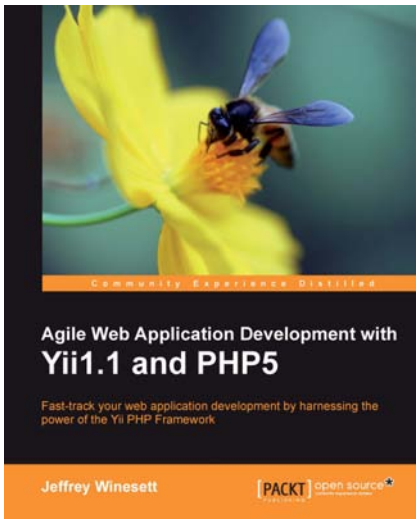
[PACKT] open source
community experience distilled
PUBLISHING

Agile Web Application Development with Yii 1.1 and PHP5

Jeffery Winesett



Chapter No.9
"Iteration 6: Adding User Comments"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.9 "Iteration 6: Adding User Comments"

A synopsis of the book's content

Information on where to buy this book

About the Author

Jeffery Winesett is the director of software engineering and application development at Control Group Inc., a New York based consulting firm specializing in delivering technology for big ideas. He has spent the last five of his twelve years of software development focused on delivering large-scale PHP-based applications. Jeffery also writes articles on the topics of PHP, web application frameworks, and software development. He has enjoyed being a Yii evangelist since its early alpha version.

I'd like to thank all of the technical reviewers, editors, and staff at Packt for their fantastic contributions, suggestions, and improvements. I'd like to thank Qiang Xue and the entire Yii Framework developer team for creating and maintaining this brilliant framework. Ryan Trammel at Scissortail design for his attention to detail and CSS assistance. My lovely wife Tiffany, for her endless patience throughout this project and Lemmy and Lucie for providing me with an endless supply of sunshine.

Agile Web Application Development with Yii 1.1 and PHP5

Yii is a high-performance, component-based application development framework written in PHP. It helps ease the complexity of building large-scale applications. It enables maximum reusability in web programming, and can significantly accelerate the development process. It does so by allowing the developer to build on top of already well-written, well-tested, and production-ready code. It prevents you from having to rewrite core functionality that is common across many of today's web-based applications, allowing you to concentrate on the business rules and logic specific to the unique application being built.

This book takes a very pragmatic approach to learning the Yii Framework. Throughout the chapters we introduce the reader to many of the core features of Yii by taking a test-first approach to building a real-world task tracking and issue management application called TrackStar. All of the code is provided. The reader should be able to borrow from all of the examples provided to get up and running quickly, but will also be exposed to deeper discussion and explanation to fully understand what is happening behind the scenes.

What This Book Covers

Chapter 1—Meet Yii introduces Yii at a high level. We learn the importance and utility of using application development frameworks, and the characteristics of Yii that make it incredibly powerful and useful.

Chapter 2—Getting Started walks through a simple Hello, World! style application using the Yii Framework.

Chapter 3—The TrackStar Application provides an introduction to the task management and issue tracking application, TrackStar, that will be built throughout the remainder of the chapters. It also introduces the Test Driven Development (TDD) approach.

Chapter 4—Iteration 1:Creating The Initial TrackStar Application demonstrates the creation of a new database-driven, Yii web application.

Chapter 5—Iteration 2: Project CRUD introduces the automated code generation features of Yii, as we work to build out the "C"reate, "R"ead, "U"pdate and "D"elete functionality for the project entity in our TrackStar application.

Chapter 6—Iteration 3: Adding Tasks introduces us to relational active record and controller class fi lters in Yii, as we add in the management issues into TrackStar.

Chapter 7—Iteration 4: User Management and Authentication covers the first part of Yii's user authentication and authorization framework, Authentication.

Chapter 8—Iteration 5: User Access Control covers the second part of the user authentication and authentication framework, Authorization. Both Yii's simple access control and role-based access control are covered.

*Chapter 9—Iteration 6: Adding User Comment*s takes a deeper dive into writing relational Active Record queries in Yii as well as introduce a basic portlet architecture for reusing content across multiple pages.

Chapter 10—Iteration 7: Adding an RSS Web Feed demonstrates how easy it is to integrate other third-party frameworks into a Yii application by integrating the Zend Framework's Web Feed library to create simple RSS feed within our application.

Chapter 11—Iteration 8: Making It Pretty: Design, Layout, Themes and Iternationalization (i18n) delves deeper into the presentation tier of Yii, introducing layout views, themes as well as internationalization and localization in Yii.

Chapter 12—Iteration 9: Modules – Adding Administration introduces the concept of a module in Yii by using one to add administrative functionality to the application.

Chapter 13—Iteration 10: Production Readiness covers error handling, logging, caching and, security as we prepare our TrackStar application for production.

9
Iteration 6: Adding User Comments

With the implementation of user management in the past two iterations, our Trackstar application is really starting to take shape. The bulk of our primary application feature functionality is now behind us. We can now start to focus on some of the nice-to-have features. The first of these features that we will tackle is the ability for users to leave comments on project issues.

The ability for users to engage in a dialogue about project issues is an important part of what any issue tracking tool should provide. One way to achieve this is to allow users to leave comments directly on the issues. The comments will form a conversation about the issue and provide an immediate, as well as historical context to help track the full lifespan of any issue. We will also use comments to demonstrate using Yii widgets and establishing a portlet model for delivering content to the user (for more information on **Portlets**, visit `http://en.wikipedia.org/wiki/Portlet`).

Iteration planning

The goal of this iteration is to implement feature functionality in the Trackstar application to allow users to leave and read comments on issues. When a user is viewing the details of any project issue, they should be able to read all comments previously added as well as create a new comment on the issue. We also want to add a small fragment of content, or portlet, to the project-listing page that displays a list of recent comments left on all of the issues. This will be a nice way to provide a window into recent user activity and allow easy access to the latest issues that have active conversations.

The following is a list of high-level tasks that we will need to complete in order to achieve these goals:

- Design and create a new database table to support comments
- Create the Yii AR class associated with our new comments table
- Add a form directly to the issue details page to allow users to submit comments
- Display a list of all comments associated with an issue directly on the issues details page
- Take advantage of Yii *widgets* to display a list of the most recent comments on the projects listing page

Creating the model

As always, we should run our existing test suite at the start of our iteration to ensure all of our previously written tests are still passing as expected. By this time, you should be familiar with how to do that, so we will leave it to the reader to ensure that all the unit tests are passing before proceeding.

We first need to create a new table to house our comments. Below is the basic DDL definition for the table that we will be using:

```
CREATE TABLE tbl_comment
(
  `id` INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `content` TEXT NOT NULL,
  `issue_id` INTEGER,
  `create_time` DATETIME,
  `create_user_id` INTEGER,
  `update_time` DATETIME,
  `update_user_id` INTEGER
)
```

As each comment belongs to a specific issue, identified by the `issue_id`, and is written by a specific user, indicated by the `create_user_id` identifier, we also need to define the following foreign key relationships:

```
ALTER TABLE `tbl_comment` ADD CONSTRAINT `FK_comment_issue` FOREIGN
KEY (`issue_id`) REFERENCES `tbl_issue` (`id`);

ALTER TABLE `tbl_comment` ADD CONSTRAINT `FK_comment_author` FOREIGN
KEY (`create_user_id`) REFERENCES `tbl_user` (`id`);
```

If you are following along, please ensure this table is created in both the `trackstar_dev` and `trackstar_test` databases.

Once a database table is in place, creating the associated AR class is a snap. We have seen this many times in previous iterations. We know exactly how to do this. We simply use the Gii code creation tool's `Model Generator` command and create an AR class called `Comment`. If needed, refer back to *Chapters 5 and 6* for all the details on using this tool to create model classes.

Since we have already created the model class for issues, we will need to explicitly add the relations to to the Issue model class for comments. We will also add a relationship as a statistical query to easily retrieve the number of comments associated with a given issue (just as we did in the Project AR class for issues). Alter the `Issue::relations()` method as such:

```
public function relations()
{
  return array(
    'requester' => array(self::BELONGS_TO, 'User', 'requester_id'),
    'owner' => array(self::BELONGS_TO, 'User', 'owner_id'),
    'project' => array(self::BELONGS_TO, 'Project', 'project_id'),
    'comments' => array(self::HAS_MANY, 'Comment', 'issue_id'),
    'commentCount' => array(self::STAT, 'Comment', 'issue_id'),
  );
}
```

Also, we need to change our newly created `Comment` AR class to extend our custom `TrackStarActiveRecord` base class, so that it benefits from the logic we placed in the `beforeValidate()` method. Simply alter the beginning of the class definition as such:

```
<?php
      /**
 * This is the model class for table "tbl_comment".
 */
class Comment extends TrackStarActiveRecord
{
```

We'll make one last small change to the definitions in the `Comment::relations()` method. The relational attributes were named for us when the class was created. Let's change the one named `createUser` to be `author`, as this related user does represent the author of the comment. This is just a semantic change, but will help to make our code easier to read and understand. Change the method as such:

```
    /**
 * @return array relational rules.
 */
public function relations()
{
  // NOTE: you may need to adjust the relation name and the related
  // class name for the relations automatically generated below.
  return array(
    'author' => array(self::BELONGS_TO, 'User', 'create_user_id'),
    'issue' => array(self::BELONGS_TO, 'Issue', 'issue_id'),
  );
}
```

Creating the Comment CRUD

Once we have an AR class in place, creating the CRUD scaffolding for managing the related entity is equally as easy. Again, use the Gii code generation tool's `Crud Generator` command with the AR class name, `Comment`, as the argument. Again, we have seen this many times in previous iterations, so we will leave this as an exercise for the reader. Again, if needed, refer back to *Chapters 5 and 6* for all the details on using this tool to create CRUD scaffolding code. Although we will not immediately implement full CRUD operations for our comments, it is nice to have the scaffolding for the other operations in place.

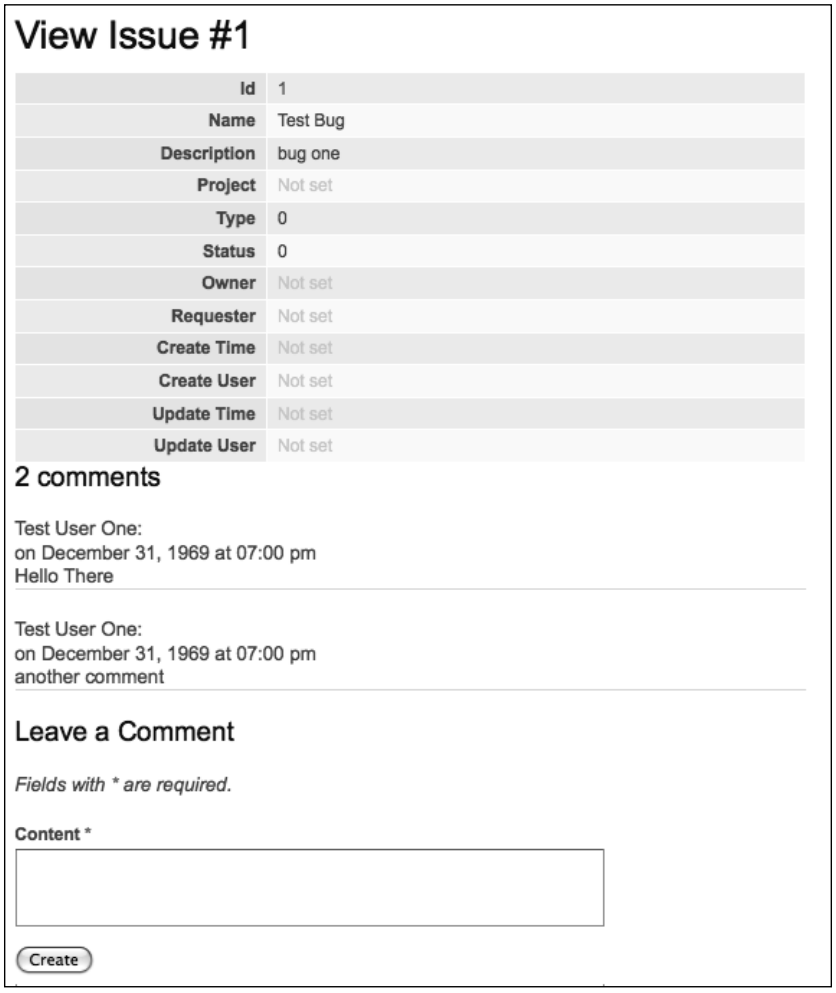
As long as we are logged in, we should now be able to view the autogenerated comment submission form via the following URL:

`http://localhost/trackstar/index.php?r=comment/create`

Altering the scaffolding to meet requirements

As we have seen many times before, we often have to make adjustments to the autogenerated scaffolding code in order to meet the specific requirements of the application. For one, our autogenerated form for creating a new comment has an input field for every single column defined in the `tbl_comment` database table.

We don't actually want all of these fields to be part of the form. In fact, we want to greatly simplify this form to have only a single input field for the comment content. What's more, we don't want the user to access the form via the above URL, but rather only by visiting an issue details page. The user will add comments on the same page where they are viewing the details of the issue. We want to build towards something similar to what is depicted in the following screenshot:



In order to achieve this, we are going to alter our `Issue` controller class to handle the post of the comment form as well as alter the issue details view to display the existing comments and new comment creation form. Also, as comments should only be created within the context of an issue, we'll add a new method to the `Issue` model class to create new comments.

Adding a comment

Let's start by writing a test for this new public method on the `Issue` model class. Open up the `IssueTest.php` file and add the following test method:

```
public function testAddComment()
{
  $comment = new Comment;
  $comment->content = "this is a test comment";
  $this->assertTrue($this->issues('issueBug')->addComment($comment));
}
```

This, of course, will fail until we add the method to our Issue AR class. Add the following method to the Issue AR class:

```
/**
  * Adds a comment to this issue
  */
public function addComment($comment)
{
  $comment->issue_id=$this->id;
  return $comment->save();
}
```

This method ensures the proper setting of the comment issue ID before saving the new comment. Run the test again to ensure it now passes.

With this method in place, we can now turn focus to the issue controller class. As we want the comment creation form to display from and post its data back to the `IssueController::actionView()` method, we will need to alter that method. We will also add a new protected method to handle the form `POST` request. First, alter the `actionView()` method to be the following:

```
public function actionView()
{
    $issue=$this->loadModel();
    $comment=$this->createComment($issue);

    $this->render('view',array(
      'model'=>$issue,
      'comment'=>$comment,
    ));
}
```

Then add the following `protected` method to create a new comment and handle the form `post` request for creating a new comment for this issue:

```
protected function createComment($issue)
{
  $comment=new Comment;
  if(isset($_POST['Comment']))
  {
    $comment->attributes=$_POST['Comment'];
    if($issue->addComment($comment))
    {
      Yii::app()->user->setFlash('commentSubmitted',"Your comment has
been added." );
      $this->refresh();
    }
  }
  return $comment;
}
```

Our new protected method, `createComment()` is responsible for handling the `POST` request for creating a new comment based on the user input. If the comment is successfully created, the page will be refreshed displaying the newly created comment. The changes made to `IssueController::actionView()` are responsible for calling this new method and also feeding the new comment instance to the view.

Displaying the form

Now we need to alter our view. First we are going to create a new view file to render the display of our comments and the comment input form. As we'll render this as a partial view, we'll stick with the naming conventions and begin the filename with a leading underscore. Create a new file called `_comments.php` under the `protected/views/issue/` folder and add the following code to that file:

```
<?php foreach($comments as $comment): ?>
<div class="comment">
      <div class="author">
    <?php echo $comment->author->username; ?>:
  </div>

  <div class="time">
    on <?php echo date('F j, Y \a\t h:i a',strtotime($comment->create_
time)); ?>
  </div>

  <div class="content">
```

```
      <?php echo nl2br(CHtml::encode($comment->content)); ?>
    </div>
       <hr>
</div><!-- comment -->
<?php endforeach; ?>
```

This file expects as an input parameter an array of comment instances and displays them one by one. We now need to alter the `view` file for the issue detail to use this new file. We do this by opening `protected/views/issue/view.php` and adding the following to the end of the file:

```
<div id="comments">
  <?php if($model->commentCount>=1): ?>
    <h3>
      <?php echo $model->commentCount>1 ? $model->commentCount . '
comments' : 'One comment'; ?>
    </h3>

    <?php $this->renderPartial('_comments',array(
      'comments'=>$model->comments,
    )); ?>
  <?php endif; ?>

  <h3>Leave a Comment</h3>

  <?php if(Yii::app()->user->hasFlash('commentSubmitted')): ?>
    <div class="flash-success">
      <?php echo Yii::app()->user->getFlash('commentSubmitted'); ?>
    </div>
  <?php else: ?>
    <?php $this->renderPartial('/comment/_form',array(
      'model'=>$comment,
    )); ?>
  <?php endif; ?>

</div>
```

Here we are taking advantage of the statistical query property, `commentCount`, we added earlier to our Issue AR model class. This allows us to quickly determine if there are any comments available for the specific issue. If there are comments, it proceeds to render them using our `_comments.php` display view file. It then displays the input form that was created for us when we used the Gii Crud Generator functionality. It will also display the simple flash message set upon a successfully saved comment.

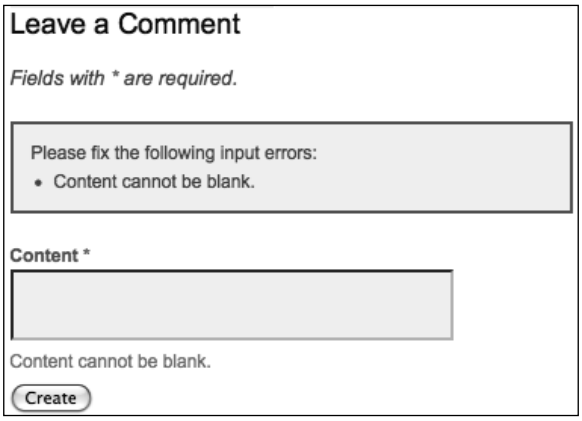
One last change we need to make is to the comment input form itself. As we have seen many times in the past, the form created for us has an input field for every column defined in the underlying `tbl_comment` table. This is not what we want to display to the user. We want to make this a simple input form where the user only needs to submit the comment content. So, open up the `view` file that houses the input form, that is, `protected/views/comment/_form.php` and edit it to be simply:

```
<div class="form">
<?php $form=$this->beginWidget('CActiveForm', array(
  'id'=>'comment-form',
  'enableAjaxValidation'=>false,
)); ?>
        <p class="note">Fields with <span class="required">*</span> are
required.</p>
        <?php echo $form->errorSummary($model); ?>
        <div class="row">
    <?php echo $form->labelEx($model,'content'); ?>
    <?php echo $form->textArea($model,'content',array('rows'=>6,
'cols'=>50)); ?>
    <?php echo $form->error($model,'content'); ?>
  </div>

  <div class="row buttons">
    <?php echo CHtml::submitButton($model->isNewRecord ? 'Create' :
'Save'); ?>
  </div>

<?php $this->endWidget(); ?>

</div>
```

With all of this in place, we can visit an issue listing page, for example `http://hostname/trackstar/index.php?r=issue/view&id=1`

And we see the following comment input form at the bottom of the page:

If we attempt to submit the comment without specifying any content, we see an error as depicted in the following screenshot:



And then, if we are logged in as **Test User One** and we submit the comment **My first test comment**, we are presented with the following display:

Creating a recent comments widget

Now that we have the ability to leave comments on issues, we are going to turn our focus to the second primary goal of this iteration. We want to display to the user a list of all of the recent comments that have been left on various issues across all of the projects. This will provide a nice snapshot of user communication activity within the application. We also want to build this small block of content in a manner that will allow it to be re-used in various different locations throughout the site. This is very much in the style of web portal applications such as news forums, weather reporting applications and sites such as Yahoo and iGoogle. These small snippets of content are often referred to as *portlets*, and this is why we referred to building a portlet architecture at the beginning of this iteration. Again, you can refer to `http://en.wikipedia.org/wiki/Portlet` for more information on this topic.

Introducing CWidget

Lucky for us, Yii is readymade to help us achieve this architecture. Yii provides a component class, called `CWidget`, which is intended for exactly this purpose. A Yii *widget* is an instance of this class (or its child class), and is a presentational component typically embedded in a `view` file to display self-contained, reusable user interface features. We are going to use a Yii widget to build a recent comments portlet and display it on the main project details page so we can see comment activity across all issues related to the project. To demonstrate the ease of re-use, we'll take it one step further and also display a list of project-specific comments on the project details page.

To begin creating our widget, we are going to first add a new `public` method on our `Comment` AR model class to return the most recently added comments. As expected, we will begin by writing a test.

But before we write the `test` method, let's update our comment fixtures data so that we have a couple of comments to use throughout our testing. Create a new file called `tbl_comment.php` within the `protected/tests/fixtures` folder. Open that file and add the following content:

```
<?php

return array(
  'comment1'=>array(
    'content' => 'Test comment 1 on issue bug number 1',
    'issue_id' => 1,
    'create_time' => '',
    'create_user_id' => 1,
    'update_time' => '',
    'update_user_id' => '',
  ),
  'comment2'=>array(
    'content' => 'Test comment 2 on issue bug number 1',
    'issue_id' => 1,
    'create_time' => '',
    'create_user_id' => 1,
    'update_time' => '',
    'update_user_id' => '',
  ),
);
```

Now we have consistent, predictable, and repeatable comment data to work with.

Create a new unit test file, `protected/tests/unit/CommentTest.php` and add the following content:

```
<?php
class CommentTest extends CDbTestCase
{
  public $fixtures=array(
    'comments'=>'Comment',
  );
  public function testRecentComments()
  {
    $recentComments=Comment::findRecentComments();
        $this->assertTrue(is_array($recentComments));
  }
}
```

This test will of course fail, as we have not yet added the `Comment::findRecentComments()` method to the Comment model class. So, let's add that now. We'll go ahead and add the full method we need, rather than adding just enough to get the test to pass. But if you are following along, feel free to move at your own TDD pace. Open `Comment.php` and add the following `public static` method:

```
public static function findRecentComments($limit=10, $projectId=null)
{
    if($projectId != null)
    {
      return self::model()->with(array(
        'issue'=>array('condition'=>'project_id='.$projectId)))-
>findAll(array(
        'order'=>'t.create_time DESC',
        'limit'=>$limit,
      ));
    }
    else
    {
      //get all comments across all projects
      return self::model()->with('issue')->findAll(array(
        'order'=>'t.create_time DESC',
        'limit'=>$limit,
      ));
    }
}
```

Our new method takes in two optional parameters, one to limit the number of returned comments, the other to specify a specific project ID to which all of the comments should belong. The second parameter will allow us to use our new widget to display all comments for a project on the project details page. So, if the input project id was specified, it restricts the returned results to only those comments associated with the project, otherwise, all comments across all projects are returned.

More on relational AR queries in Yii

The above two relational AR queries are a little new to us. We have not been using many of these options in our previous queries. Previously we have been using the simplest approach to executing relational queries:

1. Load the AR instance.
2. Access the relational properties defined in the `relations()` method.

For example if we wanted to query for all of the issues associated with, say, project id #1, we would execute the following two lines of code:

```
// retrieve the project whose ID is 1
$project=Project::model()->findByPk(1);

// retrieve the project's issues: a relational query is actually being
performed behind the scenes here
$issues=$project->issues;
```

This familiar approach uses what is referred to as a **Lazy Loading**. When we first create the project instance, the query does not return all of the associated issues. It only retrieves the associated issues upon an initial, explicit request for them, that is, when `$project->issues` is executed. This is referred to as *lazy* because it waits to load the issues.

This approach is convenient and can also be very efficient, especially in those cases where the associated issues may not be required. However, in other circumstances, this approach can be somewhat inefficient. For example, if we wanted to retrieve the issue information across **N** projects, then using this lazy approach would involve executing **N** join queries. Depending on how large **N** is, this could be very inefficient. In these situations, we have another option. We can use what is called **Eager Loading**.

The Eager Loading approach retrieves the related AR instances at the same time as the main AR instances are requested. This is accomplished by using the `with()` method in concert with either the `find()` or `findAll()` methods for AR query. Sticking with our project example, we could use Eager Loading to retrieve all issues for all projects by executing the following single line of code:

```
//retrieve all project AR instances along with their associated issue
AR instances
$projects = Project::model()->with('issues')->findAll();
```

Now, in this case, every project AR instance in the `$projects` array already has its associated issues property populated with an array of issues AR instances. This result has been achieved by using just a single join query.

We are using this approach in both of the relational queries executed in our `findRecentComments()` method. The one we are using to restrict the comments to a specific project is slightly more complex. As you can see, we are specifying a query condition on the eagerly loaded issue property for the comments. Let's look at the following line:

```
Comment::model()->with(array('issue'=>array('condition'=>'project_
id='.$projectId)))->findAll();
```

This query specifies a single join between the `tbl_comment` and the `tbl_issue` tables. Sticking with project id #1 for this example, the previous relational AR query would basically execute something similar to the following SQL statement:

```
SELECT tbl_comment.*, tbl_issue.* FROM tbl_comment LEFT OUTER JOIN
tbl_issue ON (tbl_comment.issue_id=tbl_issue.id) WHERE (tbl_issue.
project_id=1)
```

The added array we specify in the `findAll()` method simply sets an `order by` clause and a `limit` clause to the executed SQL statement.

One last thing to note about the two queries we are using is how the column names that are common to both tables are disambiguated. Obviously when the two tables that are being joined have columns with the same name, we have to make a distinction between the two in our query. In our case, both tables have the `create_time` column defined. We are trying to order by this column in the `tbl_comment` table and not the one defined in the `issue` table. In a relational AR query in Yii, the alias name for the primary table is fixed as `t`, while the alias name for a relational table, by default, is the same as the corresponding relation name. So, in our two queries, we specify `t.create_time` to indicate we want to use the primary table's column. If we wanted to instead order by the issue `create_time` column, we would alter, the second query for example, as such:

```
return Comment::model()->with('issue')->findAll(array(
  'order'=>'issue.create_time DESC',
  'limit'=>$limit,
));
```

Completing the test

Okay, now that we fully understand what our new method is doing, we need to complete testing of it. In order to fully test our new method, we need to make a few changes to our fixture data. Open each of the fixture data files: `tbl_project.php`, `tbl_issue.php`, and `tbl_comment.php` and ensure each of these entries is in place:

Add the following code in `tbl_project`:

```
'project3'=>array(
  'name' => 'Test Project 3',
  'description' => 'This is test project 3',
  'create_time' => '',
  'create_user_id' => '',
  'update_time' => '',
  'update_user_id' => '',
),
```

In `tbl_issue`, add the following code:

```
'issueFeature2'=>array(
  'name' => 'Test Feature For Project 3',
  'description' => 'This is a test feature issue associated with
project # 3 that is completed',
  'project_id' => 3,
  'type_id' => 1,
  'status_id' => 2,
  'owner_id' => 1,
  'requester_id' => 1,
  'create_time' => '',
  'create_user_id' => '',
  'update_time' => '',
  'update_user_id' => '',
),
```

Finally, add the following code in `tbl_comment`:

```
'comment3'=>array(
  'content' => 'The first test comment on the first feature issue
associated with Project #3',
  'issue_id' => 3,
  'create_time' => '',
  'create_user_id' => '',
  'update_time' => '',
  'update_user_id' => '',
),
```

We now have a total of three comments in our test database. Two of them associated with project #1 and one associated with project #3.

Now we can alter our test method to test:

- Requesting all comments
- Limiting the number of returned comments to just two
- Restricting the returned comments to only those associated with project #3

The following method tests all three scenarios:

```
public function testRecentComments()
{
      //retrieve all the comments for all projects
  $recentComments = Comment::findRecentComments();
      $this->assertTrue(is_array($recentComments));
      $this->assertEquals(count($recentComments),3);

      //make sure the limit is working
      $recentComments = Comment::findRecentComments(2);
      $this->assertTrue(is_array($recentComments));
      $this->assertEquals(count($recentComments),2);

      //test retrieving comments only for a specific project
      $recentComments = Comment::findRecentComments(5, 3);
      $this->assertTrue(is_array($recentComments));
      $this->assertEquals(count($recentComments),1);
}
```

We also need to ensure that our `CommentTest` class is using the fixture data for `comments`, `issues`, and `projects`. Make sure the following fixtures are defined at the top of our `CommentTest` class:

```
<?php
class CommentTest extends CDbTestCase
{
  public $fixtures=array(
    'comments'=>'Comment',
    'projects'=>'Project',
    'issues'=>'Issue',
  );
```

Now, if we run this test again, we should have all six assertions passing:

```
>>phpunit unit/CommentTest.php
PHPUnit 3.4.12 by Sebastian Bergmann.
.
Time: 0 seconds
OK (1 test, 6 assertions)
```

Armed with the knowledge of the benefits of Lazy Loading versus Eager Loading in Yii, we should make an adjustment to how the `Issue` model is loaded within the `IssueController::actionView()` method. Since we have altered the issues detail view to display our comments, including the author of the comment, we know it will be more efficient to use the Eager Loading approach to load our comments along with their respective authors when we make the call to `loadModel()` in this method. To do this, we can add a simple input flag to this `loadModel()` method to indicate whether or not we want to load the comments as well.

Alter the `IssueController::loadModel()` method as shown below:

```
public function loadModel($withComments=false)
  {
    if($this->_model===null)
    {
      if(isset($_GET['id']))
      {
         if($withComments)
         {
           $this->_model=Issue::model()->with(array(
           'comments'=>array('with'=>'author')))
                                    ->findbyPk($_GET['id']);
         }
         else
```

```
            {
              $this->_model=Issue::model()->findbyPk($_GET['id']);
            }
          }
        if($this->_model===null)
        throw new CHttpException(404,'The requested page does not
                                  exist.');
      }
    return $this->_model;
  }
```

Now we can change the call to this method in `IssueController::actionView()`, as such:

```
public function actionView()
  {
    $issue=$this->loadModel(true);
```

With this in place, we will load all of our comments, along with their respective author information, with just one database call.

Creating the widget

Now we are ready to create our new widget to use our new method to display our recent comments.

As we previously mentioned a widget in Yii is a class that extend from the framework class `CWidget` or one of its child classes. We'll add our new widget to the `protected/components/` directly, as the contents of this folder are already specified in the main configuration file to be auto-loaded within the application. This way we won't have to explicitly import the class every time we wish to use it. We'll name our widget `RecentComments`, so we need to add a `php` file of the same name to this directly. Add the following class definition to this newly created `RecentComment.php` file:

```
<?php
/**
 * RecentComments is a Yii widget used to display a list of recent
comments
 */
class RecentComments extends CWidget
{
  private $_comments;
  public $displayLimit = 5;
  public $projectId = null;
  public function init()
```

```
        {
            $this->_comments = Comment::model()
            ->findRecentComments($this->displayLimit,
             $this->projectId);
        }
        public function getRecentComments()
    {
        return $this->_comments;
    }
        public function run()
        {
            // this method is called by CController::endWidget()
            $this->render('recentComments');
        }
}
```

The primary work involved when creating a new widget is to override the `init()` and `run()` methods of the base class. The `init()` method initializes the widget and is called after its properties have been initialized. The `run()` method executes the widget. In this case, we simply initialize the widget by requesting recent comments based on the `$displayLimit` and `$projectId` properties. The execution of the widget itself simply renders its associated view file, which we have yet to create. `view` files, by convention, are placed in `views/` directly within the same folder where the widget resides, and have the same name as the widget, but start with a lowercase letter. Sticking with convention, create a new file whose fully qualified path is `protected/components/views/renderComments.php`. Once created, add the following markup to that file:

```
<ul>
  <?php foreach($this->getRecentComments() as $comment): ?>
    <div class="author">
      <?php echo $comment->author->username; ?> added a comment.
    </div>
    <div class="issue">
       <?php echo CHtml::link(CHtml::encode($comment->issue->name),
array('issue/view', 'id'=>$comment->issue->id)); ?>
      </div>
  <?php endforeach; ?>
</ul>
```

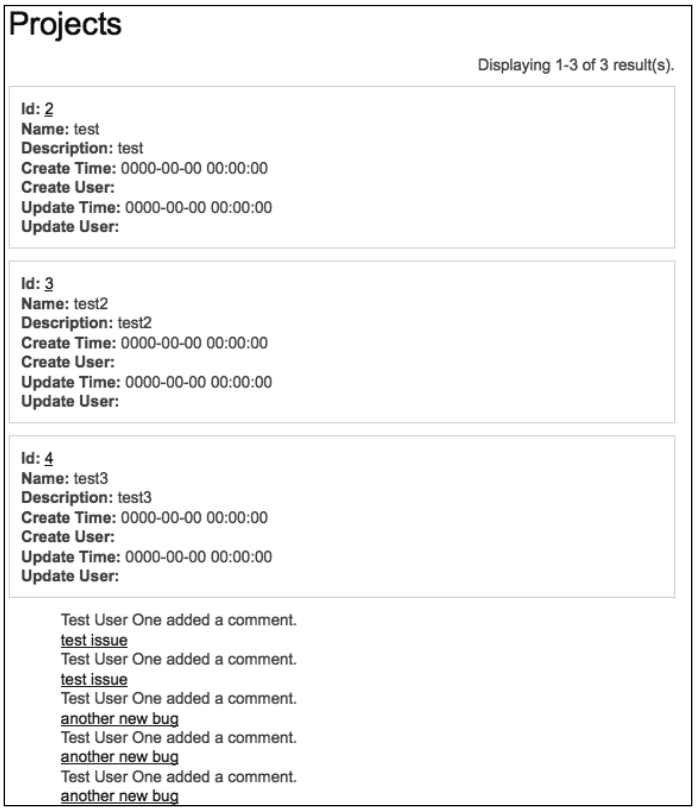
This calls the `RenderComments` widget's `getRecentComments()` method, which returns an array of comments. It then iterates over each of them displaying who added the comment and the associated issue on which the comment was left.

In order to see the results, we need to embed this widget into an existing controller view file. As previously mentioned, we want to use this widget on the projects listing page, to display all recent comments across all projects, and also on a specific project details page, to display the recent comments for just that specific project.

Let's start with the project listing page. The `view` file responsible for displaying that content is `protected/views/project/index.php`. Open up that file and add the following at the bottom:

```
<?php $this->widget('RecentComments'); ?>
```

Now if we view the projects listing page `http://localhost/trackstar/index.php?r=project`, we see something similar to the following screenshot:



We have now embedded our new recent comments data within the page simply by calling the widget. This is nice, but we can take our little widget one step further to have it display in a consistent manner with all other potential portlets in the application. We can do this by taking advantage of another class provided to us by Yii, `CPortlet`.

Introducing CPortlet

`CPortlet` is part of `zii`, the official extension class library that comes packaged with Yii. It provides a nice base class for all portlet-style widgets. It will allow us to render a nice title as well as consistent HTML markup, so that all portlets across the application can be easily styled in a similar manner. Once we have a widget that renders content (like our `RecentComments` widget), we can simply use the rendered content of our widget as the content for `CPortlet`, which itself is a widget, as it also extends from `CWidget`. We can do this by placing our call to the `RecentComments` widget between a `beginWidget()` and an `endWiget()` call for `CPortlet`, as such:

```
<?php $this->beginWidget('zii.widgets.CPortlet', array(
  'title'=>'Recent Comments',
));

$this->widget('RecentComments');

$this->endWidget(); ?>
```

Since `CPortlet` provides a title property, we set it to be something meaningful for our portlet. We then use the rendered content of the `RecentComments` widget to drive the content for the porlet widget. The end result of this is depicted in the following screenshot:

This is not a huge change from what we had previously, but we have now placed our content into a consistent container that is already being used throughout the site. Notice the similarity between the right column menu content block and our newly created recent comments content block. I am sure it will come as no surprise to you that this right column menu block is also displayed within a `CPortlet` container. Taking a peek in `protected/views/layouts/column2.php`, which is a file that the `yiic webapp` command autogenerated for us when we initially created the application, reveals the following code:

```
<?php
  $this->beginWidget('zii.widgets.CPortlet', array(
    'title'=>'Operations',
  ));
  $this->widget('zii.widgets.CMenu', array(
    'items'=>$this->menu,
    'htmlOptions'=>array('class'=>'operations'),
  ));
  $this->endWidget();
?>
```

So it seems that the application has been taking advantage of portlets all along.

Adding our widget to another page

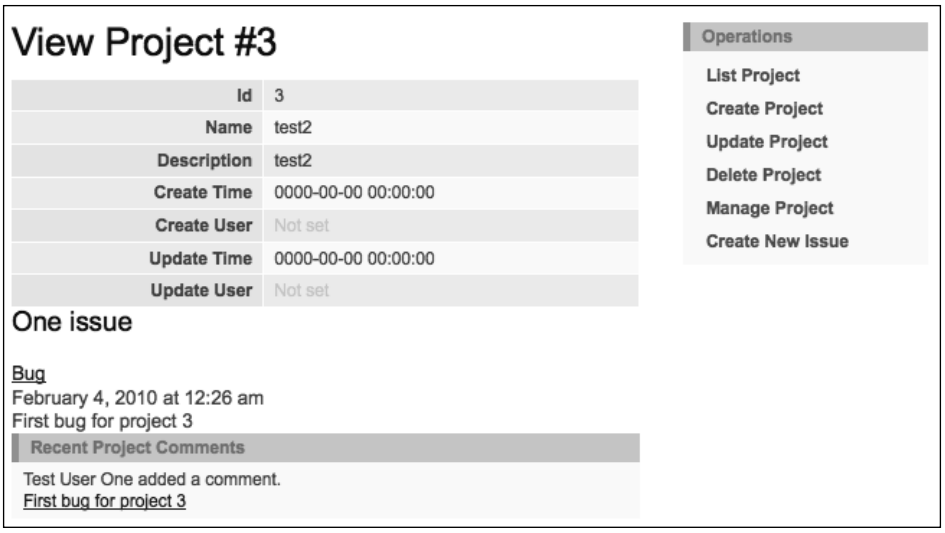
Let's also add our portlet to the project details page, and restrict the comments to just those associated with the specific project.

Add the following to the bottom of the `protected/views/project/view.php` file:

```
<?php $this->beginWidget('zii.widgets.CPortlet', array(
  'title'=>'Recent Project Comments',
));

$this->widget('RecentComments', array('projectId'=>$model->id));

$this->endWidget(); ?>
```

This is basically the same thing we added to the project listings page, except we are initializing the `RecentComments` widget's `$projectId` property by adding an array of *name*=>*value* pairs to the call.

Now if we visit a specific project details page, we should see something similar to the following screenshot:



This screenshot shows the details page for project #3, which has one associated issue with just one comment on that issue, as depicted in the picture. You may need to add a few issues and comments on those issues in order to generate a similar display. We now have a way to display recent comments with a few different configurable parameters anywhere throughout the site in a consistent and easily maintainable manner.

Summary

With this iteration, we have started to flesh out our Trackstar application with functionality that has come to be expected of most user-based web applications today. The ability for users to communicate with each other within the application is an essential part of a successful issue management system.

As we created this essential feature, we were able to deeper look into how to write relational AR queries. We were also introduced to content components called widgets and portlets. This introduced us to an approach to developing small content blocks and being able to use them anywhere throughout the site. This approach greatly increases reuse, consistency, and ease of maintenance.

In the next iteration, we'll build upon the recent comments widget created here, and expose the content generated by our widget as an RSS feed to allow users to track application or project activity without having to visit the application.

Where to buy this book

You can buy Oracle Agile Web Application Development with Yii 1.1 and PHP5 from the Packt Publishing website: `https://www.packtpub.com/yii-1-1-and-php5-for-agile-web-application-development/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com