to tell the framework how it can best present the CRUD screens to the user, so the resulting application is much closer to the user than it is to the database.

Each node contains one or more *attributes*. Attributes are like fields, although they can be much more. A relationship with another node (N:1, 1:N, N:M, 1:1) is an attribute of a node too, and you can also add virtual attributes, for example to add calculated values.

The actual screens of the application are implemented by *action handlers*. The screen that displays a list of records, the screen that views a record, the screen that edits it—all of these are action handlers. Those actions that do not have direct output, but that manipulate data, e.g. deleting a record or saving an update, are action handlers too.

If you're familiar with MVC frameworks such as the Zend Framework, you could think of an action handler as a controller and the node as the model. The views are implemented using Smarty templates. The concepts are similar but the naming is different; the first version of ATK (in the year 2000; yes, the ATK framework goes back that far!) already had this architecture, way before the MVC pattern became popular.

What is nice is that, for the most common actions, the action handlers are pre-built. They are generic classes that can operate on any node. So, instead of having to develop an action handler for each and every piece of data in your application, you basically just develop the node. The action handlers are usable out of the box, and it's often not necessary to create your own.

In the example application for this article we will be building some of these nodes and their attributes, and we will be using action handlers to work with them. We're almost there, but we'll have to discuss one last, but very important architectural aspect of ATK. It is the one thing that is the most confusing to new ATK users, as it is so wildly different from almost all the other frameworks out there. In most frameworks, you build an application by writing code, from which you call APIs. So from the user perspective, normally this is what happens:

```
User -> Your code -> Framework
```

In ATK, it works the other way round:

```
User -> Framework -> Your code
```

This may sound confusing, so let me try to explain it using an example.

When you make a Zend Framework application, you start by writing your bootstrap code, your controllers and your views and in those, you call framework code to make it do the things you need it to do. In ATK however we have a more extreme version of "*Don't reinvent the wheel*" at play, and the initial setup of the application is very similar throughout applications. So this part is all done by the framework, and the framework just does its thing until it needs input from you. For example, it can create a complete CRUD application without your coding a single line, but at some point it needs to know about the data. At that point, it will construct your node and call its constructor to get information about it.

Likewise, if you want to influence the way a certain value is displayed, in most frameworks you would call some function to format the data. In ATK, the framework calls your function whenever it needs to display a string. Basically the framework states: "Here I have this string, format it for me so I can use it". It's like working with callbacks.

This approach requires the developer to think the other way round, and I've found that it's the one thing that most developers new to ATK find the hardest to get used to. Once you get your head around this though, it becomes natural.

ATK In Action

Ok, time to get our hands dirty!

Let's build an application. Let's take the hypothetical case that Marco Tabini needs a new management application to track his magazine articles. Since Marco recently started a Python magazine, we would have to deal with multiple magazines, but for now we'll just assume there's just the one magazine with monthly issues. Each issue can have multiple articles, and each article appears only in one issue. We'll also want to categorize articles, so we'll be dealing with a data model of three entities: **issue**, **article** and **category**. Figure 1 depicts a small UML diagram for this hypothetical application.

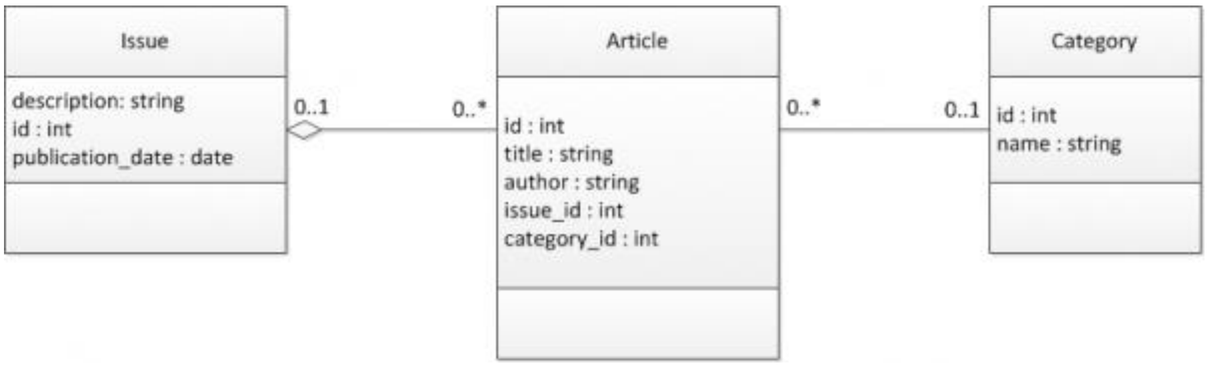


Figure 1 - UML Diagram

We need screens for managing the magazines and adding articles. We also want to be able to search our article database, and perhaps export the articles for use in other software. This might seem a little big for the scope of this article, but let's see how far we get.

ATK Installation

We need to start by installing the framework. You can download the framework from the ATK project homepage. There are two flavours of download available: the library plus a complete demo application that is very useful to get started, or the plain library. Since we'll be building this application from scratch we don't need the demo right now, so just download the plain library.

Make a directory somewhere in your document root, say *magman* (**mag**azine **man**ager), and extract (or unzip, depending on the format you downloaded) the framework to the new directory.

```
# create the directory
mkdir magman
# change to the directory
cd magman
# uncompress atk
tar -xzvf ../atk-6.5.0.tar.gz
# make sure the directory is called just 'atk'
mv atk-6.5.0 atk
```

So now we have a directory that is empty, apart from the *atk/* subdirectory. We can build the application now, but since the initial stuff is not really specific to this application, there's no need to write any code yet. We just put the bootstrap code in place.

This code is provided as a skeleton in *atk/skel/*, and with the following command we can put everything in place:

```
# Copy the skeleton files
cp -r atk/skel/* .
```

This copies everything from *atk/skel/* into the current directory. We now have a lot of files in the directory and several subdirectories. We call this stuff the "glue files", as they glue together the application. If we have to, we can change these files later on, but that's hardly ever necessary.

ATK uses the *atktmp/* directory to store temporary and cache files to improve performance. On Unix systems, use the following command to make the *atktmp/* directory writable by the web server (you may need to change **www** to the user the web server is running as):

```
chown -R www atktmp
```

Once you've done this you should be able to browse the application by navigating to http://localhost/magman (or wherever you placed the app). *"Huh? We haven't coded anything yet!"* True, but as I explained earlier, we only need to code what makes our application unique. So far we only have a browsable application, so we don't need to write any code; it just works.

If you are able to browse the application (there's one dummy menu item on the top that you can ignore), you're ready to start coding. If not, retrace the above steps and see if you may have made a mistake in one of them.

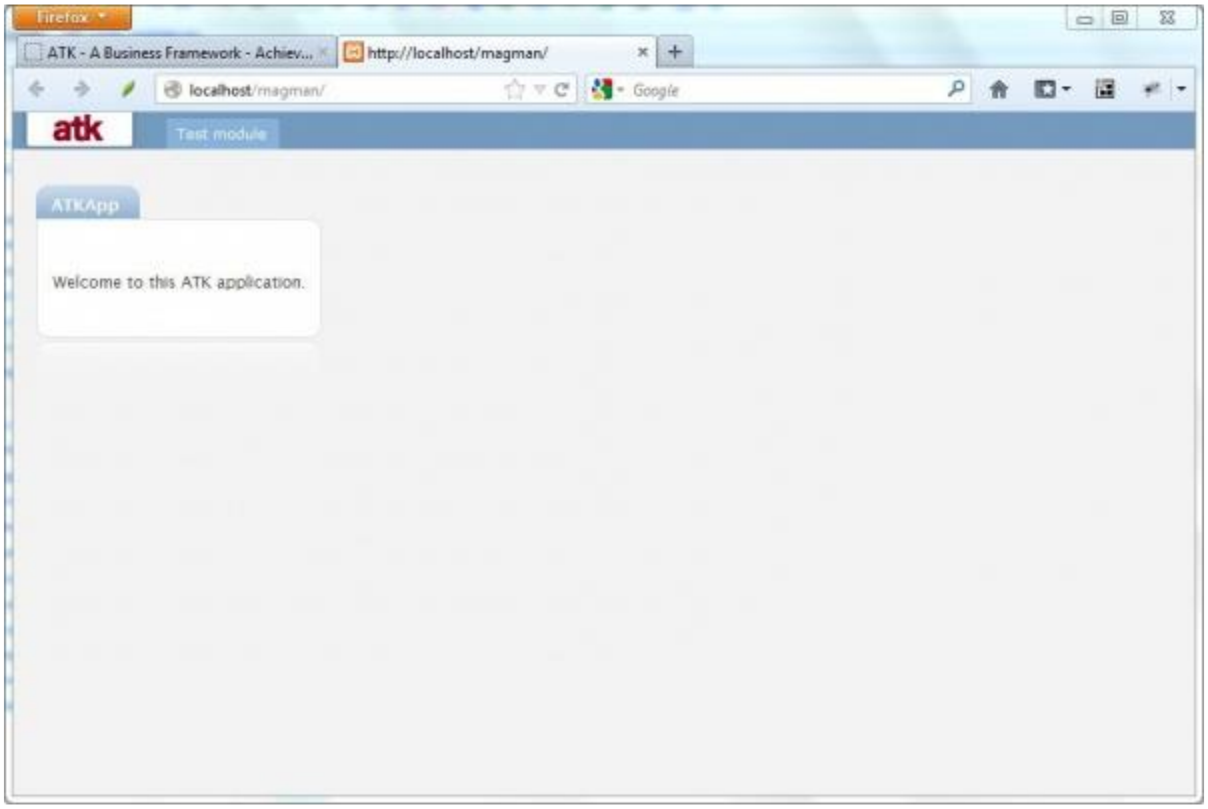


Figure 2 - The New Skeleton App

Priming the Database

Let's start coding our application. We first need a database. So create a database, and create the tables for our data model. Listing 1 contains the required SQL; I haven't included all fields that might be needed in a real application, just a few for demonstration purposes.

```
CREATE TABLE article (
  id int(11) NOT NULL,
  title varchar(255) NOT NULL,
  issue_id int(11) default NULL,
  category_id int(11) default NULL,
  author varchar(255) NOT NULL,
  PRIMARY KEY (id)
);

CREATE TABLE category (
  id int(11) NOT NULL,
  `name` varchar(255) NOT NULL,
  PRIMARY KEY (id)
);

CREATE TABLE issue (
  id int(11) NOT NULL,
  description varchar(255) NOT NULL,
  publication_date date default NULL,
  PRIMARY KEY (id)
);
```

Listing 1

You also need to insert the contents of the *init_mysql.sql* file, which you'll find in the root of your application. This will create some tables that ATK will use internally. You can delete *init_mysql.sql* afterwards if you like, it's only there to initialize your database with some tables the framework will use.

Next, we have to tell our application what the database connection settings are. We do this in the file config.inc.php, which is in the application root. Edit this file and make sure the database settings ($config_db) reflect your setup. In my case, they would be:

```
$config_db["default"]["driver"]   = "mysqli";
$config_db["default"]["host"]     = "localhost";
$config_db["default"]["db"]       = "magman";
$config_db["default"]["user"]     = "magman";
$config_db["default"]["password"] = "demo";
```

Managing Issues

An ATK application is divided into modules. To start creating the module for managing magazines, navigate to the *modules/* directory of your application, and do **mkdir magman**. This will create a **magman** module. Edit *config.modules.inc* in the modules/ directory, replacing **module("test")** with **module("magman")**. For the sake of brevity, we will put all our functionality in this module; normally, it would be a better idea to have several modules, for example a **magazine** module, a **subscriber** module, and so on.

In the *modules/magman/* directory we are now going to create the node for the **issue** entity, in a file named *class.issue.inc*. The code is pretty straightforward. We create a class that is derived from **atkNode** and add some attributes to it, according to our database model. Listing 2 contains the code.

```
<?php
// modules/magman/class.issue.inc

// although there's an autoload, it's good
// practice to 'useattrib' any attribute
useattrib("atkdateattribute");

class issue extends atkNode
{
  public function __construct()
  {
    parent::__construct("issue");

    $this->add(new atkAttribute("id", AF_AUTOKEY));
    $this->add(new atkAttribute("description", AF_OBLIGATORY));
    $this->add(new atkDateAttribute("publication_date"));

    $this->setTable("issue");
    $this->setOrder("publication_date DESC");
  }
}
```

Listing 2

As you can see, we use several different attribute types for different fields. The **id** attribute has the flag **AF_AUTOKEY**, which makes this field the primary key. It will be auto-incremented, and it will also be hidden from the user. By default, ATK doesn't use the **AUTO_INCREMENT** made available by MySQL, as this feature is not really standard across databases. It's more standard to use database sequences, and ATK emulates that behaviour when using MySQL, using the **db_sequence** table that *init_mysql.sql* added for us.

The **AF_OBLIGATORY** flag is added to attributes to make them required fields. This can be auto-detected by the framework, but by default it's done manually. This is because not everything that is required by the database should be required by the user (a calculated value, for example) or vice versa.

After the attributes, we add a line that tells ATK which table to store magazine issues in, and a further line to tell ATK to sort issues by publication date by default.

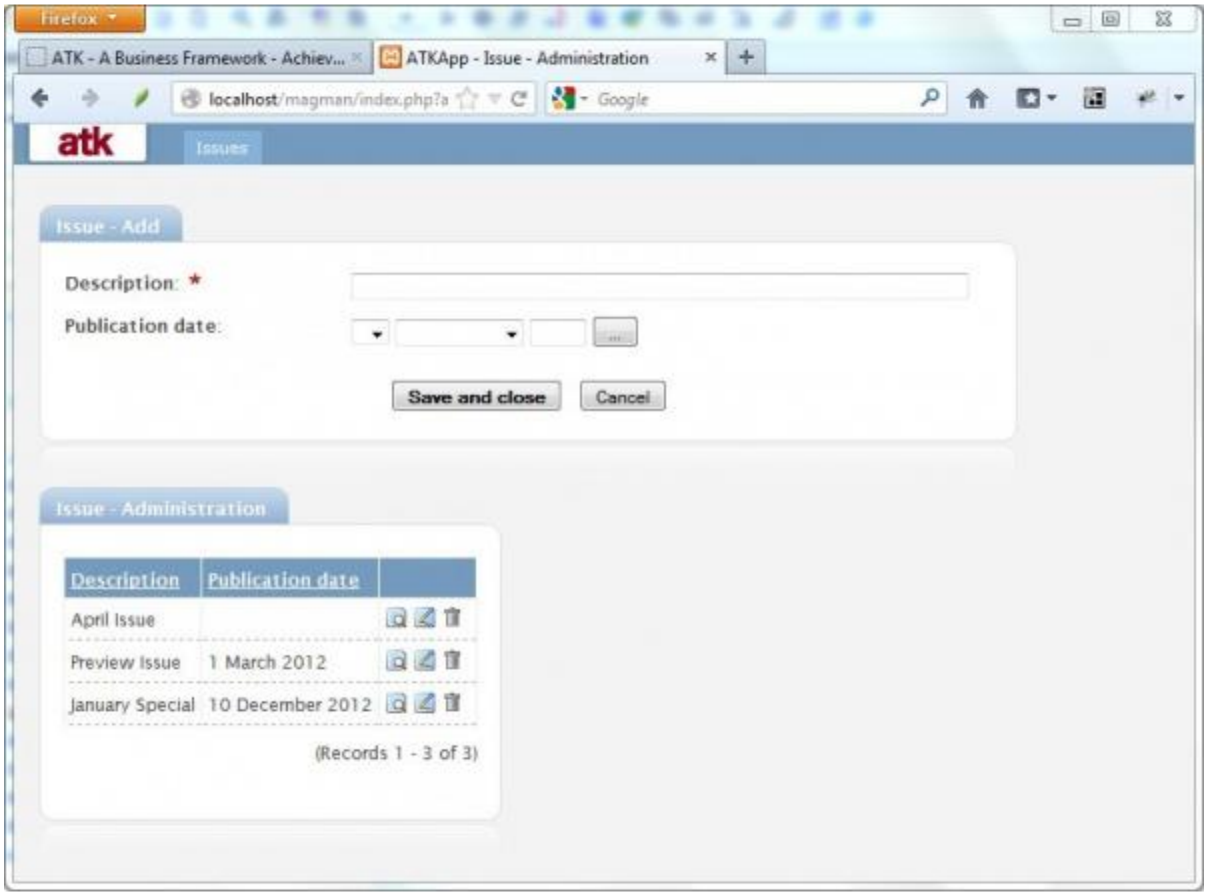
Now that we have our **issues** entity, we can hook it into the application. The basic CRUD functionality comes out of the box; no further coding is required. We only need to add it to the application menu.

The action handler that opens a screen to manage the magazine issues is named **admin**, for "administrating records". In hindsight this was a bad choice; it's often confused with "root stuff", but this is the way it's named and it's too late now to change that. Every module has a *module.inc* file that creates menu items, so we'll add a menu item there that opens up the **admin** action for our **issue** node. The code for *module.inc*, which goes into the *modules/magman/* directory, can be found in Listing 3.

```
<?php
// modules/magman/module.inc

class mod_magman extends atkModule
{
  public function getMenuItems()
  {
    $this->menuitem("issues", dispatch_url("magman.issue",
                                           "admin"));
  }
}
```

Listing 3

Browsing the application again, you should now be able to add magazine issues, delete them and edit them. That took roughly ten lines of code, hence the ATK slogan: *An application in 10 lines of code.*


Firefox
ATK - A Business Framework - Achiev...
ATKApp - Issue - Administration
localhost/magman/index.php?a
Google
atk
Issues
Issue - Add
Issue - Administration

Description: *

Publication date:

Save and close | Cancel

Description	Publication date	
April Issue		
Preview Issue	1 March 2012	
January Special	10 December 2012	

(Records 1 - 3 of 3)

Figure 3 Browsing Issues

Managing Articles

Now that we've created the **issue** node, we need to create an **article** node. The basic code for this is relatively straightforward. It is covered in Listing 4, so save the code there as *class.article.inc* under *modules/magman/*.

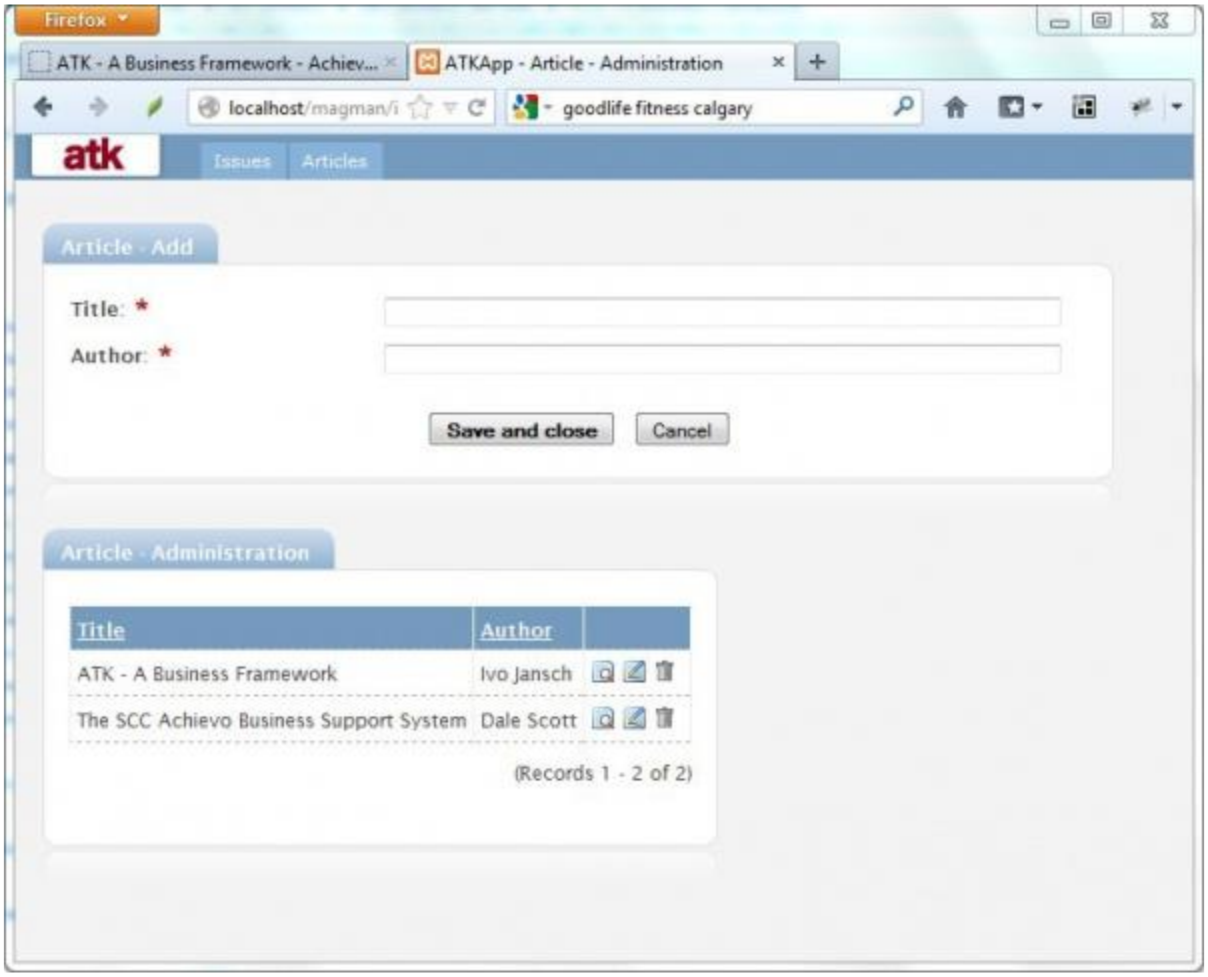
```php
<?php
// modules/magman/class.article.inc

class article extends atkNode
{
  public function __construct()
  {
    parent::__construct("article");

    $this->add(new atkAttribute("id", AF_AUTOKEY));
    $this->add(new atkAttribute("title", AF_OBLIGATORY));
    $this->add(new atkAttribute("author", AF_OBLIGATORY));

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

Listing 4

You can add articles to the menu by adding the following line to *module.inc*:

```php
$this->menuitem("articles",
                dispatch_url("magman.article",
                             "admin"));
```

If you now browse the application, you should be able to manage both articles and magazines. We only have to connect the two together, so that we can see which articles were in which issue.



Title	Author	
ATK - A Business Framework	Ivo Jansch	
The SCC Achievo Business Support System	Dale Scott	

(Records 1 - 2 of 2)

Figure 4 Browsing Articles

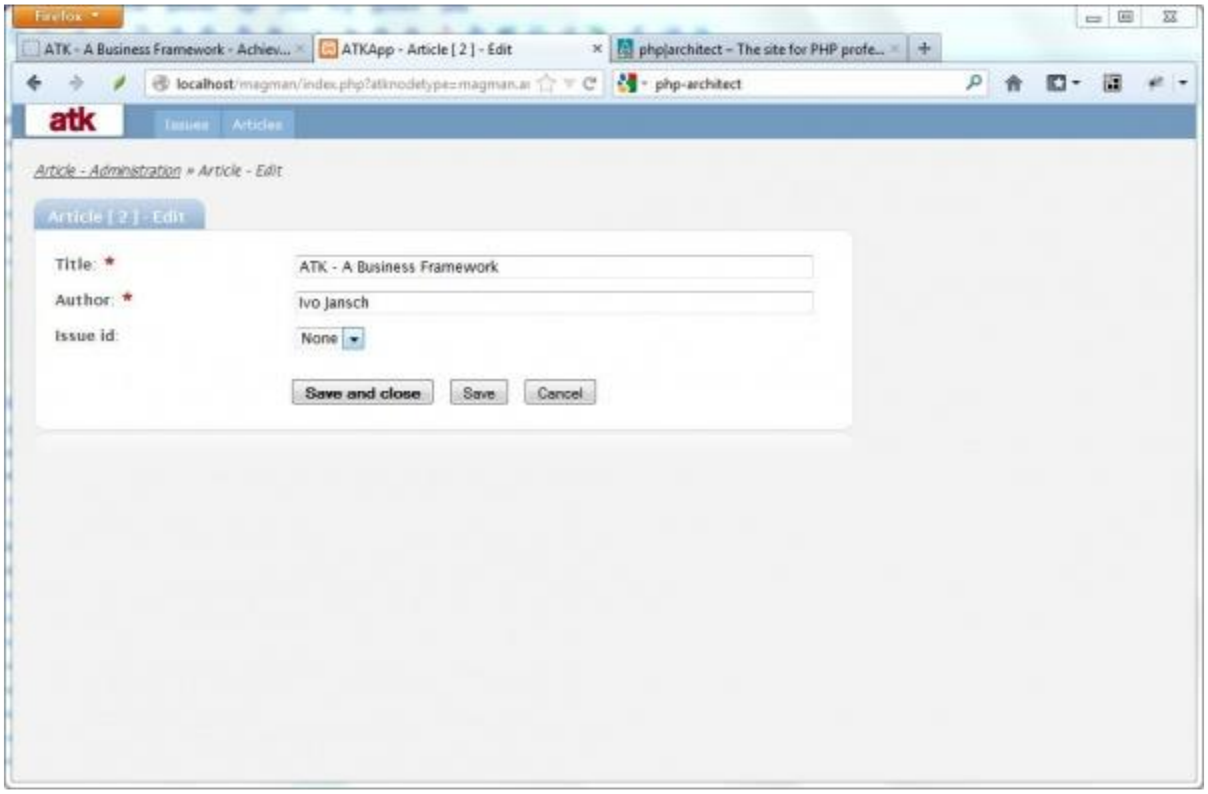
Adding the Issue to Article Relationship

In the data model, there is a one-to-many relationship between issue and article. Similarly, we could say that there is a many-to-one relation between article and issue. We will add these relationships to the constructors of our issue and article nodes:

```
// in class.article.inc
$this->add(new atkManyToOneRelation("issue_id",
                                    "magman.issue"));
// in class.issue.inc
$this->add(new atkOneToManyRelation("articles",
                                    "magman.article",
                                    "issue_id",
                                    AF_HIDE_LIST));
```

For the many-to-one relation, all we need to pass is the name of the module suffixed with the name of the node we're relating to. For the one-to-many relation, we also need to tell it the name of the referential key in the destination node (that's **issue_id** in the **article** node). We also pass the **AF_HIDE_LIST** flag, which hides the articles from the issue list view. It is sometimes convenient to expose everything, but it clutters the display a lot.

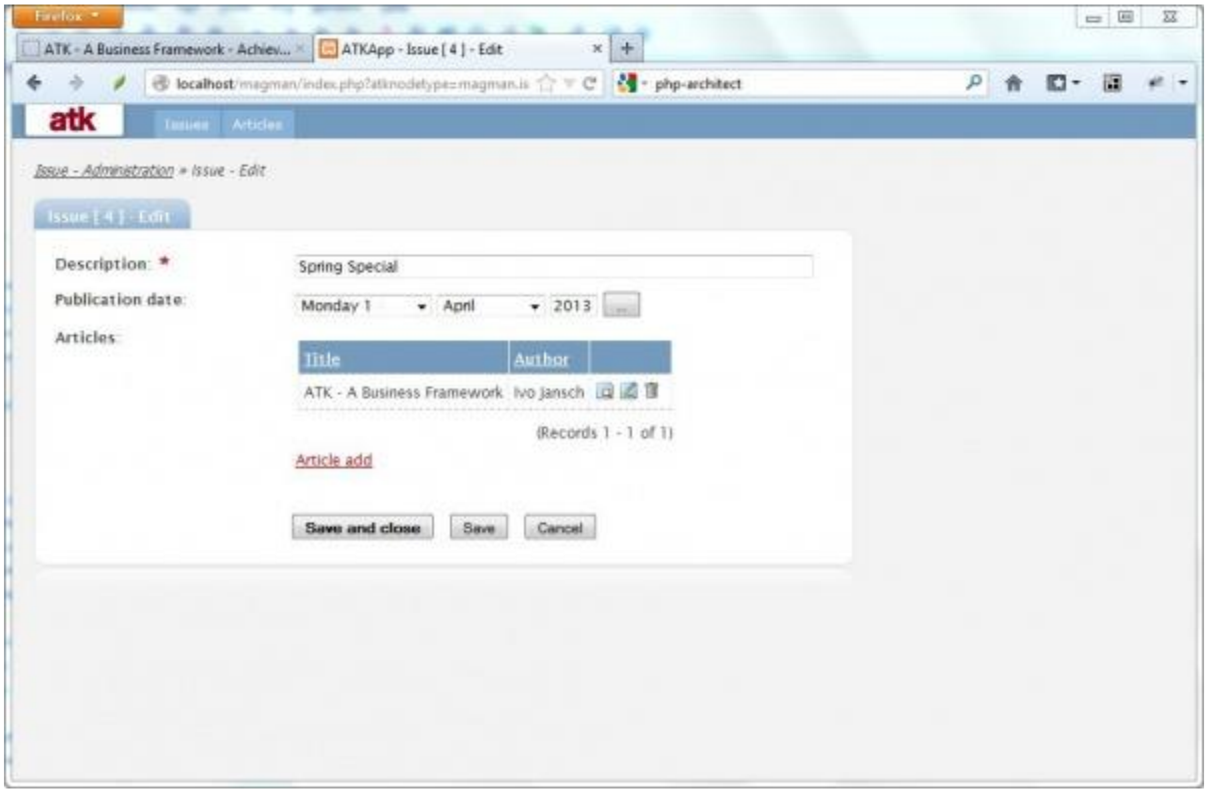
If you browse the application again, you'll find that you can now select an issue when adding an article. Also, when you edit an issue, you'll be able to add articles to it.


Firefox
ATK - A Business Framework - Achiev...
ATKApp - Article [2] - Edit
php|architect - The site for PHP profe...
localhost/magman/index.php?atknodetype=magman.ar
php-architect
atk
Issues
Articles
Article - Administration » Article - Edit
Article [2] - Edit

Title: * ATK - A Business Framework
Author: * Ivo Jansch
Issue id: None

Save and close
Save
Cancel

Figure 5 Article to Issue Relationship


Firefox
ATK - A Business Framework - Achiev...
ATKApp - Issue [4] - Edit
localhost/magman/index.php?atknodetype=magman.is
php-architect
atk
Issues
Articles
Issue - Administration » Issue - Edit
Issue [4] - Edit
Article add
Save and close
Save
Cancel

Description: * Spring Special

Publication date: Monday 1 April 2013

Articles:

Title	Author
ATK - A Business Framework	Ivo Jansch

(Records 1 - 1 of 1)

Figure 6 Issue to Article Relationship

The dropdown to select an issue for an article is displaying numeric ids, since we haven't told the framework which field we want to display in the dropdown. The way we solve this is by adding this call to the constructor of the **issue** class:

```
$this->setDescriptorTemplate(
  "[description] ([publication_date.year]-
    [publication_date.month])");
```

This code basically tells the framework: whenever a record of this node (an issue) is displayed, use this string to show the user which record he is dealing with. ATK will automatically fill in the correct values in the string every time it displays a record in a dropdown. After adding this line, you should see something like "Issue 1 (2008-10)" in the dropdown to select an issue for an article.

It is good practice to always specify a descriptor template; you never know if somebody might use your nodes in a relationship somewhere. Also, this descriptor is used in the title bar of screens so the user sees what record he is editing.

Currently, when editing an article the window title bar might be this:

```
ATKApp - Article [ 3 ] - Edit
```

but if we specify a descriptor template in the constructor of the **article** class like this:

```
$this->setDescriptorTemplate("[title] by [author]");
```

the title bar changes to:

```
ATKApp - Article [ The SCC Achievo Business Support System by Dale Scott ] - Edit
```

Relationships really demonstrate the power of the ATK framework. By adding just one or two lines of code, you can relate data, and the framework will completely build the user interface that fits the editing of such relationships. No need to code any of it.

Article Categories

One last entity we haven't implemented is the article *category*. We'll use this category to categorize articles into **review**, **column**, **feature** or **editorial**. Categories are stored in the database, so we can modify them later should the need arise.

I'm going to use the article categories to demonstrate another feature of the framework, the **atkMetaNode**. This is an enhancement of the **atkNode** that we've been using so far. It goes a step further with the *Don't Repeat Yourself* principle. *atkMetaNode* basically says: *You've already put everything that defines an article category in the database, so why code it at all?* The atkMetaNode just takes the database metadata and, under the hood, converts it into code.

To use this concept to create the node for the **article** category, all we need is a file named *class.category.inc*. You code it like this:

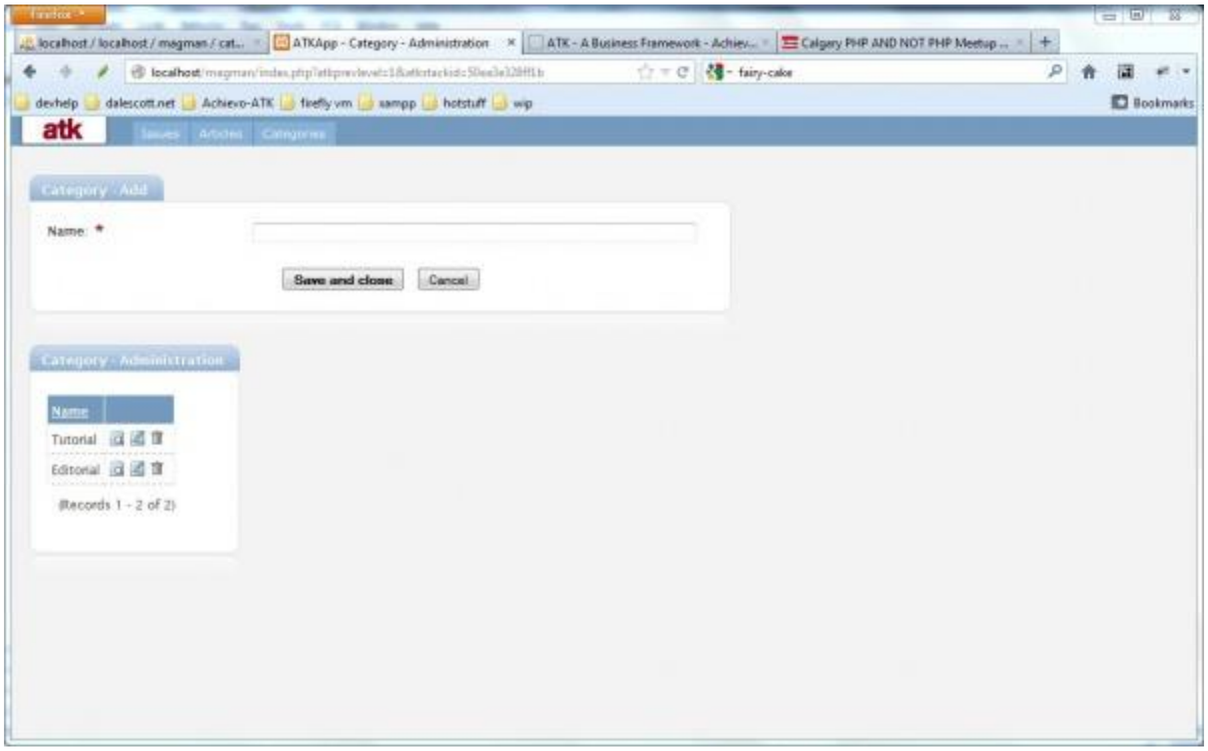
```
<?php
// modules/magman/class.category.inc

class category extends atkMetaNode
{
  protected $descriptor = "[name]";
}
```

Note the slightly different way of setting the descriptor template. The **atkMetaNode** makes much use of properties to define its' behavior.

As with the previous nodes, we add categories to the menu via *module.inc*:

```
$this->menuitem("categories",
   dispatch_url("magman.category", "admin"));
```

You can now add, edit and delete categories in our application.



Name
Tutorial
Editorial

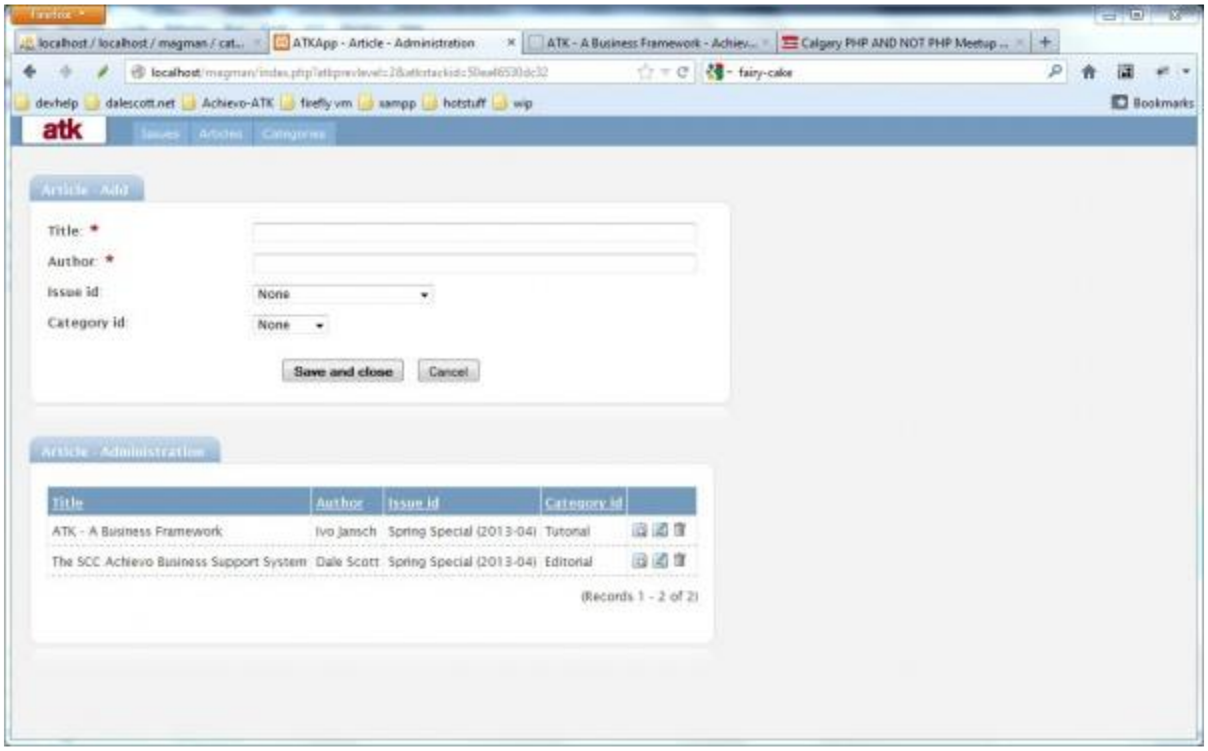
(Records 1 - 2 of 2)

Figure 7 Categories

When you're just starting to use the ATK framework, you'll mostly write "normal" nodes, as this will help you understand how the framework works. In most applications however, there are a significant number of tables that simply contain lookup data, and have no special business logic requirements. You have already seen how to use the **atkMetaNode** for these types of tables. Gradually, you'll find yourself using the **atkMetaNode** more and more, until it becomes your default. The **atkMetaNode** allows you to customize anything it detects, so eventually you'll learn to work like this: start with a default no-code setup using meta nodes, and just change the parts that differ from the assumed defaults.

We're not completely done with the categories yet. Articles have a many-to-one relationship to categories; many articles can have the same category, but a single article only has one category. To reflect this, we just need to add a line to the **article** node in *class.article.inc*:

```
$this->add(new atkManyToOneRelation("category_id",
                                     "magman.category"));
```



Title: *
Author: *
Issue id: None
Category id: None

Title	Author	Issue id	Category id
ATK - A Business Framework	Ivo Jansch	Spring Special (2013-04)	Tutorial
The SCC Achievo Business Support System	Dale Scott	Spring Special (2013-04)	Editorial

(Records 1 - 2 of 2)

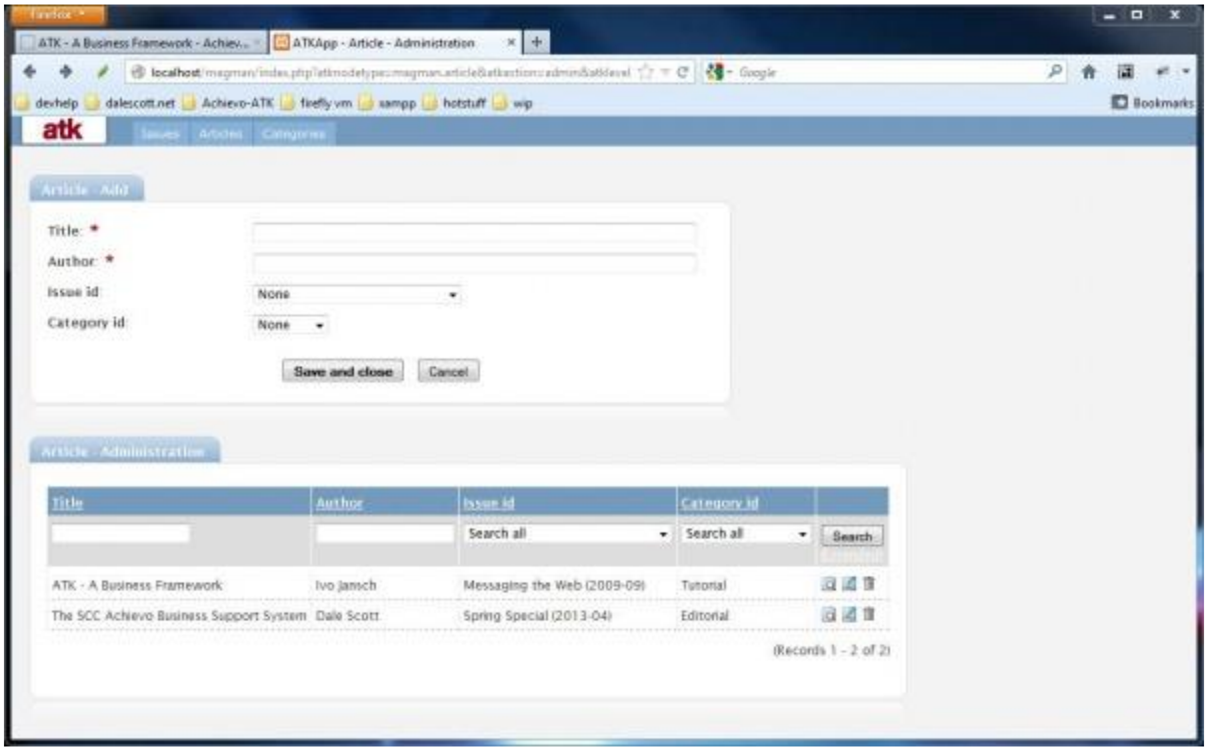
Figure 8 Article Category

Enhanced Functionality

So far we've only got a very basic CRUD application. Although it took only a few lines of code to reach this point, we'd want to make it a little more fancy than this.

Searching

The first thing to implement is search functionality. Well, *implement* is perhaps the wrong word: most of it again comes out of the box. The way we add search behavior is by adding the **AF_SEARCHABLE** flag to fields that should be searchable. We could do that for the issue **description** field, the article **title** and **author** fields, as well as the **category** and **issue** relationships. When you browse the application after adding the flag, you will see that you can now search records in the record list - note how it implements search using dropdowns for the relations. You can also click the **extended** link to have advanced search capabilities, such as searching within date ranges and/or combinations of values. This is all functionality that comes for free with ATK, without having to implement anything. After all, there's no logical difference between search in this or another data management application, so why should you need to write any code? See Figures 9 and 10.


Firefox
ATK - A Business Framework - Achiev...
ATKApp - Article - Administration
localhost/magman/index.php?atknodetype=magman.article&atkaction=admin&atklevel
Google
devhelp
dalescott.net
Achievo-ATK
firefly vm
sampp
hotstuff
wip
Bookmarks
atk
Issues
Articles
Categories

Article - Add

Title: *	
Author: *	
Issue id:	None
Category id:	None

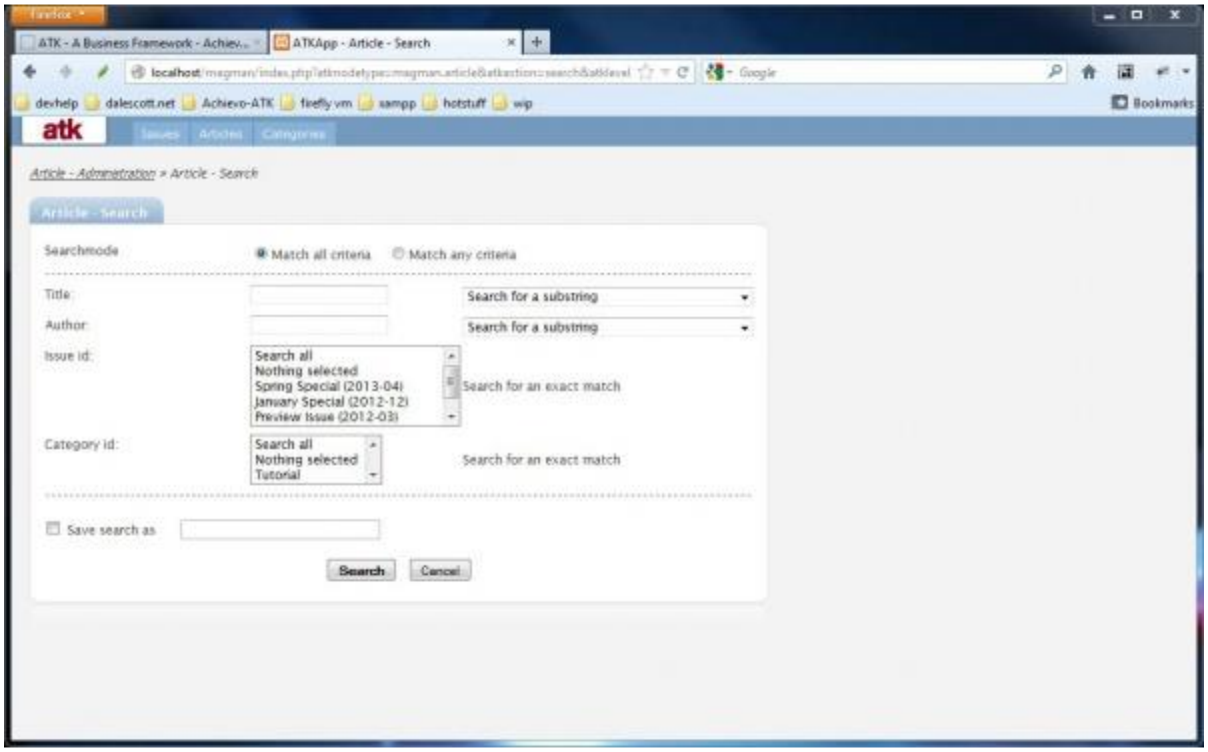
Save and close | Cancel

Article - Administration

Title	Author	Issue id	Category id	
		Search all	Search all	Search
ATK - A Business Framework	Ivo Jansch	Messaging the Web (2009-09)	Tutorial	
The SCC Achievo Business Support System	Dale Scott	Spring Special (2013-04)	Editorial	

(Records 1 - 2 of 2)

Figure 9 Article Search


Firefox
ATK - A Business Framework - Achiev...
ATKApp - Article - Search
localhost/magman/index.php?atknodetype=magman.article&atkaction=search&atklevel
Google
devhelp
dalescott.net
Achievo-ATK
firefly vm
sampp
hotstuff
wip
Bookmarks
atk
Issues
Articles
Categories

Article - Administration > Article - Search

Article - Search

Searchmode	[✓ Match all criteria] Match any criteria	
Title:		Search for a substring
Author:		Search for a substring
Issue id:	Search all Nothing selected Spring Special (2013-04) January Special (2012-12) Preview Issue (2012-03)	Search for an exact match
Category id:	Search all Nothing selected Tutorial	Search for an exact match

Save search as

Search | Cancel

Figure 10 Article Advanced Search

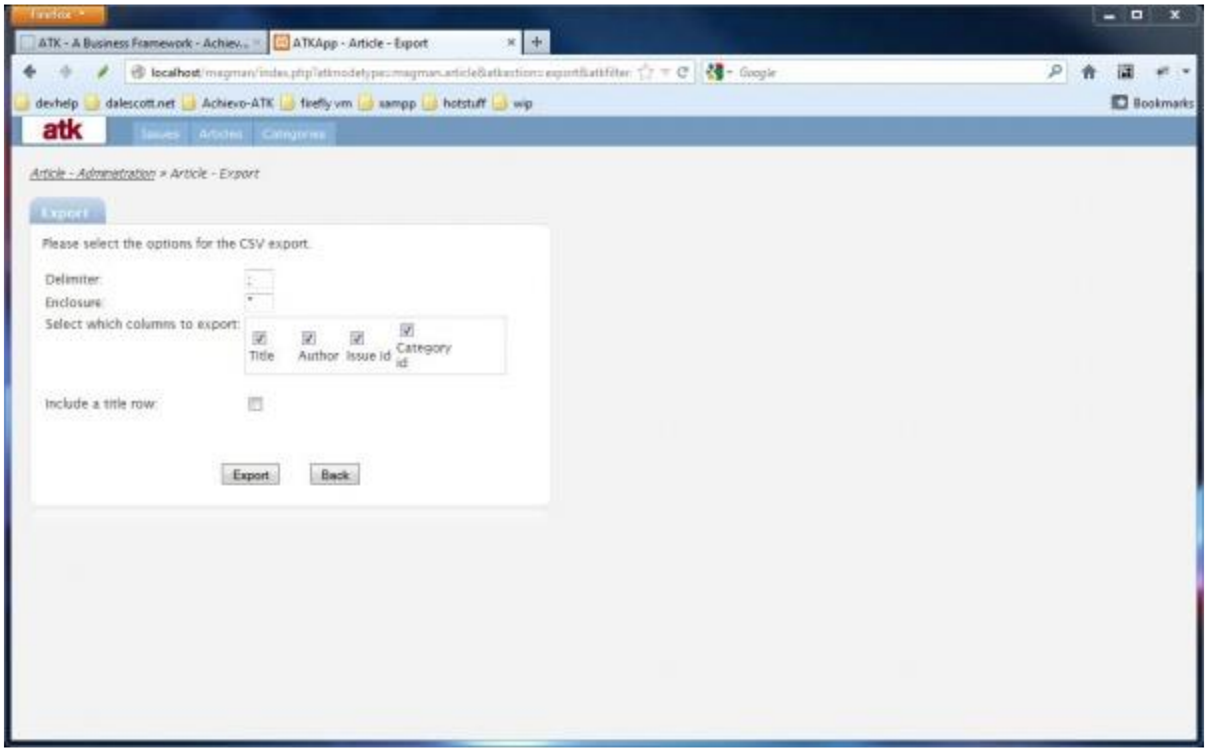
Note also that when you edit an issue, even the list of articles in that issue is now searchable.

Exporting Data

Another thing we can easily do is add export functionality. This allows the user to export an article as a comma separated value (CSV) file. Since, again, this is functionality that many applications are likely to need, export functionality is a core feature in the ATK framework. All it takes to implement export support for our **articles** node is to add the flag **NF_EXPORT** to the node. This is done by changing the call to the base constructor slightly:

```
parent::__construct("article", NF_EXPORT);
```

Browse the articles, and you will notice the *Export* link above the list of records. See Figure 11 for the export screen if you click the link.


ATK - A Business Framework - Achiev...
ATKApp - Article - Export
localhost/magman/index.php?atknodetype=magman.article&atkaction=export&atkfilter
devhelp
dalescott.net
Achievo-ATK
firefly vm
sampp
hotstuff
wip
Bookmarks
atk
Issues
Articles
Categories
Article - Administration » Article - Export
Export

Please select the options for the CSV export.

Delimiter: ;
Enclosure: "
Select which columns to export: [✓ Title] [✓ Author] [✓ Issue id] [✓ Category id]

Include a title row: ☐

Figure 11 Exporting Articles

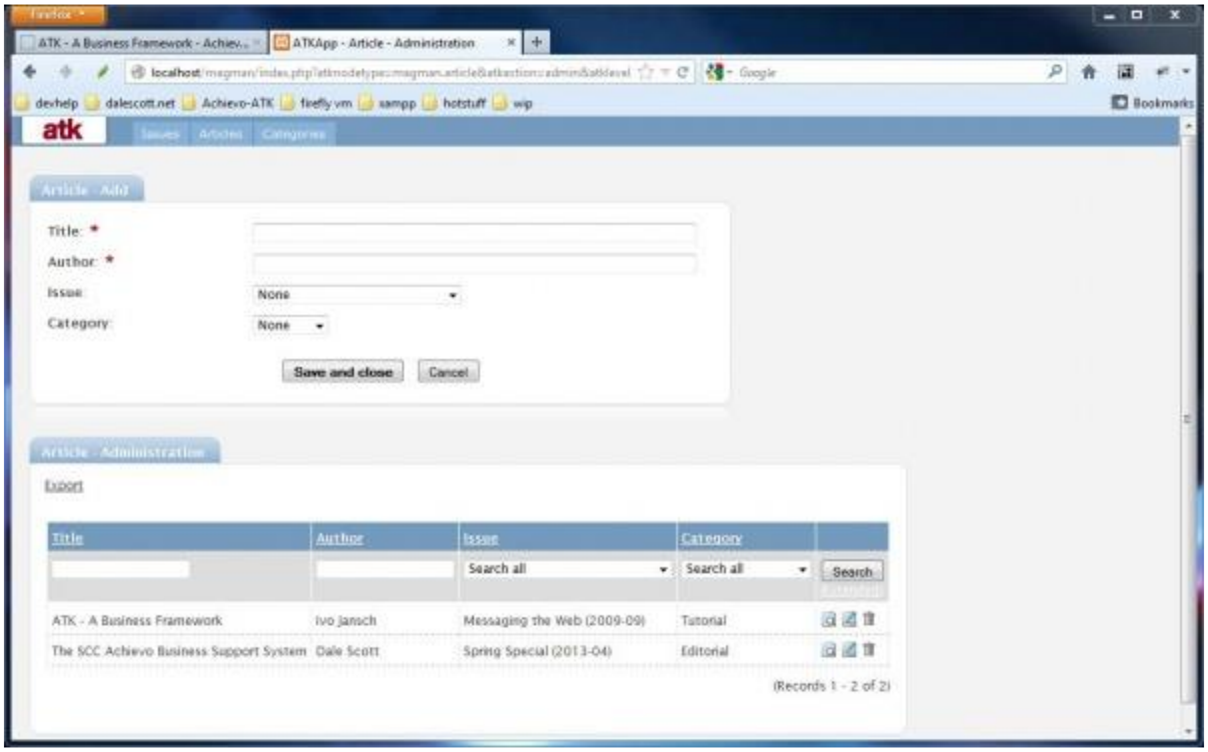
Languages

Our final enhancement consists of fixing some of the labels. As you'll have noticed, there are columns named **issue_id** and **category_id**; it would be good to tell the application how to properly name those columns for our end users. This is achieved using a simple language file, as demonstrated in Listing 5.

```
<?php
// modules/magman/languages/en.lng

$en = array("category_id" => "Category",
            "issue_id" => "Issue",
);
```

Listing 5

See Figure 12, and compare it to Figure 9.



Title: *	
Author: *	
Issue	None
Category	None

Title	Author	Issue	Category
		Search all	Search all
ATK - A Business Framework	Ivo Jansch	Messaging the Web (2009-09)	Tutorial
The SCC Achievo Business Support System	Dale Scott	Spring Special (2013-04)	Editorial

(Records 1 - 2 of 2)

Figure 12 Using a Languages File

Wrapping Up

For completeness, the entire code base is given in Listing 6. The end result should look similar to Figure 12.

```
<?php
// modules/magman/module.inc

class mod_magman extends atkModule
{
  public function getMenuItems()
  {
    $this->menuitem("issues", dispatch_url("magman.issue", "admin"));
    $this->menuitem("articles", dispatch_url("magman.article",
                                             "admin"));
    $this->menuitem("categories", dispatch_url("magman.category",
                                               "admin"));
  }
}

// modules/magman/class.issue.inc

// although there's an autoload, it's good
// practice to 'useattrib' any attribute.
useattrib("atkdateattribute");

class issue extends atkNode
{
  public function __construct()
  {
    parent::__construct("issue");

    $this->add(new atkAttribute("id", AF_AUTOKEY));
    $this->add(new atkAttribute("description",
                                AF_OBLIGATORY|AF_SEARCHABLE));
    $this->add(new atkDateAttribute("publication_date"));

    $this->add(new atkOneToManyRelation("articles",
                                        "magman.article", "issue_id",
                                        AF_HIDE_LIST));

    $this->setTable("issue");
    $this->setOrder("publication_date DESC");
    $this->setDescriptorTemplate(
      "[description] ([publication_date.year]-[publication_date.month])");
  }
}

// modules/magman/class.article.inc

class article extends atkNode
{
  public function __construct()
  {
    parent::__construct("article", NF_EXPORT);

    $this->add(new atkAttribute("id", AF_AUTOKEY));
    $this->add(new atkAttribute("title",
                                AF_OBLIGATORY|AF_SEARCHABLE));
    $this->add(new atkAttribute("author",
                                AF_OBLIGATORY|AF_SEARCHABLE));
    $this->add(new atkManyToOneRelation("issue_id",
                                        "magman.issue", AF_SEARCHABLE));
    $this->add(new atkManyToOneRelation("category_id",
                                        "magman.category",
                                        AF_SEARCHABLE));

    $this->setDescriptorTemplate("[title] by [author]");
    $this->setTable("article");
  }
}
```

```
// modules/magman/class.article.inc

class category extends atkMetaNode
{
  protected $descriptor = "[name]";
}

?>
```

Listing 6

Look at all the functionality we created—basic CRUD functionality, CSV export, data search, relationships - and how many lines of code did we write? Hardly more than fifty. Fifty! That's only fifty lines of code to maintain, which makes this a very efficient application in terms of code maintenance. Of course, once you start to add custom functionality the amount of code will go up, but the code you maintain will only ever be the code that makes your application unique; most of the basic functionality does not need you to write any code at all.

Weaknesses

So far we've only looked at the strengths of the framework, but when you start using ATK you will also need to be aware of its weaknesses. One is that ATK has a bit of a learning curve. Although you'll only need to write a few lines of code, you still have to know *which* lines to write to get the functionality you need, so you'll have a lot of *"How do I...?"* type questions. Luckily, there is a *Howto* section on the ATK Wiki, and a very active ATK Forum where you can ask those questions.

One thing that's currently missing from the documentation is a proper reference manual. There is an API reference, and ATK is heavily documented using PHPDoc, but unfortunately there is no user manual at this point in time. The ATK documentation team is currently working to plug this gap. In the meantime, the wiki, the API docs and especially the forum, are very helpful.

Further Reading

So far we haven't considered security, so our application would only be usable in an intranet environment. For many applications that's OK, but often we'll want to add authentication and authorization. It's only logical that the framework offers support for these. There isn't enough room to cover them in this article though, so I'll refer you to some further reading.

One of the most useful documents for getting started is the demo application I mentioned earlier. It contains ten lessons that explain, not only the basics already covered in this article, but also more advanced things such as security, how to work with OpenOffice documents, and how to cope with more complex data models such as many-to-many relationships. The demo application can be downloaded from the ATK project download page, and every lesson has inline source code comments that explain the code. Install it, toy with it, and read those comments. This will improve your understanding of the framework.

Next, you should dive into the *Howto* section on the project wiki and create a forum account. This will help you learn the more advanced functionality of the ATK framework, and will also teach you how to customize the functionality that the framework generates for you.