Cake php

Science, Computer Science

JAYPEE UNIVERSITY OF ENGINEERING ANDTECHNOLOGYRAGHOGARH, GUNA (M. P.) INDUSTRIAL TRAINING REPORT at ***** Software Solutions Limited JUNE-JULY 2011 SUBMITTED BY: NAME: Rohit Gupta EN NO. : 08307G YEAR, BRANCH: 2011, CSE Acknowledgement It has been a matter of great pleasure for me for having got an opportunity to convey my sincere thanks to entire people who devoted their valuable time to help me in the duration of training and project work. They extend their support, assistance and enable me to complete the project work successfully. First I would like to express the profound sense of gratitude to Mr.

SA (Training Head) for being a source of inspiration and providing me with necessary guidance whenever I needed it, despite his busy schedule. Words simply cannot express the gratitude and indebtedness to him. I would also like to thank Mr. V&&&&&&&& who was my mentor during my stay at *****, who helped me a lot in providing me with resources and helping me in all the ways he could during the training. Index Sno| Topic| page| 1| Company Profile| 4| 2| Basics of PHP5| 6| 3| Basics of cake PHP| 18| 4| Implementation of Cake PHP| 44| 5| Conclusion| 50| 6| Joining letter| 51| 7| Weekly Reports| 52| | | | | | | | | | Company Profile PHP 5 Introduction With the advent of PHP 5, the object model was rewritten to allow for better performance and more features. This was a major change from PHP 4. PHP 5 has a full object model. Some of the prominent features in PHP 5, include are
of visibility, abstract and final classes and methods, additional magic methods, interfaces, cloning and typehinting. The Basics class Basic class definitions begin with the keyword class, followed by a class name, followed

by a pair of curly braces which enclose the definitions of the properties and methods belonging to the class.

The class name can be any valid label which is a not a PHP reserved word. A valid class name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. A class may contain its own constants, variables (called " properties"), and functions (called " methods"). ---------- Example: Simple Class definition var; } } ?> The pseudo-variable $this is available when a method is called from within an object context. this is a reference to the calling object (usually the object to which the method belongs, but possibly another object, if the method is called statically from the context of a secondary object). new To create an instance of a class, the new keyword must be used. Classes should be defined before instantiation. If a string containing the name of a class is used with new, a new instance of that class will be created. If the class is in a namespace, its fully qualified name must be used when doing this. ---------- Example: Creating an instance In the class context, it is possible to create a new object by new self and new parent. When assigning an already created instance of a class to a new variable, the new variable will access the same instance as the object that was assigned. This behaviour is the same when passing instances to a function. A copy of an already created object can be made by cloning it. extends A class can inherit the methods and properties of another class by using the keyword extends in the class declaration.

It is not possible to extend multiple classes; a class can only inherit from one base class. The inherited methods and properties can be overridden by redeclaring them with the same name defined in the parent class. However, if the parent class has defined a method as final, that method may not be overridden. It is possible to access the overridden methods or static properties by referencing them with parent::.

---------- Example: Simple Class Inheritance displayVar(); ?> ---------- The above example will output: Extending class a default value Properties Class member variables are called " properties". They are defined by using one of the keywords public, protected, or private, followed by a normal variable declaration. This declaration may include an initialization, but this initialization must be a constant value--that is, it must be able to be evaluated at compile time and must not depend on run-time information in order to be evaluated.

Within class methods the properties, constants, and methods may be accessed by using the form $this-> property (where property is the name of the property) unless the access is to a static property within the context of a static class method, in which case it is accessed using the formself:: $property. The pseudo-variable $this is available inside any class method when that method is called from within an object context. $this is a reference to the calling object (usually the object to which the method belongs, but possibly another object, if the method is called statically from the context of a secondary object).

Class Constants It is possible to define constant values on a per-class basis remaining the same and unchangeable. Constants differ from normal variables in that you don't use the $ symbol to declare or use them. The value must be a constant expression, not (for example) a variable, a property, a result of a mathematical operation, or a function call. ---------- Example: Defining and using a constant

By calling this function the scripting engine is given a last chance to load the class before PHP fails with an error. ---------- Example: Autoload example ---------- This example attempts to load the classes MyClass1 and MyClass2 from the files MyClass1. php and MyClass2. php respectively. Constructors and Destructors Constructor void __construct ([mixed $args [, $...]])

PHP 5 allows developers to declare constructor methods for classes. Classes which have a constructor method call this method on each newly-created object, so it is suitable for any initialization that the object may need before it is used. Destructor void __destruct (void) PHP 5 introduces a destructor concept similar to that of other object-oriented languages, such as C++. The destructor method will be called as soon as all references to a particular object are removed or when the object is explicitly destroyed or in any order in shutdown sequence. Like constructors, parent destructors will not be called implicitly by the engine.

In order to run a parent destructor, one would have to explicitly call parent:: __destruct() in the destructor body. The destructor will be called even if script execution is stopped using exit(). Calling exit() in a destructor will

prevent the remaining shutdown routines from executing. Visibility The visibility of a property or method can be defined by prefixing the declaration with the keywords public, protected or private. Class members declared public can be accessed everywhere. Members declared protected can be accessed only within the class itself and by inherited and parent classes.

Members declared as private may only be accessed by the class that defines the member. Property Visibility Class properties must be defined as public, private, or protected. If declared using var, the property will be defined as public. Method Visibility Class methods may be defined as public, private, or protected. Methods declared without any explicit visibility keyword are defined as public. Visibility from other objects Objects of the same type will have access to each others private and protected members even though they are not the same instances.

This is because the implementation specific details are already known when inside those objects. Object Inheritance Inheritance is a well-established programming principle, and PHP makes use of this principle in its object model. This principle will affect the way many classes and objects relate to one another. For example, when you extend a class, the subclass inherits all of the public and protected methods from the parent class. Unless a class overrides those methods, they will retain their original functionality.

This is useful for defining and abstracting functionality, and permits the implementation of additional functionality in similar objects without the need to reimplement all of the shared functionality.

---------- Example #1 Inheritance Example

printItem('baz'); // Output: 'Foo: baz' $foo->

printPHP(); // Output: 'PHP is great' $bar->

printItem('baz'); // Output: 'Bar: baz' $bar->

printPHP(); // Output: 'PHP is great' ?> Scope Resolution Operator (::) The Scope Resolution Operator or in simpler terms, the double colon, is a token that allows access to static, constant, and overridden properties or methods of a class. When referencing these items from outside the class definition, use the name of the class. When an extending class overrides the parents definition of a method, PHP will not call the parent's method.

It's up to the extended class on whether or not the parent's method is called. This also applies to Constructors and Destructors, Overloading, and Magic method definitions. Static Keyword Declaring class properties or methods as static makes them accessible without needing an instantiation of the class. A property declared as static can not be accessed with an instantiated class object (though a static method can). Because static methods are callable without an instance of the object created, the pseudo-variable $this is not available inside the method declared as static.

Static properties cannot be accessed through the object using the arrow operator -;. Class Abstraction PHP 5 introduces abstract classes and methods. Classes defined as abstract may not be instantiated, and any class that contains at least one abstract method must also be abstract. Methods defined as abstract simply declare the method's signature - they cannot define the implementation. When inheriting from an abstract class, all methods marked abstract in the parent's class declaration must be defined

by the child; additionally, these methods must be defined with the same (or a less restricted) visibility.

For example, if the abstract method is defined as protected, the function implementation must be defined as either protected or public, but not private. Object Interfaces Object interfaces allow you to create code which specifies which methods a class must implement, without having to define how these methods are handled. Interfaces are defined using the interface keyword, in the same way as a standard class, but without any of the methods having their contents defined. All methods declared in an interface must be public, this is the nature of an interface. implements To implement an interface, the implements operator is used.

All methods in the interface must be implemented within a class; failureto do so will result in a fatal error. Classes may implement more than one interface if desired by separating each interface with a comma. Constants Its possible for interfaces to have constants. Interface constants works exactly like class constants except they cannot be overridden by a class/interface that inherits it. Overloading Overloading in PHP provides means to dynamically " create" properties and methods. These dynamic entities are processed via magic methods one can establish in a class for various action types.

The overloading methods are invoked when interacting with properties or methods that have not been declared or are not visible in the current scope All overloading methods must be defined as public. PHP's interpretation of " overloading" is different than most object oriented languages. Overloading traditionally provides the ability to have multiple methods with the same

name but different quantities and types of arguments. Property overloading __set() is run when writing data to inaccessible properties. __get() is utilized for reading data from inaccessible properties. _isset() is triggered by calling isset() or empty() on inaccessible properties. __unset() is invoked when unset() is used on inaccessible properties. Method overloading mixed __call (string $name , array $arguments) mixed __callStatic (string $name , array $arguments) __call() is triggered when invoking inaccessible methods in an object context. __callStatic() is triggered when invoking inaccessible methods in a static context. Patterns Patterns are ways to describe best practices and good designs. They show a flexible solution to common programming problems. Factory

The Factory pattern allows for the instantiation of objects at runtime. It is called a Factory Pattern since it is responsible for " manufacturing" an object. A Parameterized Factory receives the name of the class to instantiate as argument. ---------- Example: Parameterized Factory Method ---------- Defining this method in a class allows drivers to be loaded on the fly. If the Example class was a database abstraction class, loading a MySQLand SQLite driver could be done as follows: Singleton The Singleton ensures that there can be only one instance of a Class and provides a global access point to that instance The Singleton pattern is often implemented in Database Classes, Loggers, Front Controllers or Request and Response objects.

Magic Methods The function names __construct, __destruct, __call, __callStatic, __get, __set, __isset, __uns

et, __sleep, __wakeup, __toString, __invoke, __set_state and __clone are magical in PHP classes. You cannot have functions with these names in any of your classes unless you want the magic functionality associated with them. __sleep and __wakeup serialize() checks if your class has a function with the magic name __sleep. If so, that function is executed prior to any serialization. It can clean up the object and is supposed to return an array with the names of all variables of that object that should be serialized.

If the method doesn't return anything then NULL is serialized and E_NOTICE is issued. __toString The __toString method allows a class to decide how it will react when it is treated like a string. For example, what echo $obj; will print. This method must return a string, as otherwise a fatal E_RECOVERABLE_ERROR level error is emitted. __invoke The __invoke method is called when a script tries to call an object as a function. __set_state The only parameter of this method is an array containing exported properties in the form array('property' =; value, ...). Final Keyword

PHP 5 introduces the final keyword, which prevents child classes from overriding a method by prefixing the definition with final. If the class itself is being defined final then it cannot be extended. Object Cloning Creating a copy of an object with fully replicated properties is not always the wanted behavior. A good example of the need for copy constructors, is if your object holds a reference to another object which it uses and when you replicate the parent object you want to create a new instance of this other object so that the replica has its own separate copy.

An object copy is created by using the clone keyword (which calls the object's __clone() method if possible). An object's __clone() method cannot be called directly. ---------- $copy_of_object = clone $object; When an object is cloned, PHP 5 will perform a shallow copy of all of the object's properties. Any properties that are references to other variables, will remain references. Once the cloning is complete, if a __clone() method is defined, then the newly created object's __clone() method will be called, o allow any necessary properties that need to be changed. Type Hinting PHP 5 introduces Type Hinting. Functions are now able to force parameters to be objects (by specifying the name of the class in the function prototype) or arrays (since PHP 5. 1). However, if NULL is used as the default parameter value, it will be allowed as an argument for any later call. Objects and references One of the key-points of PHP5 OOP that is often mentioned is that " objects are passed by references by default". This is not completely true. This section rectifies that general thought using some examples.

A PHP reference is an alias, which allows two different variables to write to the same value. As of PHP5, an object variable doesn't contain the object itself as value anymore. It only contains an object identifier which allows object accessors to find the actual object. When an object is sent by argument, returned or assigned to another variable, the different variables are not aliases: they hold a copy of the identifier, which points to the same object. Cake PHP CakePHP is a free, open-source, rapid development framework for PHP.

It’s a foundational structure for programmers to create web applications. The primary goal of using cake PHP is to enable us to work in a structured and rapid manner–without loss of flexibility. CakePHP takes the monotony out of web development. It provides us with all the tools one needs to get started coding. Instead of reinventing the wheel every time one sits down to a new project, CakePHP helps make the development very easy and rapid and more modularized. CakePHP has an active developer team and community, bringing great value to the project. Understanding Model-View-Controller

CakePHP follows the MVC software design pattern. Programming using MVC separates your application into three main parts: The Model represents the application data The View renders a presentation of model data The Controller handles and routes requests made by the client A Basic MVC Request Figure: 1 shows an example of a bare-bones MVC request in CakePHP. To illustrate, assume a client just clicked on the “ Buy A Custom Cake Now! ” link on your application’s home page. Client clicks the link pointing to http://www. example. com/cakes/buy, and his browser makes a request to your web server.

The dispatcher checks the request URL (/cakes/buy), and hands the request to the correct controller. The controller performs application specific logic. For example, it may check to see if Client has logged in. The controller also uses models to gain access to the application’s data. Models usually represent database tables, but they could also
represent LDAP entries, RSS feeds, or files on the system. In this example, the controller uses a model to fetch Client’s last purchases from the

database. Once the controller has worked its magic on the data, it hands it to a view.

The view takes this data and gets it ready for presentation to the client. Views in CakePHP are usually in HTML format, but a view could just as easily be a PDF, XML document, or JSON object depending on your needs. Once the view has used the data from the controller to build a fully rendered view, the content of that view is returned to Client’s browser. Almost every request to your application will follow this basic pattern. We'll add some details later on which are specific to CakePHP, so keep this in mind as we proceed. Benefits Why use MVC?

Because it is a tried and true software design pattern that turns an application into a maintainable, modular, rapidly developed package. Crafting application tasks into separate models, views, and controllers makes your application very light on its feet. New features are easily added, and new faces on old features are a snap. The modular and separate design also allows developers and designers to work simultaneously, including the ability to rapidly prototype. Separation also allows developers to make changes in one part of the application without affecting others.

Basic principles of Cake PHP The CakePHP framework provides a robust base for our application. It can handle every aspect, from the user’s initial request all the way to the final rendering of a web page. And since the framework follows the principles of MVC, it allows us to easily customize and extend most aspects of your application. The framework also provides a basic organizational structure, from filenames to database table names, keeping

your entire application consistent and logical. This concept is simple but powerful. Cake PHP structure

CakePHP features Controller, Model, and View classes, but it also features some additional classes and objects that make development in MVC a little quicker and more enjoyable. Components, Behaviors, and Helpers are classes that provide extensibility and reusability to quickly add functionality to the base MVC classes in your applications. Controllers Extensions(Components) A Component is a class that aids in controller logic. If you have some logic you want to share between controllers (or applications), a component is usually a good fit. As an example, the core EmailComponent class makes creating and sending emails a snap.

Rather than writing a controller method in a single controller that performs this logic, you can package the logic so it can be shared. Controllers are also fitted with callbacks. These callbacks are available for your use, just in case you need to insert some logic between CakePHP’s core operations. Callbacks available include: * beforeFilter(), executed before any controller action logic * beforeRender(), executed after controller logic, but before the view is rendered * afterFilter(), executed after all controller logic, including the view render.

There may be no difference between afterRender() and afterFilter() unless you’ve manually made a call to render() in your controller action and have included some logic after that call. View Extensions (Helpers) Helper is a class that aids in view logic. Much like a component used among controllers, helpers allow presentational logic to be accessed and shared between views.

Most applications have pieces of view code that are used repeatedly. CakePHP facilitates view code reuse with layouts and elements. By default, every view rendered by a controller is placed inside a layout.

Elements are used when small snippets of content need to be reused in multiple views. Model Extensions (“ Behaviors”) Similar to Components and Helpers, " Behaviors" work as ways to add common functionality between models. For example, if you store user data in a tree structure, you can specify your " User" model as behaving like a tree, and gain free functionality for removing, adding, and shifting nodes in your underlying tree structure. Models also are supported by another class called a DataSource. DataSources are an abstraction that enable models to manipulate different types of data consistently.

While the main source of data in a CakePHP application is often a database, you might write additional DataSources that allow your models to represent RSS feeds, CSV files, LDAP entries, or iCal events. Just like controllers, models are featured with callbacks as well. A Typical CakePHP Request let’s imagine that our client just clicked on the “ Buy A Custom Cake Now! ” link on a CakePHP application’s landing page. Typical Cake Request. 1. Client clicks the link pointing to http://www. example. com/cakes/buy, and his browser makes a request to your web server. . The Router parses the URL in order to extract the parameters for this request: the controller, action, and any other arguments that will affect the business logic during this request. 3. Using routes, a request URL is mapped to a controller action (a method in a specific controller class). In this case, it’s the buy() method of the

CakesController. The controller’s beforeFilter() callback is called before any controller action logic is executed. 4. The controller may use models to gain access to the application’s data.

In this example, the controller uses a model to fetch Client’s last purchases from the database. Any applicable model callbacks, behaviors, and DataSources may apply during this operation. While model usage is not required, all CakePHP controllers initially require at least one model. 5. After the model has retrieved the data, it is returned to the controller. Model callbacks may apply. 6. The controller may use components to further refine the data or perform other operations (session manipulation, authentication, or sending emails, for example). 7.

Once the controller has used models and components to prepare the data sufficiently, that data is handed to the view using the controller’s set() method. Controller callbacks may be applied before the data is sent. The view logic is performed, which may include the use of elements and/or helpers. By default, the view is rendered inside of a layout. 8. Additional controller callbacks (like afterFilter) may be applied. The complete, rendered view code is sent to Client’s browser. CakePHP Folder Structure After downloading and extracted CakePHP, these are the files and folders one should see: * app cake * vendors * plugins * . htaccess * index. php * README Three main folders are: * The app folder will be where you work your magic: it’s where your application’s files will be placed. * The cake folder is where we’ve worked our magic. Make a personal commitment not to edit files in this folder. We can’t help you if you’ve modified the core. *

Finally, the vendors folder is where you'll place third-party PHP libraries you need to use with your CakePHP applications. The App Folder CakePHP's app folder is where you will do most of your application development.

Let's look a little closer at the folders inside of app. config| Holds the (few) configuration files CakePHP uses. Database connection details, bootstrapping, core configuration files and more should be stored here. | controllers| Contains your application's controllers and their components. | Libs| Contains 1st party libraries that do not come from 3rd parties or external vendors. This allows you to separate your organization's internal libraries from vendor libraries. | locale| Stores string files for internationalization. | models| Contains your application's models, behaviors, and datasources. plugins| Contains plugin packages. | tmp| This is where CakePHP stores temporary data. The actual data it stores depends on how you have CakePHP configured, but this folder is usually used to store model descriptions, logs, and sometimes session information. Make sure that this folder exists and that it is writable, otherwise the performance of your application will be severely impacted. In debug mode, CakePHP will warn you if it is not the case. | vendors| Any third-party classes or libraries should be placed here. Doing so makes them easy to access using the App:: import('vendor', 'name') function.

Keen observers will note that this seems redundant, as there is also a vendors folder at the top level of our directory structure. We'll get into the differences between the two when we discuss managing multiple applications and more complex system setups. | views| Presentational files

are placed here: elements, error pages, helpers, layouts, and view files. | webroot| In a production setup, this folder should serve as the document root for your application. Folders here also serve as holding places for CSS stylesheets, images, and JavaScript files| CakePHP Conventions

CakePHP’s conventions have been distilled out of years of web development experience and best practices. CakePHP Conventions File and Classname Conventions In general, filenames are underscored while classnames are CamelCased. So if you have a class MyNiftyClass, then in Cake, the file should be named my_nifty_class. php. Below are examples of how to name the file for each of the different types of classes you would typically use in a CakePHP application: Each file would be located in or under (can be in a subfolder) the appropriate folder in your app folder. Model and Database Conventions

Model classnames are singular and CamelCased. Person, BigPerson, and ReallyBigPerson are all examples of conventional model names. Table names corresponding to CakePHP models are plural and underscored. The underlying tables for the above mentioned models would be people, big_people, and really_big_people, respectively. Field names with two or more words are underscored like, first_name. Controller Convention Controller classnames are plural, CamelCased, and end in Controller. PeopleController and LatestArticlesController are both examples of conventional controller names. View Conventions

View template files are named after the controller functions they display, in an underscored form. The getReady() function of the PeopleController class

will look for a view template in /app/views/people/get_ready. ctp. Developing with CakePHP Installation Installing CakePHP can be as simple as slapping it in your web server’s document root, or as complex and flexible as you wish. There are three main installation types for CakePHP: development, production, and advanced. * Development: easy to get going, URLs for the application include the CakePHP installation directory name, and less secure. Production: Requires the ability to configure the web server’s document root, clean URLs, very secure. * Advanced: With some configuration, allows you to place key CakePHP directories in different parts of the filesystem, possibly sharing a single CakePHP core library folder amongst many CakePHP applications. Development A development installation is the fastest method to setup Cake. This example will help you install a CakePHP application and make it available at http://www. example. com/cake_1_3/. We assume for the purposes of this example that your document root is set to /var/www/html.

Unpack the contents of the Cake archive into /var/www/html. You now have a folder in your document root named after the release you've downloaded (e. g. cake_1. 3. 0). Rename this folder to cake_1_3. Your development setup will look like this on the file system: * /var/www/html * /cake_1_3 * /app * /cake * /vendors * /. htaccess * /index. php * /README If your web server is configured correctly, you should now find your Cake application accessible at http://www. example. com/cake_1_3/. ---------- Controllers Introduction

A controller is used to manage the logic for a part of your application. Most commonly, controllers are used to manage the logic for a single model. For

example, if you were building a site for an online bakery, you might have a RecipesController and a IngredientsController managing your recipes and their ingredients. In CakePHP, controllers are named after the model they handle, in plural form. Your application's controllers are classes that extend the CakePHP AppController class, which in turn extends a core Controller class, which are part of the CakePHP library.

The AppController class can be defined in /app/app_controller. php and it should contain methods that are shared between all of your application’s controllers. Controllers can include any number of methods which are usually referred to as actions. Actions are controller methods used to display views. An action is a single method of a controller. CakePHP’s dispatcher calls actions when an incoming request matches a URL to a controller’s action ---------- The App Controller As stated in the introduction, the AppController class is the parent class to all of your application's controllers.

AppController itself extends the Controller class included in the CakePHP core library. As such, AppController is defined in /cake/libs/controller/app_controller. php or /app/app_controller. php. If /app/app_controller. php does not exist then copy from /cake location before customizing for application. It contains a skeleton definition: ---------- 1. ----------

Controller attributes and methods created in your AppController will be available to all of your application's controllers. It is the ideal place to create code that is common to all of your controllers. Components (which you'll

learn about later) are best used for code that is used in many (but not necessarily all) controllers.

While normal object-oriented inheritance rules apply, CakePHP also does a bit of extra work when it comes to special controller attributes, like the list of components or helpers used by a controller. In these cases, AppController value arrays are merged with child controller class arrays.
---------- The Pages Controller CakePHP core ships with a default controller called the Pages Controller (cake/libs/controller/pages_controller. php). The home page you see after installation is generated using this controller. It is generally used to serve static pages.

Eg. If you make a view file app/views/pages/about_us. ctp you can access it using url http://example. com/pages/about_us
---------- Controller Attributes $name People using PHP4 should start out their controller definitions using the $name attribute. The $name attribute should be set to the name of the controller. Usually this is just the plural form of the primary model the controller uses. This takes care of some PHP4 classname oddities and helps CakePHP resolve naming.
---------- See comments for this section $components, $helpers and $uses

The next most often used controller attributes tell CakePHP what helpers, components, and models you'll be using in conjunction with the current controller. Using these attributes make MVC classes given by $components and $uses available to the controller as class variables ($this-; ModelName,

for example) and those given by $helpers to the view as an object reference variable ($helpername). Controllers have access to their primary model available by default. Our RecipesController will have the Recipe model class available at $this-; Recipe, and our ProductsController also features the Product model at $this-; Product.

However, when allowing a controller to access additional models through the $uses variable, the name of the current controller's model must also be included. This is illustrated in the example below. The Html and Form Helpers are always available by default. But if you choose to define your own $helpers array in AppController, make sure to include Html and Form if you want them still available by default in your own Controllers. The Session Helper and Component may be useful to manage sessions and state in your application. Let’s look at how to tell a CakePHP controller that you plan to use additional MVC classes. ---------- See comments for this section Page-related Attribute: $layout A few attributes exist in CakePHP controllers that give you control over how your view is set inside of a layout. The $layout attribute can be set to the name of a layout saved in /app/views/layouts. You specify a layout by setting $layout equal to the name of the layout file minus the . ctp extension. If this attribute has not been defined, CakePHP renders the default layout, default. ctp. If you haven’t defined one at /app/views/layouts/default. ctp, CakePHP’s core default layout will be rendered. ---------- lt;? php ---------- ---------- // Using $layout to define an alternate layout ---------- ---------- class

RecipesController extends AppController

{ ---------- function quickSave()

{ ---------- $this-> layout = 'ajax';

---------- } ---------- }

---------- ---------- ?>

See comments for this section The Parameters Attribute ($params) Controller parameters are available at $this-> params in your CakePHP controller. This variable is used to provide access to information about the current request. The most common usage of $this-> params is to get access to information that has been handed to the controller via POST or GET operations. See comments for this section form: $this-> params['form'] Any POST data from any form is stored here, including information also found in $_FILES. See comments for this section admin: $this-> params['admin']

Is set to 1 if the current action was invoked via admin routing. See comments for this section bare: $this-> params['bare'] Stores 1 if the current layout is empty, 0 if not. See comments for this section isAjax: $this-> params['isAjax'] Stores 1 if the current request is an ajax call, 0 if not. This variable is only set if the RequestHandler Component is being used in the controller. See comments for this section controller: $this-> params['controller'] Stores the name of the current controller handling the request. For example, if the URL /posts/view/1 was requested, $this-> params['controller'] would equal " posts".

See comments for this section action: $this-> params['action'] Stores the name of the current action handling the request. For example, if the URL

/posts/view/1 was requested, $this-> params['action'] would equal " view". See comments for this section pass: $this-> params['pass'] Returns an array (numerically indexed) of URL parameters after the Action. See comments for this section url: $this-> params['url'] Stores the current URL requested, along with key-value pairs of get variables. For example, if the URL /posts/view/? var1= 3&var2= 4 was called, $this-> params['url'] would contain: See comments for this section ata: $this-> data Used to handle POST data sent from the FormHelper forms to the controller.

---------- // The FormHelper is used to create a form element: ---------- echo $this-> Form-> text('User. first_name'); See comments for this section named $this-> params['named'] Stores any named parameters in the url query string in the form /key: value/. For example, if the URL /posts/view/var1: 3/var2: 4 was requested, $this-> params['named'] would be an array containing: See comments for this section ---------- Controller Methods . Interacting with Views Controllers interact with the view in a number of ways. First they are able to pass data to the views, using set(). You can also decide which view class to use, and which view file should be rendered from the controller. See comments for this section set: set(string $var, mixed $value) The set() method is the main way to send data from your controller to your view. Once you've used set(), the variable can be accessed in your view. ---------- set('color', 'pink'); ---------- ----------

//Then, in the view, you can utilize the data:

---------- ?> ----------

You have selected ;? php echo $color; ? ; icing for the cake. The set() method also takes an associative array as its first parameter. This can often be a quick way to assign a set of information to the view. See comments for this section render: render(string $action, string $layout, string $file) The render() method is automatically called at the end of each requested controller action. This method performs all the view logic (using the data you've given in using the set() method), places the view inside its layout and serves it back to the end user.

Although CakePHP will automatically call it (unless you've set $this-; autoRender to false) after every action's logic, you can use it to specify an alternate view file by specifying an action name in the controller using $action. See comments for this section Rendering a specific view In your controller you may want to render a different view than what would conventionally be done. You can do this by calling render() directly. Once you have called render() CakePHP will not try to re-render the view. ---------- lass PostsController extends AppController { ---------- function my_action() { ---------- $this-> render('custom_file'); ---------- } ---------- } See comments for this section Flow Control See comments for this section redirect: redirect(mixed $url, integer $status, boolean $exit) The flow control

method you’ll use most often is redirect(). This method takes its first parameter in the form of a CakePHP-relative URL.

When a user has successfully placed an order, you might wish to redirect them to a receipt screen. ---------- function placeOrder() { ----------

---------- //Logic for finalizing order goes here

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

if($success) { ---------- $this->

redirect(array('controller' => 'orders', 'action' => 'thanks'));

---------- else

{ ---------- $this-> redirect(array('controller' =>

'orders', 'action' => 'confirm')); ---------- }

---------- } The second parameter of redirect() allows you to define an HTTP status code to accompany the redirect. You may want to use 301 (moved permanently) or 303 (see other), depending on the nature of the redirect. The method will issue an exit() after the redirect unless you set the third parameter to false. See comments for this section flash lash(string $message, string $url, integer $pause, string $layout) Like redirect(), the flash() method is used to direct a user to a new page after an operation. The flash() method is different in that it shows a message before passing the user on to another URL. The first parameter should hold the message to be displayed, and the second parameter is a CakePHP-relative URL. CakePHP will display the $message for $pause seconds before forwarding the user on. If there's a particular template you'd like your

flashed message to use, you may specify the name of that layout in the $layout parameter. See comments for this section

Callbacks CakePHP controllers come fitted with callbacks you can use to insert logic just before or after controller actions are rendered. beforeFilter() This function is executed before every action in the controller. It's a handy place to check for an active session or inspect user permissions. beforeRender() Called after controller action logic, but before the view is rendered. This callback is not used often, but may be needed if you are calling render() manually before the end of a given action. afterFilter() Called after every controller action, and after rendering is complete. This is the last controller method to run.

CakePHP also supports callbacks related to scaffolding. _beforeScaffold($method) $method name of method called example index, edit, etc. _afterScaffoldSave($method) $method name of method called either edit or update. See comments for this section See comments for this section See comments for this section loadModel loadModel(string $modelClass, mixed $id) The loadModel function comes handy when you need to use a model which is not the controller's default model or its associated model. ---------- $this-> loadModel('Article'); ---------- recentArticles = $this-> Article-> find('all', array('limit' => 5, 'order' => 'Article. created DESC')); Components Components are packages of logic that are shared between controllers. If you find yourself wanting to copy and paste things between controllers, you might consider wrapping some functionality in a

component. CakePHP also comes with a fantastic set of core components you can use to aid in: * Security * Sessions * Access control lists * Emails * Cookies * Authentication * Request handling Each of these core components are detailed in their own chapters. For now, we'll show you how to create your own components.

Creating components keeps controller code clean and allows you to reuse code between projects. ---------- Creating Components Suppose our online application needs to perform a complex mathematical operation in many different parts of the application. We could create a component to house this shared logic for use in many different controllers. The first step is to create a new component file and class. Create the file in /app/controllers/components/math. php. The basic structure for the component would look something like this:

---------- 1. ---------- Including Components in your Controllers Once our component is finished, we can use it in the application's controllers by placing the component's name (minus the " Component" part) in the controller's $components array.

The controller will automatically be given a new attribute named after the component, through which we can access an instance of it:

---------- /* Make the new component available at $this-> Math, ---------- as well as the standard $this-> Session */ ---------- var $components = array('Math', 'Session'); 1. ---------- var

$components = array('Math', 'Session'); See comments for this section See comments for this section Using other Components in your Component

Sometimes one of your components may need to use another. You can include other components in your component the exact same way you include them in controllers: Use the $components var.

---------- Existing-> foo();

---------- } ----------

---------- function bar()

{ ---------- // ... ----------

} ---------- } ---------- ?

> ---------- Models

Models represent data and are used in CakePHP applications for data access. A model usually represents a database table but can be used to access anything that stores data such as files, LDAP records, iCal events, or rows in a CSV file. A model can be associated with other models. For example, a Recipe may be associated with the Author of the recipe as well as the Ingredient in the recipe. A Model represents your data model. In object-oriented programming a data model is an object that represents a " thing", like a car, a person, or a house. A blog, for example, may have many blog posts and each blog post may have many comments.

The Blog, Post, and Comment are all examples of models, each associated with another. This intermediate class, AppModel, is empty and if you haven't created your own is taken from within the /cake/ folder. Overriding the AppModel allows you to define functionality that should be made available to

all models within your application. To do so, you need to create your own app_model. php file that resides in the root of the /app/ folder. With your model defined, it can be accessed from within your Controller. CakePHP will automatically make the model available for access when its name matches that of the controller. ---------- Ingredient-> find('all'); ---------- $this-> set('ingredients', $ingredients); ---------- }

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

---------- ?> ----------

Retrieving Your Data from databases Find: find($type, $params) Find is the multifunctional workhorse of all model data-retrieval functions. $type can be either 'all', 'first', 'count', 'list', 'neighbors' or 'threaded'. The default find type is 'first'. $params is used to pass all parameters to the various finds, and has the following possible keys by default - all of which are optional:

---------- rray(----------

'conditions' => array('Model. field' => $thisValue), //array of conditions

---------- 'recursive' => 1, //int

---------- 'fields' => array('Model. field1', 'DISTINCT Model. field2'), //array of field names

---------- 'order' => array('Model. created', 'Model. field3 DESC'), //string or array defining order

---------- 'group' => array('Model. ield'), //fields to GROUP BY ---------- 'limit' => n, //int

---------- 'page' => n, //int

---------- 'offset'=> n, //int

---------- 'callbacks' => true //other possible values are false, 'before', 'after' ----------) 1. ---------- array(2. ---------- 'conditions' => array('Model. field' => $thisValue), //array of conditions 3. ---------- 'recursive' => 1, //int 4. ---------- 'fields' => array('Model. field1', 'DISTINCT Model. field2'), //array of field names 5. ---------- 'order' => array('Model. created', 'Model. field3 DESC'), //string or array defining order 6. ---------- 'group' => array('Model. field'), //fields to GROUP BY 7. ---------- 'limit' => n, //int 8. ---------- page' =; n, //int 9. ---------- 'offset'=> n, //int 10. ---------- 'callbacks' => true //other possible values are false, 'before', 'after' 11. ----------) See comments for this section find('first'): find('first', $params) 'first' is the default find type, and will return one result, you'd use this for any use where you expect only one result. If no results are found, false is returned. find('count'): find('count', $params) find('count', $params) returns an integer value.

There are no additional parameters used by find('count'). See comments for this section find('all'): find('all', $params) find('all') returns an array of (potentially multiple) results. It is in fact the mechanism used by all find() variants, as well as paginate. find('list'): find('list', $params) find('list', $params) returns an indexed array, useful for any use where you would want

a list such as for populating input select boxes. find('threaded'): find('threaded', $params) find('threaded', $params) returns a nested array, and is appropriate if you want to use the parent_id field of your model data to build nested results. ind('neighbors'): find('neighbors', $params) 'neighbors' will perform a find similar to 'first', but will return the row before and after the one you request. See comments for this section findBy findBy; fieldName;(string $value); The findBy magic functions also accept some optional parameters: findBy; fieldName;(string $value[, mixed $fields[, mixed $order]]); These magic functions can be used as a shortcut to search your tables by a certain field. Just add the name of the field (in CamelCase format) to the end of these functions, and supply the criteria for that field as the first parameter.

PHP5 findBy; x; Example | Corresponding SQL Fragment | $this-; Product-; findByOrderStatus(‘ 3’); | Product. order_status = 3 | $this-; Recipe-; findByType(‘ Cookie’); | Recipe. type = ‘ Cookie’ | $this-; User-; findByLastName(‘ Anderson’); | User. last_name = ‘ Anderson’ | $this-; Cake-; findById(7); | Cake. id = 7 | $this-; User-; findByUserName(‘ psychic’); | User. user_name = ‘ psychic’ | See comments for this section query query(string $query) SQL calls that you can't or don't want to make via other model methods (this should only rarely be necessary) can be made using the model's query() method.

See comments for this section field: field(string $name, array $conditions = null, string $order = null) Returns the value of a single field, specified as $name, from the first record matched by $conditions as ordered by $order. If

no conditions are passed and the model id is set, will return the field value for the current model result. If no matching record is found returns false. read(): read($fields, $id) read() is a method used to set the current model data (Model::$data)--such as during edits--but it can also be used in other circumstances to retrieve a single record from the database. fields is used to pass a single field name, as a string, or an array of field names; if left empty, all fields will be fetched. $id specifies the ID of the record to be read. By default, the currently selected record, as specified by Model::$id, is used. Passing a different value to $id will cause that record to be selected. ---------- Behaviors Model behaviors are a way to organize some of the functionality defined in CakePHP models. They allow us to separate logic that may not be directly related to a model, but needs to be there.

By providing a simple yet powerful way to extend models, behaviors allow us to attach functionality to models by defining a simple class variable. That's how behaviors allow models to get rid of all the extra weight that might not be part of the business contract they are modeling, or that is also needed in different models and can then be extrapolated. As an example, consider a model that gives us access to a database table which stores structural information about a tree. Removing, adding, and migrating nodes in the tree is not as simple as deleting, inserting, and editing rows in the table.

Many records may need to be updated as things move around. Rather than creating those tree-manipulation methods on a per model basis (for every model that needs that functionality), we could simply tell our model to use

the TreeBehavior, or in more formal terms, we tell our model to behave as a Tree. This is known as attaching a behavior to a model. With just one line of code, our CakePHP model takes on a whole new set of methods that allow it to interact with the underlying structure. ---------- Views Views are the V in MVC.

Views are responsible for generating the specific output required for the request. Often this is in the form of HTML, XML, or JSON, but streaming files and creating PDF's that users can download are also responsibilities of the View Layer. ---------- View Templates The view layer of CakePHP is how you speak to your users. Most of the time your views will be showing (X)HTML documents to browsers, but you might also need to serve AMF data to a Flash object, reply to a remote application via SOAP, or output a CSV file for a user. CakePHP view files are written in plain PHP and have a default extension of . tp (CakePHP Template). These files contain all the presentational logic needed to get the data it received from the controller in a format that is ready for the audience you’re serving to. View files are stored in /app/views/, in a folder named after the controller that uses the files, and named after the action it corresponds to. For example, the view file for the Products controller's " view()" action, would normally be found in /app/views/products/view. ctp. * layouts: view files that contain presentational code that is found wrapping many interfaces in your application. Most views are rendered inside of a layout. * elements: maller, reusable bits of view code. Elements are usually rendered inside of views. * helpers: these classes encapsulate view logic that is needed in many places in the view layer. Among other things, helpers in CakePHP can help you build

forms, build AJAX functionality, paginate model data, or serve RSS feeds. Caching Elements You can take advantage of CakePHP view caching if you supply a cache parameter. If set to true, it will cache for 1 day. Otherwise, you can set alternative expiration times. If you render the same element more than once in a view and have caching enabled be sure to set the 'key' parameter to a different name each time.

This will prevent each succesive call from overwriting the previous element() call's cached result. E. g. ---------- element('helpbox', array('cache' => array('key' => 'first_use', 'time' => '+1 day'), 'var' => $var)); ---------- ---------- echo $this-> element('helpbox', array('cache' => array('key' => 'second_use', 'time' => '+1 day'), 'var' => $differentVar)); ---------- > See comments for this section Requesting Elements from a Plugin If you are using a plugin and wish to use elements from within the plugin, just specify the plugin parameter. If the view is being rendered for a plugin controller/action, it will automatically point to the element for the plugin. If the element doesn't exist in the p