

Yii2

For Beginners

PHP Web Application Development

A step by step guide learning Yii 2
from setup to coding fundamentals

Bill Keck

Yii 2 For Beginners

A step by step guide to learning Yii 2 for beginners

Bill Keck

This book is for sale at <http://leanpub.com/yii2forbeginners>

This version was published on 2015-05-28



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 - 2015 Bill Keck

Tweet This Book!

Please help Bill Keck by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

I just bought Yii 2 For Beginners by Bill Keck

The suggested hashtag for this book is [#Yii2](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

<https://twitter.com/search?q=#Yii2>

This book is dedicated to anyone who has taken the time to help someone else learn something about programming, whether it is through an online article, blog, community forum, or book. As I have benefited from the help of others, so too, I wish to contribute...

I would also like to thank the core Yii 2 team, starting with Yii founder Qiang Xue, who, with his creation of Yii 2, has brought something beautiful and useful into the world. Also, many thanks to the tireless efforts of Samdark, Cebe, Orey, and Kartik, who take the time to help people like me, and others like me, who have a passion for programming, but sometimes need a little help filling in the details.

And finally, thank you to my wife and kids, just because without them, I would not endeavor to reach so high, even if I'm only able to grasp so little. In the end, it's the dream of accomplishing something that drives me, and it is only with their love and support that I'm able to do this at all.

Contents

Chapter One: Introduction	1
Introduction	1
Features	1
What Makes The Yii 2 Framework Special?	2
Upsides	3
Downsides	3
Why I chose Yii 2	3
Other Options	4
Yii 2 Arrives	4
Gii	5
DB-First Approach	5
MySQL	5
Improved Workflow	6
Minimum PHP Skills	6
Tools You Will Need	7
Errata	9
Contact Bill Keck	10
Summary	10
Chapter Two: Yii 2 Advanced Template Installation	12
Quick Setup of Yii2 Advanced Template	12
Step 1 - Create Folder	13
Step 2 - Apache Conf	13
Step 3 - Local Host	14
Step 4 - Restart Apache	15
Step 5 - Create Project in IDE	15
Step 6 - Find Command Line Path	18
Step 7 - Composer Self-Update	18
Step 8 - Install Yii 2	19
Step 9 - Check For Yii 2 Folder	20
Step 10 - Run Php Init	20
Step 11 - Create The Database	21
Step 12 - Set DB Connection	21

CONTENTS

Step 13 - Run Migration	22
Step 14 - Create Git Repository	23
Step 15 - Confirm App Is Working	24
Trouble-Shooting	24
Summary	25
Chapter Three: Welcome to the MVC	26
MVC Pattern	26
Index.php	28
The Application Instance	29
Routing	29
Using Gii	30
Bootstrap	30
Debugger	31
Summary	32
Chapter Four: Modifying the User Model	34
Role and Status	35
The User Model	37
Properties of the Model	45
Constants	46
Identity Interface	46
Behaviors	49
Rules	50
Identity Methods	51
Boilerplate Methods	54
Other Models Accessing User	56
SignupForm Model	56
Summary	60
Chapter Five: Creating New Models with Gii	61
Creating Tables	61
Role Table	62
Status Table	63
User Type Table	63
Gender Table	64
Profile Table	64
Synchronize	65
Configuring Gii	66
Making Models with Gii	68
Create Role Model	69
Add Records To Role Table	72
Add Relationship To Role	73

CONTENTS

Update User Model with Role	74
Create Status Model	76
Update User Model with getStatus	78
Add Records to Status Table	80
Create UserType Model	81
Update User Model with UserType	83
Add Records to user_type Table	85
Create Gender Model	85
Add Records to gender Table	87
Create Profile Model	88
The Complete Profile Model	96
Update User Model with Profile	101
Finish Up User Model	102
The Complete User Model	104
Summary	116
Chapter Six: Helpers	117
Value Helpers	117
Permission Helpers	127
Record Helpers	132
Summary	134
Chapter Seven: Site Controller	135
Behaviors	136
Actions	137
Index Action	138
Login Action	139
Login Form Model	139
Logout Action	143
Contact Action	143
Contact Form Model	144
Captcha	145
Contact View Form	148
About Action	152
Signup Action	152
Signup Form Model	153
ResetPasswordForm Model	159
Backend Site Controller	162
Beginning Access Control	164
loginAdmin Method	165
Summary	167
Chapter Eight: Profile Crud	169

CONTENTS

CRUD	169
Profile Controller	171
Profile Search	171
_search	171
_form	171
Index	171
View	172
Create	172
Update	173
Modifying Profile Controller & Views	173
Modifying the Profile Controller	174
Index Action	176
View Action	178
Create Action	180
Update Action	182
Delete Action	184
FindModel Action	185
Modifying the Profile Views	186
View.php	186
Gender	192
Form Partial	193
Create	196
Update	196
Site Layout	197
Profile Link	201
DatePicker	202
Summary	204
Chapter Nine: Upgrade and Access Control	206
Upgrade Controller	207
Upgrade View	208
Require Upgrade To	208
Access Control	209
Passing A Variable From the Controller	214
Summary	217
Chapter Ten: Homepage Social Widgets	218
Implementing Homepage Social Widgets	218
Index	218
Facebook Widget	219
Facebook App Setup	220
Facebook Configuration	226
Extensions	228

CONTENTS

HTML Helper	229
Collapse Widget	235
Modal Widget	237
Alert Widget	237
Font-Awesome	238
Asset Bundle	239
Add Font-Awesome to Layout	241
Summary	248
Chapter Eleven: Backend Creation	249
Main.php	253
Updating Backend Views	258
backend/views/profile/_form.php	258
backend/views/profile/view.php	258
backend/views/user/view.php	263
backend/views/user/_form	266
Deeper Changes to Backend	267
backend/views/user/index.php	268
backend/views/profile/index.php	270
backend/views/profile/_search.php	272
backend/views/user/_search.php	273
User Search	275
Admin UI	296
Controller Behaviors	302
Match Callback	303
Summary	306
About The Author	307
Chapter 12: Bonus Material	308
AutoResponder	308
Dropdown Navigation	325
FAQ	333
Test Controller	373
Components	374
Creating a Custom Widget	379
CDN	399
Summary	402
Chapter 13: Bonus Material PrettyUrls & Slugs	404
Pretty URLs	404
Apache Vhost	405
Restart Apache	406
.htaccess	406

CONTENTS

Slugs	408
Sluggable Behavior	408
Slug Column	409
Drop old Faqs and Create New Ones	410
Add Url Manager Rules	411
Modify View Action on FaqController	411
Modify Create and Update Actions on Backend Controller	413
Change Gridview Action Column URL	414
Summary	421
Chapter 14: Bonus Material Social Login and Register	423
Yii2 - AuthClient	423
Install yii2authclient via Composer	424
Configuration	424
Twitter Issue	425
Provider Applications	426
Facebook App	426
Github App	427
Google App	429
LinkedIn App	435
Index View Change	437
Login View Change	439
Signup View Change	440
Social Sync Dropdown	441
Auth Data Structure	443
Auth Model	444
Site Controller Actions Method	445
OnAuth Success	445
Updated OnAuth Success	447
Login and Registration Scenarios	451
Refactor For Maintainability and Extensibility	461
New Class Properties	461
New Helper Methods	465
OnAuthSuccess Method	470
Action Login	474
Action Signup	476
Pages Controller	482
Summary	485
Chapter 15: Template Migration guide	486
User	487
Role	487
UserType:	487

CONTENTS

Status	487
ValueHelpers	487
PermissionHelpers	488
RecordHelpers	488
Database Changes	488
Extra ValueHelpers	488
LoginForm Model	489
PasswordResetRequestForm	489
UserSearch	490
ProfileSearch	490
Main.php	490
Index.php	490
Troubleshooting	490
Summary	491
Chapter 16: Images and File Uploads	492
The Uploads Folder	492
Marketing Image Table	492
Marketing Image SQL	493
Marketing Image Model	494
Modify MarketingImage Model	494
PHP FileInfo	500
Modify MarketingImage Search Model	503
Modify Index View	506
Modify View	509
Modify Update View	511
Modify _search Partial	512
Modify _form Partial	513
Modifying the Controller	515
The Create Action	521
The Update Action	522
The Delete Action	524
Image Thumbnails with Imagine	525
Install Yii 2 Imagine Extension	525
Create Thumbnail Folder	525
Alter Marketing Image Table	525
Modify MarketingImage Model	526
Scenarios	532
Modify MarketingImageSearch	533
Modifying Marketing Image Controller	536
Modify Create Action	543
Modify Update Action	544
Modify Delete Action	545

CONTENTS

Modify Views	547
Modify View	547
Modify Update	549
Modify _form	550
Modify _search	550
Modify Index	550
URL Manager	553
Carousel Widget	554
CarouselWidget.php	555
carousel.php	563
Pages Index	568
Carousel Settings	574
carousel_settings table	574
CarouselSettings Model	576
CarouselSettingsSearch Model	580
CarouselSettingsController	584
CarouselSettings _form View	588
CarouselSettings view.php View	589
CarouselSettings index.php View	591
CarouselSettings _search View	592
PagesController	594
Pages Index View	597
CarouselWidget	598
carousel.php	602
Main	606
Summary	610
Chapter 17: Bonus Material Ratings Widget	612
FaqRatings Model	615
\$model->getErrors()	619
Overwrite Save Method on Model	620
Faq Controller	622
Frontend Faq view.php	623
_rating-form.php	628
FaqRatings Controller	630
Faq Model	638
Faq Index View	638
Faq View	639
Signup Form Model	640
signup.php	643
terms.php	645
termsoverflow.css	647
Frontend AppAsset.php	648

CONTENTS

Improving The Carousel	649
Modify marketing_imageTable	650
Modify MarketingImage Model	650
MarketingImage Views	652
Marketing Image view.php	652
Update View	654
MarketingImage Controller	656
Create Action	656
Update Action	658
Delete Action	660
Entire File	661
CarouselSettings Model Rules	661
CarouselWidget	662
validateSize Method	663
Entire CarouselWidget File	663
carousel.php	663
Summary	671
Chapter 18: Bonus Material Returning Calculated Values in Gridview	673
Sorting A Calculated Value In Gridview	673
Donate To Kartik	675
Average Rating For Gridview	676
Times Rated	679
Summary	681

Chapter One: Introduction

Introduction

Welcome to Yii 2 For Beginners. This book will take you step by step through setup and installation, and then onto coding in the most exciting PHP framework available today, Yii 2.

Yii 2 comes in two flavors, basic and advanced, and it might seem counterintuitive to use the advanced template in a beginner's book, but ironically, the advanced template is easier to use if your application requires a working user model that stores users in a database. Most modern web applications will need this functionality and the advanced template has a ready-made solution for that.

The other big benefit to the advanced template is that it divides the application between frontend and backend, which answers the inevitable question of "where do I put my admin area?"

Not only do I want to introduce you to this amazing php framework, but I also want you have a starting point for your projects that includes everything you need to build a robust database-driven web application. While the out-of-the-box advanced template is extremely helpful, it is missing some key pieces, which we will fill in with this book.

The goal is to provide you with a boilerplate template that you can use for all your future projects.

Features

Some of the features you get with the install of the advanced template include:

- pre-defined schema for the user table
- user login and registration forms
- forgot password functionality
- separate frontend and backend domains
- automatic code generation for models, controllers, and views
- built-in integration with Twitter Bootstrap and mobile first design
- robust widgets and helpers for data presentation

If you don't understand something in that list, don't worry, we will be covering it in detail. Just know that it really is amazing what Yii 2 does for you. But no matter how great a framework is, you still need to do more to make it support a real application.

So to all the out-of-the-box features, will be adding:

- nice frontend ui refinements like jquery datepicker
- model relation methods that make displaying related data easy
- controller methods to restrict who sees what
- extended data structure that will be common to all your future sites
- role-based access control (RBAC)
- restricting content to user based on user type, such as free or paid
- social modules that allow for sharing
- one-click Facebook signup and login

These are all things that your web application is likely to require, regardless of the type of site it is. So, as you learn Yii 2 with this book, you will be building a template that you can expand on for all of your future applications.

This book is perfect for beginning PHP programmers who are ready to move on to framework development. The Yii 2 PHP framework is highly scalable and extensible, and loaded with features. We introduce you to this wonderful framework and explain in detail everything you need to know to get up and running. You will love Yii 2!

Advanced Php artisans will be able to zip through this book and get up and running quickly on Yii 2, a phenomenal php framework. This will not only save them time on projects, but also fully leverage the benefits of an open-source framework that has an entire community behind it.

The main style of this book, however, is for beginners. There's a lot of granular detail to help people who have some experience with PHP but have not really jumped up to advanced object-oriented programming yet.

We try to make sure we fully understand how the framework works, how it uses OOP to create an intuitive development layer that allows many different level programmers to achieve the results they are striving for.

In any case, learning Yii 2 gives you hands on experience with object-oriented programming with practical results. You end up with a working website.

What Makes The Yii 2 Framework Special?

Programmers have to make decisions, it's a fact that is at the heart of what programming is. So one of the biggest decisions you will have to make as a programmer, and more likely, a decision you will make as part of a team of programmers, is whether or not to use a framework and if so, which one.

As to the question of using a PHP framework, there are so many benefits to doing so, it becomes a no-brainer.

Upsides

Here are some of the obvious benefits:

- Uses standard ways of doing things, so reduces or eliminates spaghetti code.
- Reduces time spent on plumbing tasks such as form validation and security.
- Makes it easier to work as a team by enforcing standards.
- Makes it easier to maintain code by utilizing a common architecture and methods.
- You get the benefit of an active community of developers who maintain the framework and support common tasks and new features.

Downsides

There are a couple of downsides to using a framework that should be pointed out. First, all the code that comprises the framework creates server overhead and this can be a real problem. Luckily there are caching options available which will reduce the effects of this, and for enterprise applications, you can use raw sql to minimize query time. So don't let the server overhead stop you from using a framework.

The other thing is that obviously when you are working with the framework, you are working with a vast amount of code that you didn't write and it takes time to figure out how it works. Some of the framework code can be quite cryptic depending on your level of skill and experience, so don't expect to instantly understand everything. It's not going to happen.

Of course you already knew that there was a learning curve, which is why you are reading this book. And while it takes time to learn someone else's code, which can be a pain, it would be far more painful to have to write a custom framework from scratch. All things considered, using a framework for enterprise development is a wise choice.

Ok, so the easy part is to figure out that utilizing a framework will help you develop a more organized and robust project, but now comes the hard part. You have decide which framework to use.

Why I chose Yii 2

I can't tell you which is the best framework for you, that is something for you and you alone to decide, but I can share a little of the journey that led me to Yii 2. This decision wasn't driven by the need to find the easiest way to learn PHP, that is for sure.

At my company back in 2012, I was part of team of developers who looked at various frameworks and had to decide which one to use. I never dreamed at the time that I would end up writing a book about one of them.

Anyway, we collectively researched everything we could find on the major PHP frameworks. I personally read all of the documentation and we had long engineering discussions about what we thought would work. You can't imagine my frustration with the fact that I read all this documentation and walked away from it feeling less knowledgeable than before I started reading it.

Our team of programmers did have a preference however. They felt that Yii 1.1.14 was the best choice. This was the version of Yii that was available at the time we were deciding this. So the team adopted that framework and never looked back. They loved it.

I, on the other hand, remained frustrated. Since I was only a novice programmer, I really struggled to learn it. I didn't find it very intuitive. Especially after comparing it to other frameworks, where they were trying so hard to make everything integrate beautifully, the architecture of Yii just seemed ugly.

I got so frustrated at one point, that I started looking for another option.

Other Options

I would find some beautifully written documentation for a new framework and run it past the team. I always got the same response. The team was happy with Yii.

They told me it might be difficult to learn, but it was easy to use, once you knew how it worked. Because of that, I committed myself to learning it. It was slow going and a rough ride. I wasn't getting it. I was working through chapter 10 in a book on Yii 1.1.14, thinking I would never really be able to build an application on my own in less than a hundred years. Too many roads seemed to go nowhere.

Then a miracle happened.

Yii 2 Arrives

I found out about the Yii 2 alpha. I was curious to see what the differences were in Yii 2, which had been 3 years in the making at that point. So I jumped in and to my utter and complete surprise, I instantly connected with it. I understood the structures. I could write code that actually worked! What a great feeling that was.

I have personally found Yii 2 to be the most intuitive and elegant of all the PHP frameworks that I have studied. I have so much enthusiasm for it that I want to share it with every programmer I know, and even those I don't know, so it has motivated me to write this book.

With Yii 2, even as a beginner, I was able to stand up a working website that has a data-driven user model, with both a frontend and a backend. Right out of the box, I get a working user model, with forgot password functionality, which is also integrated with Bootstrap for mobile-responsive design, without having to do any programming whatsoever. How cool is that?

Although I was a beginning programmer when I was studying the PHP frameworks, I did have experience working with databases and this is one area in my opinion where Yii 2 really shines.

Gii

Yii 2 has a code generation tool called Gii. I pronounce that with a soft “g,” but I have no idea if that is the right way to say it or not.

Anyway, Gii analyzes your database tables and automatically builds PHP models from them. Not only that, but it analyzes the relationships between tables and automatically generates the relational code into the models. For example, if you had a data structure with 30 tables, and half of them had a user_id column that was meant to reference id of the user table, Gii would build the appropriate relationships for you, each time you built a model. Not only is this a time-saver, but this also gives you very consistent code because it is always done the same way and it helps you adopt this discipline.

It’s worth mentioning that other frameworks work exactly the opposite way. With them, you build the model first, then do a migration to the database to create the table and corresponding columns. So the big difference is that you are building your data structure piecemeal as you go along, whereas in Yii 2 you have the option of having a more complete data structure to begin with.

Both approaches work, however they represent drastically different workflows. In my opinion, the migration/piecemeal approach to data structure only really works for a single developer or a very small team working on a small project. The reason why I say this is that although democracy is probably the best system politically, imagine a world where each developer makes up their own data structure and implements it. How consistent would that be? What if the right hand didn’t know what the left hand was doing? In larger teams, this is a recipe for chaos. This is why enterprise development teams usually have a database administrator, also known as the DB, and only they can create or delete data structure.

DB-First Approach

Since Yii 2 allows you to essentially import the models from the data structure, you can start your project by really thinking through your data structure. Overall I like to avoid talking about too much theory because the time is better spent working through hands-on examples, but I think it’s worth taking a moment to think about what a well thought-out data structure really means.

Whether you are a single developer or part of an enterprise level team, you are essentially being given the same task, the same overall mission. You have to serve data from a database into a browser-friendly format, typically using PHP, HTML, and Javascript. We use a PHP framework to make this task easier, and by saying that, we are admitting upfront that it’s not an easy task. Why is that?

The database is a very reliable and consistent piece of software, which allows us to create a relational data structure.

MySql

Throughout this book we use Mysql as the database, which, in addition to being free, is capable of powering enterprise data for web applications.

Because of the structure of the database, with its indexes and primary keys, a database can serve data very efficiently. In the simplest terms, this means it is very fast. It's also very deep. It can hold millions of records, which can be retrieved, if structured properly, in milliseconds.

Another key aspect of the database is that it allows us to structure the data in such a way as to connect things like the user's address and their username as if they were one record, but hold them in separate tables as separate records. The more you can break down the data structure into discrete components like that, the more powerful it is. This is called normalization of data.

The problem is that the more refined the database is, the more effectively normalized it is, the more complex it is to deal with in PHP. You end up having to connect a lot of PHP models together to represent the data correctly.

Now this might be getting too heavy on theory for a beginning book, so we won't take this much further for now, but the point is to understand the nature of the problem that the framework helps to solve. The easier it is for you to connect the models via the framework, the more power you derive from your database.

Improved Workflow

In my opinion, Yii 2 stands alone in how it helps you connect the models to the database, leading to improved workflow, efficiency, and overall design capabilities. It frees you to build a detail-rich data structure that will ultimately result in the end user being more engaged. I believe that Yii 2 does this more efficiently and deeply than any of the other PHP frameworks, that is why I'm so committed to it and so interested in sharing it.

Minimum PHP Skills

Learning a PHP framework is simple as long as you are a skilled PHP programmer. Quite often PHP is a second or third language for a programmer who is already proficient in an object oriented language like C or Java, so learning PHP is just a matter of adjusting syntax and they pick it up quickly. That's great for ninjas, but what happens if you are just learning your first programming language?

As someone who learned PHP as a first programming language, I can tell you from direct experience that it's difficult to move from beginner to advanced because there is not a lot of support for the middle ground. That fact is one of the motivations for me to write this book.

Anyway, if you search online, you either find incredibly complex examples involving multiple interfaces with nested objects or examples that are so rudimentary that, while they are easy to understand, they do nothing to advance your abilities.

To work your way through this book, you will need a decent understanding of object-oriented PHP. You can get this from a variety of sources online. I got my first taste of PHP from thenewboston.org,

which has 200 videos on PHP. Great for an introduction, but not much more. I followed that up with a quick read of Richard Reese's book on Java, which helped me understand object-oriented programming better, since everything in Java involves a class. Also, when I looked at PHP again, it seemed simpler. I also went through the basics at:

[W3 Schools](#)

W3schools.com is a great learning resource. You can play with the code online at that site.

And then of course there is [Php.net](#) itself which is where we find all the docs for the language and sometimes very complicated examples. I learned a lot there and got lost a lot too, that's the way it goes. Try it, you'll see what I mean.

At any rate, to be able to work with Yii 2, you should understand the basics about objects, arrays, and control structures like foreach loops. You should know the components of a class, properties and methods, etc. Take a look at:

[OOP for Beginners](#)

You should be able to get through that tutorial very easily. If not, go back and study it before trying to tackle Yii 2. Also, Yii 2 uses PHP 5.4 and above, which supports new array syntax and namespaces, both of which will be utilized extensively.

If you are light on programming experience, but full of enthusiasm, you should do well, as long as you are willing to do the work and are patient. At any point, if you don't understand something, you can stop and take the time to research it on [Google](#) or [stackoverflow](#) or [PHP.net](#). PHP is a well-documented and well-supported language, used by countless programmers who will try to help you.

Also, I took a lot of care to label the sub-sections of this book, so you can easily find what you are looking for, if you need to refer back to it. Many times you will want to return to a section to reference something and I've done my best to make that as intuitive as possible.

Tools You Will Need

There are a number of tools that I recommend that you use for development in Yii 2, all which, like the framework itself, are free. These are recommendations only, not necessary to follow exactly, but my instructions will assume you are using them. So if you are advanced enough, you can use whatever you wish, no big deal. As long as you have working development environment, you are fine. If not, try to use these exact tools, it will be easier for you in the long run.

Sometimes the most difficult part of a project is setting up the development environment. Because I use it personally, we will be using xampp up on a windows machine. Xampp includes PHP, Mysql, Apache, and PhpMyadmin, so it's perfect to create a development environment for project. It's also free.

[Download Xampp](#)

[Install Video for Xampp and PhpMyadmin](#)

Any alternative that will let you run those programs is fine, you do not need Xampp to follow this book. On the other hand, it's pretty easy to get up and running with Xampp, one reason why I use it. The tricky part is setting up environment variables on a Windows machine, but that is well documented and I have provided download and setup links for your convenience, so you can check those out if you need to.

Even though everything for Mysql can be done in PhpMyadmin, I also recommend setting up Mysql workbench. Workbench's EER (Enhanced Entity Relationship) Diagrams help you see the relationships and make creating tables and foreign keys a snap. We will use photos of Mysql Workbench to show you table structure later in the book.

[Download Mysql Workbench](#)

You should familiarize yourself with how to create a database, how to sync a diagram model to a database, and obviously how to create tables and columns. To build a database-driven application, you need a basic understanding of sql, nothing too deep, but you should know how basic queries work and the concept of joining tables for queries. And since we use MySQL, you need to be familiar with it. If any of that is new to you, the good news is that you can google up some tutorials and find everything you need for free. [W3 Resources MySQL tutorial](#) are a great reference.

For my IDE, I use PhpED. IDE stands for Integrated Development Environment, and helps you organize projects and code. Most developers use some form of IDE as opposed to just a text editor. I'm recommending Eclipse or Netbeans for this project, however, because both are free whereas PhpEd is a paid IDE. In order to install Eclipse, you will have to install the Java sdk first.

[Download Eclipse](#)

[Download Netbeans](#)

You will also need to install Composer, which you should do after installing xampp, which means after PHP is installed. In order to run Composer, you must first enable curl in your PHP build. You will also need to set an environment variable for it if you are using windows.

[Download composer](#)

[Enable Curl](#)

I also recommend using git, which provides version control. Version control is a handy way of saving your work so you can step backward easily if you need to. When you are dealing with a large number of files that are constantly being updated, this is a great help. Git also protects you in a team environment from someone overwriting your work because you can simply step back to a previous version.

[Download Git](#)

Lastly, I recommend console2 for Windows users, which is a command line tool that is a little prettier than the standard windows prompt. This makes it easier on the eyes and just a little easier to work with.

[Download Console 2](#)

In order to get your development environment working with Yii 2, you will need to add both a vhost entry into Apache and a local host entry into your hosts file. We will go through each step for that in detail.

Like I said earlier, if you prefer to use different tools or, for example, a linux machine for development, that is your choice.

I provided links and reference pages for installation, but for beginners, this may prove to be difficult. You can use the installation of the development environment as one of the tests to see if you are ready to tackle Yii 2. Just don't give up easily. If it doesn't go well, you can always get help from a more experienced programmer.

Tip

Also, and this is a tip for beginners, almost everything you will go through as a programmer has been gone through by other programmers before you and this is especially true for configuration errors. Don't be afraid to use [Google](#) for help in troubleshooting setup. You will end up using it more often than not.

Once you've got everything up and running, spend a little time learning your way around the tools. It will make your efforts developing in Yii 2 go a lot smoother.

Errata

Although I have poured over every line of code in this book at least a hundred times and built the examples from scratch twice just to make sure I could follow the directions, mistakes are bound to happen, such is the nature of technical writing. I am actively updating errata as I go, so I do hope to be able to quickly correct any errata I am made aware of. You can help by emailing me if you find something, everyone will appreciate it.

Formatting Tip

In certain cases, I had to format my code using two lines where one would be appropriate, in order to avoid line breaks from the wordwrapping in PDF and other formats. The wordwrapping in PDF causes special characters to appear, which break the code, so I had to avoid that the best I could. As a result, I'm not recommending you follow the code examples as an example of style. I would recommend following the PSR-2 Guide, available here: [PSR-2 Coding Style Guide](#)

You can format your code with a formatter at [Php Formatter](#), if you want to make it more readable. Obviously be careful not to break the code. I will also be supplying Gists for each block of code that we write in the book, where the block of code exceeds 3 lines. If you don't know what a Gist is, don't worry, we will cover it in detail later.

Contact Bill Keck

LeanPub provides a contact link at the bottom of the book's landing page, so feel free to contact me there:

[Email Bill Keck](#)

You can also leave a comment for me at my Yii 2 blog:

[Bill Keck's Yii 2 Blog](#)

My blog is also a good source of the latest news about Yii 2, PHP trends, tutorials, and a few random thoughts I decide to blog about on occasion. Please feel free to leave a comment.

Please note the purchase price of this book does not include technical support.

The fastest way to overcome errors is to Google it, most likely someone has come across the same problem. Also, please keep in mind that Yii 2 is continually being developed and new versions may not support the code offered in this book. This is not unusual for programming books.

I'm going to do my best to stay on top of that, but there can be times when Yii 2 has made a version change that I have not accounted for yet. Once I'm aware of an issue, I can typically fix it quickly, so please do your part and notify me if you notice a versioning problem.

I will mention it numerous times and in numerous places that updates to this book are available to you for free for the life of the book. I plan to add bonus material on regular basis, so please take advantage of that. Simply login to your leanpub.com account to get the latest version.

Update notices for major updates go out by email. Minor updates simply get published, and these will typically just cover a typo.

To see if you have the latest version, you can go to the leanpub.com landing page for the book and look at the last updated date, which will show you when my last commit was.

Also, beginners will face a high volume of error messages due to typos and missed code. It's perfectly fine and part of the learning process. You will learn more from troubleshooting bugs than you will from simply copying and pasting code.

In most cases, you will find the answers to your problems if you are patient. Yii 2 forums are an excellent source of support and there are many great programmers that will help you. Always do your best to try to solve the problem first because it would be foolish to tie up a programmer's time with support requests over typos. Nevertheless, that is bound to happen. Just remember to be polite and considerate of others and you will do great.

Summary

I know it can be a little intimidating at first, especially when you realize that Yii 2 is not just some trivial set of library files that you can master in a few days, but hang in there and be patient. We are going to tackle it one step at a time.

So let me conclude the introduction with the following thought. Learning Yii 2 will come easy for some people and they are very lucky. If you are in the other camp, the ones that have to work hard to learn it, I can tell you that I know exactly how you feel. It was hard for me too. But I can also tell you that you can be optimistic. You can do this. Just stick with it and move at your own pace. And soon you will be amazed at how you are using Yii 2 to power your applications and you will be even more amazed at what you can create with it.

Chapter Two: Yii 2 Advanced Template Installation

Quick Setup of Yii2 Advanced Template

Ok, let's jump in! We are going to use yii2build as the root directory and name of project. We will be developing and hosting on a Windows machine with xampp installed and we are using PhpED as our IDE. If you want to use a free IDE, Netbeans is popular as is Eclipse. Google or see Chapter 1 for links and download for free.

At this point, we will assume you have your development environment setup and tested, and that you have spent some time familiarizing yourself with how the tools work. you need:

- Eclipse or Netbeans or some other IDE
- Composer
- xampp or some other apache, php, mysql environment
- Mysql Workbench
- PhpMyAdmin (included with xampp)
- console 2 (optional)
- GIT or some other version control

See chapter 1 for links to free downloads on the above tools, if you have not already installed them.

If you are not at this stage, you need to go back to the introduction and make sure you have all the required tools installed.

Hate Windows or Xampp? Not a problem. Obviously, you do not need to follow on Windows to read this book. If you are working directly on a LAMP stack or something else, you just need to know the linux commands. I don't provide them here, but you can easily google them. Just to reiterate, these instructions are xampp on Windows, but there are only minor differences, so you should be able to figure it out if you are using a different system.

For your convenience, I'm also listing a link to the Yii 2 guide for Advanced App installation:

[Yii 2 Advanced App Setup](#)

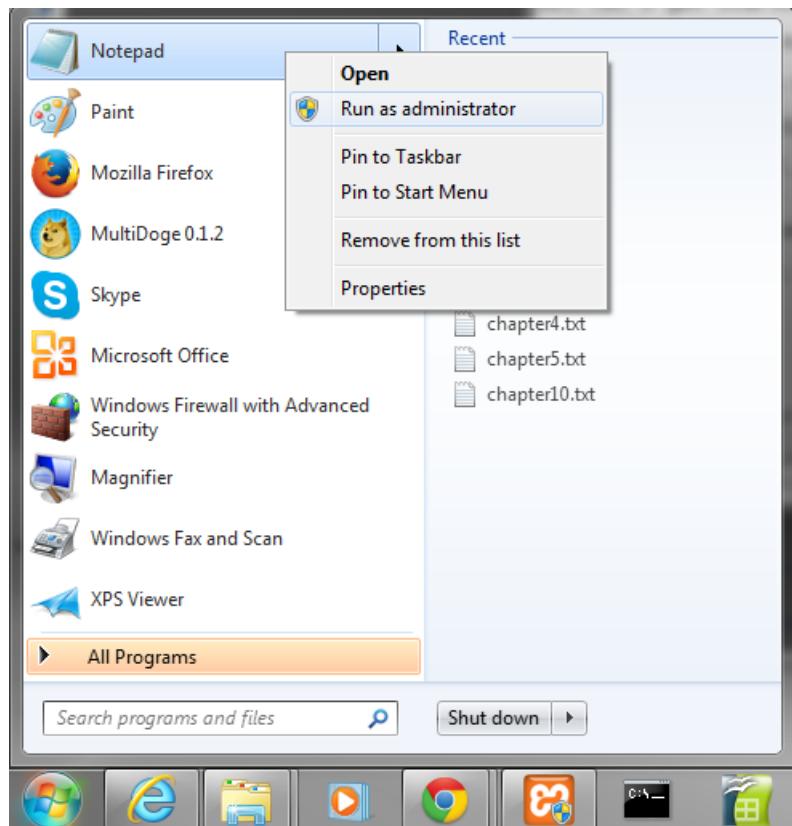
Ok, let's get started:

Step 1 - Create Folder

Go to the directory that stores the project roots, in my case it's \var\www and create a new folder named yii2build. So you should have a \var\www\yii2build folder.

Step 2 - Apache Conf

Set up apache conf. From a windows machine we will edit this from notepad, running in administrator mode. Find notepad from start button on task bar. Right click and select run in administrator mode.

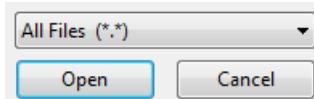


Notepad in Administrator Mode

Notepad will open. Select file open and the path to vhosts, in my case:

C:\xampp\apache\conf\extra\

Select all Files for file types:



Notepad All File Types

Then select:

httpd-vhosts.conf

Add the following entry to the file:

```
NameVirtualHost *
<VirtualHost yii2build.com>
    DocumentRoot "C:\var\www\yii2build\frontend\web"
    ServerName localhost
    ServerAlias www.yii2build.com
</VirtualHost>
NameVirtualHost *
<VirtualHost yii2build.com>
    DocumentRoot "C:\var\www\yii2build\backend\web"
    ServerName localhost
    ServerAlias backend.yii2build.com
</VirtualHost>
```

Please note that c:\var\www is where I store my project folder. If you are putting it in a different folder, c:\xampp\htdocs for example, you need to use that instead in the above host entries.



Trouble-shooting Tip

Make sure the line:

Include conf/extra/httpd-vhosts.conf

is uncommented in your xampp/apache/conf/httpd file, otherwise the above configuration will not work.

Step 3 - Local Host

Set up a local host entry:

On windows, open notepad in administrator mode and go to:

c:\Windows\System32\drivers\etc

select all Files for file types, then select:

hosts

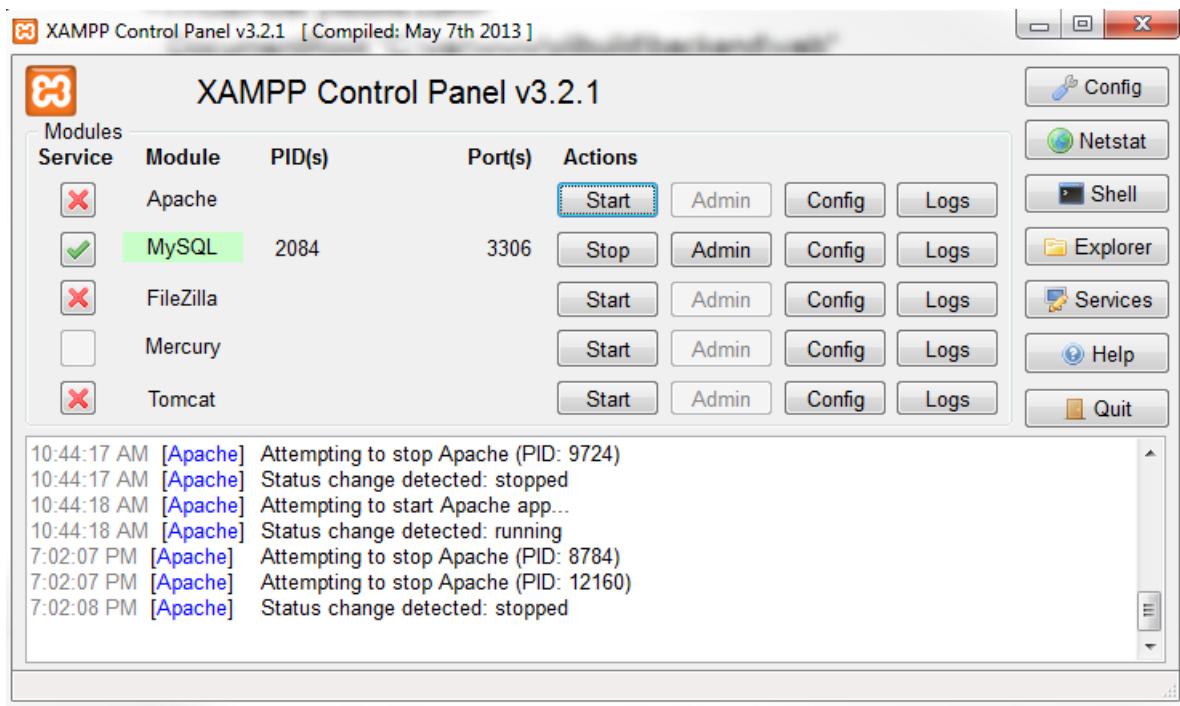
add the following and save:

```
127.0.0.1      yii2build.com    www.yii2build.com
```

```
127.0.0.1      backend.yii2build.com
```

Step 4 - Restart Apache

Click on xampp control panel and restart apache:



Xampp Control Panel

Note that we are running MySQL as a service, but not apache. If you did not set up xampp yet, obviously, you will need to do so before continuing. I recommend that you have all your tools set up and configured before proceeding and that you take some time to familiarize yourself with them. I included a xampp video link in chapter 1 that you can refer to as well.

Step 5 - Create Project in IDE

Create yii2build project in your IDE using yii2build as root folder. If you are not sure how to do this, google a tutorial for the IDE you are using. Note that I use forward slashes to designate a path

inside the IDE.

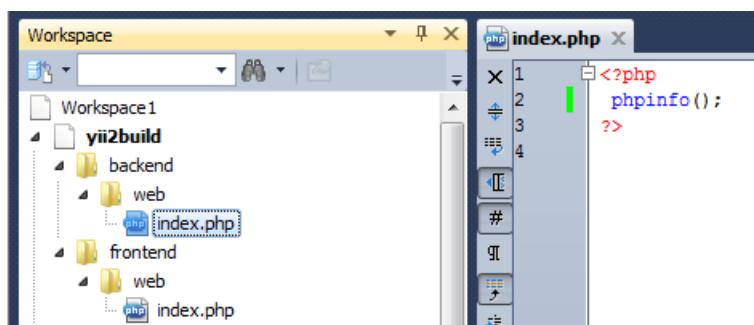
So now we can do a little test to see if we have setup our host files correctly. In your yii2build folder, create a folder named frontend and another called backend. Inside each of these folders, create a folder named web. So you should have yii2build/frontend/web/ and yii2build/backend/web/.

Now create a php file named index, with the following single line:

```
<?PHP  
phpinfo();  
?>
```

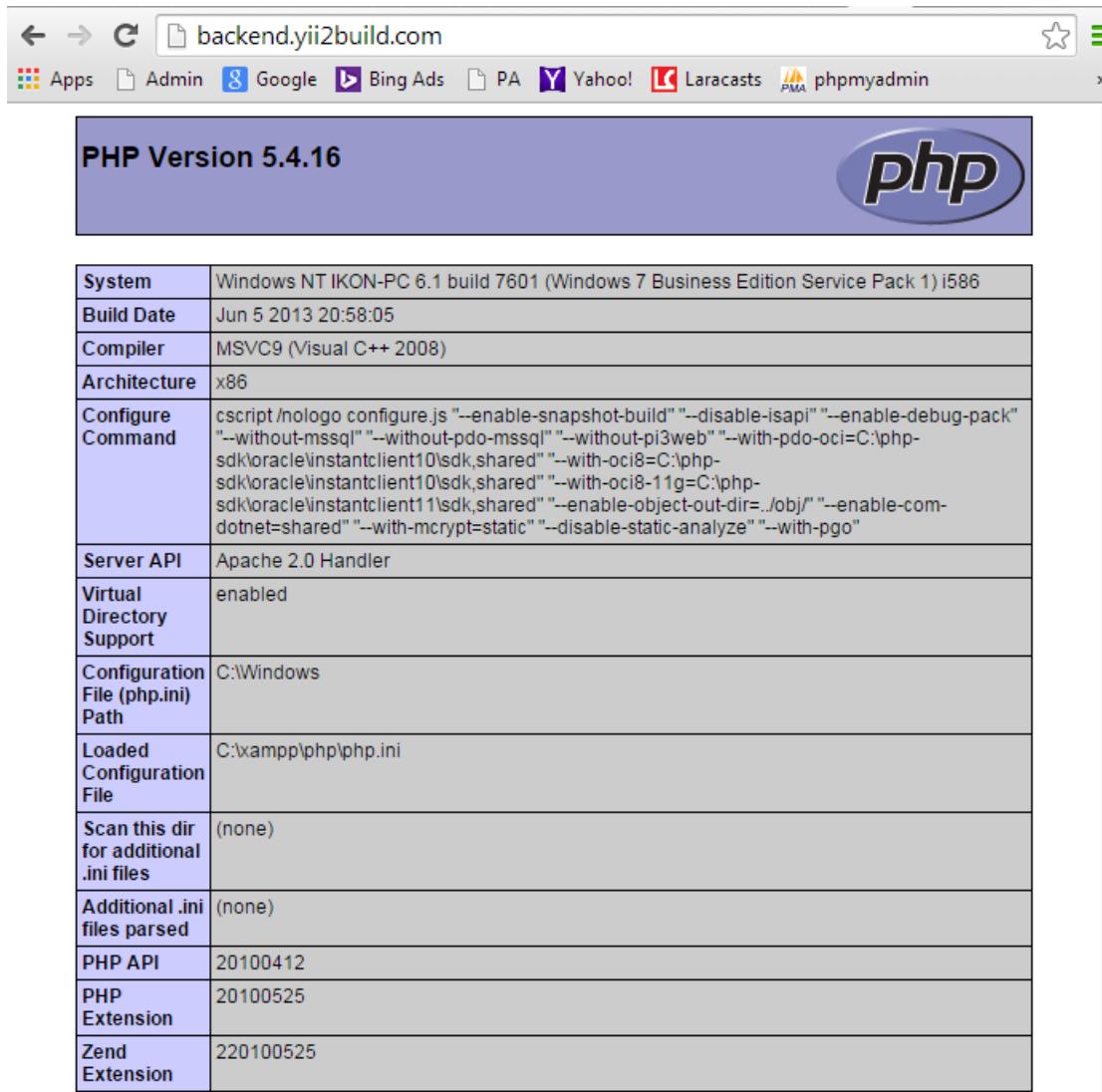
Save a copy to both folders. So you should have:

```
yii2build/frontend/web/index.php  
yii2build/backend/web/index.php
```



Folder Structure for Test Host

If you type `yii2build.com` and `backend.yii2build.com` into your browser, they should both return the `phpinfo` output, which also conveniently gives you a chance to check to see if you have PHP 5.4 or greater, which is what you need to run Yii 2.



The screenshot shows a web browser window with the URL `backend.yii2build.com` in the address bar. The page title is "PHP Version 5.4.16". The content is a table of PHP configuration parameters:

System	Windows NT IKON-PC 6.1 build 7601 (Windows 7 Business Edition Service Pack 1) i586
Build Date	Jun 5 2013 20:58:05
Compiler	MSVC9 (Visual C++ 2008)
Architecture	x86
Configure Command	cscript /nologo configure.js "--enable-snapshot-build" "--disable-isapi" "--enable-debug-pack" "--without-mssql" "--without-pdo-mssql" "--without-pi3web" "--with-pdo-oci=C:\php-sdk\oracle\instantclient10\ sdk\shared" "--with-oci8=C:\php-sdk\oracle\instantclient10\ sdk\shared" "--with-oci8-11g=C:\php-sdk\oracle\instantclient11\ sdk\shared" "--enable-object-out-dir=../obj/" "--enable-com-dotnet=shared" "--with-mcrypt=static" "--disable-static-analyze" "--with-pgo"
Server API	Apache 2.0 Handler
Virtual Directory Support	enabled
Configuration File (php.ini) Path	C:\Windows
Loaded Configuration File	C:\xampp\php\php.ini
Scan this dir for additional .ini files	(none)
Additional .ini files parsed	(none)
PHP API	20100412
PHP Extension	20100525
Zend Extension	220100525

Php Info

If the page does not resolve, go back and check your hosts file and/or your `httpd-vhosts.conf`. Make sure to restart Apache after making changes. Make sure you have local host entries for the domain, `yii2build`. Refer back to step 2 and 3 if necessary.

At this point, you should be able to see that your host entries are correct and that you are running the correct version of PHP. This is independent of Yii 2 and composer, so successfully implementing step 5 gives you a test point for the first part of our setup.

If this all checks out, you have successfully tested your host entries and you should delete these test web folders and their contents. Obviously leave the root folder, `yii2build`, in place.

Trouble-shooting tip: If you are using windows, you might have trouble deleting a folder. This is due to permissions of the file being set to read only. You can right click on the folder and use the properties menu to make adjustments. Use Google for exact details if you need help with that as it

can vary with the version of Windows you are using.

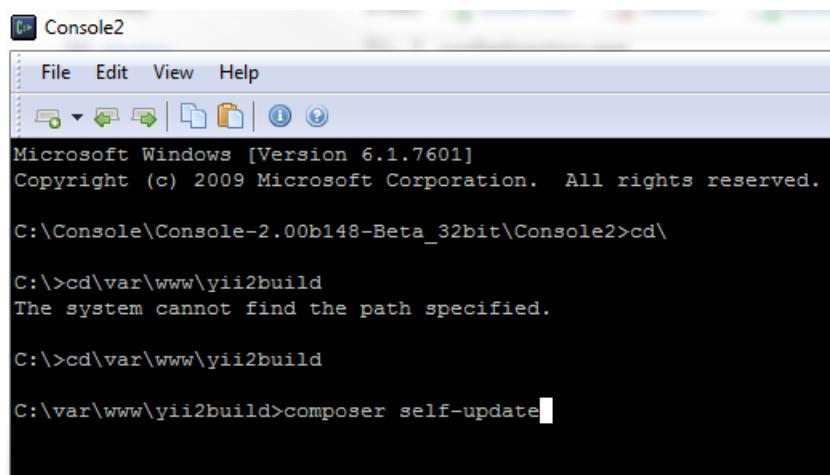
Step 6 - Find Command Line Path

From the windows command line, go to your yii2build folder. First cd\, then cd\var\www\yii2build if you have yii2build in \var\www. I know this can be a little confusing, so let me just reiterate. \var is a folder on my c: drive, within that is a folder named www, and inside www, I created a folder named yii2build, where the project will reside.

You don't have to follow this exactly, you just need to know where your root folder is and make sure you have the appropriate host entries.

Step 7 - Composer Self-Update

Make sure composer is installed and up-to-date. From the command line, in the root directory of your project, you should run: composer self-update.



```
Console2
File Edit View Help
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Console\Console-2.00b148-Beta_32bit\Console2>cd\

C:>cd\var\www\yii2build
The system cannot find the path specified.

C:>cd\var\www\yii2build

C:\var\www\yii2build>composer self-update
```

composer self-update

If you get an error message, check your installation of composer. If you don't have composer installed, Google it for instructions on installation into windows and xampp.

You will also need to make sure the following plugin is installed into composer. Issue the following command from the same directory where you did self-update:

```
composer global require "fxp/composer-asset-plugin:1.0.0-beta4"
```

```
C:\var\www\yii2build>composer self-update
Updating to version a309e1d89ded6919935a842faeaed8e888fbfe37.
  Downloading: 100%
Use composer self-update --rollback to return to version d79f2b0fd33ee9b89f3d9f1969f43dc3d570a33a
C:\var\www\yii2build>composer global require "fxp/composer-asset-plugin:1.0.*@dev"
```

Asset Plugin

If the above plugin is not installed, composer will not act correctly. The good news is that as long as you have composer working, the plugin is easy to install with the one simple command from above. Please note that in order to access the plugin, you may have to sign in with your Github account because it may ask you for your username and password. If you do not have a Github account, just go to [Github.com](https://github.com) and signup for a free account. It only takes a minute and it's free. You will only have to do this once.

Tip

I checked the Yii 2 Guide and the latest recommended version of the plugin is “fxp/composer-asset-plugin:1.0.0-beta4” If for whatever reason, that version of the plugin is out of date, use Google to find the correct version. You can also try @dev, which should work, but you never know. I will do my best to keep the book up-to-date, but these are the kinds of things that will be hard to keep track of. When going through setup in programming books, these are common problems, so this is just a heads up.

Step 8 - Install Yii 2

Install Yii 2 advanced template via composer. We do this from the command line in the above folder:

```
composer create-project --prefer-dist yiisoft/yii2-app-advanced
```

```
C:\var\www\yii2build>composer create-project --prefer-dist yiisoft/yii2-app-advanced
```

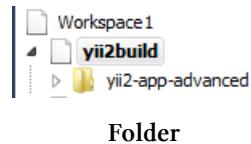
Composer Install Yii 2

Tip

The directions in the guide are slightly different in that you can set the project folder by naming it as the last parameter in the install. It can be confusing for beginners though, which is why I'm recommending that you follow these directions, which has an extra step, but allows you to check to see if the host entries are working before you install Yii 2.

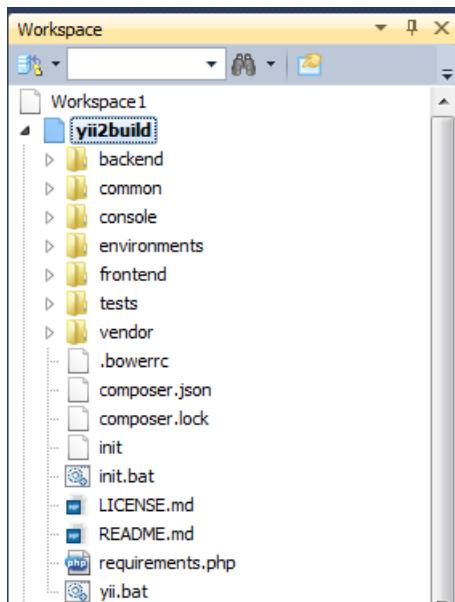
Step 9 - Check For Yii 2 Folder

You will now have a folder named yii2-app-advanced in your yii2build folder.



Folder

Using windows explorer, open this folder, and you will see all the framework files. Select all files and copy them one level up to the root yii2build folder, then **delete** the yii2-app-advanced folder. So, just to make it perfectly clear, now you should have the root folder, in this case yii2build, with the framework files inside it on the first level. There should be no yii2-app-advanced folder at this point. It should look like this after you deleted it:



App Folders

Step 10 - Run Php Init

Back to the command line. In the \path\to\yii2build, in my case it's \var\www\yii2build, run: php init.

```
C:\>cd\var\www\yii2build
C:\var\www\yii2build>php init
```

Run PHP Init

It will ask you if you wish to initialize in development or production. Select **0** for development. Then confirm **Yes**.

```
C:\var\www\yii2build>php init
Yii Application Initialization Tool v1.0

Which environment do you want the application to be initialized in?

[0] Development
[1] Production

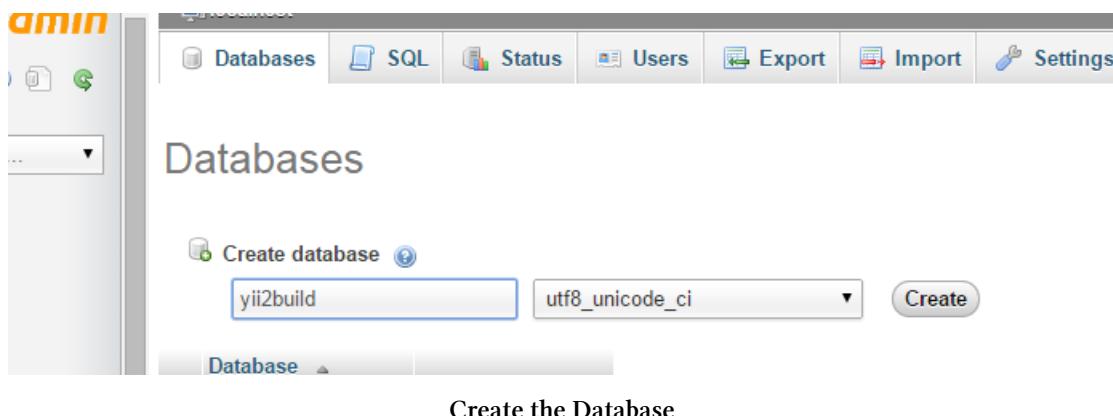
Your choice [0-1, or "q" to quit] 0

Initialize the application under 'Development' environment? [yes|no] yes
```

Devel Setup

Step 11 - Create The Database

Create the database. Go to Phpmyadmin. Go to the databases tab, create db named yii2build, with utf8_unicode_ci collation.



Step 12 - Set DB Connection

Adjust the components array in yii2build/common/config/main-local.php accordingly. It should look like this:

```
'db' => [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=yii2build',
    'username' => 'root',
    'password' => 'yourpassword',
    'charset' => 'utf8',
],
]
```

Obviously substitute your actual password into the config. Don't forget to save.

Step 13 - Run Migration

Back to the command line. You might have a different path, if so, it should be path to\rootfolder\yii2build>yii migrate. In my setup, it's \var\www\yii2build, run yii migrate. Looks like:

```
\var\www\yii2build>yii migrate
```

```
... initialization completed.

C:\var\www\yii2build>yii migrate[]
```

Migrate

Confirm yes.

```
C:\var\www\yii2build>yii migrate
Yii Migration Tool (based on Yii v2.0.0)

Creating migration history table "migration"...done.
Total 1 new migration to be applied:
    m130524_201442_init

Apply the above migration? (yes|no) [no]:
```

Confirm Yes

This will build the necessary tables in your database. You can check PhpMyadmin and you should have the following tables in the yii2build:

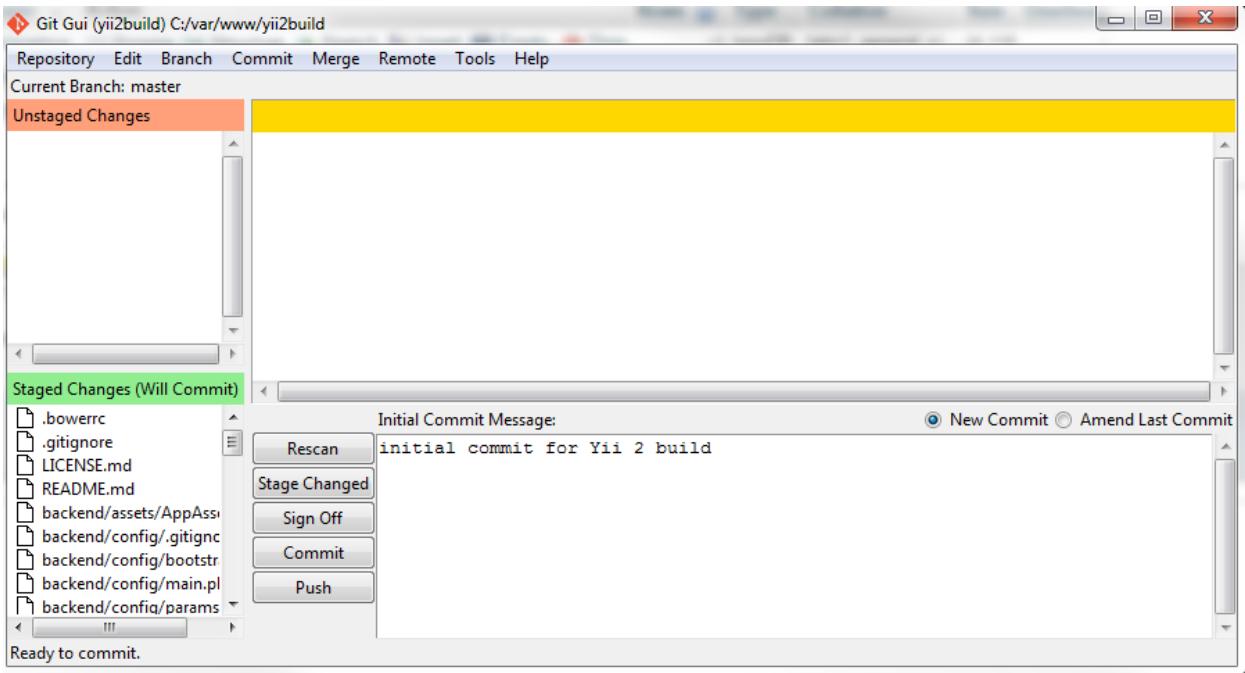
```
migrations
user
```

Table	Action	Rows	Type	Collation	Size	Overhead
migration	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	16 Kib	-
user	Browse Structure Search Insert Empty Drop	~0	InnoDB	utf8_unicode_ci	16 Kib	-
2 tables Sum		2	InnoDB	latin1_general_ci	32 Kib	0 B

Migrate Success

Step 14 - Create Git Repository

Step 14. Create GIT repository. Open Git GUI. Select create new repository. Select yii2build. Select stage changed. A warning may appear about indexes. Select continue if it does. Type initial commit for Yii 2 build into the message area, then select commit.



Commit in Git

You have to stage changes before committing. You may have to unlock index or click continue from a pop up dialog box. You will also need to enter a comment before committing.

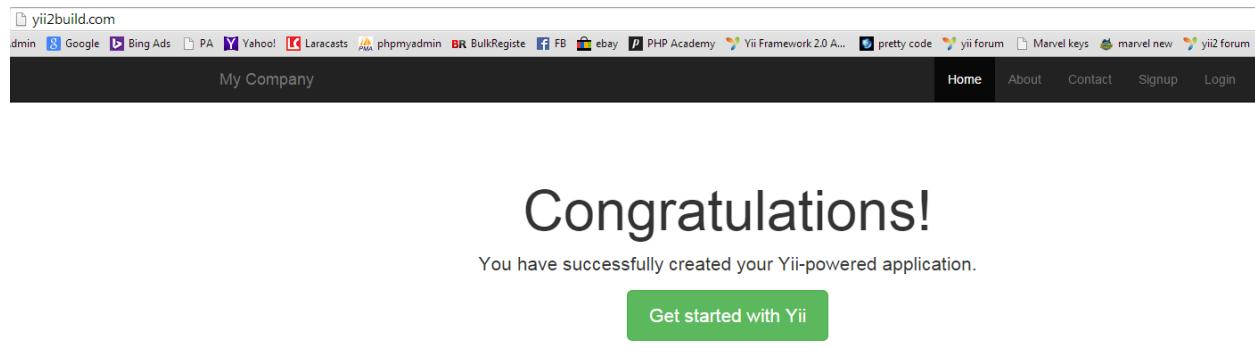
To view the repository from the repository menu, select visualize all branch history. This will show you the current master branch and its history, great for tracking your changes and stepping back if you need to. Version control is very important on a project of this size. It is unlikely that you could get through the project without it, so don't skip this step. Remember to save, at minimum, a commit at the end of every chapter of this book. You should probably do it more often than that.

Please note it is not necessary to save your project in a repository on Github, you can do that if you like, but we don't cover that in this book. We use GIT for local version control only.

Step 15 - Confirm App Is Working

Confirm the advanced application works by typing yii2build.com into your browser. You should get the advanced app template which will allow you to register a user and login with that user.

You should register a user and login to test that the application is working. Once you create a user, you can test the backend as well.



Heading

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Heading

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Heading

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Yii 2 Build

Since no access control is differentiated at this point from frontend to backend, you can log into backend by going to backend.yii2build.com and logging in. In both cases, login will simply return the index page and in the nav bar display the user name and the logout link.

Trouble-Shooting

If it's not resolving, then check your hosts file and httpd-vhosts.conf. Make sure Apache is running in xampp and has been restarted after making changes to the host files. Make sure your version of PHP is 5.4 or higher, that is required for Yii 2.

If you are seeing a directory tree, instead of the homepage, you did not successfully run the init, go back to step 10. If you can see the homepage, but get a DB error when you try to register, make sure in PhpMyAdmin that the yii2build database exists, that you have the correct password for it,

and that you have entered those settings in yii2build/common/config/main-local.php. Also make sure Mysql is running in xampp, see photo in step 4 for reference.

If you still can't get it to work, start over or at least from the point where you confirmed host entries are working and that you are running PHP 5.4.

Summary

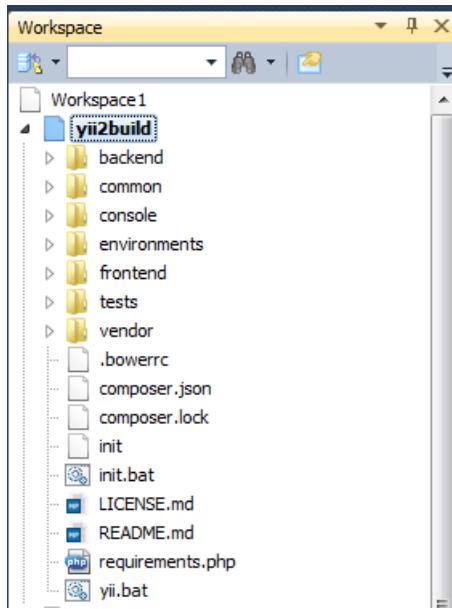
Congratulations, the hardest part of the book is over. Hopefully, this went smoothly for you. If you did have problems with setup, repeat the steps until you get it right. If you are sure everything is right, but it's still not working, consult with the individual docs of the components to see if something changed since this book was written. Google is typically very effective for this, when called to serve.

In the next chapters, we will begin working our way into development with Yii 2. We start with a brief tour of the MVC architecture, but we don't spend a lot of time on theory, unless we can use it to code. Instead, we dive in quickly in the subsequent chapters.

I've learned through personal experience that explanations of the broader concepts work better when they are coupled with practical implementation, which is why I learned almost nothing from most of my online OOP lessons, just vague impressions of interfaces and class inheritances. Not to worry. One of the great things about Yii 2 is that it pulls together so many of the principles and concepts of OOP in such an intuitive way, that you will understand the theories as you go. At least you will see them demonstrated.

Chapter Three: Welcome to the MVC

Now that we have the advanced template installed, let's take a few minutes to familiarize ourselves with our app's structure. So here we are:

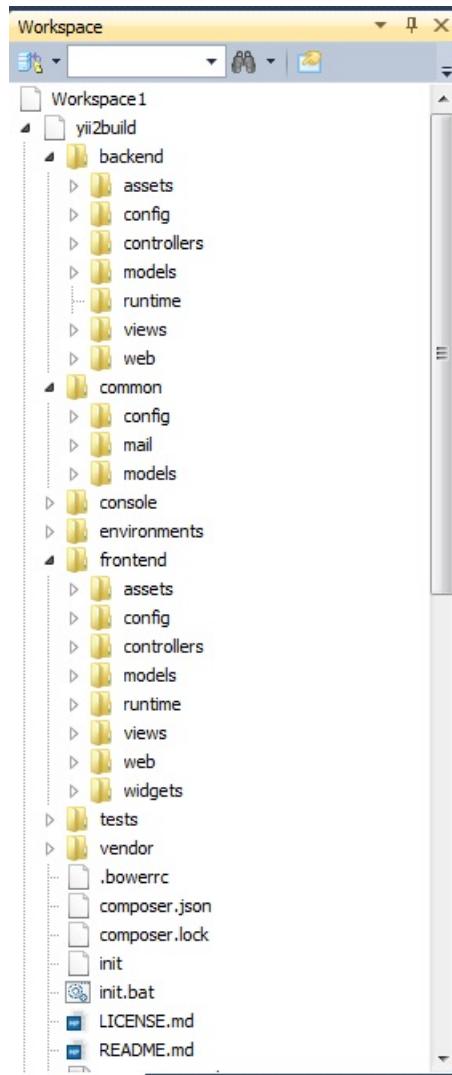


directory Structure

You can see the application is divided between backend, common, console, environments, frontend, tests, and vendor folders.

MVC Pattern

Yii 2 follows the MVC design pattern, where M stands for Model, V stands for view, and C stands for controller. We're going to discuss this briefly, but just for an overview. The best way to understand it is to work with the code and the directory structures directly, which we will do shortly. Here is another view of the structure with some of the folders open:



App Structure

You can see that the backend and frontend folders have folders named models, controllers, and views. The common folder has models, but no controller or views. You might want to take a few moments to look in all the folders to see what is there.

In Yii 2, the model is responsible for entering and retrieving data from the database. This includes any relationships that it needs from connected models, for example, a user and a user profile.

When a web request comes in, the controller typically routes it to the model, where it communicates with the database, then returns its results for display in the view. This allows for a separation of logic and presentation. You get fat models full of php, skinny controllers that mostly just do routing, and views that are light on PHP and deal more with HTML and javascript for presentation.

That's probably all we need to say about it as abstract theory. It works well and we will see how Yii 2 implements this pattern and how easy it is to understand in practice.

Index.php

There are exactly two points that should be accessible from the web in this application. Both backend and frontend have a folder named web within them and within that folder is file named index.php. If you recall, we set our hosts entries to look for this file, so that backend.yii2build.com goes to the backend folder version and yii2build.com goes to the frontend one. Each of these files is identical and looks like this:

```
<?php
defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

require(__DIR__ . '/../vendor/autoload.php');
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');
require(__DIR__ . '/../common/config/bootstrap.php');
require(__DIR__ . '/../config/bootstrap.php');

$config = yii\helpers\ArrayHelper::merge(
    require(__DIR__ . '/../common/config/main.php'),
    require(__DIR__ . '/../common/config/main-local.php'),
    require(__DIR__ . '/../config/main.php'),
    require(__DIR__ . '/../config/main-local.php')
);

$application = new yii\web\Application($config);
$application->run();
```

The first two lines check to see if the constants exist or define it for debug and dev.

```
defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');
```

Then come the require statements for files necessary to run the app, including the autoloader:

```
require(__DIR__ . '/../vendor/autoload.php');
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');
require(__DIR__ . '/../common/config/bootstrap.php');
require(__DIR__ . '/../config/bootstrap.php');
```

\$config is set by the ArrayHelper merge method, which requires the files specified:

```
$config = yii\helpers\ArrayHelper::merge(
    require(__DIR__ . '/../common/config/main.php'),
    require(__DIR__ . '/../common/config/main-local.php'),
    require(__DIR__ . '/config/main.php'),
    require(__DIR__ . '/config/main-local.php')
);
```

You can see it goes up 2 directories to find the common folder for that config. Then it goes up one directory to find the configure for frontend or backend, depending on which index.php file is doing the calling.

Then finally, we create a new instance of the application model, taking the config into the constructor, so now \$application becomes the instance of the application. Then we fire off it's run method:

```
$application = new yii\web\Application($config);
$application->run();
```

The Application Instance

The application is now available globally as Yii::\$app. There are a lot of important methods available from Yii::\$app and we will be talking about them later because it's very convenient to call them.

Don't get bogged down by this if you're not getting it right away. You will learn the architecture over time, this chapter is just meant to be an introduction.

Generally speaking, I stay away from higher level architecture concerns and focus more on what's in front of us and how it works, the nuts and bolts of getting up and running.

Routing

So let's get back to index.php, the file acts as a doorway to the application, creating the instance of it. When we are typing in a url for our application, we will always be calling index.php. Yii 2 handles all the routing for us, so when we want to get to the site home page for example, the route looks like this:

```
yii2build.com/index.php?r=site/index
```

That's not very pretty. You can set the urls to be pretty in the config, which helps their search engine friendliness and you can also eliminate the need to show index.php in the url, but we won't be covering that until chapter 13. By waiting on that, we eliminate having to debug the url or apache, if a problem with the page should arise.

Ok, let's get back to routing:

The `r=site/index` tells Yii 2 that we want the site controller and the index action. If an incoming request does not specify a route, which happens when someone just types in `yii2build.com` for example, then, the route specified by `yii\web\Application::$defaultRoute` will be used. The default is set to `site/index`, which, as we mentioned above, specifies the site controller and the index action.

If no action is specified, the controller assumes you want the index action. Example:

```
yii2build.com/index.php?r=site
```

This returns the index action of the site controller. In most cases, the action will render an associated view, a view with the same name as the controller action. Common actions and views are `index`, `view`, `create`, `update`, `delete`.

We often refer to the create, read, update, and delete actions as **CRUD**.

Using Gii

We will be using Yii 2's built-in rock star module, Gii, the all-time greatest code generation tool ever built, to help us make a lot of CRUD. And when we use Gii to create CRUD, we are often creating the controller at the same time, so we can generically expect the CRUD to include the controller. Don't worry if this is a little unclear now, it will make a lot more sense later when we are creating our files. And yes, I worship Gii, and I'm pretty sure by the time we're done, you will too.

If you look in the `views` folder under `frontend`, you can see a folder named `site`, which has an `index.php` file in it. This is the view page rendered by the index action of the site controller. The site controller itself is located at `frontend/controllers/SiteController.php`

Browse around the folders. Inside of `backend`, you will find controllers, models, views. You will find the same in the `frontend` folder. In the `common` folder, you see config, mail, and models. Overall, you can see the consistency in the naming conventions and they make Yii 2 easy to understand from an MVC point of view.

So obviously, this is quite different from simple web applications where you would have a url like `samplesite.com/about.php`. If you are tempted to skip learning Yii 2 because your current application requirements do not need to be so robust, keep in mind that over time, applications requirements tend to grow.

If today your client doesn't need a form with robust validation rules, it doesn't mean that he won't need it tomorrow.

Bootstrap

Also, with Yii 2, you get the frontend framework Bootstrap integrated out-of-the-box. If you are unfamiliar with Twitter's Bootstrap framework, I recommend you check it out, it has fast become the industry standard. You can check it out here:

Get Bootstrap

You don't need to download or do anything though, because like I said before, Yii 2 comes with it already integrated as a default. That means you get a platform-responsive css that scales to the device, allowing you to create mobile-first design from the start. And that, my friends, is just the cherry on top of the cake!

One day your client wants a nothing website and the next day he wants mobile css. You can deliver because you are already there. Anyway, I'm not trying to sound like a salesman. I truly love this platform and it shows.

In our previous projects, we might have created header and footer files that we could require in our individual pages, simple but inefficient. What if you forgot to include the file or made a typo to a previous version? What about theming and other advanced approaches?

Yii 2 has a cool solution for this by using layouts. Views are injected into the layout and there are methods available at the site config level or the controller level to specify which layout to use. A default layout is already there, so you don't need to do anything if you don't want to change it. You can also use nested layouts, if you feel that is necessary.

For our purposes, we are going to stick with the default layout, which is located at `frontend/views/layouts/main.php`. The only thing we are going to note at this time is that this tag in the middle of the page:

```
<?= $content ?>
```

This is where the view page gets injected. So now you know the header is above `$content` and the footer is below it. Don't worry, we will be making changes to this file and will be coming back to it later in the project.

Debugger

One other thing we should mention is the rather conspicuous Yii Debugger tool that sits at the bottom of the page, when you view the advanced template in the browser. This has many useful utilities, such as

- Configuration
- Logs
- Profiling
- Database
- Asset Bundles
- Mail

We don't spend time on it in this book, but in your programming workflow, this is incredibly handy. You can check to see which queries are being executed, how long they took and many other helpful

details about how your application is working. Take some time to familiarize yourself with it. You'll get it just by playing with it. If you are not using it, you can click the arrow at the bottom right of the browser and hide it.

Summary

As we said in the introduction, Yii 2 is not a trivial implementation of the MVC pattern. It's an extensive framework that is robust and easy to use, once you are familiar with it. The learning curve for beginners can be steep, but stick with it, it's worth it.

I'm going to do everything I can to help you along, method by method. It will be a little fuzzy at first, but as we go along, and we get deeper into the project, it will begin to make sense, and the pieces will start to fit.

So, what shall we build as our sample application? What would demonstrate useful features that many projects would share? And can we actually use anything we build here in a real project?

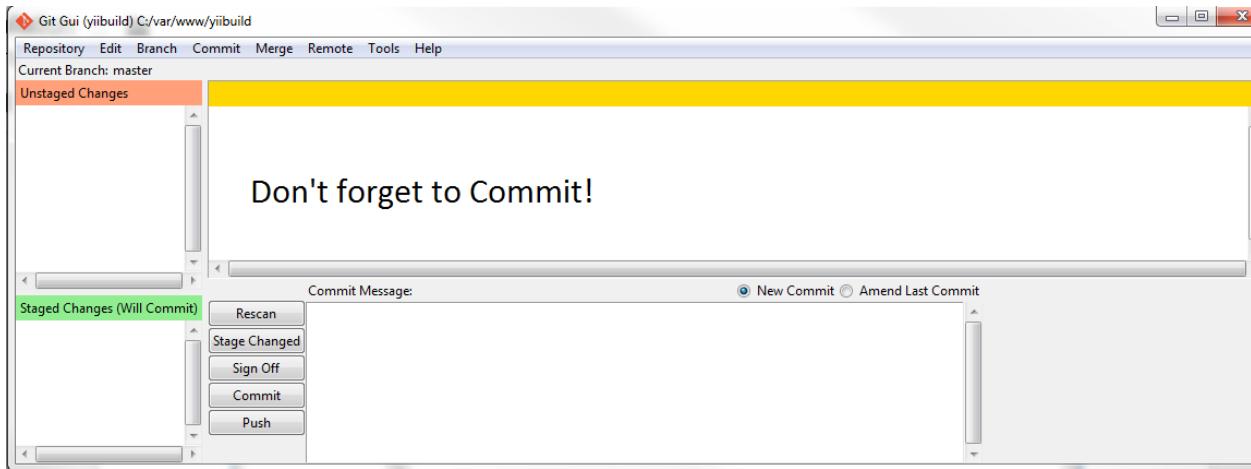
Asking myself those questions led me to conclude that this book should build an application template named Yii 2 Build. Now wait a minute, isn't the advanced application installation itself a template? Yes it is. But we are going to take things a little further.

We are going to create a basic RBAC system with UI that allows us to set roles, statuses, and user types to control access to both the frontend and backend of the application. We are going to build an upgrade controller, so that if we want a paid area of our application, we can enforce that rule.

We are also going to create a user profile model that can be extended or modified to suit your needs, but one that shows us how to control access to views that should be private to the user who owns them.

Our goal will be to create a working application that you can use as a model for future projects, one that is much further along in development than the advanced application template. As cool as the out-of-the-box template is, we can do a lot more, and learn the ins and outs of the framework along the way. Let's get to it and let's have some fun!

Also note, at the end of chapters, you will see:



Commit To Git

This is a reminder to commit your changes to version control. No need to do it now, since we didn't change anything. But stay on top of that, it will be a big help to you if you need to step backwards for any reason.

Chapter Four: Modifying the User Model

Now we're going to look at the User model. The Yii 2 advanced template gave us a working User model, where users can login and recover their password if need be, but we're going to change it. So the first thing we need to understand is why we are going to change it.

Right now our application does not treat login from frontend any differently than from the backend. But the whole point of creating separate areas for the frontend and backend is to enforce different levels of access.

Yii 2 only goes so far out-of-the-box. It leaves a wide range of implementations up to you about how you want to handle access to your application. The authentication part, determining a valid user/pass combination, is already handed to us out-of-the-box in the advanced application template. We know this because we can login and register as a user.

Authorization, determining how users are granted access to different pages is left up to us. Yii 2 has a built-in RBAC (role based access control) component that we could elect to use, but I prefer a different approach.

Normally, I try to do things the way Yii 2 intends, some of the world's best PHP programmers have worked on this framework and they really have thought of just about everything. On the other hand, a solution for RBAC involves a lot of personal choices and so sometimes it's better to custom craft it, it brings you closer to the code.

Anyway, I want to provide a working solution for RBAC, without bogging us down in it. But I still want to be able to control authorization and access through a backend UI that allows me to create roles and manage users and their status. I will explain more about that as we go.

Our goal is to build a template that we can use for many different kinds of projects, so for example, what if we had a site that needed permissions for free vs. paid users? That is more of a user type, not really a role. I think of role as something more basic like admin, user, customer service rep, etc. Role describes your relationship to the application.

User type, on the other hand, is a flexible concept that could apply to free vs. paid users, or different types of frontend users. For example, if you had a music site and some users were musicians and others were just fans.

It's also important to consider status, when assembling our schema. We need a way to determine if a user is active, pending, or retired or any other designation we come up with.

Flexibility is key, and I've found that by creating separate models for these concepts, it's easier to manipulate them. So after trial and error and a few iterations, I came up with a simple, yet

powerful approach to authorization. It will take us through multiple model setups and step us into relationships. Most of the code is really, really simple.

You'll also see how easy it is to create all this with Gii, Yii's built-in code generator. You've probably heard about Gii and it is an amazing tool, but we're not ready to use that just yet, we will come back to it next chapter.

Instead, we just need to start with some modifications to what we already have. The user table and User model were created for us automatically by the advanced application when we installed it, and while it's close to what we need, we have to make some important changes.

Role and Status

You'll notice in the database, that you have a column for status in the user table. We will also need a column for a user's role. Role and status both take an int as their data type. The problem is that we want to create a role table and a status table and we want to give these new tables the most intuitive names, which are, and I'm not trying to be funny here, role and status. Remember in Mysql, we are using lowercase as convention.

So it makes no sense, and indeed would cause ambiguation problems if we left the existing status column on the user table as status. Ambiguation is when a Mysql query can't determine the correct field/table to pull data from based on confusion in names. This can happen with the column 'id', which most of our tables are going to have. If we had a column on the user table named role and a different table named role, it would cause these kinds of issues and they can be tough to debug. So we do our best to avoid them completely.

In this case, we need to change the status column on the user table to status_id and add a new column for role_id. The best way to do this, if you are following with all the tools, is through Mysql Workbench. It's a little more graphical than PhpMyadmin, but either one will work if you have a preference.

So let's make a role_id column on the user table and make sure it has a default value of '1'. Similarly, let's change status to status_id and give that a default value of '1' as well. Now let's add one more column to the user table, and we'll call that user_type_id and also give that a default value of '1'.

Here is a screenshot from Mysql Workbench:

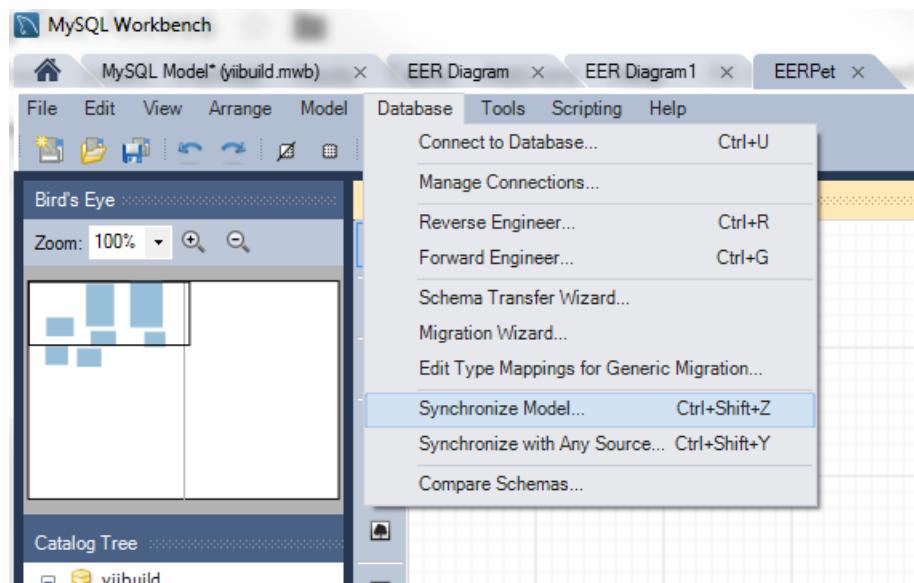
Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT(11)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
username	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
auth_key	VARCHAR(32)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
password_hash	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
password_reset_token	VARCHAR(255)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
email	VARCHAR(255)	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
role_id	SMALLINT(6)	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'1'
status_id	SMALLINT(6)	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'1'
user_type_id	SMALLINT(6)	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'1'
created_at	DATETIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
updated_at	DATETIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

User Table in Mysql Workbench

Note the use of unsigned for the id column. This means id can't be a negative number. Also make sure the created_at and updated_at fields are of the type DATETIME. For some reason the initial build is set to save as int for those fields, but we are going to work with behaviors on the User model to make sure they are saved correctly as DATETIME.

Note: If you have followed earlier instructions and added a test user, you should probably delete that record before syncing and changing the datastructure in the DB because it might not overwrite the existing record correctly.

Don't forget to sync to the database:



Synchronize to Database

Another note: If you are more comfortable using Php MyAdmin to make these changes, it is perfectly fine doing it that way, as long as you follow the data structure given.

Once we make the change, don't bother testing the site, nothing is going to work. We are going to have to change the User model before we test the site again.

 **Tip**

Before we start, a quick tip in case you didn't notice. Only view files have closing ?> tags. Do not include closing ?> tags in your models and controllers.

The User Model

Ok, let's take a serious look at the user model. When we set up our advanced template, Yii 2 did all the work for us. The upside to that is obviously, we did not have to write any code. The downside is that we are not really sure how it works. And of course if we are going to control access to users, we are going to have to know much more than we currently do about the user model.

For the sake of brevity, I'm not going to say too much about the default model you get with the advanced template, since we have so much ground to cover. So much of the core model is exactly the same with our revised model, that you will get most of it anyway.

Ok, let's do this.

If you have ever worked with code from a PDF or ebook format before, you know what a pain it can be to copy and paste correctly. Also, because of line constraints, there are ugly line separations that wouldn't normally be there. In consideration of all that, I'm providing links to gists, which will have the code in a much nicer format that's easy to copy. In most formats, it opens to a new window or you can right click on the gist link and open it in a new window, so you don't lose your place in the book.

Because I'm transferring code to a gist, we may end up with a typo. If that happens, refer to the book version, it should be authoritative at this point. Anyway, I hope to make this experience easier for you to work with.

Replace your existing User model, located within your common/models/User.php folder, with the code in the gist or from the book.

Gist:

[User Model](#)

From book:

```
<?php

namespace common\models;

use Yii;
use yii\base\NotSupportedException;
use yii\behaviors\TimestampBehavior;
use yii\db\ActiveRecord;
use yii\db\Expression;
use yii\web\IdentityInterface;
use yii\helpers\Security;

/**
 * User model
 *
 * @property integer $id
 * @property string $username
 * @property string $password_hash
 * @property string $password_reset_token
 * @property string $email
 * @property string $auth_key
 * @property integer $role_id
 * @property integer $status_id
 * @property integer $user_type_id
 * @property integer $created_at
 * @property integer $updated_at
 * @property string $password write-only password
 */

class User extends ActiveRecord implements IdentityInterface
{
    const STATUS_ACTIVE = 1;

    public static function tableName()
    {
        return 'user';
    }
}
```

```
/**  
 * behaviors  
 */  
  
public function behaviors()  
{  
    return [  
        'timestamp' => [  
            'class' => 'yii\behaviors\TimestampBehavior',  
            'attributes' => [  
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],  
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],  
                ],  
            'value' => new Expression('NOW()'),  
            ],  
        ],  
};  
  
/**  
 * validation rules  
 */  
  
public function rules()  
{  
    return [  
        ['status_id', 'default', 'value' => self::STATUS_ACTIVE],  
  
        ['role_id', 'default', 'value' => 1],  
  
        ['user_type_id', 'default', 'value' => 1],  
  
        ['username', 'filter', 'filter' => 'trim'],  
        ['username', 'required'],  
        ['username', 'unique'],  
        ['username', 'string', 'min' => 2, 'max' => 255],  
    ];  
}
```

```
[ 'email', 'filter', 'filter' => 'trim' ],
[ 'email', 'required'],
[ 'email', 'email'],
[ 'email', 'unique'],

];

}

/* Your model attribute labels */

public function attributeLabels()
{
    return [
        /* Your other attribute labels */
    ];
}

/**
 * @findIdentity
 */

public static function findIdentity($id)
{
    return static::findOne(['id' => $id, 'status_id' => self::STATUS_ACTIVE]);
}

/**
 * @inheritDoc
 */

public static function findIdentityByAccessToken($token, $type = null)
{
    throw new NotSupportedException(
        '"findIdentityByAccessToken" is not implemented.');
}
```

```
}
```

```
/**  
 * Finds user by username  
 * broken into 2 lines to avoid wordwrapping * @param string $username  
 * @return static|null  
 */  
  
public static function findByUsername($username)  
{  
  
    return static::findOne([ 'username' => $username, 'status_id' =>  
        self::STATUS_ACTIVE ]);  
  
}  
  
/**  
 * Finds user by password reset token  
 *  
 * @param string $token password reset token  
 * @return static|null  
 */  
  
public static function findByPasswordResetToken($token)  
{  
    if (!static::isPasswordResetTokenValid($token)) {  
        return null;  
    }  
  
    return static::findOne([  
        'password_reset_token' => $token,  
        'status_id' => self::STATUS_ACTIVE,  
    ]);  
}  
  
/**  
 * Finds out if password reset token is valid  
 *
```

```
* @param string $token password reset token
* @return boolean
*/
public static function isPasswordResetTokenValid($token)
{
    if (empty($token)) {
        return false;
    }
    $expire = Yii::$app->params['user.passwordResetTokenExpire'];
    $parts = explode('_', $token);
    $timestamp = (int) end($parts);
    return $timestamp + $expire >= time();
}

/**
* @getId
*/
public function getId()
{
    return $this->getPrimaryKey();
}

/**
* @getAuthKey
*/
public function getAuthKey()
{
    return $this->auth_key;
}

/**
* @validateAuthKey
*/
```

```
public function validateAuthKey($authKey)
{
    return $this->getAuthKey() === $authKey;
}

/**
 * Validates password
 *
 * @param string $password password to validate
 * @return boolean if password provided is valid for current user
 */
public function validatePassword($password)
{
    return Yii::$app->security->validatePassword($password, $this->password_hash);
}

/**
 * Generates password hash from password and sets it to the model
 *
 * @param string $password
 */
public function setPassword($password)
{
    $this->password_hash = Yii::$app->security->generatePasswordHash($password);
}

/**
 * Generates "remember me" authentication key
 */
public function generateAuthKey()
{
    $this->auth_key = Yii::$app->security->generateRandomString();
```

```
}

/**  
 * Generates new password reset token  
 * broken into 2 lines to avoid wordwrapping  
 */  
  
public function generatePasswordResetToken()  
{  
    $this->password_reset_token = Yii::$app->security->generateRandomString()  
  
    . ' ' . time();  
  
}  
  
/**  
 * Removes password reset token  
 */  
  
public function removePasswordResetToken()  
{  
    $this->password_reset_token = null;  
  
}  
}
```



Tip

If you are using the code from the book and not the Gist, the code for User.php is not formatted exactly like you would want it in your file. There are a few instances of two lines being used when there should be one. The reason is that PDF and other formats break the line with a wordwrap and insert special characters that mess up the code, so I have to proactively format the code so the line doesn't break. It doesn't always look pretty, but at least the code will function. You should find these instances and convert them to a single line by removing the white space. The public static function `findByUsername($username)` and public function `generatePasswordResetToken()` have the extra line in the body of the function.

Properties of the Model

So where are the properties of the class representing the model? You can see them in the comments, but they are not listed in the class. Why is that? It turns out that Yii 2, through its internal magic, knows the properties of the model by the column names of the tables, so you don't need to declare them. How cool is that? It certainly makes it hard to forget to include them.

This applies to the models that extend ActiveRecord. Form models, which we will explore later, extend Model and have properties that need to be declared, but we won't worry about that now.

Ok, let's move on to our user model. We didn't make too many changes, but obviously we added fields and changed status to status_id. Although we added user_type_id to the user table, we don't see much evidence of it here, except for in the rules method. And yet, as we described above, the model knows its attributes based on the table structure, so we are already adding depth to the user model.

That said, we can't really understand the User model without some idea of how we're going to support it, what other models we are going to create. Looking ahead, these are models we know we are going to create:

- Role
- Status
- UserType
- Profile
- Gender

In the next chapter, we are going to create the database tables for these models, and then the actual models themselves.

A lot of programmers will create database structure one table at a time and feel their way forward. Typically they use migrations to accomplish this. Other than the initial migration that created the original user model, we are not using migrations.

I'm a big believer in thinking through the data structure and creating it all at once, as opposed to an adhoc approach. That's not to say you can't refine and change as you go, but a little forethought goes a long way. You are of course free to use migrations if you wish, especially if you are comfortable using them. See the Yii 2 Guide for details:

[Yii 2 Migrations](#)

Constants

One thing that might pop out at you from that list of new models, especially with Role, Status, and UserType, is that these data structures could alternatively be handled by constants. While that would be probably easier to implement, I favor putting things like status values in the DB. The reason for this is that I can then create an Admin UI that allows me to update and create new values, without having to go into the code.

Take Role for example. Let's say that you have a role called admin, which grants access to the backend. It has a value of 20. You set up your constant as follows:

```
const ROLE_ADMIN_VALUE = 20;
```

But then you decide that you need an even more expansive role, let's call it SuperUser. You would have to go back to the code, find every instance where you are using the constant, create another constant, and add it to all the supporting methods that will populate the names of the Roles for dropdown lists, etc. It's easy enough to do, but in my opinion, not the best way.

I would rather have UI in the backend that allows me to simply add a DB record that defines the new role and gives it a value. Then, if I have coded my methods correctly, I have it available to me everywhere. As we progress in this book, you will see how this plays out.

Now if you check under our class declaration, you will see we left one constant in place:

```
const STATUS_ACTIVE = 1;
```

I kept the constant there for a good reason, even though it violates DRY (as far as for what we are going to build), because the status value active is vital to the registration and recover password system and we don't have our supporting tables and models built yet. I leave the constant in place, so that we can get the site up and running. The site needs this value to work and it's one of those cases where I'm willing to duplicate for ease of use. You can replace this later with a method if you choose to do so. It's a trivial matter in later stages to make the change if you wish.

Identity Interface

Going back to the class declaration for a moment:

```
class User extends ActiveRecord implements IdentityInterface
{
```

This is Yii 2 class structure that I didn't write, but it's not a problem, we can still note a few things about the model.

In this case, we extend ActiveRecord and implement IdentityInterface, which means we have to create the interface's methods in our User class. We'll look at the class and the comments written by the Yii 2 developers provide details on what the methods should do. It will give you some idea of how it works, but don't worry if you don't instantly know how to write the methods, and you will see why that is in a moment.

```
<?php

/**
 * @link http://www.yiiframework.com/
 * @copyright Copyright (c) 2008 Yii Software LLC
 * @license http://www.yiiframework.com/license/
 */

namespace yii\web;

interface IdentityInterface
{

    /**
     * Finds an identity by the given ID.
     * @param string/integer $id the ID to be looked for
     * @return IdentityInterface the identity object that
     * matches the given ID.
     * Null should be returned if such an identity cannot be found
     * or the identity is not in an active state
     *(disabled, deleted, etc.)
    */

    public static function findIdentity($id);
    /**
     * Finds an identity by the given token.
     * @param mixed $token the token to be looked for
     * @param mixed $type the type of the token. The value of this parameter depends \

```

```
on the implementation.  
* For example, [[\yii\filters\auth\HttpBearerAuth]] will set this parameter to \  
be `yii\filters\auth\HttpBearerAuth`.  
* @return IdentityInterface the identity object that matches the given token.  
* Null should be returned if such an identity cannot be found  
* or the identity is not in an active state (disabled, deleted, etc.)  
*/  
  
public static function findIdentityByAccessToken($token, $type = null);  
  
/**  
 * Returns an ID that can uniquely identify a user identity.  
 * @return string|integer an ID that uniquely identifies a user identity.  
 */  
  
public function getId();  
  
/**  
 * Returns a key that can be used to check the validity of a given identity ID.  
 *  
 * The key should be unique for each individual user, and should be persistent  
 * so that it can be used to check the validity of the user identity.  
 *  
 * The space of such keys should be big enough to defeat potential identity atta\  
cks.  
 *  
 * This is required if [[User::enableAutoLogin]] is enabled.  
 * @return string a key that is used to check the validity of a given identity ID.  
 * @see validateAuthKey()  
 */  
  
public function getAuthKey();  
  
/**  
 * Validates the given auth key.  
 */
```

```

* This is required if [[User::enableAutoLogin]] is enabled.
* @param string $authKey the given auth key
* @return boolean whether the given auth key is valid.
* @see getAuthKey()
*/

public function validateAuthKey($authKey);

}

```

An Interface is like a contract with the subclass. It says if you wish to use my interface, you must have the following methods. If you don't include them all, it will return an error. Programmers use Interfaces to control the architecture.

So this IdentityInterface is the contract our User model needs to implement. Ok, so I said we didn't have to worry about the interface, why is that? Fortunately for us, the advanced template already implements it for us, and you can find these methods already on our User model, so you don't need to write a single line of code. Thank you Yii 2 Advanced Template!

The basic template does not come with this implementation, so this is one reason why the advanced template is actually easier to implement than the basic template. It's one of the primary reasons we chose the advanced template for this book.

We have made a small change to a number of the interface methods that were provided by the template, changing the attribute 'status' to 'status_id' to reflect the changes we made in our data structure. I will point these changes out as we move through each method of the user model.

So let's get back to our User model proper. As we move through the methods, I will also point out what we included in the use statements to support the method when that is necessary.

The first method we see is:

```

public static function tableName()
{
    return 'user';
}

```

Hopefully this one is rather self-evident. I wish they were all this easy!

Behaviors

The next method is:

```

public function behaviors()
{
    return [
        'timestamp' => [
            'class' => 'yii\behaviors\TimestampBehavior',
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
            ],
            'value' => new Expression('NOW()'),
        ],
    ];
}

```

I find the concept of behaviors in Yii 2 very intuitive, written with clear syntax and beautiful code. The method tells the model how to behave, given certain events.

The first element in the array, ‘timestamp’ identifies the behavior, and we tell it what class we want to use. Then we define the events that will affect the attributes, in this case ActiveRecord::EVENT_BEFORE_INSERT and ActiveRecord::EVENT_BEFORE_UPDATE. These point to the attributes, ‘created_at’ and ‘updated_at’, which are also represented as fields in the user table.

Note that we are also defining the value and it will use:

```
'value' => new Expression('NOW()'),
```

It hands the string ‘NOW()’ to Mysql, which is a Mysql syntax for the current DateTime, so the expression class formats it for us. Without that, it would insert an integer, which is not the behavior we want. So this method will fire off whenever a record is created or updated and put the appropriate entry into the database in the correct DateTime format.

The concept of behaviors is used extensively on Controllers and we will be looking at that later. Also note that in order to use Expression, we have to include the appropriate use statement:

```
use yii\db\Expression;
```

Ok, on to our next method:

Rules

```
public function rules()
{
    return [
        ['status_id', 'default', 'value' => self::STATUS_ACTIVE],
        ['role_id', 'default', 'value' => 1],
        ['user_type_id', 'default', 'value' => 1],  
  

        ['username', 'filter', 'filter' => 'trim'],
        ['username', 'required'],
        ['username', 'unique'],
        ['username', 'string', 'min' => 2, 'max' => 255],  
  

        ['email', 'filter', 'filter' => 'trim'],
        ['email', 'required'],
        ['email', 'email'],
        ['email', 'unique'],
    ];
}
```

This is how easy Yii 2 makes it to enforce validation rules on the model. It's an array format, where the first value is the attribute, the second is the validator being called, and then come parameters or conditions. You can check the guide for a more complete list of validators and how to use them:

[Yii 2 guide on Rules](#)

The first 3 rules deal with setting defaults. I put white space in between groups for cosmetic reasons to make it easier to work on rules for a particular attribute, but the order they are in doesn't really matter.

If we look at the last set of rules, the ones for email, we see that make sure we trim spaces out, email is required, email is of email type, and email is unique. Yii 2 does all of this for you with this simple syntax, how awesome is that?

Identity Methods

The next method on the User model is `findIdentity`, which is an implementation of `IdentityInterface`, which we covered previously.

```
public static function findIdentity($id)
{
    return static::findOne(['id' => $id, 'status_id' => self::STATUS_ACTIVE]);
}
```

This is one of the places where we changed ‘status’ to ‘status_id’. This is followed by:

```
public static function findIdentityByAccessToken($token, $type = null)
{

    throw new NotSupportedException
('"findIdentityByAccessToken" is not implemented.');

}
```

Also created for us by the advanced template:

```
/*
 *broken into two lines to avoid wordwrapping
 * line break to avoid wordwrap
 * body should be single line in your IDE
 */

public static function findByUsername($username)
{
    return static::findOne(['username' => $username, 'status_id' =>
        self::STATUS_ACTIVE]);
}
```

Here again we have used the ‘status_id’ attribute instead of ‘status’. Another method from the IdentityInterface, with the same change to ‘status_id’ attribute:

```

public static function findByPasswordResetToken($token)
{
    if (!static::isPasswordResetTokenValid($token)) {
        return null;
    }

    return static::findOne([
        'password_reset_token' => $token,
        'status_id' => self::STATUS_ACTIVE,
    ]);
}

```

This followed by a method to test if the reset token is valid:

```

public static function isPasswordResetTokenValid($token)
{
    if (empty($token)) {
        return false;
    }
    $expire = Yii::$app->params['user.passwordResetTokenExpire'];
    $parts = explode('_', $token);
    $timestamp = (int) end($parts);
    return $timestamp + $expire >= time();
}

```

And yet another method from the Interface:

```

public function getId()
{
    return $this->getPrimaryKey();
}

```

And the last two methods from the Interface provided by the Advanced Template:

```
public function getAuthKey()
{
    return $this->auth_key;
}

public function validateAuthKey($authKey)
{
    return $this->getAuthKey() === $authKey;
}
```

Since the advanced app template provides the Interface methods for us, we will not cover them in greater detail. If you wish to read more on them, you can check out:

[Yii 2 Security Authentication](#)

Boilerplate Methods

The next few classes are all part of the boilerplate, which we did not change. Again, the comments provide an explanation better than I can, since these are deep framework methods that I didn't write:

```
/**
 * Validates password
 *
 * @param string $password password to validate
 * @return boolean if password provided is valid for current user
 */

public function validatePassword($password)
{
    return Yii::$app->security->validatePassword($password, $this->password_hash);
}

/**
 * Generates password hash from password and sets it to the model
 *
 * @param string $password
 */

```

```
public function setPassword($password)
{
    $this->password_hash = Yii::$app->security->generatePasswordHash($password);
}

/**
 * Generates "remember me" authentication key
 */

public function generateAuthKey()
{
    $this->auth_key = Yii::$app->security->generateRandomString();
}

/**
 * Generates new password reset token
 * line break to avoid wordwrap
 * body should be single line in your IDE
 */

public function generatePasswordResetToken()
{
    $this->password_reset_token = Yii::$app->security->generateRandomString()
        . '_' . time();
}

/**
 * Removes password reset token
 */

public function removePasswordResetToken()
{
    $this->password_reset_token = null;
}
```

If all went well with updating the user table and copying the new User model, you should be able to use the application again to register a user. If for some reason it doesn't work, retrace your steps

and check your spelling carefully. Make sure the DB is updated with the correct fields.

Note: Since we changed our field to status_id, the out-of-the-box forgot password functionality is now broken. Don't worry, we will fix it later.

Other Models Accessing User

Before we end our chapter about the User model, we should discuss the fact that a controller doesn't always access the User model the same way. There are different models that a controller can use to update the user table at different times. For example, in our application, if we are creating a user from the site registration form, the controller will use the SignupForm model located on the frontend/models/SignupForm.php, which is provided by Yii 2 as part of the advanced template.

That might sound confusing at first, but it makes a lot of sense. In advanced MVC architectures, forms typically have form models to govern their behavior. The form model works in concert with the controller to provide all the logic necessary to validate and process the form.

SignupForm Model

Let's take a look at the SingupForm model, located in frontend/models/SignupForm.php. You see there are only 3 attributes:

```
class SignupForm extends Model
{
    public $username;
    public $email;
    public $password;
```

The reason why there are no attributes or rules for role_id, status_id, user_type_id, for example, is that we are setting those by default in the background, not from the form, so they are not needed. Remember, we set the default value of role_id to 1, and it automatically gets recorded that way when a user record is created.

Often, user data will be handed to a form model to enforce validation rules or other methods. The data comes in from a controller, which gives the model the post data from a view that is typically a form. This sounds more complicated than it actually is.

It's important to know how a user is created in your application, so let's see how this works by looking at the actionSignup() method on frontend/controllers/SiteController:

```

public function actionSignup()
{
    $model = new SignupForm();
    if ($model->load(Yii::$app->request->post())) {
        if ($user = $model->signup()) {
            if (Yii::$app->getUser()->login($user)) {
                return $this->goHome();
            }
        }
    }

    return $this->render('signup', [
        'model' => $model,
    ]);
}

```

You can see the method calls an instance of the SignupForm model. The main method of SingupForm is signup(), which creates an instance of the User model if the form has passed validation:

```

public function signup()
{
    if ($this->validate()) {
        $user = new User();
        $user->username = $this->username;
        $user->email = $this->email;
        $user->setPassword($this->password);
        $user->generateAuthKey();
        $user->save();
        return $user;
    }

    return null;
}

```

It will try to validate, and if it can validate, it calls an instance of the user class, so it can set the user properties to what was handed in via form, create the hashed password, generate the auth key, save and return \$user. It's important to note that a return statement, when executed, terminates the function, so you don't need an else statement here. If there is a \$user, it gets returned and the code never executes return null. If the if statement evaluates false, it will return null. It will be false if validation fails or if there were some other problem.

Ok, back to the action on the SiteController, where we get a nice nested if statement, which we can break apart to understand:

```
if ($model->load(Yii::$app->request->post())) {
```

If the model (SignupForm) can load the post data from Yii::\$app->request->post(), which only happens if there is post data. The syntax for getting the post data is clear and concise:

```
Yii::$app->request->post()
```

This brings all the form attributes along as long as the form and form model are built correctly. The post data can only come from someone filling out the signup form on the view and being passed along by the action of the view. If that happens, then continue. In this case the view is signup.php under frontend/views/site/signup.php. We won't go into detail on the form now, but you can check it out for yourself if you want to.

Next if:

```
if ($user = $model->signup()) {
```

Call the signup method of SignupForm. The first thing the signup method does is validate, so if we don't get past the rules, it will not sign up the user and it will return an error message to the user, based on rule behavior. If all is well and we get an instance of \$user, it continues:

Then the third if:

```
if (Yii::$app->getUser()->login($user)) {
```

We are accessing getUser and login user from an application instance of Yii, which has access to those methods. We talked about creating the application instance from Index.php in chapter 3, so here it is being used to called a couple of chained methods.

tip

Note, for us to be able to use Yii::\$app, we need to have the use statement, use Yii; at the top of the file.

So if we can find the user and login the user, then:

```
return $this->goHome();
```

This simply takes you back to the Site Index.php view, but in a logged in state, otherwise, you get the signup form itself:

```
return $this->render('signup', [
    'model' => $model,
]);
```

And with all the validation and internal methods of Yii 2, if you tried to signup and something was wrong, it will display the error messages as well.

The login method from SiteController is similar:

```
public function actionLogin()
{
    if (!\Yii::$app->user->isGuest) {
        return $this->goHome();
    }

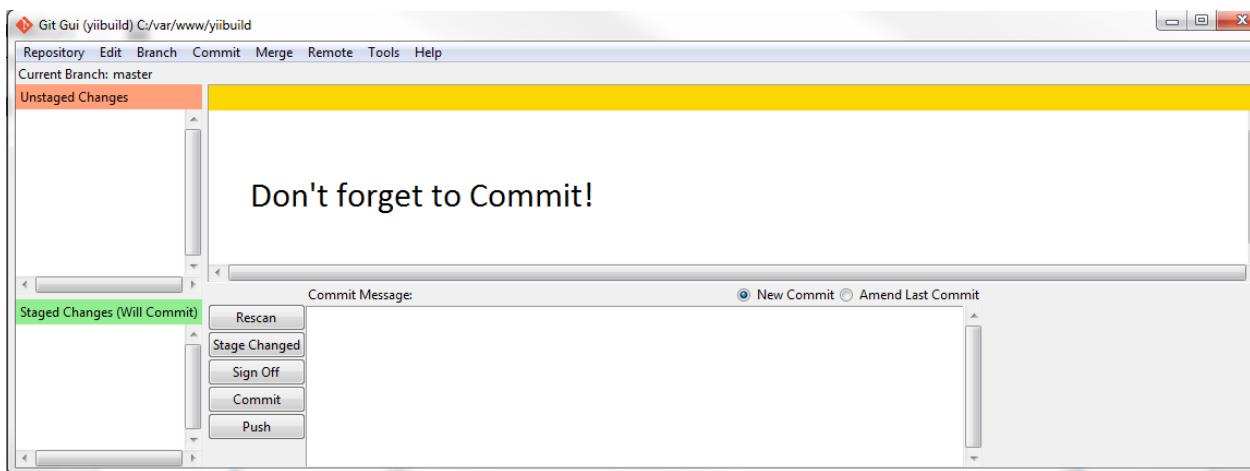
    $model = new LoginForm();
    if ($model->load(\Yii::$app->request->post()) && $model->login()) {
        return $this->goBack();
    } else {
        return $this->render('login', [
            'model' => $model,
        ]);
    }
}
```

First it tests to see if you are logged in or not by calling the `isGuest` method. We are using the `!` in front, so if not a guest, you are already logged in and you go to the home page.

Then it uses a different model, the `LoginForm` model and either logs you in and takes you back to the page you were on previously, but in a logged in state, or it shows you the login form, again with errors if you tried to login in and did it incorrectly.

Ok, so we took a quick detour from the `User` model to give you an idea of how users are created and to give you a look at the models moving user data through the site. We didn't really go into too much depth on the controller, we will cover controllers more in detail later, this was more about the models that are controlling the user. Here we had 3 distinct models, `User`, `SignupForm`, and `LoginForm` that controlled the user's data.

Summary



Don't forget to Commit!

Ok, that was a lot to absorb. If this is all new to you and you are struggling with it a bit, don't worry, it will become more clear over time as you get used to seeing the same types of methods used to move data around the site. We will see all this in detail again.

So we are building a reusable template and starting by modifying the User model, which has a lot of methods on it that reach deep into the framework.

The User model is always drastically different than other models because of things like the set password method and the other methods that are unique to users. We also touched on the fact that controllers can sometimes use other models to create and change user records.

The other models we are going to build, such as Role, Status, UserType, etc., tend to be more straightforward and easier to understand, not to mention, a lot shorter in size.

In the next section, we will use Gii, Yii 2's code generation tool, and you will see how amazing this really is and how much faster the workflow is.

Chapter Five: Creating New Models with Gii

Before we can use Gii to create new models, we have to create the tables first. As we said in the last chapter, our goal is create a data structure that allows us to manage users and control access to the website.

The models we will be creating are:

- Role
- Status
- Gender
- UserType
- Profile

Note that in the list above, since we are talking about models, we use uppercase, and you can see on UserType, that I used the format that Gii will create from the convention where the table name is user_type. We will understand that better later in the book when we create the UserType CRUD.

Creating Tables

Now it's time for us to create the rest of the tables. I'm going to provide screenshots from Mysql workbench, which will give us an easy reference for not only what fields we need, but also the constraints and data types.



Tip

MySQL CONSTRAINTS are used to define rules to allow or restrict what values can be stored in columns.

MySQL CONSTRAINTS enforce the integrity of database.

MySQL CONSTRAINTS are declared at the time of creating a table.

MySQL CONSTRAINTS are:

- NOT NULL

- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT

For a tutorial reference on Mysql, check [W3Resource](#).

Role Table

Here is the table for role:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	SMALLINT(6)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>					
role_name	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
role_value	INT(11)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					

role table

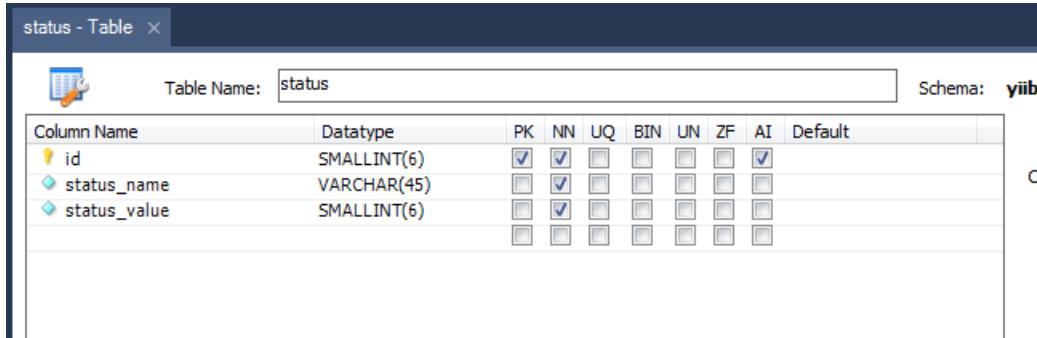
Notice that we have used lower case to name the table. If a table name requires two words, we will separate them with an underscore. We will also use underscore to separate words in column names as you can see above.

The role table is very simple. Pk stands for primary key, NN means Not Null, and AI is auto-increment. We auto-increment the record ids. We use varchar for role_name and integer for role_value. You can probably use small int for role_value, I will leave that choice up to you.

Sometimes when you building even trivial data structure, you will want created_at and updated_at, plus created_by and updated_by, just to keep track of who is doing what and when. But since this is only holding the names and values of roles, we don't need those fields.

Status Table

Ok, let's move on and now do one for status:



The screenshot shows the 'status - Table' configuration window. The 'Table Name' is set to 'status' and the 'Schema' is 'yiib'. The table structure is defined with three columns: 'id' (PK, NN), 'status_name' (UQ), and 'status_value' (NN). The 'Default' column contains checked boxes for 'id' and 'status_value'.

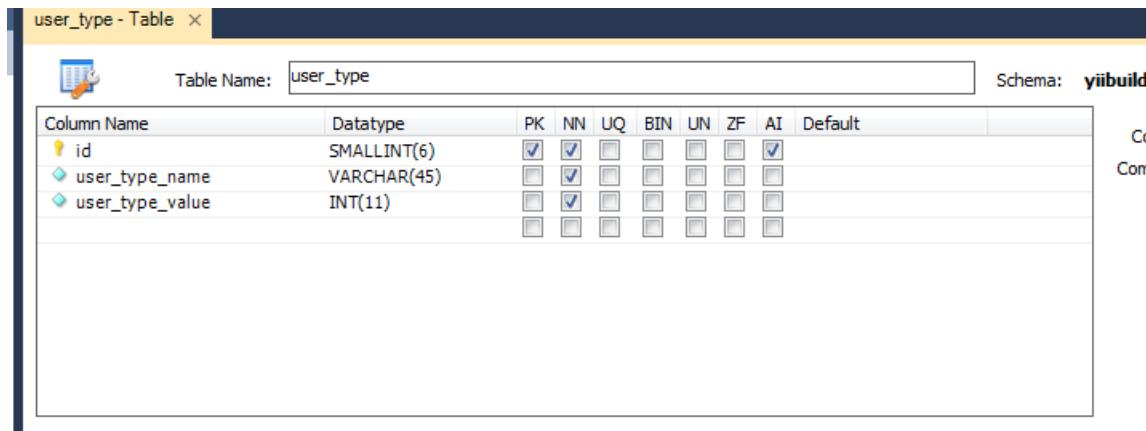
Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	SMALLINT(6)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>				
status_name	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
status_value	SMALLINT(6)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				

Status Table

This is identical to role, only it's for status. On both tables we have created so far, we are selecting PK for primary key on the first column, which is id. We also set it to NN, which is not null, meaning it is not allowed to be null.

User Type Table

Now let's do the user_type table:



The screenshot shows the 'user_type - Table' configuration window. The 'Table Name' is set to 'user_type' and the 'Schema' is 'yiibuild'. The table structure is defined with three columns: 'id' (PK, NN), 'user_type_name' (UQ), and 'user_type_value' (NN). The 'Default' column contains checked boxes for 'id' and 'user_type_value'.

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	SMALLINT(6)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>				
user_type_name	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
user_type_value	INT(11)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				

User type Table

That's the same type of data structure as the first two tables we created, only we have a table name with an underscore in it. Gii creates a specific naming convention to handle this, which we will see later when we create the model, controller, and views.

Gender Table

Here we have gender:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	SMALLINT(6)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
gender_name	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Gender Table

This one is even simpler, just id and gender_name.

Profile Table

And lastly, the profile table:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT(11)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
user_id	INT(11)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
first_name	TEXT(60)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
last_name	TEXT(60)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
birthdate	DATE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
gender_id	SMALLINT(6)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
created_at	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
updated_at	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Profile Table

Our plan is to allow each user to create a single profile, so these will have a one to one relationship with the User model. So let's add the following method to the User model at the bottom of the class.

Gist:

[Get Profile](#)

From book:

```
public function getProfile()
{
    return $this->hasOne(Profile::className(), ['user_id' => 'id']);
}
```

Make sure you updated the User model with the above. We are jumping around a bit, but that can't be avoided.

You can see in this case the id of the user is set to the user_id on the profile record. And this establishes the link between the two models.

We'll do that for our other models in a few minutes, after we have set up the new models. Then we can update the User model with all the other relationship methods it needs to talk to the other models.

Ok, back to the profile table.

Note that on the profile int columns, I checked off UN, which stands for unsigned and does not allow negative numbers.

You can also see there is a red diamond on the gender_id column and this represents a foreign key. Foreign keys are set to tie 2 tables together and Gii can read this data and setup the relationship for you when it creates the model. We will see this in action later.

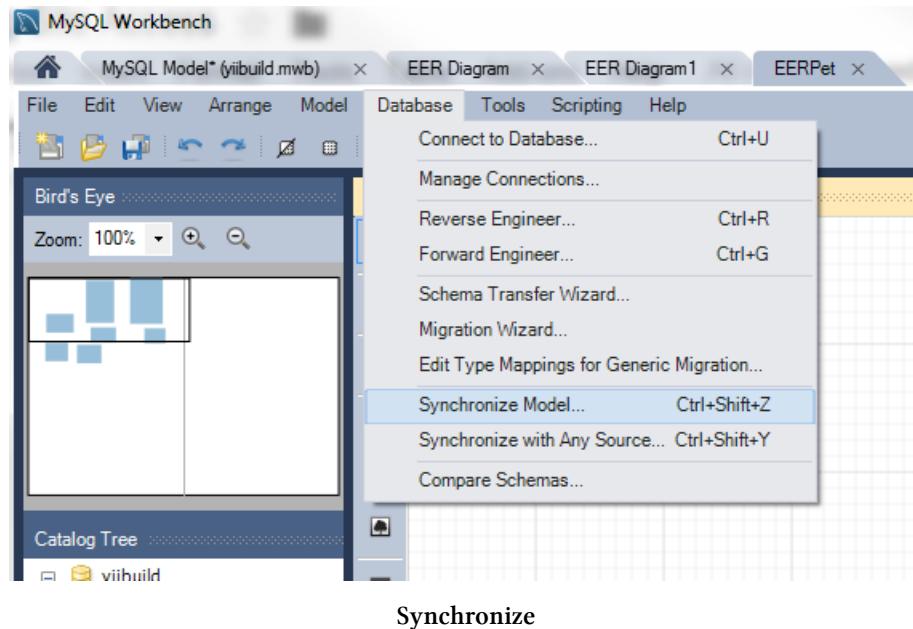
If you are unfamiliar with foreign keys in MySQL and MySQL Workbench, you should take some time to Google it and learn it, they are an important part of database structure. You don't necessarily need to use them to follow this book, but you do need to make sure that all relationships are defined in cases where Gii would have automatically created the relationship for you because of a foreign key. It's not a problem, just pay careful attention to the needed relationships when we cover them later.

Right now all you need to know, is that the foreign key for gender_id on the profile table is mapped to id on the gender table.

Note: If you are having trouble setting the foreign key, make sure the data types match exactly. Refer to the screenshots above for reference.

Synchronize

Don't forget to synchronize the model with the actual DB:



Synchronize

Make sure to check PhpMyadmin to make sure everything synced ok:

The screenshot shows the PhpMyAdmin interface for a database named 'yiibuild'. On the left, there's a tree view of databases: 'exampl', 'freindr', 'information_schema', 'keycomics', 'magento', 'mydb', 'mysql', 'performance_schema', and 'root'. The main area displays a table of tables with the following data:

Table	Action	Rows	Type	Collation	Size	Overhead
gender	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	16 Kib	-
migration	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	16 Kib	-
profile	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	32 Kib	-
role	Browse Structure Search Insert Empty Drop	~3	InnoDB	latin1_general_ci	16 Kib	-
status	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	16 Kib	-
user	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	16 Kib	-
user_type	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	16 Kib	-
7 tables	Sum	15	InnoDB	latin1_general_ci	128 Kib	0 B

PhpMyAdmin

And that's it. All in all, it's a very simple data structure and we're going to have a lot of fun with it. We're going to use Gii to create models, controllers, and views, lots of code that it will generate for us.

Configuring Gii

Of course we need to make sure we have Gii installed. Go to the following url in your browser:

yiibuild.com/index.php?r=gii

If that does not resolve, then you need to check your Composer.Json file to see if you have the Gii module required. composer.json is in your root directory and should be visible in your IDE.

Again with larger blocks of code, for your convenience:

Gist:

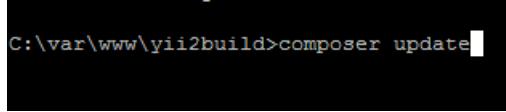
composer.json

This is what my file looks like:

```
{  
    "name": "yiisoft/yii2-app-advanced",  
    "description": "Yii 2 Advanced Application Template",  
    "keywords": ["yii2", "framework", "advanced", "application template"],  
    "homepage": "http://www.yiiframework.com/",  
    "type": "project",  
    "license": "BSD-3-Clause",  
    "support": {  
        "issues": "https://github.com/yiisoft/yii2/issues?state=open",  
        "forum": "http://www.yiiframework.com/forum/",  
        "wiki": "http://www.yiiframework.com/wiki/",  
        "irc": "irc://irc.freenode.net/yii",  
        "source": "https://github.com/yiisoft/yii2"  
    },  
    "minimum-stability": "stable",  
    "require": {  
        "php": ">=5.4.0",  
        "yiisoft/yii2": "*",  
        "yiisoft/yii2-bootstrap": "*",  
        "yiisoft/yii2-swiftmailer": "*",  
        "kartik-v/yii2-social": "dev-master",  
        "fortawesome/fontawesome-free": "4.2.0"  
    },  
    "require-dev": {  
        "yiisoft/yii2-codeception": "*",  
        "yiisoft/yii2-debug": "*",  
        "yiisoft/yii2-gii": "*",  
        "yiisoft/yii2-faker": "*",  
        "yiisoft/yii2-jui": "*"  
    },  
    "config": {  
        "process-timeout": 1800  
    },  
    "extra": {  
        "asset-installer-paths": {  
            "npm-asset-library": "vendor/npm",  
            "bower-asset-library": "vendor/bower"  
        }  
    }  
}
```

```
}
```

You can see under “require-dev”, I have the line for gii. I have a few extensions included for use later, including Karitk social, font-awesome and others. It makes sense to just copy this version of composer.json into your file, so go ahead and do that, then run composer update from the command line:



```
C:\var\www\yii2build>composer update
```

Composer Update

Now if you go back to your url:

```
yii2build.com/index.php?r=Yii
```

You should get to Gii.

Note for users who are not running from localhost, you will need to adjust your config. Please consult the guide for details if this is necessary:

[Gii Config Not LocalHost](#)

If you had Gii in the first place, make sure you run the composer update anyway, so we can pull in the extensions we are going to need later. Here is a note of caution. If you skip steps, like this composer update with the new composer.json file, it will cause problems later when it is assumed you have these extensions. Please be careful to follow instructions exactly, you will get better results.

Anyway, if you can see Gii, congrats and take a breath, you’re about to have some fun.

Making Models with Gii

One thing I should mention before we start. We are going to create 5 new models and continue to update the User model with relationships. This is a lot of information, so don’t feel like you have to memorize it or instantly understand every nuance. All of this will make sense as we go on and you see how we utilize the models and how we stitch everything together.

Also note, I will be providing complete files for the two larger models near the end of the chapter, so even if you miss something, you will have the complete models to check it against. That said, let’s fire up Gii!

Point your browser to:

```
backend.yii2build.com/index.php?r=Yii
```

Create Role Model

And let's make our first new model. We'll start with the Role model. One decision we have to make upfront is where to locate the model. The logical choices are frontend, common, and backend folders.

The guide for Yii 2 recommends using the common folder for models, and then extending them to different models for frontend and backend if necessary. I actually prefer to use common for other types of classes and models. So that means I either put the model in frontend or backend. With the use of namespaces, you have a lot of freedom to structure it how you wish.

As to whether that should go in the frontend, common, or backend folder, I'm going to put it in the backend folder. It could easily be in one of the other folders, but this choice seems intuitive to me, it's what I would think of when I go looking for it, so I'm going to put in backend.

Here is a screenshot of Gii with the role table:

Model Generator

This generator generates an ActiveRecord class for the specified database table.

Table Name
role

Model Class
Role

Namespace
backend\models

Base Class
yii\db\ActiveRecord

Database Connection ID
db

Use Table Prefix

Generate Relations

Generate Labels from DB Comments

Enable I18N

Code Template
default (C:\var\www\yiibuild\vendor\yiisoft\yii2-gii\generators\model/default)

Preview

Gii with Role table

Pay careful attention to the Namespace field. You can see we have the namespace backend\models, so the Role.php file will reside in that folder and the namespace will be attached to the file. Then whenever we want to use it, we just include a use statement:

```
use backend\models\Role;
```

Click the Preview button. It will auto-generate the file. You can review it by clicking on the file name. To actually generate the code, click on the green Generate button. Now go check backend/models and you will see Role.php, perfectly formatted for us:

```
<?php

namespace backend\models;

/**
 * This is the model class for table "role".
 *
 * @property integer $id
 * @property string $role_name
 * @property integer $role_value
 */

class Role extends \yii\db\ActiveRecord
{

    /**
     * @inheritdoc
     */

    public static function tableName()
    {
        return 'role';
    }

    /**
     * @inheritdoc
     */

    public function rules()
    {
        return [
            [['role_name', 'role_value'], 'required'],
            [['role_value'], 'integer'],
            [['role_name'], 'string', 'max' => 45]
        ];
    }
}
```

```
/**
 * @inheritdoc
 */

public function attributeLabels()
{
    return [
        'id' => 'ID',
        'role_name' => 'Role Name',
        'role_value' => 'Role Value'

    ];
}

}
```

So there it is in backend/models. It looks very familiar at this point. I've already explained in detail what the rules do on the User model, and rules function the same way here. This is such a simple model, that we don't have much to talk about.

The attributeLabels method just sets the attributes to label names that will be visible on the application.

Let's just take a moment to appreciate how clean and simple all that really is. You can see how, once you get into this flow, things can move quickly.

Ok, moving on.

Add Records To Role Table

Now that we have a role table and Role model, let's use PhpMyadmin to create a couple of role records that we can play with. It will make it easier to understand how this all fits together:

User the insert tab on the role table:

The screenshot shows the 'Insert' screen for the 'role' table in phpMyAdmin. There are two sets of input fields for inserting new records. The first record has 'role_name' set to 'User' and 'role_value' set to '10'. The second record has 'role_name' set to 'Admin' and 'role_value' set to '20'. Both records have 'id' set to 1. At the bottom, there are buttons for 'Insert as new row' and 'Go'.

Column	Type	Function	Null	Value
id	smallint(6)			
role_name	varchar(45)			User
role_value	int(11)			10

Column	Type	Function	Null	Value
id	smallint(6)			
role_name	varchar(45)			Admin
role_value	int(11)			20

PhpMyAdmin Role Insert Records

You can see we don't need to set the id field, that is auto-increment. So we are creating 2 records, 1 named User, 1 named Admin. User has a role_value of 10 and Admin has a role_value of 20. Once the records are added, you should see this on the browse tab for the role table:

The screenshot shows the 'Browse' tab for the 'role' table. It lists two records: one for 'User' with id 1, role_name 'User', and role_value 10; and another for 'Admin' with id 2, role_name 'Admin', and role_value 20. Each record has edit and delete buttons.

	+ Options	← →	id	role_name	role_value	
<input type="checkbox"/>				1	User	10
<input type="checkbox"/>				2	Admin	20

PhpMyAdmin Role Records

If you recall, we set role_id on the user table to default to 1 when a user record is created. So it makes sense that role_id will map to id, which means a new user gets a role of user when they are created. Now we just need a few methods on both models to tie this all together.

Add Relationship To Role

On Role.php, the Role model, at the top, under namespace, we are going to add:

```
use common\models\User;
```

And at the bottom of the class add:

```
public function getUsers()
{
    return $this->hasMany(User::className(), ['role_id' => 'id']);
}
```

The use statement gives us visibility on the User model. The getUsers method is a standard way of establishing the relationship from role to users. In this case, we are creating ahasMany relationship because a single role can have many users. That is also why the method is called getUsers instead of the singular getUser. In the array, you see:

```
['role_id' => 'id']
```

So the first field is that of the relationship and the field it points to in the array is from the model you are currently working on. The syntax is very intuitive, but it's worth spelling this out exactly:

```
hasMany(User::className(), ['role_id' => 'id'])
```

className is a method of the related model. The related model's field name comes first, followed by the current model's field name, so now role_id on the user table is mapped to role_value in the role table. That's it!

Update User Model with Role

Of course, we need the same type of method on the User model. Let's open the User model at common/models/User.php and add the use statement:

```
use backend\models\Role;
```

Now let's add the following methods to the bottom of the class:

Gist:

User Role Relationships

From book:

```
/**
 * get role relationship
 */
public function getRole()
{
    return $this->hasOne(Role::className(), ['id' => 'role_id']);
}
```

```


/**
 * get role name
 *
 */
public function getRoleName()
{
    return $this->role ? $this->role->role_name : '- no role -';
}

/**
 * get list of roles for dropdown
 */
public static function getRoleList()
{
    $droptions = Role::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'role_name');
}


```

Ok, let's look at these one by one. The first one is the `getRole` relationship, which is the other side of the `getUsers` relationship that we added to `Role`. In this case, `Users` only have one role, so it is a `hasOne` relationship and the method has the singular `getRole` name to it. Otherwise, the format is exactly the same as it was for the `Role` model. It simply says that `id` on the `role` table maps to `role_id` on the `user` table. This should be easy to understand.

The second method here for `User` is `getRoleName`. This allows us to return the name of the role, which we will want to do for our backend UI. We put in a ternary test to see if a role has been assigned, and if so, return the name or the string '`- no role -`'.

Finally, we want to return a list of role ids and names to use in dropdown lists in the UI. So we create the `getRoleList` method as follows:

```
public static function getRoleList()
{
    $droptions = Role::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'role_name');
}
```

Here we create the local variable \$droptions and assign an instance of Role, with all records returned as an array. Note how intuitive yii 2's syntax is, it reads just like a sentence. Then we use ArrayHelper::map method to list the role values and names. In order to use ArrayHelper, we need to include a use statement for it at the top of the file in the use statement block:

```
use yii\helpers\ArrayHelper;
```

This is a very common format for us to have these 3 relationship methods and we will do similar methods for the other models. Also note, the Yii 2 guide does a pretty good job of listing the relationship types, so refer to that as well when you are building your applications:

[Yii DB Guide](#)

Also, now that we have our getRoleList method, we can use it to enforce a validation rule on the User model. We only want the model to accept role_id values that are in the range of the values in the role_value field in the role table. We can get those using the following:

```
array_keys($this->getRoleList())
```

If you recall, in getRoleList, the keys are the id records. So this statement returns 1, 2 based on what we have in our DB so far. If we add new values, they would automatically get added to the list.

So to make a rule out of it, we do the following:

```
[['role_id'], 'in', 'range'=>array_keys($this->getRoleList())],
```

The above syntax is again very intuitive. Just pop this into your rules method on the User model and you now have a range of values enforced for entries on the role_id column in your DB. This is a very common technique and we will do the same exact thing for two other attributes on the User model, which we will see shortly.

Create Status Model

Let's move on to our next model, Status. We are going to place the Status model in backend as well. So let's fire up Gii and repeat the steps we did for Role, only this time use the status table.

Again, make sure your namespace is entered as backend\models. Click on Preview, then the Generate button. If all went well, you should have a Status.php file in backend/models that looks like this:

```
<?php

namespace backend\models;

/**
 * This is the model class for table "status".
 *
 * @property integer $id
 * @property string $status_name
 * @property integer $status_value
 */

class Status extends \yii\db\ActiveRecord
{

    /**
     * @inheritdoc
     */

    public static function tableName()
    {
        return 'status';
    }

    /**
     * @inheritdoc
     */

    public function rules()
    {
        return [
            [['status_name', 'status_value'], 'required'],
            [['status_value'], 'integer'],
            [['status_name'], 'string', 'max' => 45]
        ];
    }
}
```

```

/**
 * @inheritdoc
 */

public function attributeLabels()
{
    return [
        'id' => 'ID',
        'status_name' => 'Status Name',
        'status_value' => 'Status Value'
    ];
}

}

```

And let's create the relationship to user. First add the use statement to Status.php:

```
use common\models\User;
```

Then we add the relation method.

Gist:

[Status to User Relation](#)

From book:

```

public function getUsers()
{
    return $this->hasMany(User::className(), ['status_id' => 'id']);
}

```

I won't do a lot of explaining here, this is just like the Role model. And just as we did for Role, we need to add relationship methods for Status to the User model.

Update User Model with getStatus

So, on User.php, add the use statement:

```
use backend\models>Status;
```

Now add the following 3 methods:

Gist:

User Status Relations

From book:

```
/*
 * get status relation
 *
*/
public function getStatus()
{
    return $this->hasOne(Status::className(), ['id' => 'status_id']);
}

/*
 * get status name
 *
*/
public function getStatusName()
{
    return $this->status ? $this->status->status_name : '- no status -';
}

/*
 * get list of statuses for dropdown
 */
public static function getStatusList()
{
    $droptions = Status::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'status_name');
}
```

And now let's add the validation rule for range, since we have access to getStatusList:

```
[['status_id'], 'in', 'range'=>array_keys($this->getStatusList())],
```

The order that put this in will not matter, it will observe the rule, but for readability of code, its recommended that you put this under the other rule for status like so:

```
[ 'status_id', 'default', 'value' => self::STATUS_ACTIVE],
[[ 'status_id'], 'in', 'range'=>array_keys($this->getStatusList())],
```

Add Records to Status Table

Before we forget, let's add a couple of records to the status table via PhpMyadmin, so we will be able to test functionality later.

Add the following via the insert tab on the status table:

status_name: Active

status_value: 10

status_name: Pending

status_value: 5

Column	Type	Function	Null	Value
id	smallint(6)			
status_name	varchar(45)			Active
status_value	smallint(6)			10

Column	Type	Function	Null	Value
id	smallint(6)			
status_name	varchar(45)			Pending
status_value	smallint(6)			5

Insert Into Status

When it's done, you should see the following under the browse tab on the status table:

	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Edit	Copy	Delete	id	status_name	status_value
	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Edit	Copy	Delete	1	Active	10
	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Edit	Copy	Delete	2	Pending	5

Status Records

Create UserType Model

Ok, so now we can move on to the user_type table. As we will see in a moment, this is a little bit different because there are two words in the table name separated by underscore. Gii will choose the model name for us as follows:

UserType

That is the naming convention for the model based on a table name with underscore between two words. It's possible to change this manually, but you should have a good reason for doing so, and we don't have such a reason in this case. So we are going with Gii's model name.

Now make sure namespace is set to backend\models. Then do the preview and generate steps and we should end up with UserType.php in backend/models that looks like this:

```
<?php

namespace backend\models;

use Yii;

/**
 * This is the model class for table "user_type".
 *
 * @property string $id
 * @property string $user_type_name
 * @property integer $user_type_value
 */

class UserType extends \yii\db\ActiveRecord
{
    /**
     *
}
```

```
* @inheritdoc
*/
public static function tableName()
{
    return 'user_type';
}

/**
 * @inheritdoc
 */
public function rules()
{
    return [
        [['user_type_name', 'user_type_value'], 'required'],
        [['user_type_value'], 'integer'],
        [['user_type_name'], 'string', 'max' => 45]
    ];
}

/**
 * @inheritdoc
 */
public function attributeLabels()
{
    return [
        'id' => 'ID',
        'user_type_name' => 'User Type Name',
        'user_type_value' => 'User Type Value',
    ];
}
```

```
}
```

This is another simple model along the lines of the other two we have created. So let's go ahead and add the use statement:

```
use common\models\User;
```

Now let's add the getUsers method to setup the relation to users.

Gist:

UserType Get Users

From book:

```
public function getUsers()
{
    return $this->hasMany(User::className(), ['user_type_id' => 'id']);
}
```

Update User Model with UserType

Now let's move to the User.php file in common/models to make the changes we need there:

We add the use statement at the top:

```
use backend\models\UserType;
```

And add the following methods:

Gist:

User to UserType Relations

From book:

```
/**
 *getUserType

 */

public function getUserType()
{
    return $this->hasOne(UserType::className(), ['id' => 'user_type_id']);
}
```

```


/**
 * get user type name
 *
 */

public function getUserTypeName()
{
    return $this->userType ? $this->userType->user_type_name : '- no user type -';
}

/**
 * get list of user types for dropdown
 */


public static function getUserTypeList()
{
    $droptions = UserType::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'user_type_name');
}

/**
 * get user type id
 *
 */

public function getUserId()
{
    return $this->userType ? $this->userType->id : 'none';
}

```

Ok, we did 4 methods instead of 3. The first three should look really familiar, since we did the same types on the other models, and we can quickly add the validation rule for range on user_type_id:

```
[['user_type_id'], 'in', 'range'=>array_keys($this->getUserTypeList())],
```

One thing not so obvious is the naming convention of `getUserTypeName`. The name of the method seems logical, since we always start with a lowercase g for get and then capitalize the remaining words. Where it gets tricky is in:

```
return $this->userType ? $this->userType->user_type_name : '- no user type -';
```

`$this->userType` is using a magic get method, where get is implied, and in this case, Yii 2 doesn't want you to start with an uppercase letter, the naming convention changes for this special case. This can be confusing so you have to be very careful to follow naming conventions exactly or things will break.

The fourth method `getUserTypeId` returns the id record of the `UserType`, which we will need for future use, so not much to discuss about that now.

Add Records to `user_type` Table

Let's insert two records into the `user_type` table:

`name = Free, value = 10`

`name = Paid, value = 30`

When you're done, use the browse tab to check and see if the records are there:

			id	user_type_name	user_type_value
<input type="checkbox"/>	 Edit	 Copy	 Delete	1	Free
<input type="checkbox"/>	 Edit	 Copy	 Delete	2	Paid

User Type Records

If you've been a little unclear as to what we intended for the `UserType` model, these names should give you a pretty good idea. This data structure is going to come in very handy when we want to restrict parts of the application to Paid users only.

Create Gender Model

Now we will move on to the Gender model. I want to do this model first because it's a smaller model and is related to profile, and since we have used a foreign key to connect the profile and gender tables, Gii will even create the relationship method for us.

Since Gender is closely related to Profile, I have decided to place both of those models in the frontend. I'm choosing frontend because it seems easier for me to remember that way, since profile and gender involve user choices. It's a largely cosmetic decision, since we are not following Yii 2's recommendation to place it in common.

One reason I don't do that is I like to keep common relatively small, so I can create helper classes that are easy to find. I like to put my helper classes in common, it just seems intuitive to me to do it that way. But since models are namespaced, you can put them anywhere that the application aliases can find them, so frontend, backend, or common, it does not matter.

What does matter is that you put the intended namespace in Gii correctly when you create the model, so in this case of Gender we are using:

```
frontend\models
```

If all went well, you should see a Gender.php in frontend/models that looks like this:

```
<?php

namespace frontend\models;

use Yii;

/**
 * This is the model class for table "gender".
 *
 * @property integer $id
 * @property string $gender_name
 *
 * @property Profile[] $profiles
 */

class Gender extends \yii\db\ActiveRecord
{

    /**
     * @inheritdoc
     */

    public static function tableName()
    {
        return 'gender';
    }

    /**
     * @inheritdoc
     */

    public function rules()
    {
```

```

    return [
        [['gender_name'], 'required'],
        [['gender_name'], 'string', 'max' => 45]
    ];
}

/**
 * @inheritDoc
 */

public function attributeLabels()
{
    return [
        'id' => 'ID',
        'gender_name' => 'Gender Name',
    ];
}

/**
 * @return \yii\db\ActiveQuery
 */

public function getProfiles()
{
    return $this->hasMany(Profile::className(), ['gender_id' => 'id']);
}
}

```

You can see that because of the foreign key we added, Gii was smart enough to add the relationship for us. How cool is that?

Add Records to gender Table

This is nice and complete for us and we don't have to do anything. Let's just take a moment and add 2 records to the gender table:

1 = male 2 = female

It should look like this:

		id	gender_name
<input type="checkbox"/>	Edit Copy Delete	1	male
<input type="checkbox"/>	Edit Copy Delete	2	female

Gender Records

We don't need to update User because of this model, the User model doesn't call it. Instead, it's tightly related to Profile.

So far, most of the models we've focused are models that contain data structure that every user who registers with the application must have. The role_id, status_id, and user_type_id will be all set by default when a user registers, and we have connected them to models that have a data structure that provides depth to the user.

Create Profile Model

Now we are going to create a model named Profile. All users will be able to create a profile, but a profile will only exist if the user explicitly creates it. So of course the user profile gets its own table, which we have already created.

As a reminder, we have the following columns on the profile table:

- id
- user_id
- first_name
- last_name
- birthdate
- gender_id
- created_at
- updated_at

You can get creative with this and add other attributes, we just settled on these for demonstration purposes. Just remember if you do add more to it, to add it both in the table structure and in the model.

Ok, we already decided Profile was going in the frontend, so make sure the namespace field is set:

```
frontend\models
```

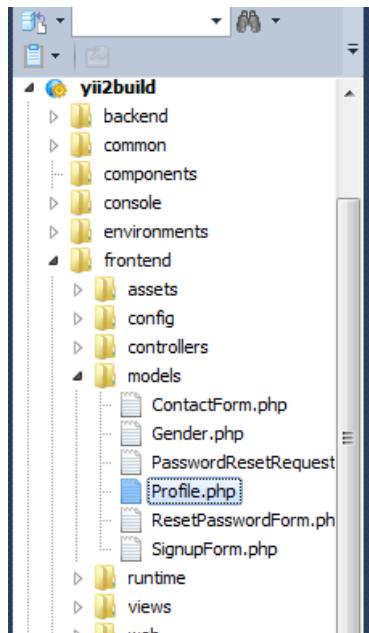
Model Generator

This generator generates an ActiveRecord class for the specified database table.

Table Name	profile
Model Class	Profile
Namespace	frontend\models
Base Class	yii\db\ActiveRecord
Database Connection ID	db

Profile Model in Gii

Now we will click through the preview and generate steps and if all has gone well, we have the following Profile.php in frontend/models. If all went well, you can see the file:



Profile in Models Directory

Note: Profile.php is in the models directory, not the search directory.

Here are the contents of the file:

```
<?php

namespace frontend\models;

use Yii;

/**
 * This is the model class for table "profile".
 *
 * @property string $id
 * @property string $user_id
 * @property string $first_name
 * @property string $last_name
 * @property string $birthdate
 * @property string $gender_id
 * @property string $created_at
 * @property string $updated_at
 *
 * @property Gender $gender
 */
class Profile extends \yii\db\ActiveRecord
{

    /**
     * @inheritdoc
     */

    public static function tableName()
    {
        return 'profile';
    }

    /**
     * @inheritdoc
     */
}
```

```
public function rules()
{
    return [
        [['user_id', 'gender_id'], 'required'],
        [['user_id', 'gender_id'], 'integer'],
        [['first_name', 'last_name'], 'string'],
        [['birthdate', 'created_at', 'updated_at'], 'safe']
    ];
}

/**
 * @inheritdoc
 */

public function attributeLabels()
{
    return [
        'id' => 'ID',
        'user_id' => 'User ID',
        'first_name' => 'First Name',
        'last_name' => 'Last Name',
        'birthdate' => 'Birthdate',
        'gender_id' => 'Gender ID',
        'created_at' => 'Created At',
        'updated_at' => 'Updated At',
    ];
}

/**
 * @return \yii\db\ActiveQuery
 */

public function getGender()
{
    return $this->hasOne(Gender::className(), ['id' => 'gender_id']);
}
```

```
}
```

By now this example of a model should look familiar, so I'm not going to explain it all again. If you need to refresh your knowledge, review the earlier models.

We need to add a few things, however. Let's start with adding the following to the use statements.

Gist:

[Profile Use Statements](#)

From book:

```
use yii\db\ActiveRecord;
use common\models\User;
use yii\helpers\Url;
use yii\helpers\Html;
use yii\helpers\ArrayHelper;
use yii\db\Expression;
```

Next up we have an addition to our rules, and don't worry, we will add the actual getGenderList method that we are calling here at the bottom of the class:

```
[['gender_id'], 'in', 'range'=>array_keys($this->getGenderList())],
```

And also, we have a rule to add to format the date correctly for birthdate:

```
[['birthdate'], 'date', 'format'=>'Y-m-d'],
```

A reader wrote in to tell me that he couldn't get the above to work unless he added the php prefix:

```
[['birthdate'], 'date', 'format'=>'php:Y-m-d'],
```

I couldn't reproduce his issue, but I thought I would mention this in case you run into a similar problem. We will do some additional formatting to the date in chapter 8, so we'll revisit the date format again in that chapter.

Moving down the class, we need to add some attribute labels for the relationship methods under the attributeLabels method, so we can use them later on our widgets throughout the app. We don't need to go too in depth at this point, other than to say we use the magic syntax of the method name and that is used by Yii::t method, which is the translate method of the app and will make the label available across the entire application:

Gist:

[Attribute Labels Addition](#)

From book:

```
'genderName' => Yii::t('app', 'Gender'),
'userLink' => Yii::t('app', 'User'),
'profileIdLink' => Yii::t('app', 'Profile'),
```

Since our Profile model deals with DATETIME on several fields, let's add the behaviors method that automatically inserts our timestamp.

Gist:

Profile Behaviors

From Book:

```
/***
 * behaviors to control time stamp, don't forget to use statement for expression
 *
 */

public function behaviors()
{
    return [
        'timestamp' => [
            'class' => 'yii\behaviors\TimestampBehavior',
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
            ],
            'value' => new Expression('NOW()'),
        ],
    ];
}
```

Let's make sure we have the appropriate use statement at the top of the file:

```
use yii\db\Expression;
```

For TimestampBehavior, we are using the full path:

```
'class' => 'yii\behaviors\TimestampBehavior'
```

So no use statement is necessary on that one.



Reminder

Just a reminder. The code in behaviors is not formatted exactly like the one you see in your IDE. The reason is that PDF and other formats break the line with a wordwrap and insert special characters that mess up the code, so I have to proactively format the code so the line doesn't break. It doesn't always look pretty, but at least the code will function.

Now we move on to relationships. The getGender relationship is already there, generated by Gii.

You can get the other ones:

Gist:

[Profile Relations](#)

From book:

```
/**
 * @return \yii\db\ActiveQuery
 */

public function getGenderName()
{
    return $this->gender->gender_name;
}

/**
 * get list of genders for dropdown
 */

public static function getGenderList()
{
    $droptions = Gender::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'gender_name');

}

/**
 * @return \yii\db\ActiveQuery
 */
```

```
public function getUser()
{
    return $this->hasOne(User::className(), ['id' => 'user_id']);
}

/**
 * @get Username
 */

public function getUsername()
{
    return $this->user->username;
}

/**
 * @getUserID
 */

public function getUserId()
{
    return $this->user ? $this->user->id : 'none';
}

/**
 * @getUserLink
 */

public function getUserLink()
{
    $url = Url::to(['user/view', 'id'=>$this->userId]);
    $options = [];
    return Html::a($this->getUserName(), $url, $options);
}
```

```

/**
 * @getProfileLink
 */

public function getProfileIdLink()
{
    $url = Url::to(['profile/update', 'id'=>$this->id]);
    $options = [];
    return Html::a($this->id, $url, $options);
}

```

The only two methods that we haven't seen an example of before are the last two. So for brevity's sake, I will skip over the ones that we already understand and focus on the new ones. However, we do need to make sure we have included the use statement at the top for the ArrayHelper class:

```
use yii\helpers\ArrayHelper;
```

We can utilize the getGenderList method to impose the restriction on values in the model like we did on the user model:

```
[['gender_id'], 'in', 'range'=>array_keys($this->getGenderList())]
```

This way, no one can add a gender id that is not valid.

Both getUserLink and getProfileIdLink utilize the Html class and the Url helper classes, so we need to make sure we have the following use statements:

```
use yii\helpers\Url;
use yii\helpers\Html;
```

Both getUserLink and getProfileIdLink do the same type of thing. They are methods that create links to the related user and to the profile id of the user. We use these in some of our UI later and its a neat way to create links that relate the models. Don't worry if you don't fully get it, you will when we work on that part.

The Complete Profile Model

You should have everything you need now for the Profile model, but just to make sure we have it all, I'm going to provide the full model for reference.

Gist:

[Full Profile Model](#)

From book:

```
<?php

namespace frontend\models;

use Yii;
use yii\db\ActiveRecord;
use common\models\User;
use yii\helpers\Url;
use yii\helpers\Html;
use yii\helpers\ArrayHelper;
use yii\db\Expression;

/**
 * This is the model class for table "profile".
 *
 * @property string $id
 * @property string $user_id
 * @property string $first_name
 * @property string $last_name
 * @property string $birthdate
 * @property integer $gender_id
 * @property string $created_at
 * @property string $updated_at
 *
 * @property Gender $gender
 */
class Profile extends \yii\db\ActiveRecord
{
    /**
     * @inheritdoc
     */
    public static function tableName()
    {
        return 'profile';
    }

    /**
     * behaviors
     */
    public function behaviors()
    {
```

```
    return [
        'timestamp' => [
            'class' => 'yii\behaviors\TimestampBehavior',
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
                ],
                ],
                'value' => new Expression('NOW()'),
                ],
                ];
}

/**
 * @inheritdoc
 */
public function rules()
{
    return [
        [['user_id', 'gender_id'], 'required'],
        [['user_id', 'gender_id'], 'integer'],
        [['gender_id'], 'in', 'range'=>array_keys($this->getGenderList())],
        [['first_name', 'last_name'], 'string'],
        [['birthdate'], 'date', 'format'=>'Y-m-d'],
        [['birthdate', 'created_at', 'updated_at'], 'safe']
    ];
}

/**
 * @inheritdoc
 */
public function attributeLabels()
{
    return [
        'id' => 'ID',
        'user_id' => 'User ID',
        'first_name' => 'First Name',
        'last_name' => 'Last Name',
        'birthdate' => 'Birthdate',
        'gender_id' => 'Gender ID',
    ];
}
```

```
        'created_at' => 'Created At',
        'updated_at' => 'Updated At',
    ];
}

/**
 * @return \yii\db\ActiveQuery
 */

public function getGender()
{
    return $this->hasOne(Gender::className(), ['id' => 'gender_id']);
}

/**
 * @return \yii\db\ActiveQuery
 */

public function getGenderName()
{
    return $this->gender->gender_name;
}

/**
 * get list of genders for dropdown
 */

public static function getGenderList()
{
    $droptions = Gender::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'gender_name');
}

/**
 * @return \yii\db\ActiveQuery
 */

public function getUser()
```

```
{  
    return $this->hasOne(User::className(), ['id' => 'user_id']);  
}  
  
/**  
 * @get Username  
 */  
  
public function getUsername()  
{  
    return $this->user->username;  
}  
  
/**  
 * @getUserID  
 */  
  
public function getUserId()  
{  
    return $this->user ? $this->user->id : 'none';  
}  
  
/**  
 * @getUserLink  
 */  
  
public function getUserLink()  
{  
    $url = Url::to(['user/view', 'id'=>$this->UserId]);  
    $options = [];  
    return Html::a($this->getUserName(), $url, $options);  
}  
  
/**  
 * @getProfileLink  
 */  
  
public function getProfileIdLink()  
{  
    $url = Url::to(['profile/update', 'id'=>$this->id]);  
    $options = [];  
    return Html::a($this->id, $url, $options);  
}
```

```
    }  
}
```

I will probably mention it a hundred times in this book, but the style of the code is set to avoid wordwrapping in PDF, so that is why it appears the way it does in the book. For a cleaner representation of the code, reference the Gist. In either case, the code has been tested and is working.

Update User Model with Profile

So now we update the user model with methods to pull in the profile relation. You should already have `getProfile`, so we won't include it here.

At the top of `User.php`, add:

```
use frontend\models\Profile;
```

Add the following methods to `User.php`:

Gist:

User Profile Relations

From book:

```
/**  
 * @getProfileId  
 *  
 */  
  
public function getProfileId()  
{  
    return $this->profile ? $this->profile->id : 'none';  
}  
  
/**  
 * @getProfileLink  
 *  
 */  
  
public function getProfileLink()
```

```
{
    $url = Url::to(['profile/view', 'id'=>$this->profileId]);
    $options = [];
    return Html::a($this->profile ? 'profile' : 'none', $url, $options);
}
```

So the getProfile method is very familiar at this point, we're simply mapping user_id on the profile table to the id column on the user table. This is how the two are associated.

The getProfileId method is a ternary statement that shows 'none' if the user does not have a profile. So when we call this method in the UI, and the user doesn't have a profile, we have an answer instead of null. Returning null when another answer is expected can lead to errors and a lot of debugging time. It's best to account for null when you can.

We need the getProfileId method to feed into our getProfileLink method in that method's Url::to method. The getProfileLink is just like the getUserLink on the profile model. It utilizes the Url helper class and the Html helper class, so we have to include the use statements at the top of the file:

```
use yii\helpers\Url;
use yii\helpers\Html;
```

Finish Up User Model

While we're here, we are going to add two more methods to User.php. These are the same kinds of methods we have already added, but they serve a specific purpose for our UI later on.

Gist:

[GetUserIdLink and GetUserLink](#)

From book:

```
/**
 * get user id Link
 */

```

```
public function getUserIdLink()
{
    $url = Url::to(['user/update', 'id'=>$this->id]);
    $options = [];
    return Html::a($this->id, $url, $options);
}
```

```

/**
 * @getUserLink
 *
 */
public function getUserLink()
{
    $url = Url::to(['user/view', 'id'=>$this->id]);
    $options = [];
    return Html::a($this->username, $url, $options);
}

```

One last bit of work on the User model. We need to add labels for all of the following methods via the attributeLabels method.

Gist:

Attribute Labels

From book:

```

/* Your model attribute labels */

public function attributeLabels()
{
    return [
        /* Your other attribute labels */

        'roleName' => Yii::t('app', 'Role'),
        'statusName' => Yii::t('app', 'Status'),
        'profileId' => Yii::t('app', 'Profile'),
        'profileLink' => Yii::t('app', 'Profile'),
        'userLink' => Yii::t('app', 'User'),
        'username' => Yii::t('app', 'User'),
        'userTypeName' => Yii::t('app', 'User Type'),
        'userTypeId' => Yii::t('app', 'User Type'),
        'userIdLink' => Yii::t('app', 'ID'),
    ];
}

```

Attribute labels tell Yii 2 how to display your attributes when they appear on the site. In some cases, as we have them used here, the attribute is the name of a method. For example, roleName is a label for the getRoleName method, which we made for this User model. It is paired with the Yii::t() method, which is the translate method for the entire app, so setting it here should make it appear this way everywhere.

If this seems a little confusing, don't worry about it now. It will become more clear when you see the attribute labels appearing in view files and widgets. I will reference it again when we come to that point.

The Complete User Model

For reference, I am going to include a copy of the User.php file, so you make sure you have everything that is necessary up to this point. This is for reference only, you should not need to copy this file. Note that it might not be in the exact order you have done yours in.

Gist:

[User Model](#)

From the Book:

```
<?php  
namespace common\models;  
  
use Yii;  
use yii\base\NotSupportedException;  
use yii\db\ActiveRecord;  
use yii\db\Expression;  
use yii\web\IdentityInterface;  
use yii\helpers\Security;  
use backend\models\Role;  
use backend\models>Status;  
use backend\models\UserType;  
use frontend\models\Profile;  
use yii\helpers\ArrayHelper;  
use yii\helpers\Url;  
use yii\helpers\Html;  
  
/**  
 * User model  
 *  
 * @property integer $id
```

```
* @property string $username
* @property string $password_hash
* @property string $password_reset_token
* @property string $email
* @property string $auth_key
* @property integer $role
* @property integer $status
* @property integer $created_at
* @property integer $updated_at
* @property string $password write-only password
*/

```

```
class User extends ActiveRecord implements IdentityInterface
{

const STATUS_ACTIVE = 1;

public static function tableName()
{
    return 'user';
}

/**
 * @inheritdoc
 */

```

```
public function behaviors()
{
    return [
        'timestamp' => [
            'class' => 'yii\behaviors\TimestampBehavior',
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
            ],
            'value' => new Expression('NOW()'),
        ],
    ];
}
```

```
}
```

```
/**
```

```
* @inheritdoc
```

```
*/
```

```
public function rules()
```

```
{
```

```
    return [
```

```
        ['status_id', 'default', 'value' => self::STATUS_ACTIVE],  
        [['status_id']], 'in', 'range'=>array_keys($this->getStatusList())),
```

```
        ['role_id', 'default', 'value' => 1],  
        [['role_id']], 'in', 'range'=>array_keys($this->getRoleList())),
```

```
        ['user_type_id', 'default', 'value' => 1],  
        [['user_type_id']], 'in', 'range'=>array_keys($this->getUserTypeList())),
```

```
        ['username', 'filter', 'filter' => 'trim'],  
        ['username', 'required'],  
        ['username', 'unique'],  
        ['username', 'string', 'min' => 2, 'max' => 255],
```

```
        ['email', 'filter', 'filter' => 'trim'],  
        ['email', 'required'],  
        ['email', 'email'],  
        ['email', 'unique'],
```

```
    ];
```

```
}
```

```
/* Your model attribute labels */
```

```
public function attributeLabels()
```

```
{
```

```
    return [
```

```
        /* Your other attribute labels */
```

```
'roleName' => Yii::t('app', 'Role'),
'statusName' => Yii::t('app', 'Status'),
'profileId' => Yii::t('app', 'Profile'),
'profileLink' => Yii::t('app', 'Profile'),
'userLink' => Yii::t('app', 'User'),
'username' => Yii::t('app', 'User'),
'userTypeName' => Yii::t('app', 'User Type'),
'userTypeId' => Yii::t('app', 'User Type'),
'userIdLink' => Yii::t('app', 'ID'),

];
}

/***
 * @inheritDoc
 */

public static function findIdentity($id)
{
    return static::findOne(['id' => $id, 'status_id' => self::STATUS_ACTIVE]);
}

/***
 * @inheritDoc
 */

public static function findIdentityByAccessToken($token, $type = null)
{
    throw new NotSupportedException
('"findIdentityByAccessToken" is not implemented.');
}

/**
 * Finds user by username
 *line break for wordwrap in pdf, should be single line
 * @param string $username
 */
```

```
* @return static|null
*/
public static function findByUsername($username)
{
    return static::findOne(['username' => $username, 'status_id' =>
        self::STATUS_ACTIVE]);
}

/**
 * Finds user by password reset token
 *
 * @param string $token password reset token
 * @return static|null
 */
public static function findByPasswordResetToken($token)
{
    if (!static::isPasswordResetTokenValid($token)) {
        return null;
    }

    return static::findOne([
        'password_reset_token' => $token,
        'status_id' => self::STATUS_ACTIVE,
    ]);
}

/**
 * Finds out if password reset token is valid
 *
 * @param string $token password reset token
 * @return boolean
 */
public static function isPasswordResetTokenValid($token)
{
    if (empty($token)) {
        return false;
    }
}
```

```
    }

    $expire = Yii::$app->params['user.passwordResetTokenExpire'];
    $parts = explode('_', $token);
    $timestamp = (int) end($parts);
    return $timestamp + $expire >= time();
}

/** 
 * @inheritDoc
 */

public function getId()
{
    return $this->getPrimaryKey();
}

/** 
 * @inheritDoc
 */

public function getAuthKey()
{
    return $this->auth_key;
}

/** 
 * @inheritDoc
 */

public function validateAuthKey($authKey)
{
    return $this->getAuthKey() === $authKey;
}

/**
```

```
* Validates password
*
* @param string $password password to validate
* @return boolean if password provided is valid for current user
*/
public function validatePassword($password)
{
    return Yii::$app->security->validatePassword($password, $this->password_hash);
}

/**
 * Generates password hash from password and sets it to the model
 *
 * @param string $password
 */
public function setPassword($password)
{
    $this->password_hash = Yii::$app->security->generatePasswordHash($password);
}

/**
 * Generates "remember me" authentication key
 */
public function generateAuthKey()
{
    $this->auth_key = Yii::$app->security->generateRandomString();
}

/**
 * Generates new password reset token
 * 2 lines to avoid wordwrapping, should be one line
 */

```

```
public function generatePasswordResetToken()
{
    $this->password_reset_token = Yii::$app->security->generateRandomString()

        . ' ' . time();

}

/***
 * Removes password reset token
 */
public function removePasswordResetToken()
{
    $this->password_reset_token = null;
}

/***
 * @getRole
 *
 */
public function getRole()
{
    return $this->hasOne(Role::className(), ['id' => 'role_id']);
}

/***
 * @getRoleName
 *
 */
public function getRoleName()
{
    return $this->role ? $this->role->role_name : '- no role -';
}
```

```
/**  
 * @getRoleList  
 */  
  
public static function getRoleList()  
{  
    $droptions = Role::find()->asArray()->all();  
    return ArrayHelper::map($droptions, 'id', 'role_name');  
}  
  
/**  
 * @getStatus  
 *  
 */  
  
public function getStatus()  
{  
    return $this->hasOne(Status::className(), ['id' => 'status_id']);  
}  
  
/**  
 * @getStatusName  
 *  
 */  
  
public function getStatusName()  
{  
    return $this->status ? $this->status->status_name : '- no status -';  
}  
  
/**  
 * @getStatusList  
 */
```

```
public static function getStatusList()
{
    $droptions = Status::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'status_name');
}

/**
 * @getProfile
 *
 */
public function getProfile()
{
    return $this->hasOne(Profile::className(), ['user_id' => 'id']);
}

/**
 * @getProfileId
 *
 */
public function getProfileId()
{
    return $this->profile ? $this->profile->id : 'none';
}

/**
 * @getProfileLink
 *
 */
public function getProfileLink()
{
    $url = Url::to(['profile/view', 'id'=>$this->profileId]);
    $options = [];
}
```

```
    return Html::a($this->profile ? 'profile' : 'none', $url, $options);
}

/**
 * @getUserType
 */

public function getUserType()
{
    return $this->hasOne(UserType::className(), ['id' => 'user_type_id']);
}

/**
 * @getUserTypeName
 *
 */
public function getUserTypeName()
{
    return $this->userType ? $this->userType->user_type_name : '- no user type -';
}

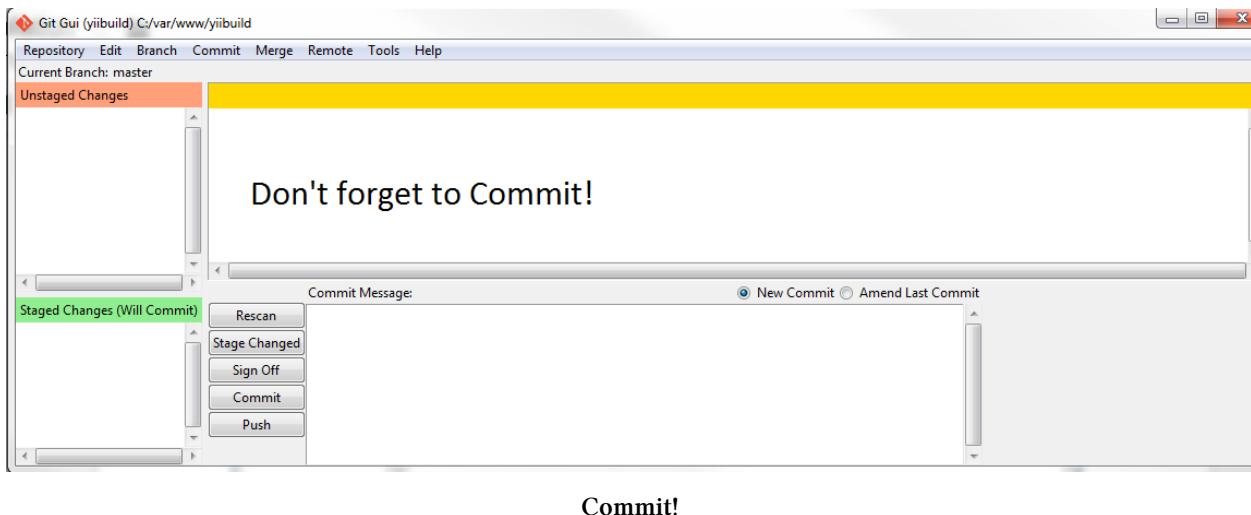
/**
 * @getUserTypeList
 */
public static function getUserTypeList()
{
    $droptions = UserType::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'user_type_name');
}

/**
 * @getUserTypeId

```

```
*  
*/  
  
public function getUserId()  
{  
    return $this->userType ? $this->userType->id : 'none';  
}  
  
/**  
 * @getUserIdLink  
 *  
 */  
  
public function getUserIdLink()  
{  
    $url = Url::to(['user/update', 'id'=>$this->id]);  
    $options = [];  
    return Html::a($this->id, $url, $options);  
}  
  
/**  
 * @getUserLink  
 *  
 */  
  
public function getUserLink()  
{  
    $url = Url::to(['user/view', 'id'=>$this->id]);  
    $options = [];  
    return Html::a($this->username, $url, $options);  
}  
}
```

Summary



Commit!

This chapter was a beast. We created 5 new tables to add to our data structure. We created 5 new models and updated the User model. We haven't done that much custom coding yet, we primarily relied on Gii to create our models for us. Then we added relationship methods, rules, use statements, behaviors and various other odds and ends to unlock the power of Yii 2.

A lot of the relationship methods we added will go a long way towards building an intuitive UI that allows us to manage users and control their access to various parts of the application. We are building the application with eye towards code reuse and extensibility. We want to make a template that would be a good starting point for any application.

We haven't seen how it all translates into an application yet, but don't worry we will. And you'll get to see just how easy Yii 2 makes this for us.

Chapter Six: Helpers

You might have noticed by now that this book is organized around key concepts, not work flow, at least up to this point. The reality of workflow is that you jump around quite a bit between models, controllers and views, and this can get really confusing when you are new to the framework.

For example, we looked at the User model extensively, but we still really don't know much about how the user logs in or even how the models we created in the previous chapter will support that. But don't worry, it's coming. We are laying the foundation for everything and it will all come together. And by the time we are done, you will have a working template, with a working user model with full admin control via UI that you can extend into a robust application.

This chapter is dedicated to helpers, special classes we create for formatting and returning certain values. This might sound trivial, but these helpers are going to rapidly accelerate our development cycle once we have them.

If you think about it, a framework is one giant set of helper classes, and Yii 2 certainly fits this description. But no matter how much framework there is, each individual programmer has a need for their own helpers. These are classes that will help you move development along quickly with reusable code that you have written yourself.

We've already talked about the fact we're planning to control user access. So we could ask ourselves some key questions. How will the controller know who has a role of Admin? How will it know the status or the user_type_name of the user? When someone clicks on the profile link in navigation, how will the controller know whether they already have a profile or if they need to create one?

In order to control access based on these types of things, we need to be able to extract the values we want succinctly and easily. To help make things easier on myself, I created a number of methods that will return the values I will need for more complex operations.

Value Helpers

I could have put all of these methods in a single Utilities or Helpers class, but recently I read Clean Code by Robert Martin, and one of the results of reading books on programming is that you try to adopt the principles that you like. In this case, I really like the idea of getting semantic help from class names and method names. So I decided to go for smaller helper classes that were more descriptive.

For example, we could have a class named ValueHelpers. We could use it to return values with methods named as follows:

- roleMatch - see if the users role matches the specified role

- `getRoleValue` - find the value of a role
- `getUsersRoleValue` - find the value of specific user's role
- `isRoleNameValid` - confirm we have a valid role name
- `statusMatch` - determine if the current user's status matches the desired status
- `getStatusId` - return the id of status based inputting the status name.
- `userTypeMatch` - determine if the current user's user type matches the desired type

These methods are going to be the building blocks of our access control, which will help us build a production-ready template.

I could have provided a much simpler solution or simply relied on Yii 2's RBAC, but I didn't do that because that's not a fit for this template. I want to provide a robust enough foundation for us to be able to continue to build upon and improve as our needs evolve.

The downside is that this is a bit more complicated than just giving a quick 'this is how it works tour of Yii 2.' The upside is that you get closer to the code that is going to manage your access control, and therefore, when you need to customize it in the future in some way we can't anticipate now, you will have a much better understanding on how to do it.

Also, as we go along creating our helper methods, we will get use Yii 2's Active Record implementation to find and return results. We will also be using raw sql in some cases, so you get to see the contrast between the different ways of accessing data from your database.

Let's go ahead and create `ValueHelpers.php` in the `common/models` folder. For the sake of consistency and to not have to bounce around all over the place, I'm simply going to give you the entire class. Then we will discuss the methods.

Gist:

[ValueHelpers](#)

From book:

```
<?php
namespace common\models;

use yii;
use backend\models\Role;
use backend\models>Status;
use backend\models\UserType;
use common\models\User;

class ValueHelpers
{

    public static function roleMatch($role_name)
```

```
{  
  
$userHasRoleName = Yii::$app->user->identity->role->role_name;  
  
    return $userHasRoleName == $role_name ? true : false;  
  
}  
  
public static function getUsersRoleValue($userId=null)  
{  
  
    if ($userId == null){  
  
        $usersRoleValue = Yii::$app->user->identity->role->role_value;  
  
        return isset($usersRoleValue) ? $usersRoleValue : false;  
  
    } else {  
  
        $user = User::findOne($userId);  
  
        $usersRoleValue = $user->role->role_value;  
  
        return isset($usersRoleValue) ? $usersRoleValue : false;  
  
    }  
  
}  
  
public static function getRoleValue($role_name)  
{  
  
    $role = Role::find('role_value')  
        ->where(['role_name' => $role_name])  
        ->one();  
  
    return isset($role->role_value) ? $role->role_value : false;  
  
}  
  
public static function isRoleNameValid($role_name)
```

```
{  
  
    $role = Role::find('role_name')  
        ->where(['role_name' => $role_name])  
        ->one();  
  
    return isset($role->role_name) ? true : false;  
  
}  
  
  
public static function statusMatch($status_name)  
{  
  
    $userHasStatusName = Yii::$app->user->identity->status->status_name;  
  
    return $userHasStatusName == $status_name ? true : false;  
  
}  
  
public static function getStatusId($status_name)  
{  
  
    $status = Status::find('id')  
        ->where(['status_name' => $status_name])  
        ->one();  
  
    return isset($status->id) ? $status->id : false;  
  
}  
  
public static function userTypeMatch($user_type_name)  
{  
  
    $userHasUserTypeName =  
        Yii::$app->user->identity->userType->user_type_name;  
  
    return $userHasUserTypeName == $user_type_name ? true : false;  
  
}
```

Please don't worry about memorizing these methods. It's not necessary for you to do that. We are however going to review them in detail, so it will give you more of an overview of how all this will work and to introduce you to Yii 2's Active Record.

Anyway, you can see we have 7 fairly simple methods that return values associated with the models we created.

So for example, if I want to know the value of Admin, I can call:

```
getRoleValue('Admin');
```

And according to what's in our DB:

Role Records				
		id	role_name	role_value
<input type="checkbox"/>		Edit		Copy
<input type="checkbox"/>		Edit		Copy
		1	User	10
		2	Admin	20

The answer is 20. So when I want to control access to the backend, for example, I have written a method to restrict access to users that have a minimum role value of 20. Or I have one to match 20 exactly if I wanted to just have one kind of role have access.

We are not making that decision or building that method yet, we are just anticipating that we need the value for decisions we will make in the future.

One thing you might want to know is what is the purpose of `role_value`, why not just use the `id` primary key? The reason I added a `role_value` column, however, was to give us more flexibility in the design of the system. You will see how that plays out later.

You'll also notice that our methods are public static methods, which means they can be called as so:

```
ValueHelpers::getRoleValue('Admin');
```

As long as you have included the use statement:

```
use common\models\ValueHelpers;
```

at the top of your file, you can use that method like that.

We've done our best with the helper methods to make them as semantically pleasing and easy to understand as possible. This makes understanding the purpose easier when you come back to it after a period of time.

These are simple methods, but let's step through them. Here is the first one:

```
public static function roleMatch($role_name)
{
    $userHasRoleName = Yii::$app->user->identity->role->role_name;

    return $userHasRoleName == $role_name ? true : false;
}
```

We're calling it roleMatch as if posing a question, so it makes sense that it returns true or false.

We could use that as follows:

```
if(ValueHelpers::roleMatch('Admin'){

    //then allow the user to do this

}
```

For the method to do that calculation, we rely on the relationship between the User and Role models in the following:

```
Yii::$app->user->identity->role->role_name;
```

So, first thing to note about that is the user attributes are always available to us via a call like so:

```
Yii::$app->user->identity->username;
```

That will return the username of the current user, who would have to be logged in for this to work. You also need:

```
use yii;
```

That must go at the top of any file in which you wish to access the user with the Yii::\$app.

So, getting back to:

```
Yii::$app->user->identity->role->role_name;
```

We are using magic get syntax for the getRole method on the User model, so that is why you see ->role instead of ->getRole(). Once the models are tied together via the relationship, as we put in place in the previous chapter, we can access the properties of the related model, in this case, it's role_name, which belongs to Role.

You can see how much power is packed into one line of code. It's just incredible. This makes the framework a pleasure to work with.

To keep the lines short, especially for this book, I'm assigning that to a variable:

```
$userHasRoleName = Yii::$app->user->identity->role->role_name;
```

From there we are doing a simple equal comparison via ternary to see if it matches the role name that was handed in and we are done:

```
return $userHasRoleName == $role_name ? true : false;
```

Nice clean intuitive syntax, and that makes it very maintainable.

Ok, next method:

```
public static function getUsersRoleValue($userId=null)
{
    if ($userId == null){

        $usersRoleValue = Yii::$app->user->identity->role->role_value;

        return isset($usersRoleValue) ? $usersRoleValue : false;
    } else {

        $user = User::findOne($userId);

        $usersRoleValue = $user->role->role_value;

        return isset($usersRoleValue) ? $usersRoleValue : false;
    }
}
```

By setting \$userId to null in the signature, we are giving ourselves the option of handing in a value for that or not. In the case of a logged in user, we don't need to hand in the \$userId, we already have it.

So in that case, we get:

```
if ($userId == null){  
  
$usersRoleValue = Yii::$app->user->identity->role->role_value;  
  
return isset($usersRoleValue) ? $usersRoleValue : false;
```

Again the relationship between User and Role is already there and easy for us to use. We check to see if \$usersRoleValue is set, and if not return false. Otherwise return \$usersRoleValue.

But in some cases, the user might not be logged in, in which case we need to hand in the user id. It's just slightly more complicated. We have to create an ActiveRecord instance of the user, so we can use the relationship to find the \$usersRoleValue:

```
$user = User::findOne($userId);
```

That translates roughly into SQL:

```
Select id FROM User WHERE id = $userId;
```

Obviously ActiveRecord formatting makes setting up a query easy. The Yii 2 guide has many examples on how to use ActiveRecord.

So rather than reproducing what's in the Yii 2 guide, I will refer you to the guide with the suggestion you take a few minutes to look it over:

Yii 2 Active Record

Don't worry though, I won't leave you stranded. I will continue to explain everything we use fully. Once we create an instance of the User, we use our relationship to role to finish this up:

```
$usersRoleValue = $user->role->role_value;  
  
return isset($usersRoleValue) ? $usersRoleValue : false;
```

So we had to jump through a few hoops to get our users role value, but it's worth it. It will give us flexibility in building out the rest of our access control. Anyway by the time we are using our helper methods in our controllers, you will be very happy we did all this.

Next, we simply want a role value without the user. We hand in the name of the role and use an instance of ActiveRecord to access it:

```
public static function getRoleValue($role_name)
{
    $role = Role::find('role_value')
        ->where(['role_name' => $role_name])
        ->one();

    return isset($role->role_value) ? $role->role_value : false;
}
```

In this ActiveRecord call, I switched to using the find method instead of findOne because it's not a primary key and findOne is a shortcut for searching on a primary key.

Also, just as a heads up, in order to use ActiveRecord to access results, you have to include use statements for the models you want to return. You can see we included the following at the top of the file:

```
use yii;
use backend\models\Role;
use backend\models>Status;
use backend\models\UserType;
use common\models\User;
```

Next we have a quick check to see if the role name is valid, this helps prevent programming errors:

```
public static function isRoleNameValid($role_name)
{
    $role = Role::find('role_name')
        ->where(['role_name' => $role_name])
        ->one();

    return isset($role->role_name) ? true : false;
}
```

In implementing access control, having control over a user's status is also important:

```
public static function statusMatch($status_name)
{
    $userHasStatusName = Yii::$app->user->identity->status->status_name;

    return $userHasStatusName == $status_name ? true : false;
}
```

The statusMatch method is exactly like the roleMatch method at the top of the class, using the same technique with relationships, so review that method if you are uncertain on how this works.

Next:

```
public static function getStatusId($status_name)
{
    $status = Status::find('id')
        ->where(['status_name' => $status_name])
        ->one();

    return isset($status->id) ? $status->id : false;
}
```

Here we are just handing in the name of status to return via ActiveRecord the id. This will come in handy when we want to remove the status constant from the user model. But don't worry about that now.

And finally, we have:

```
public static function userTypeMatch($user_type_name)
{
    $userHasUserTypeName = Yii::$app->user->identity->userType->user_type_name;

    return $userHasUserTypeName == $user_type_name ? true : false;
}
```

Like roleMatch and statusMatch, it simply returns true or false if the user's userType matches the one we specify in the signature.

Permission Helpers

As we imagine how we would use our helpers, knowing value isn't enough. If we are going to control access to areas of the application, we would want helpers that use value and define permission. So we are going to create a PermissionHelpers class in common/models.

Go ahead and create PermissionHelpers.php in common/models.

Gist:

[PermissionHelpers](#)

From book:

```
<?php
namespace common\models;

use common\models\ValueHelpers;
use yii;
use yii\web\Controller;
use yii\helpers\Url;

class PermissionHelpers
{

    public static function requireUpgradeTo($user_type_name)
    {

        if (!ValueHelpers::userTypeMatch($user_type_name)) {

            return Yii::$app->getResponse()->redirect(Url::to(['upgrade/index']));
        }
    }

    public static function requireStatus($status_name)
    {

        return ValueHelpers::statusMatch($status_name);
    }

    public static function requireRole($role_name)
```

```
{  
  
    return ValueHelpers::roleMatch($role_name);  
  
}  
  
public static function requireMinimumRole($role_name, $userId=null)  
{  
  
    if (ValueHelpers::isRoleNameValid($role_name)){  
  
        if ($userId == null) {  
  
            $userRoleValue = ValueHelpers::getUsersRoleValue();  
  
        } else {  
  
            $userRoleValue = ValueHelpers::getUsersRoleValue($userId);  
  
        }  
  
        return $userRoleValue >=  
        ValueHelpers::getRoleValue($role_name) ? true : false;  
  
    } else {  
  
        return false;  
  
    }  
  
}  
  
public static function userMustBeOwner($model_name, $model_id)  
{  
  
    $connection = \Yii::$app->db;  
  
    $userid = Yii::$app->user->identity->id;  
  
    $sql = "SELECT id FROM $model_name  
           WHERE user_id=:userid AND id=:model_id";
```

```
$command = $connection->createCommand($sql);
$command->bindValue(":userid", $userid);
$command->bindValue(":model_id", $model_id);

if($result = $command->queryOne()) {

    return true;

} else {

    return false;

}

}
```

Ok, great, only 5 methods and we will see how they utilize ValueHelpers to help us control access in ways that will be important to us. We're not going too deep, however, we will get into more detail when we actually use the methods in the application.

Remember, all of these helper methods are public static so they can be called like so:

```
PermissionHelpers::requireUpgradeTo('Paid');
```

And so we jumped right into our first example. If you do build an application that has an area for users of a different type, like in the example above, Paid users, then this is perfect for your controller. It tests the current user to see if their user type matches what you handed into the method.

It does this using our handy ValueHelpers::userTypeMatch method. So already we are reusing code from our ValueHelpers class, you have to love that.

Ok, back to the method. If the user type matches, fine, continue, if not, redirect to upgrade page. Of course if you have a different destination in mind, just put the controller/action in the method.

If you have more than one upgrade or redirect page, you could rewrite the method to take a second argument, such as:

```
public static function requireUpgradeTo($user_type_name, $redirect_destination)
{
    if (!ValueHelpers::userTypeMatch($user_type_name)) {

        return Yii::$app->getResponse()->redirect(Url::to([$redirect_destination]));
    }
}
```

In that case, you would just hand in the controller action as a string ‘upgrade/salespitch’ as an example, in the second argument. Anyway, I only included the method that has the redirect hardcoded in because I’m not anticipating the application being more complicated than that and I like the simpler syntax of the single argument.

The next 2 methods, requireStatus and requireRole simply call existing statusMatch and roleMatch ValueHelper methods and are not absolutely necessary, since they don’t do anything new.

The reason they exist is so that I can have a cosmetic and syntactic consistency that makes it easy for me to work with my access control methods.

You can see the theme emerging of PermissionHelpers::requireSomething as a permission manager. It’s just super simple for me to remember, work with, and maintain.

Anyway, both of those methods require an exact match to return true.

Next, I did a requireMinimumRole method with the `>=` operator, so if you wanted for example Admin and SuperUser to be able to access the backend, you could control access to the backend with this method.

First we check to see isRoleNameValid:

```
if (ValueHelpers::isRoleNameValid($role_name)){
```

if that fails, immediately return false. If we’re good, we then check to see if we have handed in a \$userId:

```
if ($userId == null) {

    $userRoleValue = ValueHelpers::getUsersRoleValue();
```

If it’s null, call getUsersRoleValue without handing in the \$userId and set the result to \$userRoleValue. This assumes the user is logged in and the getUsersRoleValue method will handle that scenario.

If it's not null, and we handed in a value for \$userId, then getUsersRoleValue method will handle that scenario as well.

Then we just do a simple check with `>=` to see if the user meets the requireMinimumRole that we handed in:

```
return $userRoleValue >= ValueHelpers::getRoleValue($role_name) ? true : false;
```

This one had a few moving parts, which might be harder to follow, but if you want to call it, look how intuitive the syntax is:

```
PermissionHelpers::requireMinimumRole('Admin');
```

Typically, you are going to use that as part of an if statement, since it returns true or false.

Even though we haven't completely explained it, you are starting to get some idea of how we are answering the question, "How we will control access to the backend?"

Don't worry if you don't completely get it now, you will when you see it in action.

The last method, userMustBeOwner, takes 2 parameters, model name and model id. Then it performs a query to see if the current user is the owner of that specific record.

You can use the method as an example of how to do a raw query:

```
public static function userMustBeOwner($model_name, $model_id)
{
    $connection = \Yii::$app->db;
    $userid = Yii::$app->user->identity->id;
    $sql = "SELECT id FROM $model_name
            WHERE user_id=:userid AND id=:model_id";
    $command = $connection->createCommand($sql);
    $command->bindValue(":userid", $userid);
    $command->bindValue(":model_id", $model_id);

    if($result = $command->queryOne()) {
        return true;
    } else {
        return false;
    }
}
```

If the user owns the record, it returns true, if not returns false. When using it, the syntax would look typically like this:

```
if (PermissionHelpers::userMustBeOwner ('profile', $model->id)) {  
    //do something  
}
```

So how would you use this? Well, let's say you have a group of posts or other records that are visible to everyone, but only the author can update or delete them and you want to make the navigation visible only to the owner of the record. So the "do something" in the above example, could be show the navigation in a view file. We will use this exact example later in the book.

I wrote it this way because I like the syntax and I felt like this would be a good way to work with it. But keep in mind there are typically many ways to accomplish the same thing and a helper is not always necessary. I like to use them because it also gives me consistency in the coding, but this is definitely one area where you have to use your own judgement.

Record Helpers

Ok, so we have one last helper file, RecordHelpers. Let's go ahead and create RecordHelpers.php in common/models and put the following contents in the file:

Gist:

[RecordHelpers](#)

From book:

```
<?php  
  
namespace common\models;  
  
use yii;  
  
  
class RecordHelpers  
{  
  
    public static function userHas($model_name)  
    {  
        $connection = \Yii::$app->db;
```

```

$userid = Yii::$app->user->identity->id;
$sql = "SELECT id FROM $model_name WHERE user_id=:userid";
$command = $connection->createCommand($sql);
$command->bindValue(":userid", $userid);
$result = $command->queryOne();

if ($result == null) {

    return false;

} else {

    return $result['id'];

}

}

```

This class has only one method. What I have planned for our application is a user profile and I want a Profile link that when you click on it, figures out whether or not the user has a profile or if they need to create one.

I wanted to keep the syntax in my controller very intuitive and have the result formatted to either false or the record id. That way if it comes back false, I can have the user create the record, and if it returns the id of the record because the user already has one, I can render that view. Something like:

```

If ($already_exists = RecordHelpers::userHas('profile')) {

// show profile with id with value of $already_exists

} else {

// go to form to create profile

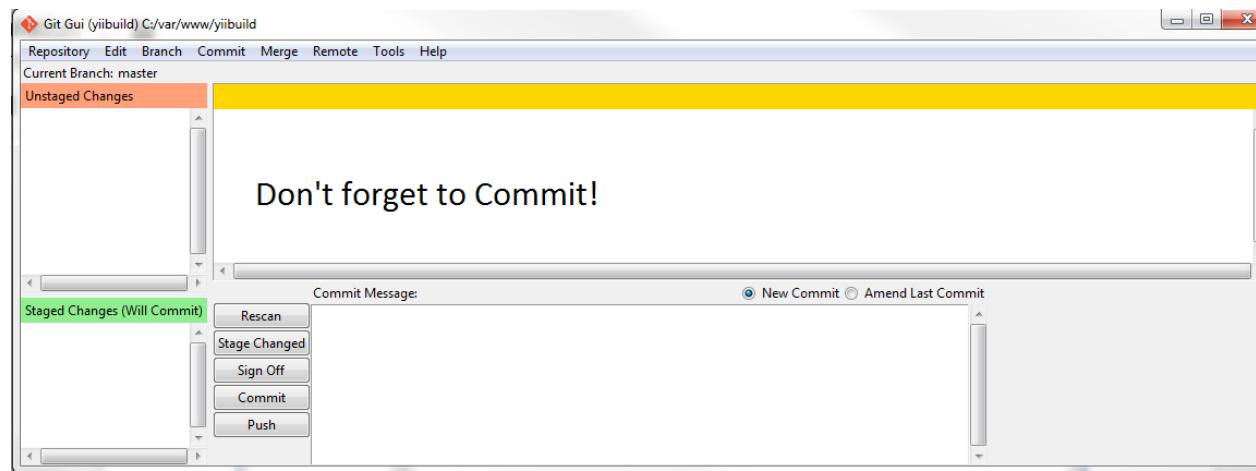
}

```

This kind of syntax makes it incredibly easy to understand what is happening here. If the if statement returns a record id, show the profile with that record id, which is now referenced by the variable \$already_exists. If it comes back false, go to the create form.

I also wrote it so I could use it with other models, I just need to hand in the model name as string. You should note that this method is written to return a single record, you would have to modify it if there were a possibility that the user could have multiple records, multiple profiles for example.

Summary



Ok, by now you are realizing that learning Yii 2 is fun, but also is a lot of work. This is a huge framework, elegant and powerful, capable of doing so many things. We've done a lot already. We set up the application, we briefly reviewed the MVC architecture, and we modified the User model. We also built 5 new models and put in place their relationships to each other and their relationship to the User model, and we built a number of helper methods to make coding easier when we dig further into the application, especially for access control.

We've done all of this, and yet our application currently does nothing more than it did when you installed it. Thank you for your patience. In the next chapters, we will begin adding features to our application.

Chapter Seven: Site Controller

We're going to continue our development with a thorough look at Site Controller and it's related views. This will introduce us to a broad number of concepts within Yii 2's controllers and models, so we can gain knowledge of how they work.

We will bounce around a bit, so don't worry if you are not instantly memorizing all of this information. Think of it as a guided tour, where we get introduced to a knowledge base that we can refer to and build upon, further enhancing our feel for how Yii 2 operates.

The first thing we should mention is that there are two site controllers, one for backend and one for frontend.

We will start by discussing the frontend site controller, which will take us through registration and login, then we'll move on to the backend site controller, pointing out the differences. And then we will add our first new functionality since we installed the advanced template. We will get the backend site controller to enforce a different level of access for the backend login.

Ok, let's get started. We will look at frontend/controllers/SiteController.php in chunks, no need to reproduce the entire file, since we aren't changing anything.

First up, namespace and use statements:

```
<?php  
namespace frontend\controllers;  
  
use Yii;  
use common\models\LoginForm;  
use frontend\models\PasswordResetRequestForm;  
use frontend\models\ResetPasswordForm;  
use frontend\models\SignupForm;  
use frontend\models>ContactForm;  
use yii\base\InvalidArgumentException;  
use yii\web\BadRequestHttpException;  
use yii\web\Controller;  
use yii\filters\VerbFilter;  
use yii\filters\AccessControl;
```

It uses quite a few models and we will see this in action. Next we have class declaration:

```
class SiteController extends Controller
```

You can see it extends Controller. When you have time, browse through Controller, it will give you a better idea of how things work, but be forewarned, the framework code is sometimes hard to follow, especially for beginners. The code you see on the surface is much friendlier than what you will see below.

Behaviors

Next we have something familiar, a behaviors method. We saw those on some of our models with TimeStamp behavior. Our controllers use AccessControl behavior:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'only' => ['logout', 'signup'],
            'rules' => [
                [
                    'actions' => ['signup'],
                    'allow' => true,
                    'roles' => ['?'],
                ],
                [
                    'actions' => ['logout'],
                    'allow' => true,
                    'roles' => ['@'],
                ],
            ],
            [
                'verbs' => [
                    'class' => VerbFilter::className(),
                    'actions' => [
                        'logout' => ['post'],
                    ],
                ],
            ],
        ];
}
```

This is Yii 2's out-of-the-box method for controlling access to the site, and it mainly controls logged in or guest. The '?' is guest and '@' is logged in. So now that we know that, we can see Yii 2's incredibly intuitive syntax at work. But let's break it down just to be sure. First thing:

Name the behavior:

```
return [
  'access' => [
```

This is called access, but you could call it any string and it would work the same. Next comes class:

```
'class' => AccessControl::className(),
```

This is just telling us what class to apply. Next we see what actions to apply the behavior to:

```
'only' => ['logout', 'signup'],
```

So these rules are only going to apply to logout and signup. Next come the rules, and this is where we can apply them to specific actions:

```
[

  'actions' => ['signup'],
  'allow' => true,
  'roles' => ['?'],

],
[
  'actions' => ['logout'],
  'allow' => true,
  'roles' => ['@'],
],
```

So, the first rule says signup, true, guest, in other words, guests are allowed to signup. Second rule says, logout, true, logged in user, in other words logged in users are allowed to access the logout action. Since we specified in the 'only' part of the method, all other actions are not controlled by these rules.

As we go through the other actions, we will see why this make sense.

Actions

The first method below the behaviors is actions:

```
public function actions()
{
    return [
        'error' => [
            'class' => 'yii\web\ErrorAction',
        ],
        'captcha' => [
            'class' => 'yii\captcha\CaptchaAction',
            'fixedVerifyCode' => YII_ENV_TEST ? 'testme' : null,
        ],
    ];
}
```

This is a configuration method, telling us which class to use for error and which class to use for Captcha. The configuration in actions makes these actions available to the controller. So if you want to use Captcha for something, you need it configured in the controller, like it is above. We also need to set it up as a widget, which we will see in action soon.

Index Action

Let's move on to actionIndex:

```
public function actionIndex()
{
    return $this->render('index');
}
```

Yes! Things had to get simpler sooner or later... This simply calls the render method to the view, in this case 'index'.

This gives us an opportunity to refresh ourselves on how the routing works. The route to index looks like this:

`yii2build.com/index.php?r=site/index`

Since we left the ugly urls in place, it's very explicit. Index.php is the bootstrap page, everything goes through that doorway. The r for route, = site/index. In this case, site is the controller, index is the action. If you leave the action off a controller, it will look for an index action and default to it. If there is no index action, it will return an error.

Also a quick note, the default to the site is set to the one above, so if you just type in the domain, `yii2build.com`, that is the route you will get.

The action in most cases will render a view, using the syntax we see on the index action above, and that is how we get to see the page. At any rate, you should have a sense now how the controller/action moves us through the site.

Login Action

The next method on the controller is actionLogin:

```
public function actionLogin()
{
    if (!\Yii::$app->user->isGuest) {
        return $this->goHome();
    }

    $model = new LoginForm();

    if ($model->load(\Yii::$app->request->post()) && $model->login()) {

        return $this->goBack();

    } else {
        return $this->render('login', [
            'model' => $model,
        ]);
    }
}
```

Ok, great, we'll see how the user logs in. The first thing that happens is the test to see if they are already logged in:

```
if (!\Yii::$app->user->isGuest) {
    return $this->goHome();
}
```

It does this by checking to see if user is not guest. If they are not a guest, that means they are already logged in and in this case, we send them to the homepage.

Login Form Model

If not logged in, we create a new instance of LoginForm:

```
$model = new LoginForm();
```

We spoke a little about the LoginForm model, when we were on Modifying the User model, but it bears taking a closer look, so we can understand exactly how this works. It's located in common/models, so that is why the first block looks like:

```
<?php  
namespace common\models;  
  
use Yii;  
use yii\base\Model;
```

Then we get our class declaration and class properties:

```
class LoginForm extends Model  
{  
    public $username;  
    public $password;  
    public $rememberMe = true;  
  
    private $_user = false;
```

Remember, this model extends Model, not User, so we have to declare the properties. We are defaulting \$rememberMe to true, this sets the flag on the form for the cookie. We default \$_user to false and we'll see why in a moment.

Next we have a rules method:

```
public function rules()  
{  
    return [  
        // username and password are both required  
        [['username', 'password'], 'required'],  
        // rememberMe must be a boolean value  
        ['rememberMe', 'boolean'],  
        // password is validated by validatePassword()  
        ['password', 'validatePassword'],  
    ];  
}
```

Yii 2 has provided comments that explain the validators we are using. You can see password uses the validatePassword method as it's validator:

```

public function validatePassword($attribute, $params)
{
    if (!$this->hasErrors()) {
        $user = $this->getUser();
        if (!$user || !$user->validatePassword($this->password)) {
            $this->addError($attribute, 'Incorrect username or password.');
        }
    }
}

```

Fairly intuitive method, if no errors, great get the user, otherwise declare error.

Next we have the login method itself as of Yii 2.0.3:

```

/**
 * 2 lines in return statement to avoid wordwrap
 */

public function login()
{
    if ($this->validate()){
        return Yii::$app->user->login($this->getUser(),
        $this->rememberMe ? 3600 * 24 * 30 : 0);

    } else {
        return false;
    }
}

```

Ok, slightly more complicated. \$this->validate() is calling the validate method of Model. and we also call getUser from the LoginForm model, which we will see in a moment. If we validate, we return the login method of user available to us from the Yii::\$app.

This is actually a reference the User model in vendor/yiisoft/yii/web/User. This is the class Yii 2 uses to manage identity and login and there is a rather complicated login method there. It gets a little confusing to have multiple models and multiple methods with the same name, but that is the nature of the beast. We can see that it takes the in the user and remember me setting in the signature and then logs in. I won't go further into that login method since it's beyond the scope of this discussion and definitely not for beginners, but you can check it out for yourself if you wish.

At least it makes the code in the LoginForm model seem very intuitive by comparison. Anyway, it works, and it logs in the user and sets the remember me cookie if that flag has been set in the form.

If something fails, it returns false, and usually validation messages will be sent to the view telling the user what the problem is. On the surface, it's very simple.

Ok, last method of this model:

```

public function getUser()
{
    if ($this->_user === false) {
        $this->_user = User::findByUsername($this->username);
    }

    return $this->_user;
}

```

Since we know the private model property `$_user` defaults to false, the condition in the if statement is going to be met if this method has not already been run. So if there is no username in `$_user`, then it uses a static method of the User model to return a model instance of the desired user and set it to `$_user`.

In order for this to work, the LoginForm model obviously has to get the values for its properties from the post, so it knows who `$this->username` refers to, and it can look up the user and set it to `$_user`. So let's see how we get the post data by returning now to the `actionLogin` method of the SiteController and picking up where we left off. So after calling a new instance of the LoginForm model, we get:

```

if ($model->load(Yii::$app->request->post()) && $model->login()) {
    return $this->goBack();
} else {
    return $this->render('login', [
        'model' => $model,
    ]);
}

```

It will get its post data from:

```
Yii::$app->request->post()
```

If we can load the post data, which will validate according to the model as we described earlier and if it can utilize the model's `login` method, it will return the user to whatever page they were on using:

```
return $this->goBack();
```

Only now they will be in a logged in state.

Otherwise, if something fails or we have not yet posted the form, we will display the form:

```

} else {
    return $this->render('login', [
        'model' => $model,
    ]);
}

```

You can also see here that it's passing an instance of \$model to the view, which we know in this case is LoginForm model that we set earlier with:

```
$model = new LoginForm();
```

That makes the model available to the view. And that's it for login, hopefully you got a good understanding of how that works.

Logout Action

The actionLogout method is significantly simpler:

```
public function actionLogout()
{
    Yii::$app->user->logout();

    return $this->goHome();
}
```

It uses the logout method of the user model buried deep in the bowels of Yii 2 and sends the user to the index page via goHome().

Contact Action

The next method routes us to a simple contact page, with the actionContact method, but there are some interesting things in here.

```
public function actionContact()
{
    $model = new ContactForm();
    if ($model->load(Yii::$app->request->post()) && $model->validate()) {

        if ($model->sendEmail(Yii::$app->params['adminEmail'])) {
            Yii::$app->session->setFlash('success',
                'Thank you for contacting us. We will respond to you as soon as possible.');

        } else {
            Yii::$app->session->setFlash('error', 'There was an error sending email.');
        }
    }
}
```

```
    }

    return $this->refresh();

} else {

    return $this->render('contact', [
        'model' => $model,
    ]);
}

}
```

Contact Form Model

The first thing actionContact method on the controller does is call a new form model, ContactForm, located in frontend/models. It's another form model that extends Model, so let's step through it:

```
<?php

namespace frontend\models;

use Yii;
use yii\base\Model;

/**
 * ContactForm is the model behind the contact form.
 */
class ContactForm extends Model
{

    public $name;
    public $email;
    public $subject;
    public $body;
    public $verifyCode;

    /**
     * @inheritdoc
     */
}
```

```
public function rules()
{
    return [
        // name, email, subject and body are required
        [['name', 'email', 'subject', 'body'], 'required'],
        // email has to be a valid email address
        ['email', 'email'],
        // verifyCode needs to be entered correctly
        ['verifyCode', 'captcha'],
    ];
}
```

So we have the namespace, the use statements, the class properties, and the first method, rules.

Notice we have an attribute verifyCode. We will use the captcha validator on this attribute. If you recall, the captcha class was configured into the controller through the actions method, so it is available as a controller action. Cool stuff.

Captcha

Let's take a minute to discuss captcha in detail, since it's a very useful feature to include in your applications. The good news is that Yii 2 makes this very easy to implement.

Using it on SiteController is default behavior, so there is a little more to it if you use on a different controller. However, it's still very simple.

If you need to implement captcha on anywhere else on your future applications, these are the steps for implementing captcha:

Step 1. Configure captcha into the new controller via actions method.

```
public function actions()
{
    return [
        'error' => [
            'class' => 'yii\web\ErrorAction',
        ],
        'captcha' => [
            'class' => 'yii\captcha\CaptchaAction',
            'fixedVerifyCode' => YII_ENV_TEST ? 'testme' : null,
        ],
    ];
}
```

Step 2. Include a captcha validator on form model rules with extra parameter for controller/captcha. This is not necessary when using it with SiteController, but in all other cases, we must define it.

For example let's say we want to use it on the contact action of a PagesController instead of SiteController. Let's imagine we are using the existing ContactForm model and modifying the rules method. It would look like this:

```
[ 'verifyCode', 'captcha', 'captchaAction' => 'pages/captcha' ]
```

Note: Make sure there is a matching \$verifyCode property on the form model class. See the current ContactForm model for example.

Step 3. Include widget on the view with same 'captchaAction' parameter in widget config:

```
<?= $form->field($model, 'verifyCode')->widget(Captcha::className(), [
    'captchaAction' => 'pages/captcha',
    'template' => '<div class="row"><div class="col-lg-3">
        {image}</div><div class="col-lg-6">{input}</div></div>',
]) ?>
```

Step 4. Include an access rule in behaviors method to allow captcha in the controller:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'only' => ['captcha'],
            'rules' => [
                [
                    'actions' => ['captcha'],
                    'allow' => true,
                    'roles' => ['?', '@'],
                ],
            ],
        ],
    ];
}
```

Adding the access rule is not necessary on SiteController, but will be needed on any other controller.

Step 5. Clear cache. It may be necessary to clear your browser cache after making changes.

This might seem like a lot of steps, but it's actually very easy to implement. I don't know of any other PHP framework that makes it this easy. And don't worry if you haven't instantly memorized this, I don't have a photographic memory either. Just use this as a reference and know that it's here for you.

Anyway, we jumped ahead, we're not looking at the view yet, plus we have more cool stuff happening here in a moment.

First let's continue with the ContactForm model. The attributeLabels method is next:

```
public function attributeLabels()
{
    return [
        'verifyCode' => 'Verification Code',
    ];
}
```

This is the label that will appear on the form in the view. No further explanation needed there.

And finally, the sendEmail method:

```
public function sendEmail($email)
{
    return Yii::$app->mailer->compose()
        ->setTo($email)
        ->setFrom([$this->email => $this->name])
        ->setSubject($this->subject)
        ->setTextBody($this->body)
        ->send();
}
```

This is calling the compose method from the mailer configured in the application which should be Swiftmailer. When you are in development mode, which is how we setup our application on init in setup, emails will be sent to frontend/runtime/mail.

Note: The folder will not exist until you create a record, which you can do by testing the contact form. So you can try your contact form to see if puts an email there. For more info on configuration options with Swiftmailer, you can check the Yii 2 guide:

[Yii 2 Email Guide](#)

And that's it for our ContactForm model. Let's return now to SiteController and its actionContact method. So we called the instance of ContactForm and set it to \$model. Then we have:

```
if ($model->load(Yii::$app->request->post()) && $model->validate()) {  
  
    if ($model->sendEmail(Yii::$app->params['adminEmail'])) {  
  
        Yii::$app->session->setFlash('success',  
        'Thank you for contacting us. We will respond to you as soon as possible.');//  
  
    } else {  
  
        Yii::$app->session->setFlash('error', 'There was an error sending email.');//  
    }  
  
    return $this->refresh();  
  
} else {  
  
    return $this->render('contact', [  
        'model' => $model,  
    ]);  
}
```

So, if we get the post data and process and validate through the form model, and if the email can be sent, we instruct the method to send a flash message. A flash message, which is text that will appear in the view, gets sent from the controller to the view via session. So here we have the setFlash method. That's all you need to send a flash message, Yii 2 will do the rest.

Note that in any event in this method, it will stay on the contact page. In cases where it processes the form, successfully or not, it refreshes the page, otherwise it shows the form, which is the same thing, but without the flash messages.

It's worth it now to take a look at the view as we have a couple of interesting things going on, including the use of captcha as verify method.

Contact View Form

So how cool is that the first view file we look at in depth is a form? My prediction is that you will be amazed at how concise this file is. And at last, you will see the full MVC cycle at work.

Ok, so in frontend/views/site folder, we have contact.php:

```
<?php

use yii\helpers\Html;
use yii\bootstrap\ActiveForm;
use yii\captcha\Captcha;

/* @var $this yii\web\View */
/* @var $form yii\bootstrap\ActiveForm */
/* @var $model \frontend\models\ContactForm */

$this->title = 'Contact';
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="site-contact">
    <h1><?= Html::encode($this->title) ?></h1>

    <p>
        If you have business inquiries or other questions, please fill out the
        following form to contact us. Thank you.
    </p>

    <div class="row">
        <div class="col-lg-5">
            <?php $form = ActiveForm::begin(['id' => 'contact-form']); ?>
                <?= $form->field($model, 'name') ?>
                <?= $form->field($model, 'email') ?>
                <?= $form->field($model, 'subject') ?>
                <?= $form->field($model, 'body')->textArea(['rows' => 6]) ?>
                <?= $form->field($model, 'verifyCode')
                    ->widget(Captcha::className(), [
                        'template' => '<div class="row"><div class="col-lg-3">
                            {image}</div><div class="col-lg-6">{input}</div></div>',
                    ]) ?>
                <div class="form-group">
                    <?= Html::submitButton('Submit', ['class' => 'btn btn-primary',
                        'name' => 'contact-button']) ?>
                </div>
            <?php ActiveForm::end(); ?>
        </div>
    </div>
</div>
```

37 lines, that's it!

You can see we start with our use statements and some comments that tell us what variables we access:

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;
use yii\captcha\Captcha;

/* @var $this yii\web\View */
/* @var $form yii\widgets\ActiveForm */
/* @var $model \frontend\models\ContactForm */
```

This is handy for reference because it's easy to get \$this and \$model confused. But not to worry, you can just look at the top of the file.

Next we get title and breadcrumbs:

```
$this->title = 'Contact';
$this->params['breadcrumbs'][] = $this->title;
?>
```

We use the params method to send the breadcrumb parameter to the layout file(frontend/views/layouts/main.php), which calls it via the Breadcrumbs widget, so it appears at the top of the page.

Also, Notice the closing php tag below. In our models and controllers, we don't use closing ?> tags. In our views, not only do we use closing ?> tags, but we have to make sure we open and close php correctly as we are interspersed with HTML.

Next we get a div, no php:

```
<div class="site-contact">
```

Then we get an <H1> with Php in it:

```
<h1><?= Html::encode($this->title) ?></h1>
```



Tip

I'm a big fan of red php tags, especially in views. I can't control that color in this book, but that's just a tip for your own coding. That color really helps it pop out against Html code.

Also note the shorter <?= opening php tag. This is the short version of:

```
<?php echo
```

So in our h1, we are echoing the title, within the Html::encode method. This will convert special characters into html entities. This is why we need to:

```
use yii\helpers\Html;
```

Then we have a <p> for instructions on filling out the form. This is just plain Html format.

Then we get two divs for our form widget to sit in:

```
<div class="row">
    <div class="col-lg-5">
```

And then we call the ActiveForm widget itself:

```
<?php $form = ActiveForm::begin(['id' => 'contact-form']); ?>
```

Note that we can't use ActiveForm without the use statement at the top of the file:

```
use yii\bootstrap\ActiveForm;
```

In the configuration array of the widget, we set 'id' => 'contact-form'. This tells the form to use the ContactForm model. So looking over the rest of the code, there's nothing to indicate the form action. How does it know what action to post to?

This is one the truly awesome features of Yii 2. It knows that the form's location is a view file called contact.php in a view folder named site. Therefore it knows that it should submit the action to site/contact. You don't even need to tell it where to post the form. And since we've mentioned that we set the id of the form to contact-form, it has the model as well, so it has everything it needs to put this together for you behind the scenes, including validation and processing.

So now just define what fields you want to post:

```
<?= $form->field($model, 'name') ?>
    <?= $form->field($model, 'email') ?>
    <?= $form->field($model, 'subject') ?>
    <?= $form->field($model, 'body')->textArea(['rows' => 6]) ?>
    <?= $form->field($model, 'verifyCode')
        ->widget(Captcha::className(),
[
    'template' => '<div class="row"><div class="col-lg-3">
        {image}</div><div class="col-lg-6">{input}</div></div>',
])
?>
```

Note: As I mentioned above, SiteController is the default controller for captcha, so in this case, we do not need to specify 'captcha' => 'site/captcha', but in all other uses, you will need to specify the controller/captcha as gave in the previous example.



Just another reminder, you need to include the closing ?> tag in views.

On the last field, we got a little tricky, we put a widget in the field. In this case, it's the Captcha widget that we have been discussing. There are other widgets that we will use in the future, such as dropDownList and DatePicker, which are easy to use as well.

Finally, we add the divs for the button and the end of the form:

```
<div class="form-group">
<?= Html::submitButton('Submit', ['class' => 'btn btn-primary',
                                'name' => 'contact-button']) ?>
</div>
<?php ActiveForm::end(); ?>
</div>
</div>
```

I don't know how you feel about that, but to me, that is so simple, it's inspiring. All the validation, all that complicated regex stuff, handled effortlessly. If working on form validation was your favorite hobby, it's time to find a new hobby.

Ok, believe it or not, we are still in a chapter about the site controller. So let's return to that now.

About Action

The next method is actionAbout:

```
public function actionAbout()
{
    return $this->render('about');
}
```

Obviously, everything is in the view file and most of it is just text.

Signup Action

Let's move on to actionSignup:

```
public function actionSignup()
{
    $model = new SignupForm();
    if ($model->load(Yii::$app->request->post())) {
        if ($user = $model->signup()) {
            if (Yii::$app->getUser()->login($user)) {
                return $this->goHome();
            }
        }
    }

    return $this->render('signup', [
        'model' => $model,
    ]);
}
```

Here we go, new form model, SignupForm. Just a quick note. I checked the signup.php view and the id of the form widget was set to form-signup, which I believe is a typo. The convention seems to be the reverse, so just to check if it would work, I changed it to signup-form, and sure enough, it still works. So most likely, Yii 2 doesn't care what order you put them in. I, on the other hand, get nervous about things like that, so I'm sticking with one way to do it, which is putting the word 'form' second. This is also an indication that when creating a form model, you should stick with the naming convention ExampleForm, for example.

Signup Form Model

We looked the SignupForm model briefly in the Modifying the User Model chapter, but let's look at the SignupForm model in more detail now. We'll start with the namespace, use statements, class declaration, and attributes:

```
<?php
namespace frontend\models;

use common\models\User;
use yii\base\Model;
use Yii;

/**
 * Signup form
 */
```

```
class SignupForm extends Model
{
    public $username;
    public $email;
    public $password;
```

Nothing that we haven't seen before. Let's look at the rules method:

```
public function rules()
{
    return [
        ['username', 'filter', 'filter' => 'trim'],
        ['username', 'required'],
        ['username', 'unique',
            'targetClass' => '\common\models\User',
            'message' => 'This username has already been taken.'],
        ['username', 'string', 'min' => 2, 'max' => 255],

        ['email', 'filter', 'filter' => 'trim'],
        ['email', 'required'],
        ['email', 'email'],
        ['email', 'unique',
            'targetClass' => '\common\models\User',
            'message' => 'This email address has already been taken.'],

        ['password', 'required'],
        ['password', 'string', 'min' => 6],
    ];
}
```

Ok, we have some rules to trim out white space, require our fields, and show us if the username or email is already in use, examples of the unique validator. Note that on the email and user unique rules, it lists a target class, along with a response message. This is the only time I've seen this and I couldn't find anything in the docs on it, so I'll take a guess here and say it needs to know which model to use for unique validation, since it has to query the database to execute the validator.

Then we have the signup method:

```

public function signup()
{
    if ($this->validate()) {
        $user = new User();
        $user->username = $this->username;
        $user->email = $this->email;
        $user->setPassword($this->password);
        $user->generateAuthKey();
        if ($user->save()) {
            return $user;
        }
    }

    return null;
}

```

This is actually fairly simple to understand. It calls an instance of the User model, then uses the User model methods to create the user, as long as the input has passed validation.

So one thing to note about controllers is that they can reference many different models. So far the site controller has used 3 different form models and the user model.

Anyway, let's return to the actionSignup method on the controller. Here it is again for reference:

```

public function actionSignup()
{
    $model = new SignupForm();
    if ($model->load(Yii::$app->request->post())) {
        if ($user = $model->signup()) {
            if (Yii::$app->getUser()->login($user)) {
                return $this->goHome();
            }
        }
    }

    return $this->render('signup', [
        'model' => $model,
    ]);
}

```

At this point, we are pretty familiar with how Yii 2 loads the post data. Then it runs the signup method on the model, which will create the user. Then it finds an instance of the user and logs them in. Finally, it returns them to the homepage.

If it fails or there is no post data, it shows the signup form.

The last two methods on the site controller deal with resetting the password. The first one is `actionRequestPasswordReset` and it calls the `PasswordResetRequestForm` model.

It's interesting to note the id on the view form is

```
<?php $form = ActiveForm::begin(['id' => 'request-password-reset-form']); ?>
```

So you can see the naming convention follows what we would expect.

Ok, back to the model, let's step through this it's pretty simple actually:

```
<?php

namespace frontend\models;

use common\models\User;
use yii\base\Model;

/**
 * Password reset request form
 */

class PasswordResetRequestForm extends Model
{
    public $email;

    /**
     * @inheritdoc
     */
}

public function rules()
{
    return [
        ['email', 'filter', 'filter' => 'trim'],
        ['email', 'required'],
        ['email', 'email'],
        ['email', 'exist',
            'targetClass' => '\common\models\User',
            'filter' => ['status' => User::STATUS_ACTIVE],
            'message' => 'There is no user with such email.'
    ];
}
```

```

],
];
}
}
```

The only attribute here is \$email. Notice on the rules in the exist array we get ‘targetClass’, ‘filter’, and ‘message’. This shows us how sophisticated validation can be. We also see the one constant I left in place on the User model being referenced. So if in the future, we want to replace that, we have to do it here as well as in the model. But wait a minute. We changed the attribute from status to status_id, so we will have to change that here. Go ahead and make the change now.

That line should look like this now:

```
'filter' => ['status_id' => User::STATUS_ACTIVE],
```

Now we have just one more method, sendEmail:

```

public function sendEmail()
{
    /* @var $user User */
    $user = User::findOne([
        'status' => User::STATUS_ACTIVE,
        'email' => $this->email,
    ]);

    if ($user) {
        if (!User::isPasswordResetTokenValid($user->password_reset_token)) {
            $user->generatePasswordResetToken();
        }

        if ($user->save()) {
            return \Yii::$app->mailer->compose(['html' => 'passwordResetToken-html',
                'text' => 'passwordResetToken-text'],
                ['user' => $user])
                ->setFrom([\Yii::$app->params['supportEmail'] => \Yii::$app->name . ' robot'])
                ->setTo($this->email)
                ->setSubject('Password reset for ' . \Yii::$app->name)
                ->send();
        }
    }

    return false;
}
```

We also need to make the attribute name change here. Go ahead and change ‘status’ to ‘status_id’. Should look like:

```
$user = User::findOne([
    'status_id' => User::STATUS_ACTIVE,
    'email' => $this->email,
]);
```

The sendemail method looks up the user by email address to see if they are active. If we're good there, we test to see if there is a valid token, and if not generate one. If we can save it, we send the token in the email. Otherwise return false.

So back to the site controller and the actionRequestPasswordReset method. So after attempting to post the data and validate, it tries to send the email, and if good, sets the flash message of success. If it passed validation but for some reason the email could not be sent, it shows a flash message for error.

If the data is not posted, it shows the form.

Ok, one more action on the site controller, actionResetPassword(\$token). This one requires the get variable from the url for the token:

```
public function actionResetPassword($token)
{
    try {
        $model = new ResetPasswordForm($token);
    } catch (InvalidParamException $e) {
        throw new BadRequestHttpException($e->getMessage());
    }

    if ($model->load(Yii::$app->request->post())
        && $model->validate() && $model->resetPassword()) {
        Yii::$app->getSession()->setFlash('success', 'New password was saved.');

        return $this->goHome();
    }

    return $this->render('resetPassword', [
        'model' => $model,
    ]);
}
```

No surprise, we have another form model. Now because we are expecting the token from the get variable, we wrap the call to the model in a try catch block so that we can handle the error if we don't get the expected token.

ResetPasswordForm Model

Let's look at the ResetPasswordForm model:

```
<?php
namespace frontend\models;

use common\models\User;
use yii\base\InvalidArgumentException;
use yii\base\Model;
use Yii;

/**
 * Password reset form
 */

class ResetPasswordForm extends Model
{
    public $password;

    /**
     * @var \common\models\User
     */

    private $_user;
}
```

We see the namespace, use statements, class declaration and the class properties. There's a comment telling us we will reference the User model. Then we get a constructor that takes two arguments, the token and a \$config that defaults to an empty array. I'm fairly certain the empty array is there because of the parent constructor of Model, the class which is being extended.

```
/*
 * Creates a form model given a token.
 *
 * @param string $token
 * @param array $config name-value pairs that will be used to initialize
 * the object properties
 * @throws \yii\base\InvalidArgumentException if token is empty or not valid
 * avoiding line-wrap in function. do not breakup lines in your code.
 */

public function __construct($token, $config = [])
{
    if (empty($token) || !is_string($token)) {
        throw new InvalidArgumentException('Password reset token cannot be blank.');
    }
    $this->_user = User::findByPasswordResetToken($token);
    if (!$this->_user) {
        throw new InvalidArgumentException('Wrong password reset token.');
    }
    parent::__construct($config);
}
```

Obviously again, we are avoiding wordwrap, so there are line breaks where there normally wouldn't be. It makes the code look sloppy but there is nothing I can do about it. So let's just move on.

So two main things the constructor is doing. 1. It checks to see if the token is empty or not a string. 2. It does a User::findByPasswordResetToken(\$token) call to set the user to \$_user.

If you remember when we went through the User model in detail, we saw the findByPasswordResetToken(\$token) method.

The constructor finishes by calling the parent.

Next we have the rules method:

```
public function rules()
{
    return [
        ['password', 'required'],
        ['password', 'string', 'min' => 6],
    ];
}
```

Easy enough to understand.

We finish up the model with the resetPassword method:

```
/*
 * Resets password.
 *
 * @return boolean if password was reset.
 */

public function resetPassword()
{
    $user = $this->_user;
    $user->password = $this->password;
    $user->removePasswordResetToken();

    return $user->save();
}
```

This is fairly easy to get. The password will be set by the post data which the controller will feed to the model. Then remove the token.

Ok, so to close out the controller actionResetPassword method:

```
if ($model->load(Yii::$app->request->post()) && $model->validate()
    && $model->resetPassword()) {

    Yii::$app->getSession()->setFlash('success', 'New password was saved.');

    return $this->goHome();
}

return $this->render('resetPassword', [
```

```
'model' => $model,  
]);  
}
```

Post the data to the model, reset password and go home, or, show the view form.

Obviously Site Controller covered a lot of ground, but we still have a little more ground to cover. That was the frontend.

Backend Site Controller

We have a backend Site Controller in backend/controllers/SiteController.php. This one however is quite a bit smaller. Let's dig in.

```
<?php  
namespace backend\controllers;  
  
use Yii;  
use yii\filters\AccessControl;  
use yii\web\Controller;  
use common\models\LoginForm;  
use yii\filters\VerbFilter;  
  
/**  
 * Site controller  
 */  
  
class SiteController extends Controller  
{
```

We are so used to this by now, we don't need to really comment here, other than to point out that we will be using the same LoginForm model as the frontend did.

Next method, behaviors:

```

public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'rules' => [
                [
                    'actions' => ['login', 'error'],
                    'allow' => true,
                ],
                [
                    'actions' => ['logout', 'index'],
                    'allow' => true,
                    'roles' => ['@'],
                ],
            ],
        ],
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'logout' => ['post'],
            ],
        ],
    ];
}

```

Nothing new to discuss here.

Now we move on to actions method, configuring just for error, since we don't need captcha:

```

public function actions()
{
    return [
        'error' => [
            'class' => 'yii\web\ErrorAction',
        ],
    ];
}

```

Then actionIndex:

```
public function actionIndex()
{
    return $this->render('index');
}
```

Super simple as frontend was.

Now comes actionLogin:

```
public function actionLogin()
{
    if (!\Yii::$app->user->isGuest) {
        return $this->goHome();
    }

    $model = new LoginForm();
    if ($model->load(Yii::$app->request->post()) && $model->login()) {
        return $this->goBack();
    } else {
        return $this->render('login', [
            'model' => $model,
        ]);
    }
}
```

Again, same as frontend.

And finally, actionLogout:

```
public function actionLogout()
{
    Yii::$app->user->logout();

    return $this->goHome();
}
```

No explanation needed at this point (I hope).

Beginning Access Control

So finally, buried deep at the end of a long chapter, a grand tour of sorts, we get to make some changes and influence our application's behavior. We are going to start by a very small change to the backend site controller.

You won't need a Gist, there's just a one word change. Modify the actionLogin to the following:

```

public function actionLogin()
{
    if (!\Yii::$app->user->isGuest) {
        return $this->goHome();
    }

    $model = new LoginForm();
    if ($model->load(Yii::$app->request->post()) && $model->loginAdmin()) {
        return $this->goBack();
    } else {
        return $this->render('login', [
            'model' => $model,
        ]);
    }
}

```

There's only one small change. Instead of calling login(), we call loginAdmin() from the LoginForm model.

loginAdmin Method

We have not created that method, so let's go ahead and do so now. Insert the following method into common/models/LoginForm.php.

Gist:

[LoginAdmin](#)

From book:

```

public function loginAdmin()
{
    if (($this->validate())
        && PermissionHelpers::requireMinimumRole('Admin',
        $this->getUser()->id)) {

        return Yii::$app->user->login($this->getUser(),
            $this->rememberMe ? 3600 * 24 * 30 : 0);

    } else {

        throw new NotFoundHttpException('You Shall Not Pass.');
    }
}

```

```
    }  
  
}
```

Please look at the Gist if you want to see the proper formatting, that got really ugly because I had to avoid wordwrap in the book.

Also add to the top of the file:

```
use yii\web\NotFoundHttpException;  
use common\models\PermissionHelpers;
```

So what we have done here is added more to the if statement. Not only do we validate the user, but we also use:

```
PermissionHelpers::requireMinimumRole('Admin', $this->getUser()->id)
```

This will ensure the user has a minimum role of admin. The reason we are handing in \$this->getUser()->id is because the user is not yet logged in, so we need to tell it explicitly which user we want to check. If you recall, we accounted for this scenario in the helper classes.

We don't need to check their status at this point because the getUser method calls findByUsername and that method does it for us.

With one relatively minor change, we now control access to the admin login by requiring the user to have at least a role of admin and an active status. And look how simple it was!

Our helper classes came in very handy and with those methods, we did not need verbose code to get the values we wanted.

So now you can play around with this by registering users and setting their role_id via PhpMyadmin. Make some that are User and some that are Admin and log in and out to both the frontend and the backend. It's pretty cool.

Before we end the chapter, we can do one bit of cleanup. Let's discuss the constant on the user table:

```
const STATUS_ACTIVE = 1;
```

As I said in chapter four, leaving the constant in place violates DRY. But for ease of setup, I have left it in place.

Now that we have our helper classes in place, we can replace it. Obviously this assumes you have an Active status record in your status table.

If you wish to replace the constant, you can do this by finding every instance of:

```
'status_id' => self::STATUS_ACTIVE
```

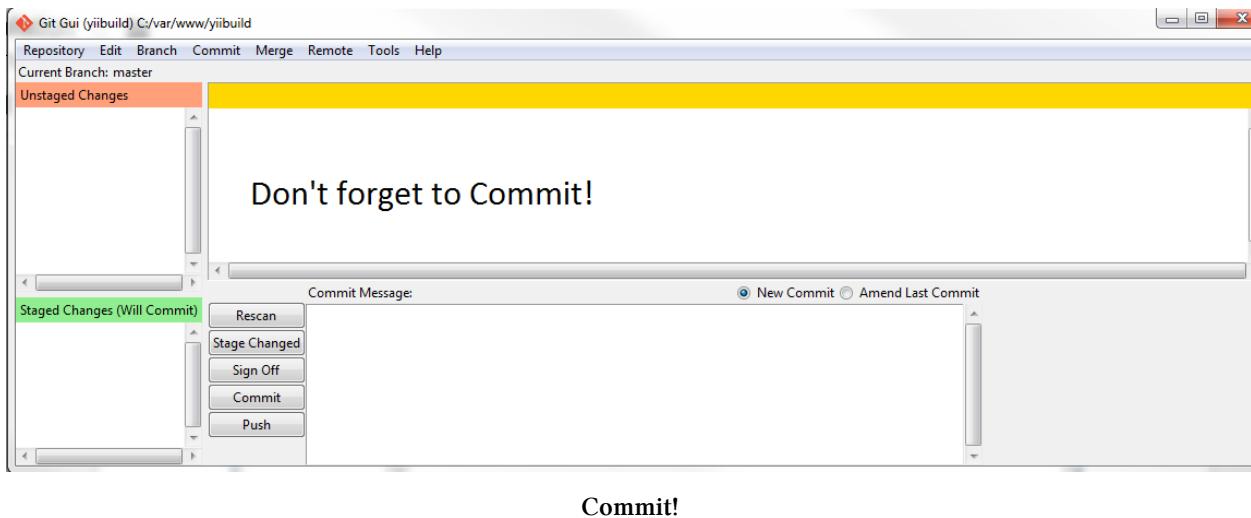
Then replace that with:

```
'status_id' => ValueHelpers::getStatusId('Active')
```

Just substitute that for the constant in the appropriate places in the User model and the ResetPasswordRequestForm model and don't forget to include the use statements for ValueHelpers.

Then you can remove the constant entirely. If you remove the constant and you get an error, it means you missed a reference to it in one of the models, so just go back over and find the missing reference.

Summary



We covered a lot of ground by examining Site Controller in great detail. We learned a lot about controllers, actions, views, and even more on form models. Now we know, having seen so many examples, that the typical implementation of a form can involve a form view, a form model and a controller.

We learned about the ActiveForm widget, which resides in the view and makes it easy for us to render the form. We learned how to implement captcha and send flash messages from the controller to the view.

We also learned about some of the more interesting actions, like resetPassword and their associated form models. We saw how we can target the user class in the validation rules.

Finally, we got to implement a couple of changes that now control access to the backend by enforcing a minimum value to the role of the user. We did that simply from using Helper methods from a

previous chapter, so that it seemed like it was a nothing change, and yet it was the beginning of our building a control system throughout the application.

We've taken care to explain everything in as much detail as possible. Hopefully enough of it is sticking with you. If not, give it time, it will. Yii 2 requires patience and persistence to learn. We're finally on our way to building the application. And so onward we march!

Chapter Eight: Profile Crud

Ok, after so much patience learning what the Yii 2 handed us out of the box, we're ready to expand our application and create something. We going to start with creating the code to allow our users to create a profile.

Profiles will be unique to each user and only that user should see their Profile. This will require changes to what Gii will create for us, but we will of course use Gii to get us started by helping us make the Profile CRUD.

CRUD

CRUD stands for create, read, update and delete, a simple acronym to remember it. When you use Gii to create CRUD, you get a controller, forms, and a search model. It's awesome!

So let's navigate back to Gii:

```
yii2build.com/index.php?r=gii
```

The screenshot shows the Gii CRUD Generator interface. On the left, there's a sidebar with a list of generators: Model Generator, CRUD Generator (which is selected and highlighted in blue), Controller Generator, Form Generator, Module Generator, and Extension Generator. The main area is titled "CRUD Generator" and contains the following configuration fields:

- Model Class:** frontend\models\Profile
- Search Model Class:** frontend\models\search\ProfileSearch
- Controller Class:** frontend\controllers\ProfileController
- View Path:** (empty input field)
- Base Controller Class:** yii\web\Controller
- Widget Used in Index Page:** GridView
- Enable I18N:** (checkbox is unchecked)
- Code Template:** default (C:\var\www\yii2builder\vendor\yiisoft\yii2-gii\generators\crud/default)

At the bottom right of the configuration area is a blue "Preview" button.

Gii CRUD

You can see that the filename conventions to follow. You need to create the search folder inside of frontend/models before you use Gii to create the CRUD, so make sure the folder named search is there before proceeding .

We already decided to put the Profile model in the frontend folder, so we will be doing the same for the crud. Make sure the fields are filled in as follows:

Model Class is frontend\models\Profile

Search Class is frontend\models\search\ProfileSearch

Controller Class it is frontend\controllers\ProfileController

You can leave the view path blank if you are using defaults in the file structure, which is what we are using.

We are providing Gii with a just little information and this time it's going to create 8 files for us:

- ProfileController.php
- ProfileSearch.php
- views/profile/_form.php
- views/profile/_search.php
- views/profile/create.php

- views/profile/index.php
- views/profile/update.php
- views/profile/view.php

After you have created the search folder in frontend/models, go ahead and run the CRUD generator for Profile. If you follow the preview/generate cycle on Gii, you should end up with those 8 files. Let's briefly discuss what each file does.

Profile Controller

This is ProfileController.php and is found under frontend/controllers/ProfileController.php. We are going to look at this in great detail in few minutes.

Profile Search

ProfileSearch.php is located in frontend/models/search/ProfileSearch.php and contains the logic that powers the search form. This is basically a special extension of the base Profile model that is for search. Because we won't be searching through profiles on the frontend, we won't be using it. Users will only have one profile and therefore no need to search. I included it here because I wanted to show you how to create this kind of file. We will use this kind of file for our backend later.

_search

The search form itself is a partial named _search.php that gets rendered to the Profile index page. Since we are not allowing other users to search profiles, we will not be using this.

_form

_form.php is another partial view that contains the form that is rendered to the create and update views. The same form partial is rendered into both create and update view pages.

Index

Index.php is the index view includes a ready made widget to display paginated record results in organized columns, along with _search.php partial on top of the form. Although we won't need this for frontend users, who only ever have one profile, we will need this for backend admin users who can review the profiles of all users. Just to be perfectly clear, we will not be using this file at all, but we will use one like it for the backend. We will leave it in place for now for demonstration purposes.

Once the file is made through Gii, you can reach this page at:

yii2build.com/index.php?r=profile/index

View

The view.php file is the details of an individual record and utilizes the DetailView widget. This page will return a 400 error since we are not passing an id, which the controller is expecting. Just ignore this for now, since we are going to change it anyway.

Create

Calling create.php renders the _form.php partial so you can input the needed data to create the record.

Once the file is made through Gii, you can reach this page at:

yiibuild.com/index.php?r=profile/create

Here is a photo of the page rendered by the above url:

Create Profile

User ID

First Name

Last Name

Birthdate

Gender ID

Created At

Updated At

Create Profile

Although this form would work if you formatted the input correctly, I don't recommend testing it now. We are going to make deep changes to the controller and other changes to the form, so it's pointless to test it now.

Update

Obviously, this page also fails because it is expecting id as a parameter.

When you look at the create view page, you will notice that there are a lot of problems. Fields like created_at show up as text fields and gender returns a number.

As I mentioned, if you try some of these urls without having created a profile, you will get errors. You can create a profile by forcing your way through, but this is not what you want to do. We have a lot of work to do before this UI is ready.

Also, because we haven't stitched together the right navigation, the form doesn't know what user is the right user to associate with the profile. Entering that manually exposes the question of security and url manipulation, because even if you associate the right record manually, you have to prevent users from hijacking records they don't own.

We need to do a bit of work on the controller and the views to get exactly what we need, the code generation only goes so far.

The good news is that even in the state everything is in, Gii has provided us with much of what we will need. It has also given us an easy-to-follow architecture that, which, with some modification on our part, will create a robust template for user-owned records.

So, for whatever application requirements you have in the future, when it involves a user-owned record, such as profile, preferences, blog entry, etc., you will have a working example that is easily replicated. And once you get into the rhythm of this, you will be amazed at how it flows.

By starting with a user-owned record like profile, we are creating a structure that Gii does not hand to us in perfect condition, that's the bad news. The good news is that when we build the backend version of profile, to help admin manage profiles, it is a much closer fit to the boilerplate, so things will become progressively easier as we go.

Modifying Profile Controller & Views

Ok, let's address our needs. We can create a profile for the user, but the form doesn't look right and there is no navigation to it. Also, there is no access control over the CRUD, so the user's profile, which should be private, is wide open and can be manipulated by anyone.

We're going to do the following to correct this:

1. We will modify the controller to control access to the CRUD actions. For example, only the user who owns the profile record should be able to view, create, update or delete the profile record.
2. We will modify the views, so they have dropdown lists where appropriate. We will remove unnecessary fields as well. We will bring in the appropriate name for gender instead of simply listing an id number.

3. We will modify our main layout view so that we have navigation to the user profile.

When we're done with this section, our Yii2build template will have taken a huge step forward.

Out of the box, users are able to register and login to the application. We have already extended the Advanced Application Template to respect the difference between role level for frontend and backend, so only users with a minimum role level of admin can log into the backend.

Tip

Just for the sake of clarity, I will mention that the frontend profile CRUD has nothing to do with backend and will not appear there. We will build that separately when we create the backend admin area.

By building our fully functional frontend profile model, users will be required to log in to build a profile. The application will know whether or not they already have a profile, and if not, when they click on the profile link, it will take them to the profile create page. The application will also enforce rules to make sure the user is only able to access their own profile. We will also provide navigation in the view pages that knows when it is appropriate to show the profile link.

So when we're done with this part of the project, we will have tight template and an example to follow if we want to create any other type of user-owned records that need to stay private to the users. This is good stuff.

Modifying the Profile Controller

We need to start by including a couple of more use statements, so copy this over the existing use statements at the top of the ProfileController.php file.

Gist:

[Profile Use Statement](#)

From book:

```
namespace frontend\controllers;

use Yii;
use frontend\models\Profile;
use frontend\models\search\ProfileSearch;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;
use common\models\PermissionHelpers;
use common\models\RecordHelpers;
```

This makes reference to our RecordHelpers class and our PermissionHelpers which we wrote in a previous chapter.

Currently, the behaviors method only has a restriction on delete that says it must be done by the post method:

```
public function behaviors()
{
    return [
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'delete' => ['post'],
            ],
        ],
    ];
}
```

We want to add some basic access control logic that restricts the user from the actions on this controller unless they are logged in. So change behaviors to this:

Gist:

[Controller Behaviors](#)

From book:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => \yii\filters\AccessControl::className(),
            'only' => ['index', 'view', 'create', 'update', 'delete'],
            'rules' => [
                [
                    'actions' => ['index', 'view', 'create', 'update', 'delete'],
                    'allow' => true,
                    'roles' => ['@'],
                ],
            ],
        ],
        'verbs' => [
    ];
```

```

    'class' => VerbFilter::className(),
    'actions' => [
        'delete' => ['post'],
    ],
],
];
}

```

The @ symbol means logged in, so the listed actions can only be performed when the user is logged in. That's not really enough access control, but it's start. We will do more later.

Please note that roles in this case does not refer to the role_id column on the user record, the two have nothing to do with each other.

Index Action

Ok, now we're ready for the actions. The index action, which is the default action of the controller, looks like this:

```

/**
 * Lists all Profile models.
 * @return mixed
 */

public function actionIndex()
{
    $searchModel = new ProfileSearch();
    $dataProvider = $searchModel->search(Yii::$app->request->queryParams);

    return $this->render('index', [
        'searchModel' => $searchModel,
        'dataProvider' => $dataProvider,
    ]);
}

```

This is meant to return a list of results and uses a different model, ProfileSearch, which extends the profile model to provide search functionality. But in the case of our user profile, we only allow one profile per user, so we won't be using this code.

We could just disable this action, cut it out of the controller completely, and delete the index.php file, but some of the bread crumb navigation that is built into Yii 2 will then return a page not found

exception and that is not the behavior we want. Also, someone could type in r=profile into the url and it would return the same page not found error, again behavior we do not want. We want all our navigation super tight, so instead we will redirect Index.php to view.php.

View.php, if you recall, lists the details of the record, and this would be appropriate for someone who wanted view their own profile.

Of course it's not as simple as redirecting. We also have to apply the controlling logic that determines whether the appropriate user has a profile, if so, show it to them, and if not, send them to the Create page.

So let's replace the actionIndex method with the following:

Gist:

Index Action

From book:

```
public function actionIndex()
{
    if ($already_exists = RecordHelpers::userHas('profile')) {
        return $this->render('view', [
            'model' => $this->findModel($already_exists),
        ]);
    } else {
        return $this->redirect(['create']);
    }
}
```

That first line should look a little familiar to you. We talked about doing something like this when we built the RecordHelpers method userHas. Now we're ready to use it.

Just a quick reminder, the userHas method checks for a user's record on the model supplied, so in this case we are checking to see if the user has a profile record. If there is no record, it returns false, if true, it returns the id of the record.

Let's describe exactly how this works. The first thing we are doing is calling the userHas method from the RecordHelpers class and setting the result to \$already_exists, wrapped in an if statement.

So if userHas evaluates true, it returns to the view file, with the correct model instance held in the variable \$already_exists.

It then uses the controller's findModel method to return that instance to the view, in this case it's named view.php, as it is passed along in the array:

```
'model' => $this->findModel($already_exists),
```

Now if \$already_exists evaluates false, we redirect to the create view, since the user doesn't have a profile, and they need to create one:

```
} else {  
  
    return $this->redirect(['create']);  
  
}
```

By taking the extra effort to create the helper class, we extracted out some logic that keeps the controller simple and clean. If we wanted to use Yii's relationship syntax in making the if condition, we might write:

```
if($already_exists = Profile::find()->where(['user_id' => Yii::$app->user->identity->id])->one())
```

Personally, I like this syntax better:

```
if ($already_exists = RecordHelpers::userHas('profile'))
```

For the most part, it's a trivial cosmetic difference, but we have the potential to use this syntax in many places in the project. We can use it on other models simply by handing in a different model name.

I mentioned in an earlier chapter about Robert C. Martin's book called Clean Code. He makes the point that programmers spend most of their time reading code, not writing it. So little syntactic differences that make code easier to read, when scaled out over a large project, make a huge difference in the speed and clarity of those who have to work on the system.

Another big change in the way we built our controller methods is that we eliminated the get variable being handed into the method, so the record id can't be hijacked through the browser. This adds an extra layer of security to the system. We will be adding more security later.

Now that we understand how RecordHelpers::userHas('profile') checks the association between the user and the profile record, we are ready to step through the rest of the actions.

View Action

The view action:

```

/**
 * Displays a single Profile model.
 * @param string $id
 * @return mixed
 */

public function actionView($id)
{
    return $this->render('view', [
        'model' => $this->findModel($id),
    ]);
}

```

Let's change this to:

Gist:

[ActionView](#)

From book:

```

public function actionView()
{
    if ($already_exists = RecordHelpers::userHas('profile')) {

        return $this->render('view', [
            'model' => $this->findModel($already_exists),
        ]);
    } else {

        return $this->redirect(['create']);
    }
}

```

Hey, that's exactly the same as the index action, what's up with that? Remember we said that we were changing the default index action to be the same as the view action because we don't need a list of profiles, only the correct profile for the user or the create form if they don't have one. So you caught a break there and we just copied the code.

Create Action

Ok, on to the Create action. Here's what you get out of the box:

```
public function actionCreate()
{
    $model = new Profile();

    if ($model->load(Yii::$app->request->post()) && $model->save()) {
        return $this->redirect(['view', 'id' => $model->id]); } else {
        return $this->render('create', [
            'model' => $model,
        ]);
    }
}
```

Basically this is saying if the form is loaded, save it and go to view or display the create form. Well, we have a little more to our version of this:

Gist:

[ActionCreate](#)

From book:

```
public function actionCreate()
{
    $model = new Profile;

    $model->user_id = \Yii::$app->user->identity->id;

    if ($already_exists = RecordHelpers::userHas('profile')) {

        return $this->render('view', [
            'model' => $this->findModel($already_exists),
        ]);
    } elseif ($model->load(Yii::$app->request->post()) && $model->save()){

        return $this->redirect(['view']);
    }
}
```

```

} else {

    return $this->render('create', [
        'model' => $model,
    ]);
}
}
}

```

We start by calling a new instance of the model, then we set the user_id attribute of the model to the current user via Yii::\$app->user->identity->id. As I've said before, the current user id is always available to us via this static call of the Yii application class.

Next we check to see if the user already has a profile by running our RecordHelpers::userHas('profile') method and setting the result to \$already_exists.

```

if ($already_exists = RecordHelpers::userHas('profile')) {
    return $this->render('view', [
        'model' => $this->findModel($already_exists),
    ]);
}
}

```

If we get an model id in response, we show the view file of that id. We need to put this test on this method because a user might be able to navigate directly to the create action, without going to index or view first.

If \$already_exists evaluates false, the next thing the code does is call the load method from the post data and attempt to save it:

```

elseif ($model->load(Yii::$app->request->post()) && $model->save()) {

    return $this->redirect(['view']);
}

```

But this will only happen if the load method has received data from the form that was posted. If it receives it and can validate, it would save and then return the appropriate view page.

You might be wondering how it knows what user_id to assign in the newly created record, since we are not setting it on the form, which means it's not being passed in via post. It gets set on the second line of the action method when we assign that \$model->user_id to the current user. And when \$model calls the load method, it remembers this attribute.

Lastly, if there is no post data or if there are validation errors, we show the form:

```

else {

    return $this->render('create', [
        'model' => $model,
    ]);
}

```

Update Action

Next we move on to the actionUpdate method. Here is what Gii gave us:

```

/**
 * Updates an existing Profile model.
 * If update is successful, the browser will be redirected to the 'view' page.
 * @param string $id
 * @return mixed
 */

public function actionUpdate($id)
{
    $model = $this->findModel($id);

    if ($model->load(Yii::$app->request->post()) && $model->save()) {
        return $this->redirect(['view', 'id' => $model->id]);
    } else {
        return $this->render('update', [
            'model' => $model,
        ]);
    }
}

```

We need to modify this to:

Gist:

[ActionUpdate](#)

From book:

```


/**
 * Updates an existing Profile model.
 * If update is successful, the browser will be redirected to the 'view' page.
 * @param string $id
 * @return mixed
 *if statement in two lines due to avoid wordwrap
 */

public function actionUpdate()
{
    if($model =  Profile::find()->where(['user_id' =>
        Yii::$app->user->identity->id])->one()) {

        if ($model->load(Yii::$app->request->post()) && $model->save()) {

            return $this->redirect(['view']);

        } else {

            return $this->render('update', [
                'model' => $model,
            ]);
        }
    } else {

        throw new NotFoundHttpException('No Such Profile.');
    }
}


```

Right away we see something different because I used:

```
if($model =  Profile::find()->where(['user_id' => Yii::$app->user->identity->id])->one())
```

I did that because I felt that since the variable name was \$model, it was a little more syntactically logical to follow that with the model name we are interested in, instead of the helper class:

```
if($model = RecordHelpers::userHas('profile'))
```

the longer syntax just seems clearer that we are looking for a profile record that matches the current user. Ok, so now we know why we are using:

```
if($model = Profile::find()->where(['user_id' => Yii::$app->user->identity->id])->one())
```

So, let's break that down. Set the instance of the Profile model, where the user_id is the current user to the variable \$model, if you can. If it evaluates false, we jump down to the last else statement and throw:

```
throw new NotFoundHttpException('No Such Profile.');
```

Otherwise, if we get our \$model variable set with the correct instance of the model, we call the data from the post form and try to save it via the next if statement.

```
if ($model->load(Yii::$app->request->post()) && $model->save()) {  
  
    return $this->redirect(['view']);  
  
}
```

If the update is successful, how does it know which view to display? The redirect method, which in this case is only supplied with an action, is not rendering. It is routing it to the named action of the current controller, so in this case, the view action will determine the right view to display.

Of course, this also assumes there is post data.

If there is no post data, no change from what is there already, it will return to the form, with our model handed in so it can pre-populate correctly.

```
else {  
    return $this->render('update', [  
        'model' => $model,  
    ]);  
}
```

Once again, no get variables were used, so there will be no hijacking the records from get variables.

Delete Action

Our final change to the ProfileController will be on the delete action. We got this from Gii:

```
public function actionDelete($id)
{
    $this->findModel($id)->delete();

    return $this->redirect(['index']);
}
```

Let's change this to:

Gist:

[ActionDelete](#)

From book:

```
public function actionDelete()
{
    $model = Profile::find()->where([
        'user_id' => Yii::$app->user->identity->id
    ])->one();

    $this->findModel($model->id)->delete();

    return $this->redirect(['site/index']);
}
```

Here we set the \$model using same method from update for the same reasons, since we don't have a get variable. Then we use the delete method, handing in the \$model->id.

Then we redirect to the site index page. If we just put 'index' as the value, the controller would assume it was 'profile/index', which is not what we want.

FindModel Action

The last method in the Profile controller, we are not changing:

```
protected function findModel($id)
{
    if (($model = Profile::findOne($id)) !== null) {
        return $model;
    } else {
        throw new NotFoundHttpException('The requested page does not exist.');
    }
}
```

This is the method to find a particular instance of the model. You hand in the \$id you are looking for and Yii returns that instance of the model, a very useful method indeed.

One thing we can appreciate about Yii 2 is the controller code is very clean, clear and concise. All the heavy lifting is abstracted out and we are simply providing a small set of instructions for the actions to follow. This is modern PHP at its best.

Modifying the Profile Views

Our profile controller functions as we wish, but our forms have date fields that we don't need and text inputs instead of dropdowns, so we have to go through our views and fix this.

We're going to do quite a few changes before we test everything. This is a little out of real-world workflow, but on the other hand, you won't get stuck inserting records manually to test with that might throw errors, if you miss something.

So just stick with doing these changes, and when we're done, we'll be able to add a profile to the user via UI and test it for real.

View.php

Ok, let's start with view.php. Here is what we got from Gii:

```
<?php

use yii\helpers\Html;
use yii\widgets\DetailView;

/* @var $this yii\web\View */
/* @var $model frontend\models\Profile */

$this->title = $model->id;
$this->params['breadcrumbs'][] = ['label' => 'Profiles', 'url' => ['index']];
```

```

$this->params['breadcrumbs'][] = $this->title;
?>
<div class="profile-view">

    <h1><?= Html::encode($this->title) ?> Profile</h1>

    <p>
        <?= Html::a('Update', ['update', 'id' => $model->id],
                    ['class' => 'btn btn-primary']) ?>
        <?= Html::a('Delete', ['delete', 'id' => $model->id], [
                    'class' => 'btn btn-danger',
                    'data' => [
                        'confirm' => 'Are you sure you want to delete this item?',
                        'method' => 'post',
                    ],
                ]) ?>
    </p>

    <?= DetailView::widget([
        'model' => $model,
        'attributes' => [
            'id',
            'user_id',
            'first_name:text',
            'last_name:text',
            'birthdate',
            'gender_id',
            'created_at',
            'updated_at',
        ],
    ]) ?>

</div>

```

We start the file by pulling in a couple of helpers:

```

use yii\helpers\Html;
use yii\widgets\DetailView;

```

This is going to be our first in-depth look at the `DetailView` widget, which we will see in a minute. Next we have some comments from Yii, `$this` is the view model, and `$model` is the profile model that's been handed in through the controller, so it represents the exact instance of the profile model that we need.

```
/* @var $this yii\web\View */
/* @var $model frontend\models\Profile */
```

You can always refer to those comments if you are confused about which variables represent which models.

Next we set the title with \$model->id:

```
$this->title = $model->id;
```

Then come the breadcrumbs in a nice simple to use format:

```
$this->params['breadcrumbs'][] = ['label' => 'Profiles', 'url' => ['index']];
$this->params['breadcrumbs'][] = $this->title;
```

\$this->params comes from the view base class. When you render a view, in this case, view.php, you are calling an instance of yii\webView, which extends the yii\base\view class, and through the magic of Yii 2's routing, making the object available as \$this, hence the comment:

```
/* @var $this yii\web\View */
```

It's a very powerful architecture. You also hand in your model or models via the controller as well, so you have a lot of capabilities for manipulating data in the views. Yii 2 does all of this, and at the same time, makes it look simple, keeping as much PHP coding and logic out of the view as possible.

Ok, moving on. When we set the div, the h1, and some nav:

```
<div class="profile-view">

    <h1><?= Html::encode($this->title) ?> Profile</h1>

    <p>
        <?= Html::a('Update', ['update', 'id' => $model->id],
                    ['class' => 'btn btn-primary']) ?>
        <?= Html::a('Delete', ['delete', 'id' => $model->id], [
            'class' => 'btn btn-danger',
            'data' => [
                'confirm' => 'Are you sure you want to delete this item?',
                'method' => 'post',
            ],
        ]) ?>
    </p>
```

The button style on Delete is btn btn-danger and you hand in the additional data parameters of confirm and method. This results in a confirmation alert, pretty handy for delete functionality and it makes for great UI, right out of the box.

Finally we have the detailview widgets:

```

<?= DetailView::widget([
    'model' => $model,
    'attributes' => [
        'id',
        'user_id',
        'first_name:text',
        'last_name:text',
        'birthdate',
        'gender_id',
        'created_at',
        'updated_at',
    ],
]) ?>

</div>

```

You can easily add or subtract attributes and we'll show you by example. Let's go ahead and replace the entire view file with:

Gist:

[Profile View](#)

From book:

```

<?php

use yii\helpers\Html;
use yii\widgets\DetailView;
use common\models\PermissionHelpers;

/**
 * @var yii\web\View $this
 * @var frontend\models\Profile $model
 */

$this->title = $model->user->username . "'s Profile";
$this->params['breadcrumbs'][] = ['label' => 'Profiles', 'url' => ['index']];
$this->params['breadcrumbs'][] = $this->title;

?>
<div class="profile-view">

    <h1><?= Html::encode($this->title) ?></h1>

```

```
<p>
<?Php

//this is not necessary but in here as example

if (PermissionHelpers::userMustBeOwner('profile', $model->id)) {

    echo Html::a('Update', ['update', 'id' => $model->id],
                 ['class' => 'btn btn-primary']);
} ?>

<?= Html::a('Delete', ['delete', 'id' => $model->id], [
    'class' => 'btn btn-danger',
    'data' => [
        'confirm' => Yii::t('app', 'Are you sure to delete this item?'),
        'method' => 'post',
    ],
]) ?>

</p>

<?= DetailView::widget([
    'model' => $model,
    'attributes' => [
        '/>'id',
        'user.username',
        'first_name',
        'last_name',
        'birthdate',
        'gender.gender_name',
        'created_at',
        'updated_at',
        '/>'user_id',
    ],
]) ?>

</div>
```

Now let's step through the changes. To start, we made some cosmetic changes to the title to display username, which are trivial, and we pulled in an additional use statement:

```
use common\models\PermissionHelpers;
```

This will give us access to the PermissionHelpers::userMustBeOwner() method. This is one of the helper methods we created back in the Helpers chapter, which returns true or false to determine if the current user is the owner of the record.

We are going to use userMustBeOwner(), to put an extra layer of security on our navigation in frontend/views/profile/view.php. In this case, we wrap the method in an if statement, and if true, we display the navigation:

```
<p>
<?Php

//this is not necessary but in here as example

if(PermissionHelpers::userMustBeOwner( 'profile' , $model->id)) {

    echo Html::a('Update' , [ 'update' , 'id' => $model->id] ,
                [ 'class' => 'btn btn-primary' ]);
}
?>
```

The method userMustBeOwner() takes 2 arguments, first, the name of the model handed in as a string, second, the id of the model instance, available to us as \$model->id because we sent the model instance to the view through the controller.

As the comment in the code indicates, this test on the user is not really necessary because for profile/view, you can't get to the view page without being the owner of the record and the update action also tests to limit access to the owner only.

I included this here simply as an example for cases where you might have view records visible to all users, but actions like update only available to record owners and in those cases, you would only want the link to update visible to record owners. A blog where authors can update their posts, but other users can only read them would be an example of this.

I could have done the same test on the delete link, but again, not necessary, since the controller tests for the owner and only record owners get to be on the view page where this nav resides.

Finally, we have some changes to the DetailView widget:

```
<?= DetailView::widget([
    'model' => $model,
    'attributes' => [
        //'id',
        'user.username',
        'first_name',
        'last_name',
        'birthdate',
        'gender.gender_name',
        'created_at',
        'updated_at',
        //'user_id',
    ],
]) ?>
```

We commented out the “*id*” and “*user_id*” fields because these numbers will not mean anything to end users. Instead we popped in *user.username* and *gender.gender_name*. We are accessing these properties through lazy-loaded relationships and that is the syntax for that. Pretty simple right?

Touching briefly on what lazy load means, it means that there will be a query for each row, in this case 2 separate queries. That’s probably fine on a page like this that has a small number of queries. For large lists with multiple queries, it would highly inefficient and we need to use eager loading in those cases, and we will show you how to do that when we are dealing with those kinds of results.

We need to do a little housework on the Gender model in regards to attribute labels, so it displays what we want correctly on the page.

Gender

So let’s start with the simple one, *Gender.php* located in *frontend/models*.

We have one change only to the attribute labels. I won’t provide a Gist, since it’s a one word change:

```
public function attributeLabels()
{
    return [
        'id' => 'ID',
        'gender_name' => 'Gender Name',
    ];
}
```

should be changed to:

```
public function attributeLabels()
{
    return [
        'id' => 'ID',
        'gender_name' => 'Gender',
    ];
}
```

That's the one attribute label I left to work on when we built the new models in the New Models chapter. All the other ones, we already have in place, as well as all the additional relationships.

You can see we put those other labels together outside of normal workflow, otherwise we would be bouncing around between models and views. I thought it was better to keep a tighter focus on one thing at a time, so people who are new to the framework, will be able to absorb the information easier.

Form Partial

Ok, next we're going to modify the `_form.php`, which is a partial. A partial, which in Yii 2 is designated by the underscore in front of the filename, is a view gets included into another view, in this case via:

```
<?= $this->render('_form', [ 'model' => $model, ])??>
```

Through the magic of Yii 2's routing and file structure, it knows which `_form` you are referring to. This makes for very concise code. Let's take a look at how we use it.

The above code is called in `Update.php`, for example. This is a perfect time to mention that the `_form.php` is simpler than the contact one we looked at earlier.

For example, in this case, since we are just doing straight CRUD actions, we don't need a separate form model. We don't even need to specify a form model at all because once again Yii 2 knows from the file structure, and from the model being handed into the view from the controller, which model to update. This is very cool and saves you a lot of time.

So for straight creating, updating, deleting, you typically don't need a separate form model. It's only when there is some complicated validation or other processes will you need a form model.

With our example here, `Update.php`, `$model` is the `Profile` model.

Ok, let's get back to the `_form`. This is what Gii gave us on boilerplate:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $model frontend\models\Profile */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="profile-form">

    <?php $form = ActiveForm::begin(); ?>

    <?= $form->field($model, 'user_id')->textInput(['maxlength' => 11]) ?>

    <?= $form->field($model, 'first_name')->textarea(['rows' => 6]) ?>

    <?= $form->field($model, 'last_name')->textarea(['rows' => 6]) ?>

    <?= $form->field($model, 'birthdate')->textInput() ?>

    <?= $form->field($model, 'gender_id')->textInput(['maxlength' => 10]) ?>

    <?= $form->field($model, 'created_at')->textInput() ?>

    <?= $form->field($model, 'updated_at')->textInput() ?>

    <div class="form-group">
        <?= Html::submitButton($model->isNewRecord ? 'Create' : 'Update', ['class' =>
            $model->isNewRecord ? 'btn btn-success' : 'btn btn-primary']) ?>
    ?>
    </div>

    <?php ActiveForm::end(); ?>

</div>
```

This is nice concise code and you have to love the framework for that. But there are some things we don't need. The user id does not need to be displayed in the form, it is set on the model in the controller, so it gets saved correctly without form input. We can also delete the fields for created_at and updated_at as we have added behaviors on the Profile model which automatically insert those for us.

Then we have just 2 other changes, we will put a note under the date field to explain the input and create a dropdown list for Gender. Here is what the entire file should look like:

Gist:

Form Partial

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/**
 * @var yii\web\View $this
 * @var frontend\models\Profile $model
 * @var yii\widgets\ActiveForm $form
 */
?>

<div class="profile-form">

    <?php $form = ActiveForm::begin(); ?>

    <?= $form->field($model, 'first_name')->textInput(['maxlength' => 45]) ?>

    <?= $form->field($model, 'last_name')->textInput(['maxlength' => 45]) ?>
<br/>

    <?= $form->field($model, 'birthdate')->textInput() ?>
    * please use YYYY-MM-DD format
<br/>

    <?= $form->field($model, 'gender_id')->dropDownList($model->genderList,
        ['prompt' => 'Please Choose One' ]); ?>

<div class="form-group">

    <?= Html::submitButton($model->isNewRecord ? 'Create' : 'Update',
        ['class' => $model->isNewRecord ? 'btn btn-success' : 'btn btn-primary']) ?>
</div>

<?php ActiveForm::end(); ?>
```

```
</div>
```

Note that <?= is the short statement for <?php echo. Each form field gets a separate line and you can see how we did it above.

Going back to _form, we should pay attention to the field for gender_id:

```
<?= $form->field($model, 'gender_id')->dropDownList($model->genderList, [ 'prompt' =>  
'Please Choose One' ]);?>
```

Two things to note. We inserted the dropDownList method using \$model->genderList, which is using a magic get, hence the lowercase g in gender. We can do this because of the relationship method getGenderList that we added to Profile in a previous chapter.

We also added in a parameter inside of an array for the prompt because we don't want the list to default to the first value, which is what it would do if the prompt were not there.

Now onto the small remaining view changes that we have in mind for the frontend Profile views.

Create

Open create.php:

and change this line:

```
$this->params['breadcrumbs'][] = ['label' => 'Profiles', 'url' => ['index']];
```

to this line:

```
$this->params['breadcrumbs'][] = ['label' => 'Profile', 'url' => ['index']];
```

Just dropping the 's' on Profile there.

Update

Now onto update.php. Chop out the 2nd breadcrumb line and change the title, so it looks like this:

```
<?php

use yii\helpers\Html;

/* @var $this yii\web\View */
/* @var $model frontend\models\Profile */

$this->title = 'Update ' . $model->user->username . "'s Profile ";
$this->params['breadcrumbs'][] = ['label' => 'Profile', 'url' => ['index']];

$this->params['breadcrumbs'][] = 'Update';
?>
<div class="profile-update">

<h1><?= Html::encode($this->title) ?></h1>

<?= $this->render('_form', [
    'model' => $model,
]) ?>

</div>
```

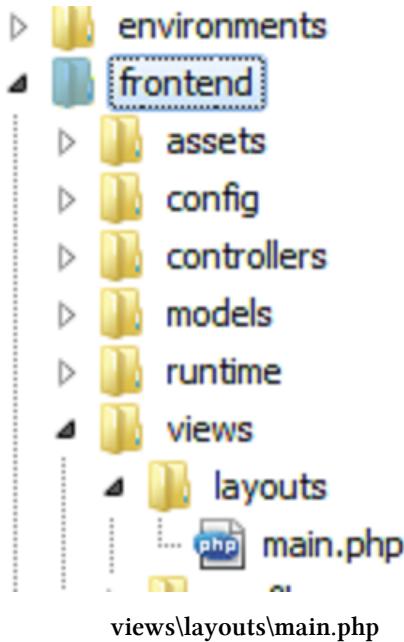
And just for consistency, let's take that 's' out of the word Profile on the view.php breadcrumbs as well:

```
$this->params['breadcrumbs'][] = ['label' => 'Profile', 'url' => ['index']];
$this->params['breadcrumbs'][] = $this->title;
```

That wraps up the changes for the Profile views. You'll notice that we didn't update index.php or _search.php. We don't need those files, they were auto-generated by Gii. Instead of getting rid of them now, we will wait until later in the project to delete because if we change our minds and want to use them, we have them and don't have to recreate. If your sense of workflow is offended by this, then feel free to delete them now.

Site Layout

It's time we modified the site layout to include a link to Profile in the header. We want this link to appear next to the logout link, which is what appears when you are logged in. So we need to modify frontend/views/layout/main.php



This is where the file is located. There is a similar one in the backend, so make sure you are in the right place.

This is what you get out of the box with the advanced template:

```

<?php
use yii\helpers\Html;
use yii\bootstrap\Nav;
use yii\bootstrap\NavBar;
use yii\widgets\Breadcrumbs;
use frontend\assets\AppAsset;
use frontend\widgets\Alert;

/* @var $this \yii\web\View */
/* @var $content string */

AppAsset::register($this);
?>
<?php $this->beginPage() ?>
<!DOCTYPE html>
<html lang="<?= Yii::$app->language ?>">
<head>
    <meta charset="<?= Yii::$app->charset ?>"/>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <?= Html::csrfMetaTags() ?>
    <title><?= Html::encode($this->title) ?></title>

```

```
<?php $this->head() ?>
</head>
<body>
    <?php $this->beginBody() ?>
    <div class="wrap">
        <?php
            NavBar::begin([
                'brandLabel' => 'My Company',
                'brandUrl' => Yii::$app->homeUrl,
                'options' => [
                    'class' => 'navbar-inverse navbar-fixed-top',
                ],
            ]);
            $menuItems = [
                ['label' => 'Home', 'url' => ['/site/index']],
                ['label' => 'About', 'url' => ['/site/about']],
                ['label' => 'Contact', 'url' => ['/site/contact']],
            ];
            if (Yii::$app->user->isGuest) {
                $menuItems[] = ['label' => 'Signup', 'url' => ['/site/signup']];
                $menuItems[] = ['label' => 'Login', 'url' => ['/site/login']];
            } else {
                $menuItems[] = [
                    'label' => 'Logout (' . Yii::$app->user->identity->username . ')',
                    'url' => ['/site/logout'],
                    'linkOptions' => ['data-method' => 'post']
                ];
            }
            echo Nav::widget([
                'options' => ['class' => 'navbar-nav navbar-right'],
                'items' => $menuItems,
            ]);
            NavBar::end();
        ?>

        <div class="container">
            <?= Breadcrumbs::widget([
                'links' => isset($this->params['breadcrumbs']) ?
                    $this->params['breadcrumbs'] : []
            ]) ?>
            <?= Alert::widget() ?>
            <?= $content ?>
        </div>
```

```

</div>

<footer class="footer">
    <div class="container">
        <p class="pull-left">&copy; My Company <?= date('Y') ?></p>
        <p class="pull-right"><?= Yii::powered() ?></p>
    </div>
</footer>

<?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

This is a nice concise template that does quite a bit. The `AppAsset::register($this);` pulls in the style sheets and js. for the template. Yii 2 utilizes a publishing system to cache assets. You have to pay careful attention to your config files.

Later in the book, we will cover implementing a new asset, but we won't go too deep. This book doesn't really cover fronted decoration in great detail.

But we can spend a minute on describing how layout works. If you look in the body of the file, you'll see:

```
<?= $content ?>
```

That rather inconspicuous statement places the view inside the layout. Yii 2's routing mechanisms know which view and which layout to use. You can use multiple layouts and themes and this subject gets deep quickly, but like I said, we're only covering the surface. You can see the section near the bottom is:

```
<footer class="footer">
```

Everything below that is in the footer section.

Everything above `<?= $content ?>` will be in the header.

Let's pop in our template name into the NavBar widget. It's a single word change, so no Gist:

```
NavBar::begin([
    'brandLabel' => 'Yii 2 Build',
    'brandUrl' => Yii::$app->homeUrl,
    'options' => [
        'class' => 'navbar-inverse navbar-fixed-top',
    ],
]);
```

Profile Link

So the next thing we are going to do is add a link to Profile. Now, the way we setup the Profile controller is that the index action first tests to see if a record exists. And if so, it redirects to the view page and if not, redirects to the create view. So we really only need one link to Profile and it covers everything. The update and delete views are already linked from within the view.php file, so no need to create that nav.

What we do want to test for however, is that the user is logged in. We do not want to show the profile link if the user is not logged in. Also note, we do not need to provide a get variable. We eliminated the need for that with the way we wrote the controller logic.

Insert the following:

```
$menuItems[] = ['label' => 'Profile', 'url' => ['/profile/view']];
```

Put this in the else statement between the opening and closing NavBar:

```
if (Yii::$app->user->isGuest) {
$menuItems[] = ['label' => 'Signup', 'url' => ['/site/signup']];
$menuItems[] = ['label' => 'Login', 'url' => ['/site/login']];
} else {
$menuItems[] = ['label' => 'Profile', 'url' => ['/profile/view']];
$menuItems[] = [
    'label' => 'Logout (' . Yii::$app->user->identity->username . ')',
    'url' => ['/site/logout'],
    'linkOptions' => ['data-method' => 'post']
];
}
```

So we're using Yii::\$app->isGuest method to determine whether or not the user is logged in, then if so, we show them the profile link. The \$menuItems array format works because it is in between NavBar::begin and NavBar::end.

Ok, so with all these changes in place, we can login as a user and play with creating a profile, updating, deleting, etc. Go ahead and make sure everything is working properly.

Try appending an invalid get variable to the url, such as:

```
http://yii2build.com/index.php?r=profile/update&id=7
```

You'll see that no matter you put into that id, it will return only the current user's update link. So at this point, this should all be working as expected.

If not, retrace your steps and try to find the typo.

Now before we close out the chapter, let's get rid of the ugly date input on the _form partial.

DatePicker

Let's make sure we remember to include the use statement on `_form.php`:

```
use yii\jui\DatePicker;
```

This is a 3rd party library that is not included in the base install of Yii. When we updated composer to make sure we had Gii, we were supposed to also install the jui library. You should have:

```
"yiisoft/yii2-jui": "*"
```

Your `composer.json` file in `yii2build` should look like the following:

```
"require-dev": {
    "yiisoft/yii2-codeception": "*",
    "yiisoft/yii2-debug": "*",
    "yiisoft/yii2-gii": "*",
    "yiisoft/yii2-faker": "*",
    "yiisoft/yii2-jui": "*"
},
```

You can see the last line we added “`yiisoft/yii2-jui`”: “`*`” Don't forget the comma on the previous line.

If that was not there, then you don't have it, so you need to add that line now. Then go to your command line and type in:

```
\var\www\yii2build>composer update
```

This will bring in the `yii2-jui` dependencies. So now we are set to use it.

The next thing is to replace the form input line in `_form.php` with:

```
<?php echo $form->field($model, 'birthdate')->widget(DatePicker::className(),
    ['clientOptions' => ['dateFormat' => 'yy-mm-dd']]); ?>
```

Go ahead and make the change and refresh your page. If all went well, you should now see a difference on your update and create form from your Profile.

Please note that is not going to work correctly. I wrote that from an earlier version of the widget, Yii 2.0 and the configuration is now a bit different, since we are working in Yii 2.0.3 or above.

Having to solve the problem the old config created is instructive, so for the moment, I'm going to pretend like the solution doesn't exist and we need to come up with a fix. It's not that uncommon for you to face such issues and this is a simple scenario to work with and learn how to overcome it.

The problem here boiled down to the fact the widget was not respecting the date format and the default format it uses would not pass validation. So essentially, update and create are dead until we fix it.

The default date format of the widget is:

MMM d, Y

This translates to a date like Oct 14, 2014

The format we need for datetime in MySQL is:

Y-m-d

This translates to 2014-10-14.

So obviously a conversion problem. We can solve this by creating a before validate method on the Profile model. Yii 2 has this really handy method named beforeValidate, and it will automatically be called as long as the method calls the parent class. Here is the code, which to be clear, goes on the Profile Model in frontend/models/Profile.php:

Gist:

[BeforeValidate Profile](#)

From book:

```
public function beforeValidate()
{
    if ($this->birthdate != null) {

        $new_date_format = date('Y-m-d', strtotime($this->birthdate));
        $this->birthdate = $new_date_format;
    }

    return parent::beforeValidate();
}
```

So if the birthdate attribute is not empty, take the \$this instance of birthdate and format it using:

date('Y-m-d', strtotime(\$this->birthdate))

Those are two built-in PHP functions there, date and strtotime. Anyway, we set \$this->birthdate to the \$new_date_format, which is now holding the birthdate that was handed in, in the correct date format.

Then we return Parent::beforeValidate(); to insure the method gets called and that's it.

Ok, so you can see how to work around a formatting issue like that.

A reader was kind enough to supply the newer config for the widget, which takes care of the problem, and he also added a cool option to make the year easier to set.

Gist:

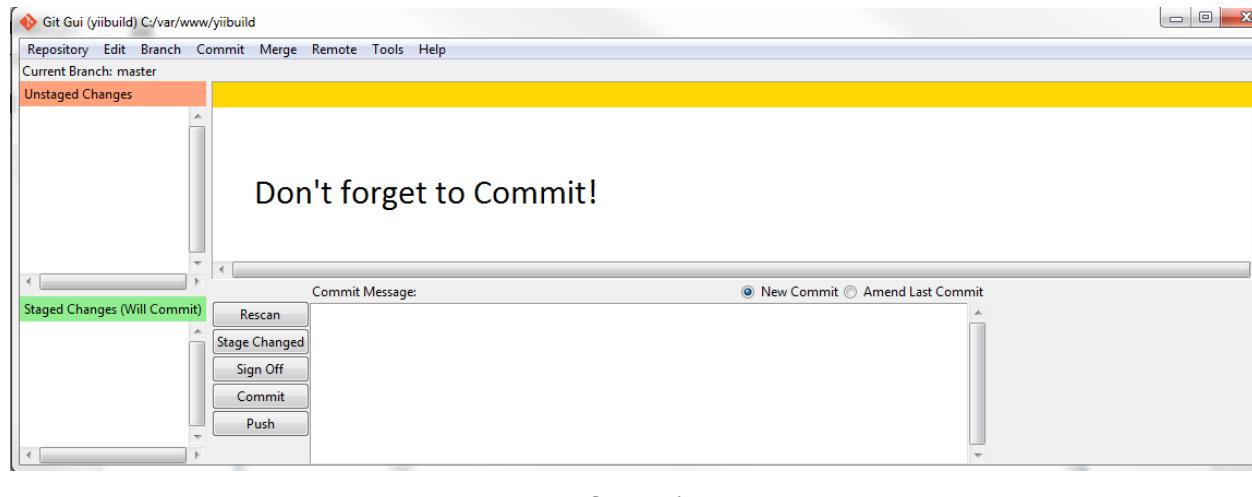
[DatePicker Solution](#)

From book:

```
<?php echo $form->field($model, 'birthdate')->
    widget(DatePicker::className(), [
        'dateFormat' => 'yyyy-MM-dd',
        'clientOptions' => [
            'yearRange' => '-115:+0',
            'changeYear' => true
        ]
    ]) ?>
```

Please note that when you use the dropdown to select year, you must also select the day or it won't be set. Also, don't forget to remove the beforeValidate method from Profile.php if you added it there, it was just for demonstration and is not necessary.

Summary



So there you have it, a working user profile. You can just click on the link now and test all the pages. You can see we finally made the leap into development, no longer confined to just learning about how things work.

You saw that we got to use our helper class to help us test whether or not a user has a profile. We built our helpers in advance because I covered the concept as a whole outside of workflow. When you are developing an application, you will think of things like that as you go, which is perfectly fine.

You also got to see a large number of efficiencies handed to us by Yii 2. We got to use the Gii tool, which handed us a ready-made architecture that only needed a little tweaking. The controller needed the most work, but only because these are private, user-owned records and the template is geared towards public records. When we do the backend, the Gii output will more closely match what we want.

And finally we modified our views, making the application more intuitive. And right away we feel the difference. It's starting to come together.

Chapter Nine: Upgrade and Access Control

We want to be able to manage content on our application that requires an upgrade. If you recall, on our `PremissionHelpers::requireUpgradeTo($user_type_name)` method, we test to see if the user type matches the string handed in the signature, and if not, it redirects them to ‘upgrade/index.’

This is a simple to implement feature if you want to build an application that has content reserved for paying members. But we can’t demonstrate this until we build the upgrade controller and the corresponding view file, so let’s do that now.

Point your browser to:

```
http://www.yii2build.com/index.php?r=gii
```

Let’s put our new controller in the frontend because our frontend users will be the ones who need to upgrade. Put this in the field for Controller Class:

```
frontend\controllers\UpgradeController
```

Whether you are creating a frontend or backend controller, they will follow the above convention, just use the appropriate starting folder.

Make sure the following fields are set if they do not auto-populate:

Action Ids: index

Leave the view path empty, it will know what the correct path is because we are using the default setup.

It should look like this:

Controller Generator

This generator helps you to quickly generate a new controller class with one or several controller actions and their corresponding views.

Controller Class
frontend\controllers\UpgradeController

Action IDs
index

View Path

Base Class
yii\web\Controller

Code Template
default (C:\war\www\yii2build\vendor\yisoft\yii2-gii\generators\controller/default)

Preview

Upgrade Controller in Gii

Ok, let's generate the code. This will not only create the controller, but also the corresponding view folder and file. In this case we only have one action we want to create, which is index. You can do more than one action by separating them with commas. In this case, the index action will render the index view, which you would use to offer the user payment options.

We're only going to be mocking up the payment options page, we'll just put a little content on our index view, that's as far as our instructions will go for this book. If you actually want to implement real payment options, I would recommend checking out Stripe. For small companies, this solution makes sense, and they have a lot of documentation for integration:

[Stripe](#)

Also, there is:

[PayPal](#)

You may want to offer both options.

Upgrade Controller

Ok, let's go back to Gii. Once you run generate on Gii, you will get:

UpgradeController.php

```
<?php

namespace frontend\controllers;

class UpgradeController extends \yii\web\Controller
{
    public function actionIndex()
    {
        return $this->render('index');
    }

}
```

Upgrade View

Pretty simple, it just renders the view, which is in views/upgrade/index.php:

```
<?php
/* @var $this yii\web\View */
?>
<h1>upgrade/index</h1>

<p>
    You may change the content of this page by modifying
    the file <code><?= __FILE__; ?></code>.
</p>
```

Again, nothing to it. So since we're just mocking up, we could leave it at that. Now to test how this works, we can simply add one line in our ProfileController.php file.

Require Upgrade To

Ok, now that we've jumped back to our **Profile Controller** in frontend/controllers/ProfileController.php, let's make it the first line of the actionUpdate:

```
PermissionHelpers::requireUpgradeTo('Paid');
```

So the entire method should look like this:

```

public function actionUpdate()
{
    PermissionHelpers::requireUpgradeTo('Paid');

    if ($model = Profile::find()->where(['user_id' =>
        Yii::$app->user->identity->id])->one()) {

        if ($model->load(Yii::$app->request->post()) && $model->save()) {
            return $this->redirect(['view', 'id' => $model->id]);
        } else {

            return $this->render('update', [
                'model' => $model,
            ]);
        }
    } else {

        throw new NotFoundHttpException('No Such Profile.');
    }
}

```

So we made no change except for adding the first line, which now tells the controller that the user has to be a user_type_name ‘Paid’. Make sure you save the change, then you can test this if you login with an existing user that only has the default user_type of Free.



Tip

You should register several users through the application and then play around with their user_type_id settings in PhpMyAdmin, so you can try different scenarios.

Now that you have a test user that is a ‘Free’ user_type, you can try to update your profile, and when you do, it will redirect you to the upgrade view. How simple is that?

Access Control

While we’re here in the Profile controller, we can talk a little more about access control. One of the more not-so-obvious ideas would be to require the user to have an Active status to access the Profile controller. Now of course the application requires it for login, so no one gets to login if they do not have active status. But what if someone downgrades their status, then hits the back button? Theoretically they can still access everything and this is sloppy control over the content.

So we're going to add an access rule that requires a status of active. Don't worry, this will be really simple because we already anticipated this and built the helper for it. And also, Yii 2's behaviors class includes access control, which we've already seen in action. Now we're just going to add a little more to it:

Change the existing behaviors method in the Profile controller to the following:

Gist:

Profile Controller Behaviors

From book:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => \yii\filters\AccessControl::className(),
            'only' => ['index', 'view', 'create', 'update', 'delete'],
            'rules' => [
                [
                    'actions' => ['index', 'view', 'create', 'update', 'delete'],
                    'allow' => true,
                    'roles' => ['@'],
                ],
            ],
        ],
        'access2' => [
            'class' => \yii\filters\AccessControl::className(),
            'only' => ['index', 'view', 'create', 'update', 'delete'],
            'rules' => [
                [
                    'actions' => ['index', 'view', 'create', 'update', 'delete'],
                    'allow' => true,
                    'roles' => ['@'],
                    'matchCallback' => function ($rule, $action) {
                        return PermissionHelpers::requireStatus('Active');
                    }
                ],
            ],
        ],
    ],
}
```

```
'verbs' => [
    'class' => VerbFilter::className(),
    'actions' => [
        'delete' => ['post'],
    ],
],
];
}
```

If you get an error on not finding the VerbFilter class, make sure to have this use statement at the top of the file:

```
use yii\filters\VerbFilter;
```

This was a class path that got changed by the Yii 2 Framework itself during the course of time when I was writing the book, so if I missed that anywhere else, that is the fix for it. When you are developing an application, it's easy to miss a use statement. The good news is that Yii 2 will complain nicely and point right to the missing file, you just need to format the use statement properly.

The complete use statements should look like this:

```
use Yii;
use frontend\models\Profile;
use frontend\models\search\ProfileSearch;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;
use common\models\PermissionHelpers;
use common\models\RecordHelpers;
```

I included the one for ProfileSearch in case you want to play around with it. Feel free to eliminate any use statement that is not necessary to the controller.

Anyway, back to the behaviors. You can see that we added another array labeled 'access2':

```
'access2' => [
    'class' => \yii\filters\AccessControl::className(),
    'only' => ['index', 'view', 'create', 'update', 'delete'],
    'rules' => [
        [
            'actions' => ['index', 'view', 'create', 'update', 'delete'],
            'allow' => true,
            'roles' => ['@'],
            'matchCallback' => function ($rule, $action) {
                return PermissionHelpers::requireStatus('Active');
            }
        ],
    ],
],
```

Now the key of the array, access2, is just a string and you can call it anything you want. So you can see what we did here. We copied the access array and added the ‘matchCallback’ element, which is a php callable that is looking for true or false.

Fortunately, we have a ready-made PermissionHelpers method that returns true or false, in this case, checking to see if the current user has a status of active. If not, it does not allow access, and since we have set the rule to all actions, they cannot access anything that the Profile Controller controls. It’s all done very simply.

Now I have created it this way because I want to demonstrate that you can have multiple layers of access rules. But in reality, since the matchCallback applies to all actions, as does the other rules, you could have simply added matchCallback to the first array, instead of creating access2.

You can also nest arrays under rules when you have rules that only apply to certain actions, for example:

```
return [
    'access' => [
        'class' => \yii\filters\AccessControl::className(),
        'only' => ['index', 'view', 'create', 'update', 'delete'],
        'rules' => [
            [
                'actions' => ['create', 'update', 'delete'],
                'allow' => true,
                'roles' => ['@'],
            ],
        ],
],
```

```
    ],
    'rules' => [
        [
            'actions' => ['index', 'view', 'create', 'update', 'delete'],
            'allow' => true,
            'roles' => ['@'],
            'matchCallback' => function ($rule, $action) {
                return PermissionHelpers::requireStatus('Active');
            }
        ],
    ],
],
],
```

The access control class will iterate for each set of rules. So this is a very flexible and easy to use method of access control.

You can see that when we simply want to restrict access, we can use behaviors. When we need to restrict based on a condition and redirect or do some other action, we can build a helper that will keep our controller code very clean.

Let's return now to the Upgrade Controller. This time we will use the more concise version of behaviors, for cleaner code:

Gist:

Upgrade Behaviors

From book:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => \yii\filters\AccessControl::className(),
            'only' => ['index'],
            'rules' => [
                [
                    'actions' => ['index'],
                    'allow' => true,
                    'roles' => ['@'],
                ],
            ],
        ],
    ];
}
```

```

        'allow' => true,
        'roles' => [ '@' ],
        'matchCallback' => function ($rule, $action) {
            return PermissionHelpers::requireStatus('Active');
        }
    ],
],
];

'verbs' => [
    'class' => VerbFilter::className(),
    'actions' => [
        'delete' => ['post'],
    ],
],
];
}
}

```

Now we will have to add some use statements, when it's complete it should look like this,

Gist:

Use Statements

From book:

```

use Yii;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;
use common\models\PermissionHelpers;
use common\models\RecordHelpers;
use frontend\models\Profile;

```

Passing A Variable From the Controller

One other small change I'm going to make to the Upgrade controller is that I'm going to add a variable to pass to the view:

```
$name = Yii::$app->user->identity->username;
```

I will do this in actionIndex:

```
public function actionIndex()
{
    $name = Yii::$app->user->identity->username;

    return $this->render('index', ['name' => $name]);
}
```

So you can see I've simply set \$name to the username of the current user. I'm doing this to show you how easy it is to move variables and objects into the view in the render method. You just add the array with the name of the element and the variable as value, like so:

```
return $this->render('index', ['name' => $name]);
```

If you have more than one object or variable, you can separate them by commas in the same array.

So now that we have the username available to us, let's just test it in the view. Replace index.php with:

Gist:

Upgrade Index

From book:

```
<?php
/* @var $this yii\web\View */
?>
<h1>Hey "= echo $name; ?&gt;, " This Requires Upgrade&lt;/h1&gt;

&lt;p&gt;
    You can get the access you want by upgrading, but &lt;?php echo $name; ?&gt;,
    that's not all. You get to go everywhere, isn't that cool?
&lt;/p&gt;</code
```

This is just goofy fun, but you get the point. So that's just a simple variable, let's try an object:

Gist:

Upgrade Index 2

From book:

```

public function actionIndex()
{
    $name = Profile::find()->where(['user_id' =>
        Yii::$app->user->identity->id])->one();

    return $this->render('index', ['name' => $name]);
}

```

Here we are setting \$name to an instance of Profile, where the user_id is the current user. To get this to work, we will have to make sure we have a use statement to access Profile:

```
use frontend\models\Profile;
```

Then in our view, we can change it to:

Gist:

Upgrade Index View

From book:

```

<?php
/* @var $this yii\web\View */
?>
<h1>Hey " <?php echo $name->first_name; ?>, " This Requires Upgrade</h1>

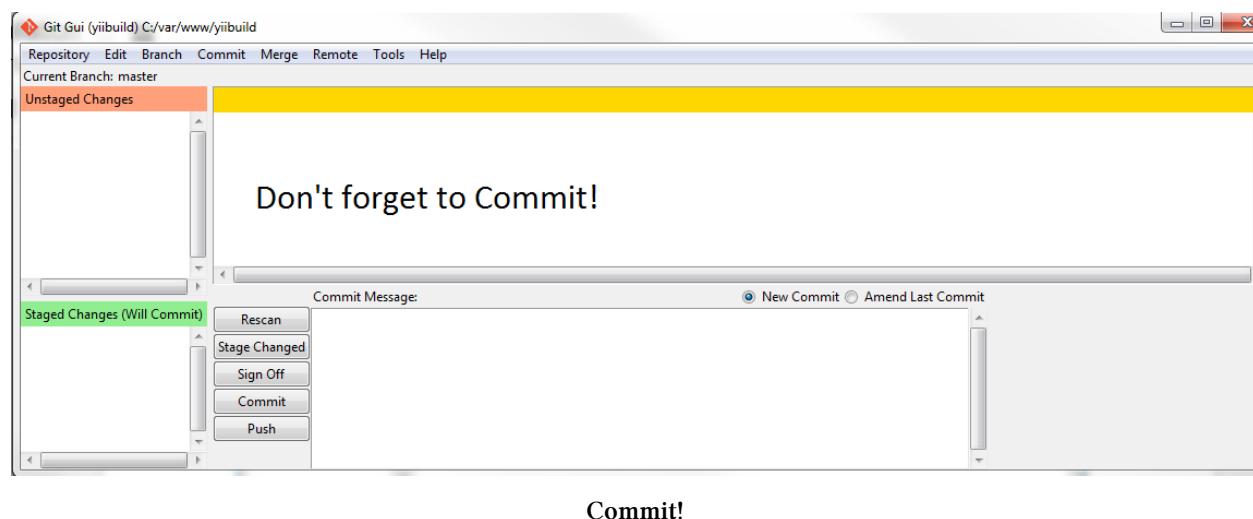
<p>
    You can get the access you want by upgrading, but
    <?php echo $name->first_name; ?> , that's not all.
    You get to go everywhere, isn't that cool?
</p>

```

So now we have personalized the output to the user's first name. You can have all kinds of fun with this, but the point is that it is extremely simple to bring in an object and access it. Make sure you are logged in when you try this or it won't work.

Later, when we code the backend, we will return objects holding lists of users and profiles, using Yii 2's built-in iterator and widgets.

Summary



Commit!

In this chapter, we took a little more control over our application. We created an upgrade controller and view and then enforced rules to bring the user to the page if they didn't meet the minimum user type allowed for access. We did this by adding a `PermissionsHelper` method, `requireUpgradeTo('Paid')` to the update method on the profile controller.

We also added a requirement for the user to have a status of active. This tightens up security and access and did this by adding a `matchCallback` requirement to the access rules in the behaviors method. These were very simple changes that didn't bloat or confuse the code, in part due to the fact that we extracted out logic to our helpers.

Finally, we played around with moving variables and objects from the controller to the view, to give us an idea of easy it is to work with the data that is accessible to us.

It took a while for us to get to a point where we could utilize the models we built and have a little fun with it, but at least by now you should be starting to get a sense of what development in Yii 2 is like.

Chapter Ten: Homepage Social Widgets

Implementing Homepage Social Widgets

We going to decorate the Home page just a little. We want to add some social widgets and a couple of little things that spice up the template. We can't do too much because this is just meant to be a starting point for other applications.

Index

Open up frontend/views/site/index.php. We will start by adding a couple of use statements:

Gist:

[Use Statements Site Index](#)

From book:

```
use \yii\bootstrap\Modal;
use kartik\social\FacebookPlugin;
use \yii\bootstrap\Collapse;
use \yii\bootstrap\Alert;
use yii\helpers\Html;
```

Those go at the top of the file under the opening Php tag.

If you have not already imported the Kartik social extension, we need to do it now. Check for the following in your composer.json file:

```
"minimum-stability": "stable",
"require": {
    "php": ">=5.4.0",
    "yiisoft/yii2": "*",
    "yiisoft/yii2-bootstrap": "*",
    "yiisoft/yii2-swiftmailer": "*",
    "kartik-v/yii2-social": "dev-master",
    "fortawesome/fontawesome-free": "4.2.0"
},
}
```

You can see it on the 2nd to last line. If you don't by this point have the font-awesome line in your composer.json, you should add that too, we will need it later. Then run composer update from the command line, like we have done many times before.



Composer Update

That should import the extension, if you did not have it already. You can check under your vendor directory for a folder named Kartik-v, which is the folder for the extension.



Tip

You can find many useful extensions by Kartik at his site, [Krajee.com](#). He is a superstar developer. As of this writing, Kartik has 28 extensions/goodies that cover everything from the social widget we're using here to GridView extensions and more. If you do use his extensions, be kind and donate if you can. Donations keep him working on new things to add to the framework and that helps everyone.

Facebook Widget

Next, let's go back to the Site view file Index.php and change the title to:

```
$this->title = 'Yii 2 Build';
```

Find the first div and replace it.

Gist:

[1st Div Site Index](#)

From book:

```
<div class="site-index">

    <div class="jumbotron">

        <?php if (Yii::$app->user->isGuest) {
            echo Html::a('Get Started Today', ['site/signup'],
                ['class' => 'btn btn-lg btn-success']);} ?>
        </p>

        <h1>Yii 2 Build</h1>

        <p class="lead">Use this Yii 2 Template to start Projects.</p>

        <br/>

        <?php echo FacebookPlugin::widget(['type'=>FacebookPlugin::LIKE,
            'settings' => []]); ?>

    </div>
```

Now if you save that and hit refresh, you will get the following error:

```
Invalid Configuration - yii\base\InvalidConfigException
The Facebook 'appId' has not been set.
```

Facebook App Setup

So what we need to do is set up our Facebook app, both in the Yii 2 configuration and by setting up an actual Facebook app.

Note that you will need an actual Facebook account to follow along. If you do not have one and do not want or cannot get one, then you will have to skip this lesson and remove all social widgets from the project.

I'm going to continue on, assuming that you have a Facebook account. So start by going to your facebook account and logging in. At the bottom of the page, find the developers link.

Then follow step by step for setting up an app. I will provide a lot of screenshots here for reference, but please keep in mind that things change over time, and it might not look the same. At any rate, it's fairly intuitive, so you should be able to figure it out.

Go to facebook and find your settings link from the down arrow all the way the right:



Facebook Nav

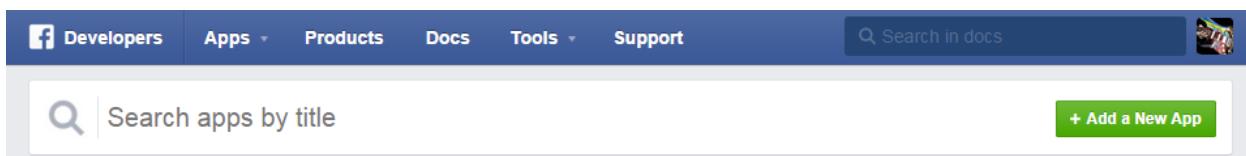
Once you are on your settings page, you can get to the footer and the developers link:



Facebook Footer

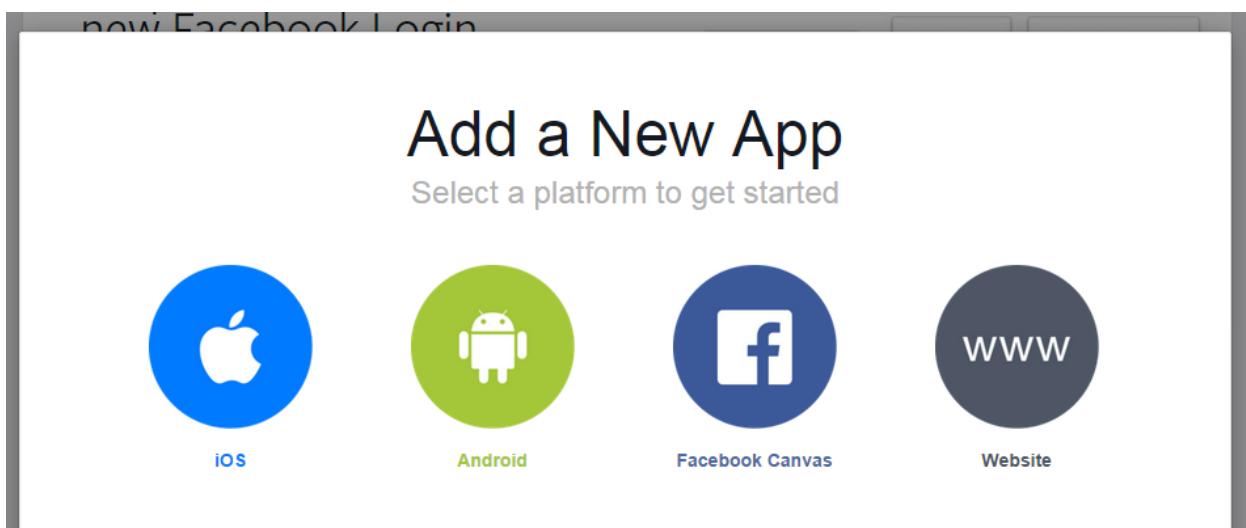
or you can just go to:

<https://developers.facebook.com/apps>



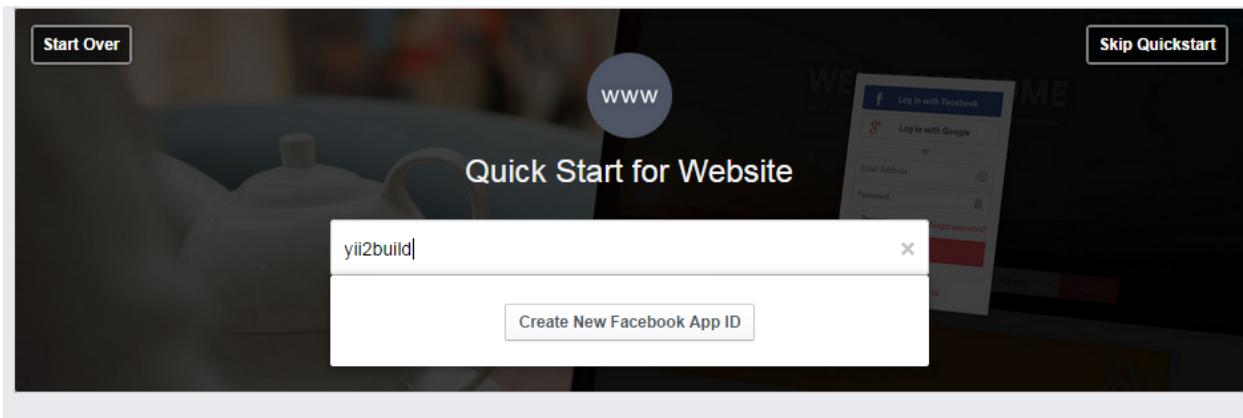
Facebook App Screen

Select add new app:



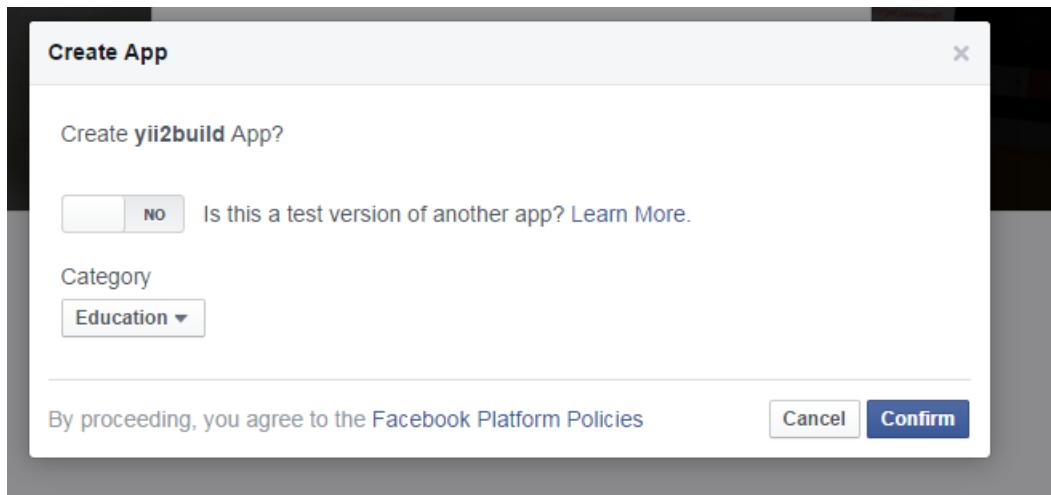
Facebook Platform

select website as platform and then choose name for app:



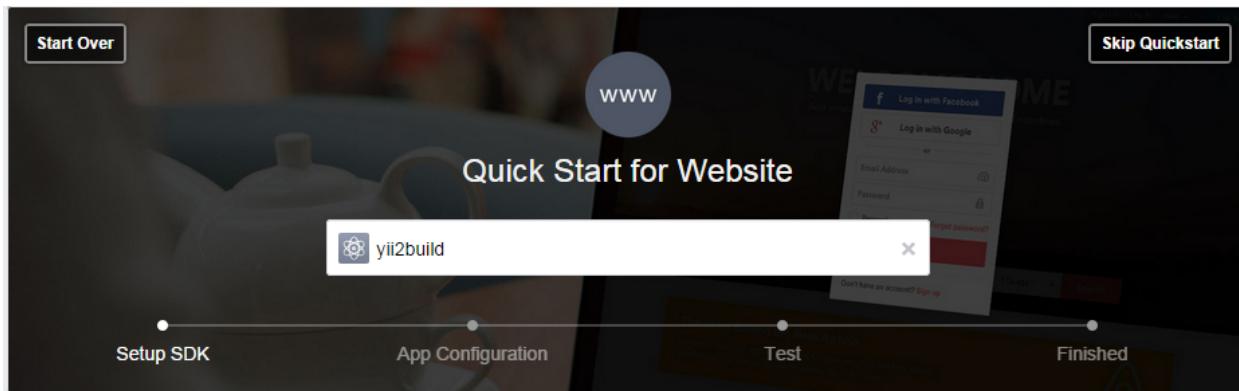
Facebook Select

Select category and confirm:



Facebook Confirm

You get the quick start screen:

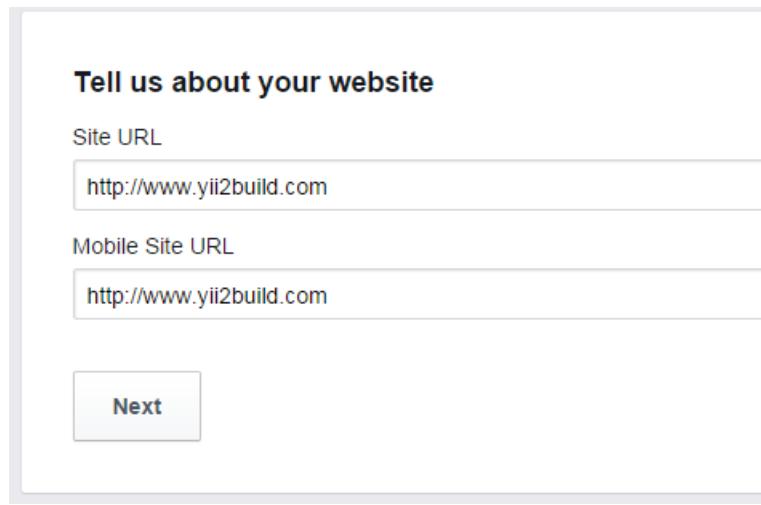


The screenshot shows the 'Quick Start for Website' interface. At the top, there are 'Start Over' and 'Skip Quickstart' buttons. A large circular 'www' icon is in the center. Below it, the title 'Quick Start for Website' is displayed. A progress bar at the bottom shows four steps: 'Setup SDK' (selected), 'App Configuration', 'Test', and 'Finished'. A modal window titled 'yii2build' is open in the center. The main content area has a heading 'Setup the Facebook SDK for JavaScript' and a snippet of code:

```
<script>
  window.fbAsyncInit = function() {
    FB.init({
      ...
    });
  };
</script>
```

Facebook Quick Start

At the bottom of the quick start screen, fill in your site url. Please note that I'm using [Yii2build.com](http://www.yii2build.com) and you will not be able to use that. In the example implementation that we see later, make sure you use your domain and not [Yii2build.com](http://www.yii2build.com). Your domain does not need to be a live site, so go ahead and enter your domain:



The form is titled 'Tell us about your website'. It has two fields: 'Site URL' containing 'http://www.yii2build.com' and 'Mobile Site URL' also containing 'http://www.yii2build.com'. A 'Next' button is at the bottom.

Facebook Quick Start Form

We get a SDK setup finished (but we are not done, click on the Skip to Developer Dashboard link under Next Steps).

Test your Facebook Integration

Now that you've got the SDK setup, you can use it to perform a few common tasks. Social Plugins such as the Like Button and Comments Plugin can be inserted into HTML pages using the JavaScript SDK.

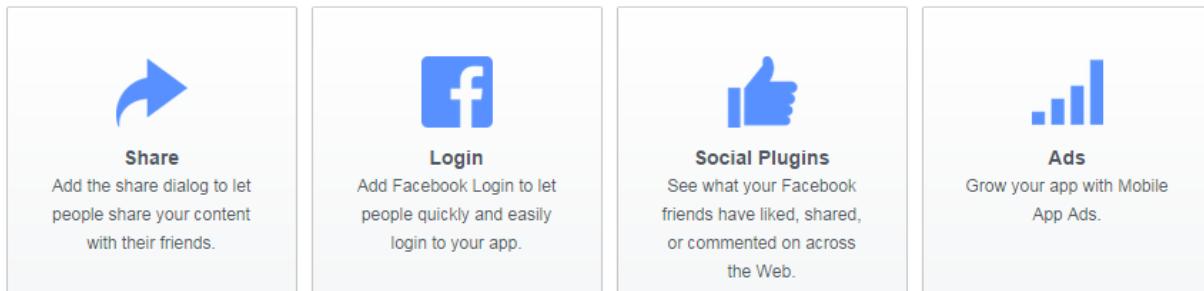
Let's try adding a Like button, just copy and paste the line of code below anywhere inside the `<body>` of your page:

```
<div  
  class="fb-like"  
  data-share="true"  
  data-width="450"  
  data-show-faces="true">  
</div>
```

Reload your page, and you should see a Like button on it.

Next Steps

Congratulations! You have added the Facebook SDK to your project. You are now in the next stage in integrating your app with Facebook. What do you want to do next? Skip to Developer Dashboard or Documentation



Facebook Quick Start Form Finished

That will bring you to the Dashboard, where you click on settings:

Facebook App Dashboard

Obviously, I scratched out my app id. Yours will appear there. In order to copy the app secret, select the show button.

You will need to copy the app id and the app secret into the appropriate area in config as I describe below, but don't try to do it yet.

```
// the global settings for the facebook plugins widget

'facebook' => [
    'appId' => 'your id',
    'secret' => 'your secret',
],
```

We haven't added that yet, but we will in a moment. Also, remember to provide a working email address inside the Facebook settings page, otherwise the app won't work.

Facebook Configuration

Ok, on to setting up our common config settings to recognize Kartik's social module, which will allow us to use his social widgets. If you recall, we included:

```
"kartik-v/yii2-social": "dev-master",
```

in our composer.json file. Now we need to tell config that it is there. The original common/config/main.php is:

```
<?php
return [
    'vendorPath' => dirname(dirname(__DIR__)) . '/vendor',
    'components' => [
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
    ],
];
];
```

If you are not seeing that, make sure you are in the right file. Yii2 has several files named main.php in config. You need the one that is in the common folder. You need to change this to:

Gist:

[Common/Config/Main Update](#)

From book:

```
<?php
return [
    'vendorPath' => dirname(dirname(__DIR__)) . '/vendor',
    'extensions' => require(__DIR__ . '/../../.vendor/yiisoft/extensions.php'),
    'modules' => [
        'social' => [
            // the module class
            'class' => 'kartik\social\Module',

            // the global settings for the disqus widget
            'disqus' => [
                'settings' => ['shortname' => 'DISQUS_SHORTNAME'] // default settings
            ],
            // the global settings for the facebook plugins widget
        ],
    ],
];
```

```
'facebook' => [
    'appId' => 'your id',
    'secret' => your 'secret',
],

// the global settings for the google plugins widget
'google' => [
    'clientId' => 'GOOGLE_API_CLIENT_ID',
    'pageId' => 'GOOGLE_PLUS_PAGE_ID',
    'profileId' => 'GOOGLE_PLUS_PROFILE_ID',
],

// the global settings for the google analytic plugin widget
'googleAnalytics' => [
    'id' => 'TRACKING_ID',
    'domain' => 'TRACKING_DOMAIN',
],

// the global settings for the twitter plugin widget
'twitter' => [
    'screenName' => 'TWITTER_SCREEN_NAME'
],
],
// your other modules
],

'components' => [
    'cache' => [
        'class' => 'yii\caching\FileCache',
    ],
],
];
];
```

Take your Facebook appId and your secret and copy them into:

```
// the global settings for the facebook plugins widget

'facebook' => [
    'appId' => 'your id',
    'secret' => 'your secret',
],
]
```

Please note that when you see the facebook widgets implemented in my examples, I use Yii2Build.com as my domain. Obviously you will use your own domain example.

Extensions

Just a note about how Kartik's social module is referenced in config. Under vendor/yiisoft is a file named extensions.php and this holds the alias for the module:

```
'kartik-v/yii2-social' =>
array (
    'name' => 'kartik-v/yii2-social',
    'version' => '9999999-dev',
    'alias' =>
        array (
            '@kartik' => $vendorDir . '/kartik-v/yii2-social',
        ),
),
),
```

You can see kartik-v/yii2-social is the same name as it has in composer.json when we included it there. Composer automatically entered it into the extensions file for us. Also note the array for alias '@kartik' => \$vendorDir . '/kartik-v/yii2-social , which allows for this line in config:

```
return [
    'vendorPath' => dirname(dirname(__DIR__)) . '/vendor',
    'extensions' => require(__DIR__ . '/../../vendor/yiisoft/extensions.php'),
    'modules' => [
        'social' => [
            // the module class

            'class' => 'kartik\\social\\Module',
        ],
    ],
],
```

I included the vendor path for reference, but that's not new, we did not add that, it was already there. We did tell it to use extensions in the path specified in the config and that is where it finds the alias to connect everything.

So basically, it says for the social module, use the class kartik\\socialModule. There is no kartik folder, but kartik is an alias for \$vendorDir . '/kartik-v/yii2-social, which when combined with /social/Module, provides the location of the class.

The actual widget is named FacebookPlugin and that is named spaced at the top of the file:

```
use kartik\social\FacebookPlugin;
```

The alias used in the extension works in the namespace as well. So that is how Yii knows where to find everything. Anyway, the site index page should not be returning an error, once you have all the above changes in place.

HTML Helper

Ok, moving back to the index.php file for site. Let's make note of the fact that I made the 'Get Started Today' button conditional on being logged out, no need to display it if logged in, since it links to the signup form. Also, I used the HTML class that we identified in our Use statement:

```
use yii\helpers\Html;
```

Then I used the a method of the Html class to format the link:

```
<p>

<?php

if (Yii::$app->user->isGuest){

    echo Html::a('Get Started Today', ['site/signup'],
        ['class' => 'btn btn-lg btn-success']);

}

?>

</p>
```

3 parameters for method a, the first one is the text of the link, 'Get Started Today.' The next one is the controller/action, 'site/signup' in this case. And then we get the class for the css, which, as we see in the code, is a button.

A couple of things to note. The a method is pretty smart. It knows that if you are in a profile view.php file, you can just hand it the action you want, for example, update, and it will know will route you to the correct update action. Also, note that you can add, separated by a comma, an option parameter in that array to hand in the get variable, for example:

```
[ 'update', 'id' => $model->id],
```

Ok, moving down the remainder of the page, remove this code:

```
<div class="body-content">

    <div class="row">
        <div class="col-lg-4">
            <h2>Heading</h2>

            <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
                do eiusmod tempor incididunt ut labore et
                dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
                exercitation ullamco laboris nisi ut aliquip
                ex ea commodo consequat. Duis aute irure dolor in
                reprehenderit in voluptate velit esse cillum dolore eu
                fugiat nulla pariatur.</p>

        <p><a class="btn btn-default" href="http://www.yiiframework.com/doc/">
            Yii Documentation &raquo;</a></p>
        </div>
        <div class="col-lg-4">
            <h2>Heading</h2>

            <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
                do eiusmod tempor incididunt ut labore et
                dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
                exercitation ullamco laboris nisi ut aliquip
                ex ea commodo consequat. Duis aute irure dolor in
                reprehenderit in voluptate velit esse cillum dolore eu
                fugiat nulla pariatur.</p>

        <p><a class="btn btn-default" href="http://www.yiiframework.com/forum/">
            Yii Forum &raquo;</a></p>
        </div>
        <div class="col-lg-4">
            <h2>Heading</h2>

            <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
                do eiusmod tempor incididunt ut labore et
                dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
                exercitation ullamco laboris nisi ut aliquip
                ex ea commodo consequat. Duis aute irure dolor in
```

```
        reprehenderit in voluptate velit esse cillum dolore eu  
        fugiat nulla pariatur.</p>  
  
<p><a class="btn btn-default" href="http://www.yiiframework.com/extensions/">  
        Yii Extensions &raquo;</a></p>  
    </div>  
</div>  
  
</div>  
</div>
```

Replace it with:

Gist:

[Site Index Remainder](#)

From book:

```
<?php  
  
echo Collapse::widget([  
  
    'items' => [  
  
        [  
  
            'label' => 'Top Features' ,  
            'content' => FacebookPlugin::widget([  
  
                'type'=>FacebookPlugin::SHARE,  
                'settings' => ['href'=>'http://www.yii2build.com','width'=>'350']  
  
            ]),  
  
            // open its content by default  
            // 'contentOptions' => ['class' => 'in']  
  
        ],  
  
        // another group item  
  
        [  
    ]
```

```
'label' => 'Top Resources',
'content' => FacebookPlugin::widget([
    'type'=>FacebookPlugin::SHARE,
    'settings' => ['href'=>'http://www.yii2build.com','width'=>'350']

]),

// 'contentOptions' => [],
// 'options' => [],

],
]

]);


Modal::begin([
    'header' => '<h2>Latest Comments</h2>',
    'toggleButton' => ['label' => 'comments'],
    ]);

echo FacebookPlugin::widget([
    'type'=>FacebookPlugin::COMMENT,
    'settings' => ['href'=>'http://www.yii2build.com','width'=>'350']
]);

Modal::end();

?>

<br/>
<br/>

<?Php
```

```
echo Alert::widget([

    'options' => [
        'class' => 'alert-info',
    ],
    'body' => 'Launch your project like a rocket...',
]);

?>

<div class="body-content">

    <div class="row">
        <div class="col-lg-4">
            <h2>Free</h2>

            <p>
<?php

if (!Yii::$app->user->isGuest) {

    echo Yii::$app->user->identity->username . ' is doing cool stuff. ';
}

?>
```

Starting with this free, open source Yii 2 template and it will save you a lot of time. You can deliver projects to the customer quickly, with a lot of boilerplate already taken care of for you, so you can concentrate on the complicated stuff.</p>

```
<p>

<a class="btn btn-default"
href="http://www.yiiframework.com/doc-2.0/guide-index.html">
    Yii Documentation &raquo;
</a>

</p>

<?php

echo FacebookPlugin::widget([
```

```
'type'=>FacebookPlugin::LIKE,
'settings' => []

]);


?>

</div>
<div class="col-lg-4">
    <h2>Advantages</h2>

<p>

Ease of use is a huge advantage. We've simplified RBAC and given you Free/Paid
user type out of the box. The Social plugins are so quick and easy to install,
you will love it!

</p>

<p>

<a class="btn btn-default"
href="http://www.yiiframework.com/forum/">Yii Forum &raquo;</a>

</p>

<?php

echo FacebookPlugin::widget([
    'type'=>FacebookPlugin::COMMENT,
    'settings' => [ 'href'=>'http://www.yii2build.com', 'width'=>'350' ]
]);

?>
</div>
<div class="col-lg-4">
    <h2>Code Quick, Code Right!</h2>

<p>
```

Leverage the power of the awesome Yii 2 framework with this enhanced template. Based Yii 2's advanced template, you get a full frontend and backend implementation that features rich UI for backend management.

```
</p>

<p>

<a class="btn btn-default"
href="http://www.yiiframework.com/extensions/">Yii Extensions &raquo;</a>

</p>

        </div>
    </div>

    </div>
</div>
```

Just a reminder, and this is not part of the code or page, don't forget to save. Also, make sure you use your domain in the social widgets, not [Yii2build.com](#).

Collapse Widget

You can see in the above code that I referenced a collapse widget, so you need to put the use statement at the top of the file:

```
use \yii\bootstrap\Collapse;
```

Which allows you to reference the widget directly through the static call:

```
echo Collapse::widget
```

The widget code is fairly simple to follow:

```
<?php

echo Collapse::widget([
    'items' => [
        [
            'label' => 'Top Features',
            'content' => FacebookPlugin::widget([
                'type'=>FacebookPlugin::SHARE,
                'settings' => ['href'=>'http://www.yii2build.com', 'width'=>'350']
            ]),
            // open its content by default
            // 'contentOptions' => ['class' => 'in']
        ],
        // another group item
        [
            'label' => 'Top Resources',
            'content' => FacebookPlugin::widget([
                'type'=>FacebookPlugin::SHARE,
                'settings' => ['href'=>'http://www.yii2build.com', 'width'=>'350']
            ]),
            // 'contentOptions' => [],
            // 'options' => []
        ],
    ]
]);
```

We give the items labels, ‘Top Features’ and ‘Top Resources’ and then for content we plug in the

FacebookPlugin::widget for sharing. The other options are commented out.

Modal Widget

After that we use a modal with a button to hold facebook comments, again very easy code to understand:

```
Modal::begin([
    'header' => '<h2>Latest Comments</h2>',
    'toggleButton' => ['label' => 'comments'],
]);

echo FacebookPlugin::widget([
    'type'=>FacebookPlugin::COMMENT,
    'settings' => ['href'=>'http://www.yii2build.com','width'=>'350']
]);

Modal::end();
```

Alert Widget

Then I just wanted to play around with an alert widget, so I included:

```
<?Php

echo Alert::widget([
    'options' => [
        'class' => 'alert-info',
    ],
    'body' => 'Launch your project like a rocket...',
]);
?>
```

Probably not how we would actually use an alert, it would be tied to an action, but including it like this gives us an idea of what it looks like.

Then the last thing we did on index.php was this line in the first <p>

```
<p>

<?php

if (!Yii::$app->user->isGuest) {

echo Yii::$app->user->identity->username . ' is doing cool stuff. ';
}

?>
```

Just another little difference to test being logged in or out and to make sure it's getting the right username. And that's it for our starting template for the index page.

Font-Awesome

Yii 2 has a somewhat complex set of methods for handling assets like bootstrap, jquery, etc., and they did this to maximize efficiency by caching resources. I'm not going to cover it much in this book, but we will dabble in it to the extent that we want to have access to font-awesome, a popular css icon library.

Ok, so here's what we need to make this work.

First, if you have not already done so, we are going to pull in font-awesome via composer, add the following to your composer.json file:

```
"minimum-stability": "stable",
"require": {
    "php": ">=5.4.0",
    "yiisoft/yii2": "*",
    "yiisoft/yii2-bootstrap": "*",
    "yiisoft/yii2-swiftmailer": "*",
    "kartik-v/yii2-social": "dev-master",
    "fortawesome/fontawesome": "4.2.0"

},
```

Now if you look closely, it's fortawesome, that is not a typo. Go ahead and run composer update and this will pull it in for you.

Asset Bundle

Then you need to create and add the following file, in two locations, frontend/assets and backend/assets, with corresponding namespaces. The file name is FontAwesomeAsset.php. Here is the entire file:

Gist:

[FontAwesomeAsset.php file](#)

From book:

```
<?php

/**
 * @link http://www.yiiframework.com/
 * @copyright Copyright (c) 2008 Yii Software LLC
 * @license http://www.yiiframework.com/license/
 */

namespace frontend\assets;

use yii\web\AssetBundle;

/**
 * @author Joao Marques<joao@jmf.com>
 */

class FontAwesomeAsset extends AssetBundle
{
    # sourcePath points to the composer package.

    public $sourcePath = '@vendor/fortawesome/fontawesome';

    # CSS file to be loaded.
    public $css = [
        'css/fontawesome.min.css',
    ];

    /**

```

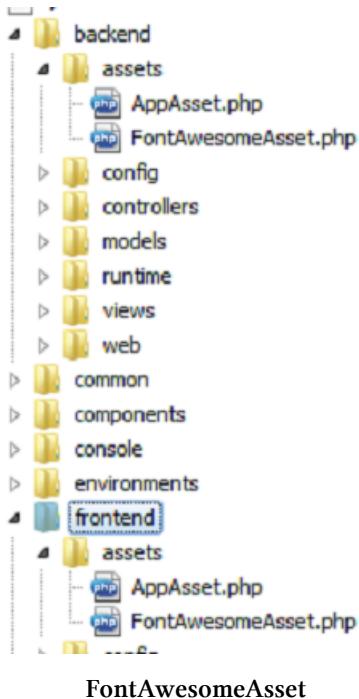
```
* Sets the publishOptions property.  
* Needed because it's necessary to  
*concatenate  
* the namespace value.  
*/  
  
public function init()  
{  
    $this->publishOptions = [  
        'forceCopy' => YII_DEBUG,  
        'beforeCopy' => __NAMESPACE__ .  
            '\FontAwesomeAsset::filterFolders'  
    ];  
  
    parent::init();  
}  
  
/**  
 * Filters the published files and folders.  
 * It's not necessary publish all files and folders  
 * from the font-awesome package  
 * Just the CSS and FONTS folder.  
 * @param string $from  
 * @param string $to  
 * @return bool true to publish to file/folder.  
 */  
  
public static function filterFolders($from, $to)  
{  
    $validFilesAndFolders = [  
        'css',  
        'fonts',  
        'font-awesome.css',  
        'font-awesome.min.css',  
        'FontAwesome.otf',  
        'fontawesome-webfont.eot',  
        'fontawesome-webfont.svg',  
        'fontawesome-webfont.ttf',  
        'fontawesome-webfont.woff',  
    ];
```

```
$pathItems = array_reverse(explode(DIRECTORY_SEPARATOR, $from));

if (in_array($pathItems[0], $validFilesAndFolders)) return true;
else return false;
}
```

Just copy a second version to backend\assets and set the namespace of that copy to:

```
namespace backend\assets;
```



So this is an asset bundle. I should note that I got this from the Yii 2 forum, the author of this file is listed near the top of the file. He did a wonderful job of commenting the code:

<http://www.yiiframework.com/forum/index.php/topic/57902-using-fontawesome/>

You can also check out more on this subject from the Yii 2 guide:

<http://www.yiiframework.com/doc-2.0/guide-structure-assets.html>

Add Font-Awesome to Layout

In frontend/views/layout/main.php, add the following use statement at the top:

```
use frontend\assets\FontAwesomeAsset;
```

And then below that, near the other register call at the additional call to register, like so:

```
AppAsset::register($this);
FontAwesomeAsset::register($this);
```

And that should do it, we should now have access to font-awesome. So let's test this by inserting the following:

```
<i class="fa fa-plug"></i>
```

We're going to put that in two places:

In main.php, the first div, 'brand label':

```
'brandLabel' => 'Yii 2 Build <i class="fa fa-plug"></i>',
```

Then let's go to frontend\views\site\index.php and add it in the first `<h1>` tag like so:

```
<h1>Yii 2 Build <i class="fa fa-plug"></i></h1>
```

That will do it. Now you should have a home page that looks like this:

The screenshot shows the Yii 2 Build homepage. At the top, there is a dark header bar with the site name "Yii 2 Build" and a small icon. To the right are links for "Home", "About", "Contact", "Signup", and "Login". Below the header is a green button labeled "Get Started Today". The main title "Yii 2 Build" is displayed prominently, followed by a large black plug icon. A subtext below the title reads "Use this Yii 2 Template to start Projects.". Underneath the main content area, there are several interactive elements: a "Top Features" section, a "Top Resources" section, a "comments" section, and a "Launch your project like a rocket..." input field with a close button. On the left, under the "Free" heading, there is a paragraph about the template's ease of use and a "Yii Documentation »" link with a "Like" button. On the right, under the "Advantages" heading, there is a paragraph about RBAC and a "Yii Forum »" link with a "Like" button. Both sections also feature a "Add a comment..." input field and a "Also post on Facebook" checkbox.

And just for troubleshooting purposes, I will provide the entire frontend/views/site/index.php for reference. You should not need to do anything at this point, but if you are missing something, you can reference this file.

Gist:

Frontend Site Index

From book:

```
<?php

use \yii\bootstrap\Modal;
use kartik\social\FacebookPlugin;
use \yii\bootstrap\Collapse;
use \yii\bootstrap\Alert;
use yii\helpers\Html;

/* @var $this yii\web\View */
$this->title = 'Yii 2 Build';
?>
<div class="site-index">

    <div class="site-index">

        <div class="jumbotron">

            <?php if (Yii::$app->user->isGuest) {
                echo Html::a('Get Started Today', ['site/signup'],
                    ['class' => 'btn btn-lg btn-success']);
            }
            ?>
        </p>

        <h1>Yii 2 Build <i class="fa fa-plug"></i></h1>

        <p class="lead">Use this Yii 2 Template to start Projects.</p>

        <br/>

            <?php echo FacebookPlugin::widget(['type'=>FacebookPlugin::LIKE,
                'settings' => []]); ?>
```

```
</div>

<?php

echo Collapse::widget([
    'items' => [
        [
            'label' => 'Top Features',
            'content' => FacebookPlugin::widget([
                'type'=>FacebookPlugin::SHARE,
                'settings' => ['href'=>'http://www.yii2build.com','width'=>'350']
            ]),
            // open its content by default
            // 'contentOptions' => ['class' => 'in']
        ],
        // another group item
        [
            'label' => 'Top Resources',
            'content' => FacebookPlugin::widget([
                'type'=>FacebookPlugin::SHARE,
                'settings' => ['href'=>'http://www.yii2build.com','width'=>'350']
            ]),
            // 'contentOptions' => [],
            // 'options' => []
        ],
    ]
])
```

```
]);  
  
Modal::begin([  
  
    'header' => '<h2>Latest Comments</h2>',  
    'toggleButton' => ['label' => 'comments'],  
  
]);  
  
echo FacebookPlugin::widget([  
  
    'type'=>FacebookPlugin::COMMENT,  
    'settings' => ['href'=>'http://www.yii2build.com', 'width'=>'350']  
  
]);  
  
Modal::end();  
  
?>  
  
<br/>  
<br/>  
  
<?Php  
  
echo Alert::widget([  
  
    'options' => [  
        'class' => 'alert-info',  
        ],  
    'body' => 'Launch your project like a rocket...',  
]);  
?>  
  
<div class="body-content">  
  
<div class="row">  
    <div class="col-lg-4">  
        <h2>Free</h2>
```

```
<p>
<?php

if (!Yii::$app->user->isGuest) {

    echo Yii::$app->user->identity->username . ' is doing cool stuff. ';
}

?>
```

Starting with this free, open source Yii 2 template and it will save you a lot of time. You can deliver projects to the customer quickly, with a lot of boilerplate already taken care of for you, so you can concentrate on the complicated stuff.</p>

```
<p>

<a class="btn btn-default"
href="http://www.yiiframework.com/doc-2.0/guide-index.html">
Yii Documentation &raquo;</a>

</p>
```

```
<?php

echo FacebookPlugin::widget([
    'type'=>FacebookPlugin::LIKE,
    'settings' => []
]);
```

```
?>

</div>
<div class="col-lg-4">
    <h2>Advantages</h2>

<p>
```

Ease of use is a huge advantage. We've simplified RBAC and given you Free/Paid

user type out of the box. The Social plugins are so quick and easy to install, you will love it!

</p>

<p>

```
<a class="btn btn-default"
    href="http://www.yiiframework.com/forum/">
    Yii Forum &raquo;</a>
```

</p>

<?php

```
echo FacebookPlugin::widget([
    'type'=>FacebookPlugin::COMMENT,
    'settings' => [ 'href'=>'http://www.yii2build.com', 'width'=>'350' ]
]);
```

?>

```
</div>
<div class="col-lg-4">
    <h2>Code Quick, Code Right!</h2>
```

<p>

Leverage the power of the awesome Yii 2 framework with this enhanced template. Based Yii 2's advanced template, you get a full frontend and backend implementation that features rich UI for backend management.

</p>

<p>

```
<a class="btn btn-default"
    href="http://www.yiiframework.com/extensions/">
    Yii Extensions &raquo;</a>
```

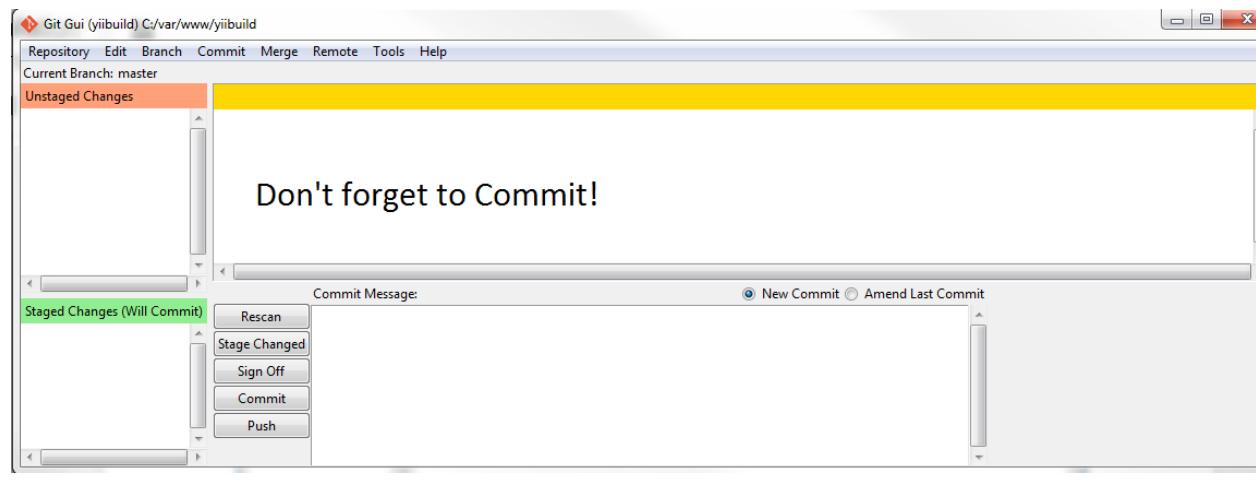
</p>

```

        </div>
    </div>

</div>
</div>
```

Summary



Commit!

In this chapter, we configured our Facebook app and installed our Kartik social widget extension so we could make use of it. With the ease of use of composer, installing the extension was easy. Kartik's extension is very useful, and as a reminder, you can check many of his other great extensions at:

<http://www.krajee.com>

We inserted the social widget inside the collapse widget to have a little fun decorating the home page. You can get more bootstrap widgets to use for your projects at:

<http://www.yiiframework.com/doc-2.0/guide-widget-bootstrap.html>

Since we already brought in the jui extension for DatePicker, we have access to all the widgets listed at:

<http://www.yiiframework.com/doc-2.0/guide-widget-jui.html>

Yii 2 supports a nice number of Jquery and Bootstrap widgets like collapse and modal. This book isn't really about front end development, so we didn't go too deep, but at least you got a sample to play with.

Finally, we add an app asset for font-awesome, so you can add some sizzle to presentation, without too much effort. You can find a lot of wonderful icons to add to your presentation at:

[Font-Awesome](#)

Chapter Eleven: Backend Creation

Ok, we're ready to move into creating our backend admin area. Before we can create all the files we need, we need to add a directory under backend/models, we want to add a search folder, so we have backend/models/search. Let's do that now.

Now we can work with Gii's CRUD generator to create controllers and CRUD views for all of our models. You might wonder if we can just use the frontend controllers and views that we have already made, however, we can't.

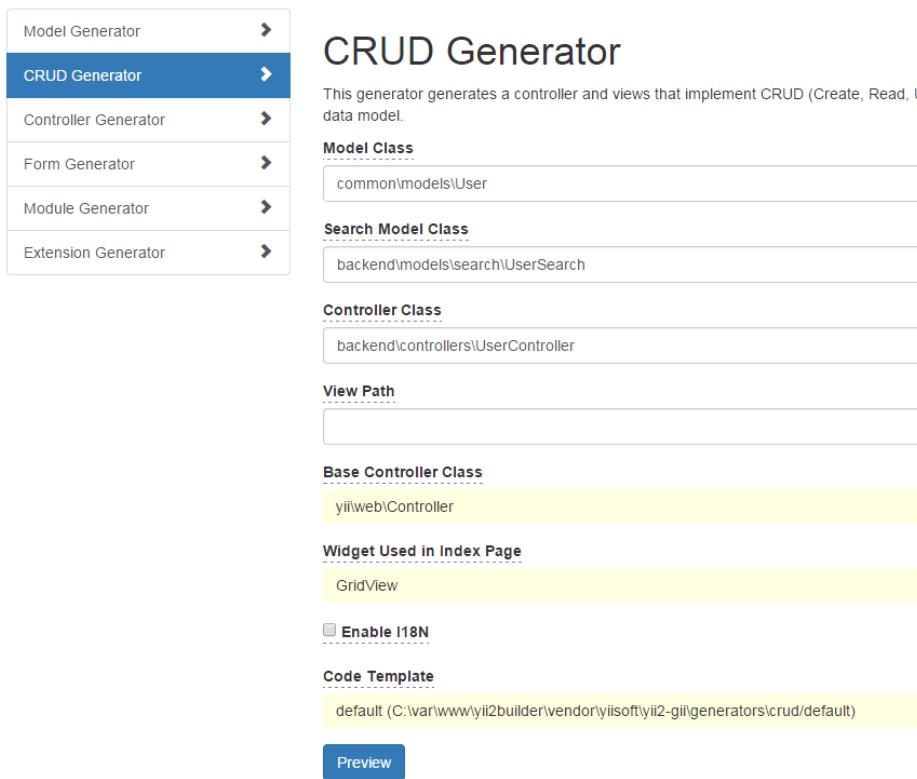
The backend operates differently from the frontend, which is why we have made a separate section of the application structure for it. I will point out the differences as we go, but for now, let's make a controller and crud for each of the following models:

- Profile
- Role
- Status
- User
- UserType

The url for Gii is:

`http://backend.yii2build.com/?r=gii`

The user model namespace is as follows:



User CRUD

If the above image is not clear, the sample looks like this:

Model Class: common\models\User

Search Model Class: backend\models\search\UserSearch

Controller Class: backend\controllers\UserController

We reference common\models\user, since that is where our user model resides, but we are creating all these other files into the backend. You can see that we also need to provide an entry for a search model. Make sure you have created the search folder within backend/models before running Gii.

Yii2 provides a separate class for search parameters and I'm really glad they did this. It separates out a lot of code out of the base model, which makes it easier to follow. You will see how it works later.

Note that we can leave the view path blank because we are conforming to the default.

To make sure you have the right namespaces and file locations generated, I will list out all the remaining models that you need to generate CRUD from, with namespaces specified.

Profile:

Model Class: frontend\models\Profile

Search Model Class: backend\models\search\ProfileSearch

Controller Class: backend\controllers\ProfileController

User:

Model Class: common\models\User

Search Model Class: backend\models\search\UserSearch

Controller Class: backend\controllers\UserController

Role:

Model Class: backend\models\Role

Search Model Class: backend\models\search\RoleSearch

Controller Class: backend\controllers\RoleController

Status:

Model Class: backend\models>Status

Search Model Class: backend\models\search>StatusSearch

Controller Class: backend\controllers>StatusController

User Type:

Model Class: backend\models\UserType

Search Model Class: backend\models\search\UserTypeSearch

Controller Class: backend\controllers\UserTypeController

Since the process of creating CRUD is exactly the same for each model listed above, we won't go through each one here. At this point, we will just assume you have created the files as we move on from here.

One thing to note. The naming convention for views with multiple words in the name is to put a - in between the two words, so the view folder for UserType model is user-type. The urls for controllers are this way too. Even though the controller file is named UserTypeController, to reach it by url, you would call, for example"

backend.yi2build.com/index.php?r=user-type/index

Ok, you can check the results individually by typing in a url, for example:

```
backend.yii2build.com/index.php?r=user
```

This will call the index action and assuming you have at least one record in there, it will display the record. Obviously, you would have to login to backend.yii2build.com and the user logging in would have to have a role_id that matches a role named ‘Admin,’ since our PermissionHelper enforces that rule.

The user index page also has links to view and update and delete, those are the icons you see on the right of the grid, so you can check to see if these are working as well. Then try the same url above with the different models you created to make sure it’s all working.

If you get errors, check the locations of your files and check each file for namespaces. For example, in ProfileController.php:

```
namespace backend\controllers;

use Yii;
use frontend\models\Profile;
use backend\models\search\ProfileSearch;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;
```

You can see it’s using frontend\models for the model, but backend\models\search for the search model.

While it’s possible to duplicate a model into more than one location, it’s definitely not recommended, it defeats the purpose of having a common folder and violates the principle of DRY.

I used the frontend\models for Profile, but could have used common\models. I just chose the former because of workflow. But you can see how easy it is to reference the correct folder via namespace when you are creating the controller and views.

We are going to make some changes to the view files and to the search model, but the controllers will mostly be left as is, except for behaviors. That is because the controllers and views that Gii creates lend themselves to a backend approach, that is one logged-in user who can search a list of users, update all records, etc. The out of the box controllers allow for this, so the good news that the auto-code generation is really helpful and a great time-saver.

So by simply controlling access to the admin area by enforcing a minimum value to role_id on the user record, only admin-level users can access the backend controllers. We will also demonstrate how easy it is to add a role above admin that can change records that admin can’t. In our template, users with Admin-level access can use the backend UI to change user records.

We will of course be restricting that somewhat, for example, admin will not be able to change a users password, they won’t even see it. We will make several changes in our backend UI to make moving

around the related records easier, for example, we will want to have access to a user's role_id and the ability to change it, so we can grant additional privileges to users.

You can see just how powerful Yii 2 really is by allowing us to set all this up quickly. And because we took the time earlier to make models for things like role and status, we are now going to have a full backend UI to control them.

Before we change the individual view files, let's make some changes to backend/views/layout/main.php. We will add numerous links to make it easy for us to navigate through the different views.

Main.php

Go to backend/views/layout/main.php.

Change main.php to:

Gist:

[backend main view change 1](#)

```
<?php

use backend\assets\AppAsset;
use yii\helpers\Html;
use yii\bootstrap\Nav;
use yii\bootstrap\NavBar;
use yii\widgets\Breadcrumbs;
use common\models\PermissionHelpers;
use backend\assets\FontAwesomeAsset;

/**
 * @var \yii\web\View $this
 * @var string $content
 */

AppAsset::register($this);
FontAwesomeAsset::register($this);

?>

<?php $this->beginPage() ?>

<!DOCTYPE html>
```

```
<html lang="= Yii::$app-&gt;language ?&gt;"&gt;

&lt;head&gt;
    &lt;meta charset="<?= Yii::$app-&gt;charset ?&gt;"/&gt;

    &lt;meta name="viewport"
        content="width=device-width,
        initial-scale=1"&gt;

        &lt;?= Html::csrfMetaTags() ?&gt;

&lt;title&gt;&lt;?= Html::encode($this-&gt;title) ?&gt;&lt;/title&gt;

        &lt;?php $this-&gt;head() ?&gt;

&lt;/head&gt;

&lt;body&gt;
    &lt;?php $this-&gt;beginBody() ?&gt;

    &lt;div class="wrap"&gt;

        &lt;?php

            if (!Yii::$app-&gt;user-&gt;isGuest){

                $is_admin = PermissionHelpers::requireMinimumRole('Admin');

                NavBar::begin([
                    'brandLabel' =&gt; 'Yii 2 Build &lt;i class="fa fa-plug"&gt;&lt;/i&gt; Admin',
                    'brandUrl' =&gt; Yii::$app-&gt;homeUrl,
                    'options' =&gt; [
                        'class' =&gt; 'navbar-inverse navbar-fixed-top',
                    ],
                ]);
            } else {
        </pre
```

```
NavBar::begin([
    'brandLabel' => 'Yii 2 Build <i class="fa fa-plug"></i>',
    'brandUrl' => Yii::$app->homeUrl,
    'options' => [
        'class' => 'navbar-inverse navbar-fixed-top',
    ],
]);
}

$menuItems = [
    ['label' => 'Home', 'url' => ['site/index']],
];
}

if (!Yii::$app->user->isGuest && $is_admin) {

    $menuItems[] = ['label' => 'Users', 'url' => ['user/index']];
    $menuItems[] = ['label' => 'Profiles', 'url' => ['profile/index']];
    $menuItems[] = ['label' => 'Roles', 'url' => ['role/index']];
    $menuItems[] = ['label' => 'User Types', 'url' => ['user-type/index']];
    $menuItems[] = ['label' => 'Statuses', 'url' => ['status/index']];
}

if (Yii::$app->user->isGuest) {

    $menuItems[] = ['label' => 'Login', 'url' => ['site/login']];
} else {

    $menuItems[] = [
        'label' => 'Logout (' . Yii::$app->user->identity->username . ')',
        'url' => ['/site/logout'],
        'linkOptions' => ['data-method' => 'post']
    ];
}
```

```
}

echo Nav::widget([
    'options' => ['class' => 'navbar-nav navbar-right'],
    'items' => $menuItems,
]);

NavBar::end();

?>

<div class="container">

<?= Breadcrumbs::widget([
    'links' => isset($this->params['breadcrumbs']) ? $this->params['breadcrumbs'] : [],
])?>

<?= $content ?>

</div>
</div>

<footer class="footer">

    <div class="container">

        <p class="pull-left">&copy; Yii 2 Build <?= date('Y') ?></p>

        <p class="pull-right"><?= Yii::powered() ?></p>

    </div>

</footer>

<?php $this->endBody() ?>
```

```
</body>
</html>

<?php $this->endPage() ?>
```

You'll note I've used a lot of whitespace to make the code easier to read. Keep in mind, readability is impacted by being in book format.

We also added a use statement:

```
use common\models\PermissionHelpers;
```

We did that because we are going to use PermissionHelpers::requireMinimumRole('Admin'); to add an additional layer of security.

You can see that instead of just adding the links to the different views, we hide the links if somehow the user has gotten to this page, but doesn't have admin-level access:

```
if (!Yii::$app->user->isGuest && $is_admin) {

    $menuItems[] = ['label' => 'Users', 'url' => ['user/index']];

}
```

The \$is_admin variable holds the value of:

```
PermissionHelpers::requireMinimumRole('Admin');
```

The logic behind this is if not a guest and the current user has role_id greater than or equal to the one need for 'admin,' display the link. It's a very simple way to control access to the links.

```
if (!Yii::$app->user->isGuest && $is_admin) {

    $menuItems[] = ['label' => 'Users', 'url' => ['user/index']];

    $menuItems[] = ['label' => 'Profiles', 'url' => ['profile/index']];

    $menuItems[] = ['label' => 'Roles', 'url' => ['/role/index']];

    $menuItems[] = ['label' => 'User Types', 'url' => ['/user-type/index']];

    $menuItems[] = ['label' => 'Statuses', 'url' => ['/status/index']];

}
```

We use the \$menuItems array to hold the url because we are working in the NavBar widget.

Note, we skipped over the block where we added an if statement to see if the user was logged in or not, then show them either:

```
'brandLabel' => 'Yii 2 Build Admin',
```

Or, if they are not logged in:

```
'brandLabel' => 'Yii 2 Build',
```

I just did that because even though it's only cosmetic, I don't even like acknowledging the word 'admin' to users who are not logged in.

Updating Backend Views

To complete our basic functionality of the backend, we need to update the view forms and grid views. In the case of the forms, like we did in the frontend views, we need to add dropdown options and remove unwanted fields.

The changes to GridView, which is the name of the main widget in Index.php, are a little deeper. We are going to change the columns being displayed, as well as add columns from related models, so that, for example, a list of users has a link to the users profiles. On the user grid, we should display the users role name, etc. These are little things, which will make the UI more useful and easier to comprehend.

backend/views/profile/_form.php

Let's start with the easiest first. We will simply copy frontend/views/profile/_form.php into backend/views/profile/_form.php.

So now we have the jui datepicker and the drop down list for gender in the form, which is exactly what we wanted.

backend/views/profile/view.php

Next, let's work on view.php for backend/views/profile/view.php. This is what Gii gave us out of the box:

```
<?php

use yii\helpers\Html;
use yii\widgets\DetailView;

/* @var $this yii\web\View */
/* @var $model frontend\models\Profile */

$this->title = $model->id;
$this->params['breadcrumbs'][] = ['label' => 'Profiles', 'url' => ['index']];
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="profile-view">

    <h1><?= Html::encode($this->title) ?></h1>

    <p>
        <?= Html::a('Update', ['update', 'id' => $model->id],
                    ['class' => 'btn btn-primary']) ?>
        <?= Html::a('Delete', ['delete', 'id' => $model->id],
                    ['class' => 'btn btn-danger',
                     'data' => [
                         'confirm' => 'Are you sure you want to delete this item?',
                         'method' => 'post',
                     ],
                ]) ?>
    </p>

    <?= DetailView::widget([
        'model' => $model,
        'attributes' => [
            'id',
            'user_id',
            'first_name:ntext',
            'last_name:ntext',
            'birthdate',
            'gender_id',
            'created_at',
            'updated_at',
        ],
    ]) ?>
```

```
</div>
```

Let's change this to:

Gist:

[Backend Profile View.php](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\DetailView;
use common\models\PermissionHelpers;

/**
 * @var yii\web\View $this
 * @var frontend\models\Profile $model
 */

$this->title = $model->user->username;

$show_this_nav = PermissionHelpers::requireMinimumRole('SuperUser');

$this->params['breadcrumbs'][] = ['label' => 'Profiles', 'url' => ['index']];
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="profile-view">

    <h1>Profile:  <?= Html::encode($this->title) ?></h1>

    <p>

        <?php if (!Yii::$app->user->isGuest && $show_this_nav) {>
            echo Html::a('Update', ['update', 'id' => $model->id],
                         ['class' => 'btn btn-primary']);?>

        <?php if (!Yii::$app->user->isGuest && $show_this_nav) {>
            echo Html::a('Delete', ['delete', 'id' => $model->id], [
                'class' => 'btn btn-danger',
                'data' => [
```

```
'confirm' => Yii::t('app', 'Are you sure you want to delete this item?'),
    'method' => 'post',
],
])}?)>

</p>

<?= DetailView::widget([
'model' => $model,
'attributes' => [
['attribute'=>'userLink', 'format'=>'raw'],
'first_name',
'last_name',
'birthdate',
'gender.gender_name',
'created_at',
'updated_at',
'id',
],
])
})?)>

</div>
```

Ok, so a few changes. We included at the top:

```
use common\models\PermissionHelpers;
```

It doesn't hurt to maintain consistency in wrapping the nav in if statements that check to see if the user has permission to do the action, before showing the links.

So to demonstrate this fully, we are going to require a higher role than admin to update or delete profiles. Let's call it SuperUser.

Go ahead and create a user in the application and through PhpMyAdmin, enter a role record for SuperUser and assign the new user that value. Don't forget it has to be a higher value than 20, which is what we gave Admin. Let's use 30 in the role table for the role value for example.

			id	role_name	role_value	
<input type="checkbox"/>	 Edit	 Copy	 Delete	1	user	10
<input type="checkbox"/>	 Edit	 Copy	 Delete	2	admin	20
<input type="checkbox"/>	 Edit	 Copy	 Delete	3	SuperUser	30

role table

Later, we will also modify our controller later to not allow anyone without admin access to view this page and I like not showing the link if it is not available to the user, so we have consistent behavior between the nav and access rules on the controller.

Ok, moving along through the changes, we changed the title to:

```
$this->title = $model->user->username;
```

Under that, we added the call for the user to be at least SuperUser role in order to see the update link:

```
$show_this_nav = PermissionHelpers::requireMinimumRole('SuperUser');
```

Next change is on the <h1> tag:

```
<h1>Profile: <?= Html::encode($this->title) ?></h1>
```

So now it displays user name instead of profile id number, much more user friendly.

Then we added our links to update and delete:

```
<p>

<?php if (!Yii::$app->user->isGuest && $show_this_nav) {
    echo Html::a('Update', ['update', 'id' => $model->id],
                 ['class' => 'btn btn-primary']);?>

<?php if (!Yii::$app->user->isGuest && $show_this_nav) {
    echo Html::a('Delete', ['delete', 'id' => $model->id], [
        'class' => 'btn btn-danger',
        'data' => [
            'confirm' => Yii::t('app', 'Are you sure you want to delete this item?'),
            'method' => 'post',
        ],
    ]);?>

</p>
```

Moving on, we changed the DetailView::widget. We changed the format of the first attribute to:

```
[ 'attribute'=>'userLink', 'format'=>'raw' ],
```

Our app knows what we are referring to because we added the getUserLink method to the Profile model and created the label:

```
'userLink' => Yii::t('app', 'User'),
```

to our attribute labels in common/models/User.php.

The method on the model:

```
public function getUserLink()
{
    $url = Url::to(['user/view', 'id'=>$this->id]);
    $options = []; //
    return Html::a($this->username, $url, $options);
}
```

This method returns the link to the user view page that we want. I'm showing the label and method here because this is probably where in your workflow you would have created them, since this is where you would see that you need them. We obviously built these in advance, so they are already in place. In the future, if you can anticipate the need for these kinds of methods, you can create them in advance as we did, as part of a bunch of boilerplate methods that you always add to model, when you create a model. Your workflow decisions, however, are best left up to you.

The other big change is a lazy loading relationship:

```
'gender.gender_name',
```

That simply tells it return the gender_name attribute from the Gender model, so we get 'male' instead of '1', which again, is much more friendly to the user. We have access to this because on the Profile model, we have the following method:

```
public function getGender()
{
    return $this->hasOne(Gender::className(), ['id' => 'gender_id']);
}
```

You might be wondering at this point what we meant by lazy loading relationship. A lazy loading query will do a DB query for each row of results, which is highly inefficient. A 1000 results would require 1001 queries (also known as the n+1 problem).

It's ok to do lazy loading when there is only one result, but you have to be careful about it. We will demonstrate the eager loading version of a query when we do the Index page.

backend/views/user/view.php

Moving on to the backend/views/user/view.php page, let's change it to the following:

Gist:

[Backend User View.php](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\DetailView;
use common\models\PermissionHelpers;

/* @var $this yii\web\View */
/* @var $model common\models\User */

$this->title = $model->username;
$show_this_nav = PermissionHelpers::requireMinimumRole('SuperUser');

$this->params['breadcrumbs'][] = ['label' => 'Users', 'url' => ['index']];
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="user-view">

    <h1><?= Html::encode($this->title) ?></h1>

    <p>

        <?php if (!Yii::$app->user->isGuest && $show_this_nav) {
            echo Html::a('Update', ['update', 'id' => $model->id],
                ['class' => 'btn btn-primary']);?>

        <?php if (!Yii::$app->user->isGuest && $show_this_nav) {
            echo Html::a('Delete', ['delete', 'id' => $model->id], [
                'class' => 'btn btn-danger',
                'data' => [
                    'confirm' => Yii::t('app', 'Are you sure you want to delete this item?'),
                    'method' => 'post',
                ],
            ]);?>

    </p>

    <?= DetailView::widget([
        'model' => $model,
        'attributes' => [
            ['attribute'=>'profileLink', 'format'=>'raw'],
            // 'username',
        ],
    ])?>
</div>
```

```

    // 'auth_key',
    // 'password_hash',
    // 'password_reset_token',
    'email:email',
    'roleName',
    'statusName',
    'userTypeName',
    'created_at',
    'updated_at',
    'id',
],
]) ?>

</div>

```

The use statements are exactly the same as the profile view page. Slight change to \$title, since we want to display the user name, it's an attribute of the current model:

```
$this->title = $model->username;
```

The rest is the same except for the DetailView::widget:

```

<?= DetailView::widget([
    'model' => $model,
    'attributes' => [
        ['attribute'=>'profileLink', 'format'=>'raw'],
        // 'username',
        // 'auth_key',
        // 'password_hash',
        // 'password_reset_token',
        'email:email',
        'roleName',
        'statusName',
        'userTypeName',
        'created_at',
        'updated_at',
        'id',
    ],
]) ?>

```

We got rid of unwanted fields displaying by commenting them out. We could cut them out entirely, but I like to leave these in for debug purposes, if I ever need to recall them.

So obviously our first attribute is a link to profile, which because of the methods we placed on the user model in the beginning, allow us to reference them as profileLink. Since this is identical to what we did previously in the Profile view, I will not explain that further, but you can look at the methods on the user model to refresh your knowledge.

Note on the email attribute, we use ‘email@email’ and this formats a mailto link on the address as it displays in the view, a nice handy feature.

We see ‘roleName’ and ‘statusName’ are brought in through relationships again defined on the user model.

When you check this all out in your browser on your project, notice how easy it is to move from the user to profile views by having those items linked. This is good UI practice and end users will appreciate it.

backend/views/user/_form

Let’s get the _form for user view updated to have dropdowns. Replace the existing file with the following:

Gist:

[Backend User _form View](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/**
 * @var yii\web\View $this
 * @var common\models\User $model
 * @var yii\widgets\ActiveForm $form
 */
?>

<div class="user-form">

<?php $form = ActiveForm::begin(); ?>

<?= $form->field($model, 'status_id')->dropDownList($model->statusList,
    [ 'prompt' => 'Please Choose One' ]);?>
```

```

<?= $form->field($model, 'role_id')->dropDownList($model->roleList,
    [ 'prompt' => 'Please Choose One' ]); ?>

<?= $form->field($model, 'user_type_id')->dropDownList($model->userTypeList,
    [ 'prompt' => 'Please Choose One' ]); ?>

<?= $form->field($model, 'username')->textInput(['maxlength' => 255]) ?>

<?= $form->field($model, 'email')->textInput(['maxlength' => 255]) ?>

<div class="form-group">
<?= Html::submitButton($model->isNewRecord ? 'Create' : 'Update',
['class' => $model->isNewRecord ? 'btn btn-success' : 'btn btn-primary']) ?>
</div>

<?php ActiveForm::end(); ?>

</div>

```

This is a simple use of the ActiveForm widget, which we've seen before when we were looking at our first view file. You should be used to this format by now. Note that I used 2 lines instead of one to avoid wordwrapping errors that happen when this book tries to adjust wordwrap in code.

Note the use of the Html helper class on the submit button. A nice ternary statement determines if the record is new or needs to be updated.

One other thing to point out on this. We do not need to id the action on the form. Because of Yii 2's framework logic, it knows what model and action to associate this form with, based on the location of the file and the model passed to the view.

It's only when your forms are more complicated that you need to create a separate form model that's when it needs an id on the form, so the controller knows which model to use. We saw examples of this from site controller, where there were numerous form models being used for things like contact, requestPasswordReset, etc.

Deeper Changes to Backend

You probably noticed that we conspicuously avoided changing the index view for backend/views/profile and backend/views/user. This is because we want to make changes to the main widget GridView. These changes are a little deeper because they will involve modifying the search model and changes to the GridView config.

backend/views/user/index.php

So let's replace the file in backend/views/user/index.php with:

Gist:

[Backend User Index View](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\grid\GridView;
use \yii\bootstrap\Collapse;

/* @var $this yii\web\View */
/* @var $searchModel backend\models\search\UserSearch */
/* @var $dataProvider yii\data\ActiveDataProvider */

$this->title = 'Users';
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="user-index">

    <h1><?= Html::encode($this->title) ?></h1>

    <?php echo Collapse::widget([
        'items' => [
            // equivalent to the above
            [
                'label' => 'Search',
                'content' => $this->render('_search', ['model' => $searchModel]),
                // open its content by default
                //'contentOptions' => ['class' => 'in']
            ],
        ]
    ]);;

?>
```

```

<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],

        // 'id',
        ['attribute'=>'userIdLink', 'format'=>'raw'],
        ['attribute'=>'userLink', 'format'=>'raw'],
        ['attribute'=>'profileLink', 'format'=>'raw'],

        'email:email',
        'roleName',
        'userTypeName',
        'statusName',
        'created_at',

        ['class' => 'yii\grid\ActionColumn'],

        // 'updated_at',
    ],
]); ?>

</div>

```

So obviously, in the use statement, we included:

```
use \yii\bootstrap\Collapse;
```

This lets us use the collapse widget, which we use to hold render statement, which brings in the form partial for search. The net effect is that it cleans up the page. In the view, when you mouseover the word search, it turns into a link. Click it, and the search form drops down. Since we already covered the collapse widget in a previous chapter, we won't go over it again.

We will be making changes to our search form, so at this point no need to bother testing it.

In the GridView widget, we left some attributes commented out for reference. You can see we added userIdLink, userLink, ProflieLink, email:email, roleName, userTypeName, and statusName. These are the labels we gave the methods on the User model attributes method. In the case of userIdLink, userLink, and profileLink, we have a specific format that we have to use to return the link. The

email:email format creates a mailto link, handy if you want to email the user. The method for userLink displays the username, in case you are wondering about that.

backend/views/profile/index.php

Let's do something similar for Profile, while we're at it. Paste in the following to backend/views/profile/index.php:

Gist:

[Backend Profile Index View](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\grid\GridView;
use \yii\bootstrap\Collapse;

/* @var $this yii\web\View */
/* @var $searchModel backend\models\search\ProfileSearch */
/* @var $dataProvider yii\data\ActiveDataProvider */

$this->title = 'Profiles';
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="profile-index">

<h1><?= Html::encode($this->title) ?></h1>

<?php echo Collapse::widget([
    'items' => [
        // equivalent to the above
        [
            'label' => 'Search',
            'content' => $this->render('_search', ['model' => $searchModel]) ,
            // open its content by default
            //'contentOptions' => ['class' => 'in']
        ],
    ]
]); ?>
```

```
    ]
]); ?>

<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],

        // 'id',
        ['attribute'=>'profileIdLink', 'format'=>'raw'],
        ['attribute'=>'userLink', 'format'=>'raw'],
        'first_name',
        'last_name',
        'birthdate',
        'genderName',
        ['class' => 'yii\grid\ActionColumn'],

        // 'created_at',
        // 'updated_at',
        // 'user_id',

    ],
]); ?>

</div>
```

The GridView widget is similar to the one in the user index view, but with less columns.

When I was doing this in workflow, I noticed we could link the id attribute to the update profile view, which would give us a fast way in to update profile. So I added the following method to the profile model:

```
public function getProfileIdLink()
{
    $url = Url::to(['profile/update', 'id'=>$this->id]);
    $options = [];
    return Html::a($this->id, $url, $options);
}
```

Also added on the profile model, the following attribute label:

```
'profileIdLink' => Yii::t('app', 'Profile'),
```

Then finally, I commented out id and added ‘profileIdLink’ to the widget in backend/views/profile/index:

```
[ 'attribute'=>'profileIdLink', 'format'=>'raw' ],
```

But most of this was obviously done previously, when we created the models. At least now you know why we did it.

While this gets us our link, we have no sort capabilities. Since sorting is something we want, we will make those changes, but we will wait to add that because we will have to make other changes to the search model at the same time.

backend/views/profile/_search.php

Now lets update backend/views/profile/_search.php. Replace the entire contents of the file with the following:

Gist:

[Backend Profile _search View](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\bootstrap\ActiveForm;
use frontend\models\Profile;
/**
 * @var yii\web\View $this
 * @var backend\models\search\ProfileSearch $model
 * @var yii\widgets\ActiveForm $form
 */
?>
```

```
<div class="profile-search">

<?php $form = ActiveForm::begin([
    'action' => ['index'],
    'method' => 'get',
]); ?>

<?= $form->field($model, 'first_name') ?>

<?= $form->field($model, 'last_name') ?>

<?= $form->field($model, 'birthdate') ?>

<?= $form->field($model, 'gender_id')->dropDownList(Profile::getgenderList(),
    [ 'prompt' => 'Please Choose One' ]); ?>

<?php // echo $form->field($model, 'created_at') ?>

<?php // echo $form->field($model, 'updated_at') ?>

<?php // echo $form->field($model, 'user_id') ?>

<div class="form-group">
    <?= Html::submitButton('Search', ['class' => 'btn btn-primary']) ?>
    <?= Html::resetButton('Reset', ['class' => 'btn btn-default']) ?>
</div>

<?php ActiveForm::end(); ?>

</div>
```

Don't bother testing search accuracy yet, we have to work on the search model, we will get to that shortly.

backend/views/user/_search.php

Ok, so now we move on to working on the _search view file for user and backend/models/search/UserSearch.php file. The UserSearch.php file provides the model for _search.php in the backend/views/user/_search.php, which itself is rendered inside of backend/views/user/index.php.

Basically, it's the search form at the top of the index file.

Let's start by replacing the contents of _search.php with the following:

Gist:

Backend User _search View

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;
use common\models\User;

/* @var $this yii\web\View */
/* @var $model backend\models\search\UserSearch */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="user-search">

<?php $form = ActiveForm::begin([
    'action' => ['index'],
    'method' => 'get',
]); ?>

<?= $form->field($model, 'id') ?>

<?= $form->field($model, 'username') ?>

<?php echo $form->field($model, 'email') ?>

<?= $form->field($model, 'role_id')->dropDownList(User::getroleList(),
    [ 'prompt' => 'Please Choose One' ]); ?>

<?= $form->field($model, 'user_type_id')->dropDownList(User::getuserTypeList(),
    [ 'prompt' => 'Please Choose One' ]); ?>

<?= $form->field($model, 'status_id')->dropDownList($model->statusList,
    [ 'prompt' => 'Please Choose One' ]); ?>
```

```
<div class="form-group">
    <?= Html::submitButton('Search', ['class' => 'btn btn-primary']) ?>
    <?= Html::resetButton('Reset', ['class' => 'btn btn-default']) ?>
</div>

<?php ActiveForm::end(); ?>

</div>
```

One thing you might have noticed is that on our ActiveForm::begin method, we are listing action and method. The reason we are doing this is that we expect dynamic data from the user. They are going to send get variables along to the controller, so we need to specify ‘get’ as the method. This is how Gii hands it to us.

And you can see we have barely changed the file otherwise, except to make dropdown lists from methods on the User model.

Now if you try this in the browser, it works great, but you will notice that the dropdown for userType shows the options, but does not filter the results. Also, we need to make sure that we eager load our results.

Eager loading, if you recall, is how we avoid the n+1 problem, where a query is made for each row of results. In a database where there are large numbers of results, an n+1 problem can render the page useless because it will take forever, if ever, to return results.

We get around that by eager loading. We will do this when we modify the UserSearch model.

User Search

The UserSearch model is an extension of the User model, in this case User, that the controller uses to instruct it on how to query the model.

The file is located at backend/models/search/UserSearch.php. The main method is search(\$params), so let’s look at that:

```

public function search($params)
{
    $query = User::find();

    $dataProvider = new ActiveDataProvider([
        'query' => $query,
    ]);

    if (!($this->load($params) && $this->validate())) {
        return $dataProvider;
    }

    $query->andFilterWhere([
        'id' => $this->id,
        'role_id' => $this->role_id,
        'status_id' => $this->status_id,
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ]);

    $query->andFilterWhere(['like', 'username', $this->username])

    ->andFilterWhere(['like', 'auth_key', $this->auth_key])
    ->andFilterWhere(['like', 'password_hash', $this->password_hash])
    ->andFilterWhere(['like', 'password_reset_token', $this->password_reset_token])
    ->andFilterWhere(['like', 'email', $this->email]);

    return $dataProvider;
}
}

```

\$params is being handed in by the form. This happens via the UserController, which starts the index action as follows:

```

public function actionIndex()
{
    $searchModel = new UserSearch();
    $dataProvider = $searchModel->search(Yii::$app->request->queryParams);

```

To display results, we call an instance of the model, in this case new UserSearch(), then set \$dataProvider variable as an instance of the model, with search method handing in the query

parameters through Yii::\$app->request->queryParams. \$dataProvider will be used by the view widget to display results.

The important thing to note is that when the Index action is called, it will look for the parameters from the form and run the search method, even if there are no search parameters. Without the search parameters, it simply returns all records from the DB.

So now let's look at the search method in detail because it's important to know how this works. The search method on the UserSearch model, starts out by setting \$query to User::find method.

After setting the query to the model instance, we create an instance of Yii 2's ActiveDataProvider class, handing in the query inside the config array.

```
$dataProvider = new ActiveDataProvider([
    'query' => $query,
]);
```

ActiveDataProvider creates a powerful iterator out of the object results, in this case user, where we find all results, since \$query was originally set to User::find();

Once we instantiate the instance of ActiveDataProvider, we check to see if we added any search parameters through the form, and if not, return the unfiltered result of \$query, which as we have already stated, will return all the users.

```
if (!($this->load($params) && $this->validate())) {
    return $dataProvider;
}
```

If there are \$params handed in from the form, then we evaluate false, which means we don't yet return \$dataProvider, and we move to the next block and then call the andFilterWhere() method on the user Model, to filter the parameters.

```
$query->andFilterWhere([
    'id' => $this->id,
    'role_id' => $this->role_id,
    'status_id' => $this->status_id,
    'created_at' => $this->created_at,
    'updated_at' => $this->updated_at,
]);
```

Next we see another call to the same method to cover 'like' as a parameter:

```
$query->andFilterWhere(['like', 'username', $this->username])
->andFilterWhere(['like', 'auth_key', $this->auth_key])
->andFilterWhere(['like', 'password_hash', $this->password_hash])
->andFilterWhere(['like', 'password_reset_token',
    $this->password_reset_token])
->andFilterWhere(['like', 'email', $this->email]);
```

You can see how the method calls are chained together successively, with the semicolon on the last line. Then finally, we return \$dataProvider:

```
return $dataProvider;
```

So that's the out of the box version. But we need a more robust version of this. We have to get related data from Roles, UserType, etc. and we need eager loading, so what we need is a little more complex. Replace the old UserSearch model with the following:

Gist:

Backend User Search Model

From book:

```
<?php

namespace backend\models\search;

use Yii;
use yii\base\Model;
use yii\data\ActiveDataProvider;
use common\models\User;

/**
 * UserSearch represents the model behind the
 * search form about `common\models\User`.
 */

class UserSearch extends User
{
    /**
     * @var mixed
}
```

```
*/  
  
    public $roleName;  
    public $userTypeName;  
    public $user_type_name;  
    public $user_type_id;  
    public $statusName;  
    public $profileId;  
  
/**  
 * @inheritDoc  
 */  
  
public function rules()  
{  
    return [  
  
        [['id', 'role_id', 'status_id', 'user_type_id'], 'integer'],  
        [['username', 'email', 'created_at', 'updated_at', 'roleName',  
            'statusName', 'userTypeName', 'profileId',  
            'user_type_name'], 'safe'],  
    ];  
}  
  
/**  
 * @inheritDoc  
 */  
  
public function scenarios()  
{  
    // bypass scenarios() implementation in the parent class  
    return Model::scenarios();  
}  
/**
```

```
* Creates data provider instance with search query applied
*
* @param array $params
*
* @return ActiveDataProvider
*/
public function search($params)
{
    $query = User::find();
    $dataProvider = new ActiveDataProvider([
        'query' => $query,
    ]);

    /**
     * Setup your sorting attributes
     * Note: This is setup before the $this->load($params)
     * statement below
    */

    $dataProvider->setSort([
        'attributes' => [
            'id',
            'userIdLink' => [
                'asc' => ['user.id' => SORT_ASC],
                'desc' => ['user.id' => SORT_DESC],
                'label' => 'User'
            ],
            'userLink' => [
                'asc' => ['user.username' => SORT_ASC],
                'desc' => ['user.username' => SORT_DESC],
                'label' => 'User'
            ],
            'profileLink' => [
                'asc' => ['profile.id' => SORT_ASC],
                'desc' => ['profile.id' => SORT_DESC],
                'label' => 'Profile'
            ],
        ],
    ]);
}
```

```
'roleName' => [
    'asc' => ['role.role_name' => SORT_ASC],
    'desc' => ['role.role_name' => SORT_DESC],
    'label' => 'Role'
],
'statusName' => [
    'asc' => ['status.status_name' => SORT_ASC],
    'desc' => ['status.status_name' => SORT_DESC],
    'label' => 'Status'
],
'userTypeName' => [
    'asc' => ['user_type.user_type_name' => SORT_ASC],
    'desc' => ['user_type.user_type_name' => SORT_DESC],
    'label' => 'User Type'
],
'created_at' => [
    'asc' => ['created_at' => SORT_ASC],
    'desc' => ['created_at' => SORT_DESC],
    'label' => 'Created At'
],
'email' => [
    'asc' => ['email' => SORT_ASC],
    'desc' => ['email' => SORT_DESC],
    'label' => 'Email'
],
],
]);
}

if (!$this->load($params) && $this->validate())) {
    $query->joinWith(['role'])
        ->joinWith(['status'])
        ->joinWith(['profile'])
        ->joinWith(['userType']);
}

return $dataProvider;
}

$this->addSearchParameter($query, 'id');
```

```
$this->addSearchParameter($query, 'username', true);
$this->addSearchParameter($query, 'email', true);
$this->addSearchParameter($query, 'role_id');
$this->addSearchParameter($query, 'status_id');
$this->addSearchParameter($query, 'user_type_id');
$this->addSearchParameter($query, 'created_at');
$this->addSearchParameter($query, 'updated_at');

// filter by role

$query->joinWith(['role' => function ($q) {
    $q->andFilterWhere(['=' , 'role.role_name', $this->roleName]);
}])

// filter by status

->joinWith(['status' => function ($q) {
    $q->andFilterWhere(['=' , 'status.status_name', $this->statusName]);
}])

// filter by user type

->joinWith(['userType' => function ($q) {
    $q->andFilterWhere(['=' , 'user_type.user_type_name', $this->userTypeName]);
}])

// filter by profile

->joinWith(['profile' => function ($q) {
    $q->andFilterWhere(['=' , 'profile.id', $this->profileId]);
}]);

return $dataProvider;
}
```

```

protected function addSearchParameter($query, $attribute, $partialMatch = false)
{
    if (($pos = strrpos($attribute, '.')) !== false) {
        $modelAttribute = substr($attribute, $pos + 1);
    } else {
        $modelAttribute = $attribute;
    }

    $value = $this->$modelAttribute;

    if (trim($value) === '') {
        return;
    }

/*
 * The following line is additionally added for right aliasing
 * of columns so filtering happen correctly in the self join
 */
}

$attribute = "user.$attribute";

if ($partialMatch) {
    $query->andWhere(['like', $attribute, $value]);
} else {
    $query->andWhere([$attribute => $value]);
}
}
}

```

Ok, let's tackle this beast. It seems like a lot, but it's not bad once you break it down. Also, I should note that I learned this by following a tutorial in the wiki by Kartik, the same author who wrote the social widget that we are using on our application's home page. I refactored only slightly for cosmetic appeal, and we probably didn't gain much clarity from that, but at least I tried.

The first thing to note is that class UserSearch extends User, and we made sure to include in our list of attributes those attributes which are referenced by a method on the model, roleName for example, because we know we are going to use the dropdown list to return role names. If this is not explicitly listed as an attribute, the form breaks and the page does not render. So if you are running into that type of problem, make sure you have a complete list of attributes that it needs. This is not always obvious because the parent model of User is supposed to know all of its attributes from reflection somewhere in the base classes.

What I found, putting this together, is that I need to declare the attributes `$user_type_id` and `$user--`

`type_name`. I'm just not sure why. I use these attributes in the where clauses, so maybe that is the source of the problem, perhaps it can't use the parent model at that location to identify the attribute. This of course, is just a guess.

When you're working with a large framework like Yii 2, you are occasionally going to run up against things that you don't completely understand. It's ok, it happens to everyone, I can certainly attest to that personally. The important thing is that we try to learn as much as we can as we go along because when it comes to using a framework, knowledge is power.

Ok, next up are the rules used by validation. The first array tells us which attributes are integer only. The second array tells which attributes are safe. We need this rule because of the `setAttributes` method in the `/yii/base/Model` class, which ignores attributes if an attribute does not have at least one validation rule and is not marked safe by the safe rule. So when the contents of `$_Post` are sent to the method, only the accepted values will be allowed in.

Anyway, Gii includes a safe array in the rules that it auto-generates, so I have kept this array, and made sure that it contains the current attributes, which are in addition to those on the parent model. Again I had to use trial and error to make sure I had what I needed.

Next we have the scenarios method:

```
public function scenarios()
{
    // bypass scenarios() implementation in the parent class
    return Model::scenarios();
}
```

This method allows you to bypass the parent scenarios, which would allow you create your own scenarios. We won't be using this, so we'll just leave it in place, since this is how Gii gave it to us.

Next we would have the `attributeLabels` method, but we don't need one because we are inheriting all the attribute labels we need from the parent model and we haven't added anything new that would require one.

Ok, let's move on the search method:

```
public function search($params)
{
    $query = User::find();
    $dataProvider = new ActiveDataProvider([
        'query' => $query,
    ]);
}
```

This is exactly like the code Gii generates for us, so no changes there. We create an instance of `User` with the `find` method, which will return all results. Then we create an instance of `ActiveDataProvider`

and inject the user model via \$query. So now \$dataProvider is loaded up with the User model and all it's records. Later, we will use the controller to pass this \$dataProvider to the view, where the GridView widget can use it.

Ok, back to the UserSearch model and its search method.

Next we take the setSort method of \$dataProvider and configure it so that the columns we want to be sortable on the Grid will have the behavior that we want:

```
$dataProvider->setSort([
    'attributes' => [
        'id',
        'userIdLink' => [
            'asc' => ['user.id' => SORT_ASC],
            'desc' => ['user.id' => SORT_DESC],
            'label' => 'ID'
        ],
        'userLink' => [
            'asc' => ['user.username' => SORT_ASC],
            'desc' => ['user.username' => SORT_DESC],
            'label' => 'User'
        ],
        'profileLink' => [
            'asc' => ['profile.id' => SORT_ASC],
            'desc' => ['profile.id' => SORT_DESC],
            'label' => 'Profile'
        ],
        'roleName' => [
            'asc' => ['role.role_name' => SORT_ASC],
            'desc' => ['role.role_name' => SORT_DESC],
            'label' => 'Role'
        ],
        'statusName' => [
            'asc' => ['status.status_name' => SORT_ASC],
            'desc' => ['status.status_name' => SORT_DESC],
            'label' => 'Status'
        ],
        'userTypeName' => [
            'asc' => ['user_type.user_type_name' => SORT_ASC],
            'desc' => ['user_type.user_type_name' => SORT_DESC],
            'label' => 'User Type'
        ]
    ]
]);
```

```

        'label' => 'User Type'
    ],

    'created_at' => [
        'asc' => ['created_at' => SORT_ASC],
        'desc' => ['created_at' => SORT_DESC],
        'label' => 'Created At'
    ],

    'email' => [
        'asc' => ['email' => SORT_ASC],
        'desc' => ['email' => SORT_DESC],
        'label' => 'Email'
    ],

]
]);

```

This is clean and easy to understand.

Then we get an if statement:

```

if (!($this->load($params) && $this->validate())) {

    $query->joinWith(['role'])
        ->joinWith(['status'])
        ->joinWith(['profile'])
        ->joinWith(['userType']);

    return $dataProvider;
}

```

Let's look at the if:

```
if (!($this->load($params) && $this->validate()))
```

We saw this before when we looked at the version Gii gave us. This one operates the same way. What it says is load the parameters and run validation method, then evaluate true or false. The ! can be confusing, so I will explain it fully.

If the statement evaluates to true, there is only a small amount of code that follows to a return statement. This is very easy to read. Just to be clear, no parameters would evaluate true, then you would execute the small block of code with the return statement.

If the if statement evaluates false, and there are parameters being handed in from the search form, and we move onto the next block of code to follow.

Ok, so let's deal with the true condition first:

If no parameters evaluates true from the if statement, add the relationships via `joinWith` method (for eager loading) and return the model with it's relationships, stored in the `$dataProvider`, since `$query` is already injected into `$dataProvider`. The controller will pass `$dataProvider` to the view.

```
{  
    $query->joinWith(['role'])  
        ->joinWith(['status'])  
        ->joinWith(['profile'])  
        ->joinWith(['userType']);  
  
    return $dataProvider;  
  
}
```

If you look carefully, you see that `userType` has the capital 'T', which has to do with how the model relation is named because there are two words involved in its composition. If you name that wrong or if there is no get method for the relation on the user model, you will get an error. The naming convention gives a capital letter to the second word in a model name when there is more than one word in the model.

`$query` is an instance of the User model and is configured into `$dataProvider`, so even if we hand in no parameters, we can still return unfiltered results. So again, to be perfectly clear, if there are no parameters for search, we return all user records.

Note we have chained together the user model with 4 other models, role, status, profile, and `userType`, so we can eager load our results, which means we won't have to do a separate query for each row of results. These joins allow us sync the User with the appropriate profile, role, etc.

Eager loading is the opposite of lazy load, and for large record sets, like the records in the User Model for example, this is preferable because it puts less strain on the DB.

Now let's look at the longer, more complex possibility of the if statement. If the `if (!($this->load($params) && $this->validate()))` statement evaluates false, this means we have parameters for the search, and we move on to the next block of code, where we use a method named `addSearchParameter` to add conditions to the query:

```
$this->addSearchParameter($query, 'id');
$this->addSearchParameter($query, 'username', true);
$this->addSearchParameter($query, 'email', true);
$this->addSearchParameter($query, 'role_id');
$this->addSearchParameter($query, 'status_id');
$this->addSearchParameter($query, 'user_type_id');
$this->addSearchParameter($query, 'created_at');
$this->addSearchParameter($query, 'updated_at');
```

You can see we run one instance of the method for each attribute. So let's look at the `addSearchParameter` method to get a better idea of what's going on:

```
protected function addSearchParameter($query, $attribute, $partialMatch = false)
{
    if (($pos = strpos($attribute, '.')) !== false) {
        $modelAttribute = substr($attribute, $pos + 1);
    } else {
        $modelAttribute = $attribute;
    }

    $value = $this->$modelAttribute;

    if (trim($value) === '') {
        return;
    }

/*
 * The following line is additionally added for right aliasing
 * of columns so filtering happen correctly in the self join
 */
$attribute = "user.$attribute";

    if ($partialMatch) {
        $query->andWhere(['like', $attribute, $value]);
    } else {
        $query->andWhere([$attribute => $value]);
    }
}
```

The first part determines if there is a '.' in the attribute. I added whitespace to make it easier to read:

```

if (($pos = strpos($attribute, '.')) !== false) {

    $modelAttribute = substr($attribute, $pos + 1);

} else {

    $modelAttribute = $attribute;
}

```

If the parameter has a ‘.’, then the method positions the attribute to the search parameter correctly, so it knows what is the model and what is the attribute.

The method is eliminating what goes before the period, which sets it up to eliminate an ambiguation problem, since the role table for example, also has an id column. This is tricky stuff and it won’t work right if we don’t do it exactly this way.

Whether it has a ‘.’ or not, it sets \$attribute to \$modelAttribute.

The next line:

```
$value = $this->$modelAttribute;
```

Sets the value of the attribute. Just a quick reminder on how this works.

The attribute is handed in to this method as string to the variable \$attribute, where it is formatted to account for whether or not there was a period.

In either case, whether or not it had a period or not, the variable is renamed \$modelAttribute. But this still represents the string that was handed in through \$attribute. So when we call \$this->\$modelAttribute, we are inserting the variable where a string would normally go. The variable \$value picks up the result of this expression, whatever type it may be, string, integer, bool.

For example, if we read \$value = \$this->username, it would be more intuitive for us to expect the \$value has the actual username, which is a string. Instead we got \$value = \$this->\$modelAttribute, which is great because we can use it for all the attributes and it will pass the correct format to the \$value variable.

\$this is referring to an instance of UserSearch, which as we know, extends User, so \$this can have an attribute named username or any of the other ones we provided when we called the method.

Now you may be asking yourself, if you are using a string, for example, how does it know which specific value to return? This is a real brain teaser for me, not so obvious from staring at the code. The answer is that it has already acquired the value from the form:

```
if (!($this->load($params) && $this->validate()))
```

Remember that the not ! statement only evaluates true, forcing the return statement, if there are no parameters. If there are parameters, it successfully runs the load and validate methods, so by the time we are using these field names to set up our query in the addSearchParemter method, the model, our friendly \$this, already has the values we need.

Ah, so simple once we see how it works. I don't know if you struggled over it as I did, but for both our sakes, I'm glad I finally got it. Also note, if the parameters can't load because of validation failure, it will show the form with errors. However this action takes place in the controller, not the model.

Anyway, back to the addSearchParameter method. We haven't finished it yet:

Return if empty:

```
if (trim($value) === '') {  
    return;  
}
```

In this case, that's another way of saying do nothing, so you don't end up with a bunch of blank where statements on the query. Again to be clear, if the field is empty, it does not get added as a search parameter.

Otherwise:

```
/*  
 * The following line is additionally added for right aliasing  
 * of columns so filtering happen correctly in the self join  
 */  
  
$attribute = "user.$attribute";
```

The comment above the line explains part of it. We set the table name in front of the attribute to avoid ambiguation problems. Since this method runs one attribute at a time, we can safely assume that \$attribute holds the string we intended. Since we stripped out anything in front of the period earlier in the method, there can be no confusion about which table we are referring to, since we are explicitly telling it to use user.

Ok, on to \$partialMatch. The default value is set to false. So the statement if (\$partialMatch) will check to see if it is true. The only way it can be true is if it is handed in that way. If you check the list of calls to the addSearchParmeter method, you can see that username and email are set to true.

Partial matches are handy, especially on strings, where the user doesn't want or sometimes even know how to type in the full match.

Anyway, if \$partialMatch is true, then use the like operator in the andWhere method (which has been inherited from somewhere else in the framework) to add a partial match to search on:

```
if ($partialMatch) {  
  
    $query->andWhere(['like', $attribute, $value]);  
  
}
```

Else, use the andWhere method to hand in the attribute and its value to the query:

```
$query->andWhere([$attribute => $value]);
```

Ok, so the attributes are added. Now we are back to where we left off in the search method and we come to the joins that will allow us to filter. You can see that for each one, we add a closure, an anonymous function, that binds the andFilterWhere method to the model being joined:

```
// filter by role  
  
$query->joinWith(['role' => function ($q) {  
  
    $q->andFilterWhere(['=' , 'role.role_name' , $this->roleName]);  
  
})  
  
// filter by status  
  
->joinWith(['status' => function ($q) {  
  
    $q->andFilterWhere(['=' , 'status.status_name' , $this->statusName]);  
  
})  
  
// filter by user type  
  
->joinWith(['userType' => function ($q) {  
  
    $q->andFilterWhere(['=' , 'user_type.user_type_name' , $this->userTypeName]);  
  
})  
  
// filter by profile  
  
->joinWith(['profile' => function ($q) {  
  
    $q->andFilterWhere(['=' , 'profile.id' , $this->profileId]);  
  
})
```

```
});
```

Note that we stacked the `->joinWith` methods, but we have comments in between, be careful, the closing semicolon only comes at the very end. This is very intuitive syntax in the `andFilterWhere` method. The first parameter gives us our operator, in this case `=` means equal because we are building a sql query. Second parameter gives us tablename and field. Third parameter is value we want bound to the query. Again, we know the model already has the input values from the form, so when you see `$this->statusName`, for example, it is using the value coming in from the form.

And thank God that's over with. I'm exhausted. Learning programming is fun, but it's also hard work.

We need to make similar changes to `ProfileSearch.php` and we need to make sure we add our sort for the `profileIdLink`:

Let's take the entire `ProfileSearch.php` file and replace it with:

Gist:

Backend Profile Search Model

From book:

```
<?php

namespace backend\models\search;

use Yii;
use yii\base\Model;
use yii\data\ActiveDataProvider;
use frontend\models\Profile;

class ProfileSearch extends Profile
{
    public $genderName;
    public $gender_id;
    public $userId;

    public function rules()
    {
        return [
            ['id', 'gender_id'], 'integer'],
        ];
    }
}
```

```
[['first_name', 'last_name', 'birthdate', 'genderName', 'userId'], 'safe'],

];

}

/***
 * @inheritDoc
 */

public function attributeLabels()
{
    return [
        'gender_id' => 'Gender',
    ];
}

public function search($params)
{
    $query = Profile::find();
    $dataProvider = new ActiveDataProvider([
        'query' => $query,
    ]);

    $dataProvider->setSort([
        'attributes' => [
            'id',
            'first_name',
            'last_name',
            'birthdate',
            'genderName' => [
                'asc' => ['gender.gender_name' => SORT_ASC],
                'desc' => ['gender.gender_name' => SORT_DESC],
                'label' => 'Gender'
            ],
            'profileIdLink' => [
                'asc' => ['profile.id' => SORT_ASC],
                'desc' => ['Profile.id' => SORT_DESC],
                'label' => 'ID'
            ]
        ]
    ]);
}
```

```
        ],
        'userLink' => [
            'asc' => [ 'user.username' => SORT_ASC],
            'desc' => [ 'user.username' => SORT_DESC],
            'label' => 'User'
        ],
    ],
]);
}

if (!($this->load($params) && $this->validate())) {
    $query->joinWith(['gender'])
        ->joinWith(['user']);

    return $dataProvider;
}

$this->addSearchParameter($query, 'id');
$this->addSearchParameter($query, 'first_name', true);
$this->addSearchParameter($query, 'last_name', true);
$this->addSearchParameter($query, 'birthdate');
$this->addSearchParameter($query, 'gender_id');
$this->addSearchParameter($query, 'created_at');
$this->addSearchParameter($query, 'updated_at');
$this->addSearchParameter($query, 'user_id');

// filter by gender name

$query->joinWith(['gender' => function ($q) {
    $q->andFilterWhere(['=' , 'gender.gender_name', $this->genderName]);
}])

// filter by user

->joinWith(['user' => function ($q) {
    $q->andFilterWhere(['=' , 'user.id', $this->user]);
}]);
}
```

```
        return $dataProvider;
    }

protected function addSearchParameter($query, $attribute, $partialMatch = false)
{
    if (($pos = strrpos($attribute, '.')) !== false) {
        $modelAttribute = substr($attribute, $pos + 1);
    } else {
        $modelAttribute = $attribute;
    }

    $value = $this->$modelAttribute;
    if (trim($value) === '') {
        return;
    }

/*
 * The following line is additionally added for right aliasing
 * of columns so filtering happen correctly in the self join
 */
}

$attribute = "profile.$attribute";

if ($partialMatch) {
    $query->andWhere(['like', $attribute, $value]);
} else {
    $query->andWhere([$attribute => $value]);
}
}
```

So now, we can column sort on id when we on backend/views/profile/index.php. And with that, we should have everything we need for ProfileSearch.

Admin UI

Let's make a change to backend/views/site/index. We want navigation to make our admin tasks easier, so let's replace the old file with:

Gist:

[Backend Site Index View](#)

From book:

```
<?php

use yii\helpers\Html;
use common\models\PermissionHelpers;

/**
 * @var yii\web\View $this
 */

$this->title = 'Admin Yii 2 Build';

$is_admin = PermissionHelpers::requireMinimumRole('Admin');

?>

<div class="site-index">

    <div class="jumbotron">

        <h1>Welcome to Admin!</h1>

        <p class="lead">

            Now you can manage users, roles, and more with
            our easy tools.

        </p>

        <p>
```

```
<?php  
  
if (!Yii::$app->user->isGuest && $is_admin) {  
  
echo Html::a('Manage Users', ['user/index'],  
['class' => 'btn btn-lg btn-success']);  
  
}  
  
?>
```

```
</p>  
  
</div>  
  
<div class="body-content">  
  
<div class="row">  
  <div class="col-lg-4">  
  
    <h2>Users</h2>  
  
    <p>
```

This is the place to manage users. You can edit status and roles from here.
The UI is easy to use and intuitive, just click the link below to get started.

```
</p>  
  
<p>  
  
<?php  
  
if (!Yii::$app->user->isGuest && $is_admin) {  
  
echo Html::a('Manage Users', ['user/index'],  
['class' => 'btn btn-default']);  
  
}  
  
?>
```

```
</p>

</div>
<div class="col-lg-4">

    <h2>Roles</h2>

    <p>

        This is where you manage Roles. You can decide who is admin and w\
ho is not. You can
        add a new role if you like, just click the link below to get started.

    </p>

    <p>

<?php

    if (!Yii::$app->user->isGuest && $is_admin) {

        echo Html::a('Manage Roles', ['role/index'],
            ['class' => 'btn btn-default']);

    }

?>

    </p>

    </div>
    <div class="col-lg-4">

        <h2>Profiles</h2>

        <p>

            Need to review Profiles? This is the place to get it done.
            These are easy to manage via UI. Just click the link below to manage profiles.

        </p>
```

```
<p>

<?php

if (!Yii::$app->user->isGuest && $is_admin) {

    echo Html::a('Manage Profiles', ['profile/index'],
        ['class' => 'btn btn-default']);

}

?>

</p>

</div>
</div>

<div class="row">
<div class="col-lg-4">

    <h2>User Types</h2>

    <p>

        This is the place to manage user types. You can edit user
        types from here. The UI is easy to use and intuitive, just
        click the link below to get started.

    </p>

    <p>

        <?php

if (!Yii::$app->user->isGuest && $is_admin) {

    echo Html::a('Manage User Types', ['user-type/index'],
        ['class' => 'btn btn-default']);

}

```

```
?>
```

```
</p>
```

```
</div>
<div class="col-lg-4">
```

```
<h2>Statuses</h2>
```

```
<p>
```

This is where you manage Statuses. You can add or delete.
You can add a new status if you like, just click the link
below to get started.

```
</p>
```

```
<p>
```

```
<?php
```

```
if (!Yii::$app->user->isGuest && $is_admin) {

echo Html::a('Manage Statuses', ['status/index'],
['class' => 'btn btn-default']);
```

```
}
```

```
?>
```

```
</p>
```

```
</div>
<div class="col-lg-4">
```

```
<h2>Placeholder</h2>
```

```
<p>
```

Need to review Profiles? This is the place to get it done.
These are easy to manage via UI. Just click the link below

```
to manage profiles.

</p>

<p>

<?php

if (!Yii::$app->user->isGuest && $is_admin) {

    echo Html::a('Manage Profiles', ['profile/index'],
        ['class' => 'btn btn-default']);

}

?>

</p>

</div>
</div>
</div>
</div>
```

We brought in a couple of classes to help us:

```
use yii\helpers\Html;
use common\models\PermissionHelpers;
```

The helpers class let's us format the urls with the following:

```
echo Html::a('Manage Roles', ['role/index'], ['class' => 'btn btn-default']);
```

Like we did in the last chapter, we are using the a method of the Html class.

Since we want to be consistent, we have wrapped each link in our If statement, to test if the user is in fact admin or greater:

```

if (!Yii::$app->user->isGuest && $is_admin) {

echo Html::a('Manage Profiles', ['profile/index'],
['class' => 'btn btn-default']);

}

```

Like we did with the previous page, we set the \$is_admin variable near the top of the file under the title variable:

```
$is_admin = PermissionHelpers::requireMinimumRole('Admin');
```

This is not a complete security solution because if someone had somehow logged into the backend without having a role of Admin or greater, they could still type in the url, so we will have to add logic to the controllers as well.

Controller Behaviors

We will do this now, through the behaviors method on backend/controllers/SiteController.php. As we have shown in a previous chapter, Yii 2 provides a matchCallback parameter on rules in AccessControl and it works perfectly for our purposes. Let's replace the existing behaviors method (don't forget the use statement for PermissionHelpers) with the following:

Gist:

[Backend Site Controller Behaviors](#)

From book:

```

public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'rules' => [
                [
                    'actions' => ['login', 'error'],
                    'allow' => true,
                ],
                [
                    'actions' => ['index'],
                    'allow' => true,
                    'roles' => ['@'],
                    'matchCallback' => function ($rule, $action) {

```

```

        return PermissionHelpers::requireMinimumRole('Admin')
        && PermissionHelpers::requireStatus('Active');
    }
],
[
    'actions' => ['logout'],
    'allow' => true,
    'roles' => ['@'],
],
],
],
],
'verbs' => [
    'class' => VerbFilter::className(),
    'actions' => [
        'logout' => ['post'],
    ],
],
];
}
}

```

Like I mentioned above, don't forget to add the use statement at the top of the file:

```
use common\models\PermissionHelpers;
```

Match Callback

We added the matchCallback rule to cover conditions where a users status or access level changes after they have logged in. If you have to drop someone's access level, you don't want them to have any residual access that they shouldn't have.

We need to make changes to our other controllers too, and we will explain the rules in detail after the changes are done. The backend controllers for User, UserType, Status, Profile, and Role are a little different from the previous example and need the following code added under the behaviors method:

Gist:

[Backend Behaviors For All Other Controllers](#)

From book:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => \yii\filters\AccessControl::className(),
            'only' => ['index', 'view', 'create', 'update', 'delete'],
            'rules' => [
                [
                    'actions' => ['index', 'create', 'view'],
                    'allow' => true,
                    'roles' => ['@'],
                    'matchCallback' => function ($rule, $action) {
                        return PermissionHelpers::requireMinimumRole('Admin')
                            && PermissionHelpers::requireStatus('Active');
                    }
                ],
                [
                    'actions' => ['update', 'delete'],
                    'allow' => true,
                    'roles' => ['@'],
                    'matchCallback' => function ($rule, $action) {
                        return PermissionHelpers::requireMinimumRole('SuperUser')
                            && PermissionHelpers::requireStatus('Active');
                    }
                ],
            ],
        ],
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'delete' => ['post'],
            ],
        ],
    ],
]
```

```
];
}
```

Make sure that behavior method is in place for each of the controllers mentioned above, and once again don't forget the use statement for PermissionHelpers:

```
use common\models\PermissionHelpers;
```

For matchCallback, we are supplying a condition:

```
'matchCallback' => function ($rule, $action) {
    return PermissionHelpers::requireMinimumRole('Admin')
        && PermissionHelpers::requireStatus('Active');
```

if that evaluates to false, we don't match and we get a "You do not have permission to view this page" response. This is another reuse of the PermisssionHelpers method, which means we are getting some good code reuse out of it. The fact that we made it a public static function means we can call it anywhere we need it, as long as we include:

```
use common\models\PermissionHelpers;
```

So with our controller methods in place, we have a decent amount of security to stop someone who is not admin from accessing admin processes. As we thought it through, we realized needed to check for status as well. What if someone's status is downgraded during an open session? They would still have access to the controller methods because status was only checked when they logged in. This is important on the frontend too. So, for example, if you built a site for example, where users could cancel their account, you would not want them to get around to areas of the site that required active status.

Notice that we are checking to see if the user's status_id is equal to the Active status.

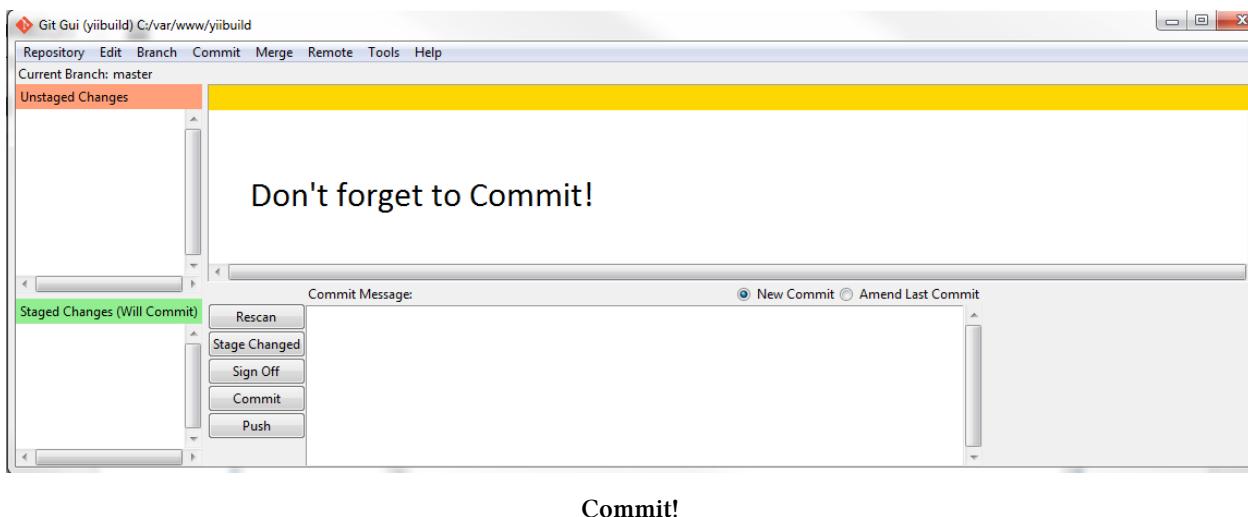
We were supposed to change the behaviors in the frontend in a previous chapter, so if you have not please go back and do so now.

Ok, finally we have our working back end. Now we can check out our new index page for admin when we are logged in:

The screenshot shows the Yii 2 Admin interface. At the top, there's a navigation bar with links for Home, Users, Profiles, Roles, User Types, Statuses, and Logout (happy). Below the navigation, a large heading says "Welcome to Admin!". A sub-header below it reads "Now you can manage users, roles, and more with our easy tools." A prominent green button labeled "Manage Users" is centered. The main content area is divided into several sections:

- Users**: A brief description and a "Manage Users" button.
- Roles**: A brief description and a "Manage Roles" button.
- Profiles**: A brief description and a "Manage Profiles" button.
- User Types**: A brief description and a "Manage User Types" button.
- Statuses**: A brief description and a "Manage Statuses" button.
- Placeholder**: A brief description and a "Manage Profiles" button.

Summary



Commit!

Well, that's it. You now have an extended template made from Yii 2's advanced template. We covered a lot of ground with this book, enough to get you up and running, which was our goal. You should be able to use the template from this book to start your own projects.

This book was a starter book, so this really is only the beginning for you on your journey with Yii 2. To continue the journey, you should consult the guide and the forums for more information on how to use the framework.

Please note I have already added bonus chapters and I will be updating the book to keep up with changes and to add new material, so look for that when you can. All updates are free to anyone who has purchased this book for the life of the book. I hope it's a long life.

If you like this book, please recommend it to friends. You can visit my [blog](#) and leave feedback. Any positive comments, links, and reviews are greatly appreciated.

Thanks for taking this journey with me. I hope to see you soon.

About The Author

Bill Keck has been developing web applications since 1999. In 2005, he was invited to speak at the Google campus in Mountain View because of a private white paper he authored. He is currently CEO of SERRF Corp, a company that utilizes Yii extensively in its products. He is a strong believer in the Yii 2 framework and consults with developers on how to take advantage of all the efficiencies in development that Yii 2 offers.

Chapter 12: Bonus Material

Congratulations again on having made it through the basics of the Yii 2 Framework. You now have an extended template that you can build upon for future projects.

So let's build on what we have already. One of the themes I talk about in my blog is pleasing the client, and you can do this by meeting and exceeding their expectations. And one of the great things about having the template ready is that so much will already be built before they even hire you.

Now if you are using Yii 2 under other circumstances, working for a venture, or a company that does development, where you never meet the client, don't worry. All of what we will cover will be useful no matter which development environment you are in.

AutoResponder

Often a client will have a need for an autoresponder somewhere in their application. It could be registration, sending in a support request, any of a number of things. And inevitably, when they want an autoresponder, two things will happen:

1. They will want to revise the text.
2. They will think of more autoresponders they want in the future.

So what we need is a solution we can implement that scales easily to the clients needs. And this can save you a lot of headaches too.

Imagine the client wants to change a single word in the text of their autoresponder and they want you to push a new version of the site for it, and to top it off, they need that done over the weekend, or civilization as we know it will come to an end.

The client gets over-excited, yet at the same time, they are paying the bills, so we have to listen. It's a nightmare. We've all been there.

So I thought about how I could avoid that scenario completely. Wouldn't it be great if they could just enter what they need into an admin page in the backend and never even bother to call me?

Ok, on one hand, when they call, we get paid, so we don't want them to stop calling us. On the other hand, the more power we give them over their project, the more they will love us for the work we are doing and the more they will come back for more, especially an enthusiastic client who appreciates the attention to detail.

So obviously, if we were going to let the client update the text for the email via UI, we would most likely be storing that text in our DB. And we know writing that UI with Gii's auto-generated code

is a snap. So it's not hard to imagine that part coming together quickly. We just need a simple data structure and that will work nicely.

What else would we need? Well, we need a model that looks up the record and fires off the email. And then here comes the buzz kill. How does it know which email to send and where do we put the code that calls the appropriate method?

My first thought was to put this on a behavior on the controller, but that wasn't quite right. The implementation of behaviors that I've seen in controllers allow you to pick specific actions, which is good, but defeats the purpose of applying to all actions, which is what I wanted.

If only there were a way in Yii 2 to apply a set of instructions to all actions. Actually there is. It's called an `afterAction`. They also have a `beforeAction` method, but that's not what we were looking for. So the `afterAction` method looks like this:

```
public function afterAction($action, $result)
{
    //your code here
    return parent::afterAction($action, $result);
}
```

It automatically takes in two arguments and as long as you call the parent, it will run after each action on the controller.

So I thought, hmm, maybe this will work. Thinking out the code before writing it, I thought that we could simply do a check on the action and controller name, see if there is a record for that specific action/controller pair, and if so, return it and run the `afterAction`, which would have a method that would send the email. Or if not, don't run the parent and return false. Or something like that.

So, eager to try this, I set up my data structure for the email messages. And because I'm thinking a little more ahead, I named the table `status_message`. I figured I might be able to reuse the body of the emails for other messages at some point, and I wanted to give myself flexibility for extensibility. So that's why I didn't call it `email_message`.

Also, note I used the singular. When choosing a name of a table, I try to stick with the convention set by Yii 2, using the `User` table as an example. So I always go singular.

Feel free, however, to name it how you wish. Just make sure if you do pick a different name, to reference it correctly in the rest of the tutorial.

So the table structure looks like this:

```

id (int, Primary Key, Not Null, Auto Increment)

controller_name (varchar(105), Not Null)

action_name (varchar(105) Not Null)

status_message_name (varchar(105) Not Null)

subject (varchar(255) Not Null)

body (varchar(2025) Not Null)

status_message_description (varchar(255) Not Null)

created_at (DateTime)

updated_at (DateTime)

```

The screenshot shows the MySQL Workbench interface with the table definition for 'status_message'. The table has the following structure:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>				
controller_name	VARCHAR(105)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
action_name	VARCHAR(105)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
status_message_name	VARCHAR(105)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
subject	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
body	VARCHAR(2025)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
status_message_description	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
created_at	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
updated_at	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Status Message Instructions

Synchronize the model the model to the DB or if you are just using Php MyAdmin, create the table with those fields and constraints.

Also, note, as I stated in the beginning of the book, I do not use migrations, that is a personal choice, but that is why I don't provide the migration. I use MySql Workbench and PhpMyAdmin for these kinds of tasks. If you are following along and don't use either of those two, feel free to use the method/tool of your choice.

It should look like this when you are done:

SELECT * FROM `status_message` LIMIT 0 .. 30

Profiling [Inline] [Edit] [Explain SQL] [Create PHP Code] [Refresh]

#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	id	int(11)			No	None	AUTO_INCREMENT	Primary
2	controller_name	varchar(105)	latin1_general_ci		No	None		Primary
3	action_name	varchar(105)	latin1_general_ci		No	None		Primary
4	status_message_name	varchar(105)	latin1_general_ci		No	None		Primary
5	subject	varchar(255)	latin1_general_ci		No	None		Primary
6	body	varchar(2025)	latin1_general_ci		No	None		Primary
7	status_message_description	varchar(255)	latin1_general_ci		No	None		Primary
8	created_at	datetime			Yes	NULL		Primary
9	updated_at	datetime			Yes	NULL		Primary

Check All / Uncheck All With selected: Primary

Add 1 column(s) At End of Table At Beginning of Table After id Go

Status Message Table

Ok, nothing too crazy there. I gave myself a description field, so I could describe the purpose of the message.

Now this data structure may evolve over time, but to get things up and running, I've kept it simple. So after creating the data structure, I could see that I needed a simple check to see if a record existed, and if so, return either false, indicating there is no message for that controller/action or return the id of the message, which I could then use in another method to retrieve the parts of the message that I wish to send via email.

Could I have done it all in one method? Yes, very easily, but by breaking it apart into multiple parts, the code is easier to digest, easier to write, and easier to reuse.

We already have the perfect place to put this method, in our RecordHelpers class, located in common/models/RecordHelpers.php.

First add the use statement at the top that will give us access to the model via ActiveRecord:

```
use backend\models\StatusMessage;
```

Then let's add the method:

Gist:

[FindStatusMessage](#)

From Book:

```

public static function findStatusMessage($action_name, $controller_name)
{
    $result = StatusMessage::find('id')
        ->where(['action_name' => $action_name])
        ->andWhere(['controller_name' => $controller_name])
        ->one();

    return isset($result['id']) ? $result['id'] : false;
}

```

As you can see, it does a simple lookup via ActionRecord, when you hand in the action_name and controller name. Obviously we have a second parameter and we used andWhere() for that.

We use a ternary to see if we have a result, if we're good, return the result. If not return false. The result will give us the id of the record which we will hand into other methods. Very simple stuff.

Here are the other two methods that retrieve the message subject and message body, respectively. Let's add them now to RecordHelpers.php:

Gist:

[GetMessageSubject](#) and [GetMessageBody](#)

From book:

```

public static function getMessageSubject($id)
{
    $result = StatusMessage::find('subject')
        ->where(['id' => $id])
        ->one();

    return isset($result['subject']) ? $result['subject'] : false;
}

public static function getMessageBody($id)
{
    $result = StatusMessage::find('body')
        ->where(['id' => $id])
        ->one();
}

```

```

    return isset($result['body']) ? $result['body'] : false;
}

}

```

In both cases, we give the message_id that we want and we get back the message subject or body that we need for the email. Later we will attach these results to our sendTheMail method, which will be part of our MailCall class that we are going to create. Don't worry it sounds much more complicated than it actually is, it's very simple stuff.

Ok, back to our new methods. Note that we are using public static methods. This makes it easy to place the method inside another method without having to use the longer instantiation syntax. You will see what I mean in a minute or two.

Anyway, so now we have the record helpers that will help us return the data from the table.

You can see we are working our way backwards from the data. Obviously, we will need a mail method to send the mail, once we have the bits of the message we need.

I needed to figure out how Yii 2 sends mail, so, remembering that there is a contact page on the advanced template that sends a mail message, I used it as a guide for what I wanted to build. I only mention this to point out that you can use a lot of what Yii 2 hands you in its templates as a guide for what you want to build.

After studying the ContactForm model, I came up with my mail method. Let's create a new model for this named MailCall. Go ahead and create MailCall.php inside of common/models.

Create an empty MailCall class with the following namespace :

```
namespace common\models;
```

Then let's add the following method:

Gist:

[SendTheMail](#)

From book:

```

public static function sendTheMail($message_id)
{
    return Yii::$app->mailer->compose()
        ->setTo(\Yii::$app->user->identity->email)
        ->setFrom(['no-reply@yii2build.com' => 'Yii 2 Build'])
        ->setSubject(RecordHelpers::getMessageSubject($message_id))
        ->setTextBody(RecordHelpers::getMessageBody($message_id))
        ->send();
}

```

Ok, so Yii 2 has a mailer class, accessed from the application instance Yii::\$app. Now that should set off a little reminder for you to include a use statement for Yii, otherwise, we are not going to have access to Yii::

So below namespace add:

```
use yii;
```

Ok, you can see we have a whole bunch of methods chained together. And so now we see the logic behind what we were doing if it was not already clear. Mailer's setTo method takes a parameter of the current user's email address. For now we are hard-coding the from address and name. For setSubject, we use our handy static call to:

```
RecordHelpers::getMessageSubject($message_id)
```

So again, that should indicate that we will use RecordHelpers, so we need to pull that in as well. Add the following use statement:

```
use common\models\RecordHelpers;
```

This is all nice concise code and for the message body, same type of thing:

```
RecordHelpers::getMessageBody($message_id)
```

You can see that the method takes \$message_id as a parameter, so now all we need is a method that will call sendTheMail, that has the ability to hand in the \$message_id.

So Now I go back to the beginning question because before I build the method, I need to know how I'm going to call it.

I was very happy with something like:

```
public function afterAction($action, $result)
{
    MailCall::isMailable($action->id, getUniqueId());
    return parent::afterAction($action, $result);
}
```

The plan was to create a method named isMailable on my newly created MailCall class. Since I'm using a static method, I can just pop it in there. I do that a lot for these helper methods.

\$action->id returns 'action', so if this were the site controller and the index action were being called, it would return 'index.' And getUniqueId() returns the controller name. So that gave me the two things I need to look up a record, see if it existed, if so, return the message id, and hand it into the sendTheMail method.

So I was pretty happy. So much so, I took a break and went for a walk. And as I was enjoying the nice cool ocean air, clearing my head, I realized I was making a mistake.

The method `afterAction` fires after every action, which is what I wanted, but it has no way to know what I intended the success of the action to be. For example, let's say you had an action that says save a record or show form for input, which we see a lot in our controllers. The `afterAction` method will fire in both cases, no matter what, because as far as it is concerned, it has called the action. But you would only want to send an email in certain cases, saving for example, not showing the form, so using `afterAction` just went out the window.

And so it goes in programming. I needed a more discrete way of determining what would trigger the email, and since that could vary greatly from action to action, the only way to do it correctly is to place the method call inside of the action at precisely the point where I want it to execute.

So I thought again about my original idea, which was to anticipate the client's needs and give them control over the content of the mail messages sent from different actions. The way I saw it, I had a choice, for one way, I could embed the method call to send the mail only when explicitly asked to do so by the client.

Because I built out a fairly robust way of storing the messages and sending them, it would become a very trivial matter for me to add more locations on request by the client, who could then edit them as they wished from the UI. This would be the standard approach.

Or, as an alternative, I could just embed the call wherever I thought it might be needed in the future, since I was planning to have it test for the existence of the message and return false otherwise.

Of course I wouldn't throw it in everywhere, just in spots likely to require an autoresponder. So then the question becomes is the DB overhead worth it. The extra call wherever I place it would slow down the site a tiny bit.

To give a more concrete example of what I'm talking about. Let's say the client wants an autoresponder on registration. The user registers and gets an email confirming it. Our class and methods, which we have not completed yet, but will shortly, will handle this beautifully. Perfect.

But looking over our template, we see that we have a contact form and wouldn't it be cool to be able to send an auto-response whenever someone contacts us? This is standard functionality on most sites and the client might not realize it yet, but they will probably want this too.

So as a compromise, we could embed the method calls to send mail everywhere we think the client will request them during development, then, after a final review with the client, remove the unused ones from the code. This creates a little cleanup work, but would actually be very simple to do.

This way, during site development, when the client begs you to add an autoresponder, you can tell them to just add the DB record via backend UI, with no additional coding required. This would keep them happy and you would be one step ahead.

Of course the level of implementation is up to you. I only mention all this because you can see that as a project progresses, things tend to change. I wanted to build a behavioral method that checked every action, but that turned out not to be realistic, so I compromised and settled for an inline action method instead.

Ok, so let's look at what we have so far.

We can use RecordHelpers::findStatusMessage(\$action_name, \$controller_name) to retrieve the \$message_id, if a message for that action and controller exists in our DB.

We can set the mail settings based on \$message_id, which includes the subject and body of the email, using RecordHelpers::getMessageSubject(\$id) and RecordHelpers::getMessageBody(\$id) methods.

We can send the mail using the MailCall::sendTheMail(\$message_id) method.

So now we need to build the method that is going to put everything into action. To get an idea of what we want to do, let's focus on the registration auto-response. Here is the snippet from the site controller on the signup action:

```
public function actionSignup()
{
    $model = new SignupForm();
    if ($model->load(Yii::$app->request->post())) {
        if ($user = $model->signup()) {

            if (Yii::$app->getUser()->login($user)) {

                return $this->goHome();
            }
        }
    }

    return $this->render('signup', [
        'model' => $model,
    ]);
}
```

Nothing special here, this is what you get out of the box on the advanced template. So we need to add something like:

```
MailCall::onMailableAction('signup', 'site');
```

That seems concise and easy to understand. I really try to boil it down to as little as possible and yet still be intuitive. MailCall is the name of the class we created for the sendTheMail method. The onMailableAction method, we have not created yet. But we can see that it takes the action and controller names as arguments, which are also names of fields in our DB.

We will use the action name and controller name to find the specific message we want to mail out for this action.

It's worth noting that I thought of another way to reference the arguments:

```
MailCall::onMailableAction(__METHOD__, getUniqueId());
```

The syntax on METHOD is a magic method to return the name of the current method and like I said before, getUniqueId() returns the name of the controller. So that would get us, ‘actionSignup’ and ‘site’, which is not quite right. So in order to use this approach, we would have to get rid of the word action.

So I did this, rather incorrectly, as the following:

```
$method_name = str_replace("action", "", $method_name);  
$action_name = $method_name;
```

Cool, it strips out action and replaces it with nothing. Yeah, but what happens if you have an action named actionTraction? It will strip action out of traction and return an error. Obviously I could just remove the first six characters from the string, which would be the correct way to do it. But I never bothered to write it that way, here’s why.

I didn’t want to have to remember what the magic METHOD and getUniqueId() were returning. Also, if someone else had to maintain the code, they would not be able to instantly understand what was happening. Plus, to explicitly name the controller and action instead actually required less code.

Sometimes it just comes down to a preference of what you want to stare at. You will spend far more time reading code than writing it. So the easier that code is to read and get at first glance, the better.

Now in terms of naming the method, I use onMailableAction because it is slightly ambiguous, which I think is appropriate because nothing is going to happen if there is no corresponding message in the DB.

Ok, so let’s provide the last method on MailCall to wrap this up. Add this to your MailCall class.

Gist:

OnMailableAction

From book:

```
public static function onMailableAction($action_name, $controller_name)  
{  
  
    if ($message_id = RecordHelpers::findStatusMessage  
        ($action_name, $controller_name)){  
  
        static::sendTheMail($message_id);  
  
    }  
  
}
```

Obviously the if statement is broken into two lines because of wordwrap. You should use it as one line in your IDE.

It's so simple. It hands in the action name and controller name, then uses our helper method to try to get a record in the DB and set it to \$message_id. If it can't set \$message_id, then it never calls MailCall. If it can set \$message_id correctly, then it calls the sendTheMail method with the \$messgae_id handed in.

For the sake of consistency, I will provide the entire MailCall.php file.

Gist:

[MailCall.php](#)

From book:

```
<?php

namespace common\models;

use yii;
use common\models\RecordHelpers;

class MailCall
{

    public static function sendTheMail($message_id)
{
    return Yii::$app->mailer->compose()
        ->setTo(\Yii::$app->user->identity->email)
        ->setFrom(['no-reply@yii2build.com' => 'Yii 2 Build'])
        ->setSubject(RecordHelpers::getMessageSubject($message_id))
        ->setTextBody(RecordHelpers::getMessageBody($message_id))
        ->send();
}

public static function onMailableAction($action_name, $controller_name)
{
    if ($message_id = RecordHelpers::findStatusMessage
        ($action_name, $controller_name)){
        static::sendTheMail($message_id);
    }
}
```

```
}
```

```
}
```

Again the if statement is broken into two lines because of wordwrap. You should use it as one line in your IDE.

So to call this correctly in our signup action on the site controller:

Gist:

Action Signup

From book:

```
public function actionSignup()
{
    $model = new SignupForm();
    if ($model->load(Yii::$app->request->post())) {
        if ($user = $model->signup()) {

            if (Yii::$app->getUser()->login($user)) {

                MailCall::onMailableAction('signup', 'site');

            }

        }

    }

    return $this->render('signup', [
        'model' => $model,
    ]);
}
```

You see that we popped in our onMailableAction method right after the user is logged in. This is because, as you recall from our sendTheMail method, we are using:

```
->setTo(\Yii::$app->user->identity->email)
```

And you only have access to Yii::\$app->user->identity->email if the user is logged in.

Also note, this is now one line of code in the controller, everything else is extracted out into other classes. How awesome is that? Nice clean controller code.

Just don't forget to pull MailCall into SiteController:

```
use common\models\MailCall;
```

At this point, since we have no records in our DB, nothing should happen, except normal registration of a user. However you should register a user and see if it is throwing any errors.

Now that we've tested ok so far, let's use Gii to build out our UI in the backend, then we can add some records and play with this.

We can move quickly on this, since we're so used to Gii at this point, and this is where you see just how awesome that code generation is for building code quickly.

Ok, first step, build the model. I won't provide a screenshot because you should know how to do this by now. I will mention, however, that my choice is to put the model in the backend, that seems logical to me.

I know I've mentioned it before, that Yii 2 recommends putting models in common and then extending them in frontend and backend, but typically that is not my personal choice. And you can see as we add models to common, how handy it is to have that folder reserved for helpers and other models.

Ok, so I'm going to assume that built your status_message model, and then built your CRUD from StatusMessage. I'm sure I don't have to point out how amazing Gii is at this point, but I will anyway. 8 files are being created for you that will only require minor changes. Not only do you get standardization, but you pick up an amazing amount of efficiency in both time and effort.

So let's move through some minor changes. Let's add our timestamp behavior to the model, StatusMessage.php:

Gist:

[StatusMessage Behaviors](#)

From book:

```
public function behaviors()
{
    return [
        'timestamp' => [
            'class' => 'yii\behaviors\TimestampBehavior',
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
            ],
            'value' => new Expression('NOW()'),
        ],
    ];
}
```

```

        ],
    ];
}

```

And of course you know that means you will have to pull in:

```
use yii\db\ActiveRecord;
```

But you will also need:

```
use yii\db\Expression;
```

To be consistent, let's make the behaviors in our StatusMessageController the same as for our other admin controllers, such as role. So replace the behaviors on the StatusMessageController with the following:

Gist:

StatusMessageController

From book:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => \yii\filters\AccessControl::className(),
            'only' => ['index', 'view', 'create', 'update', 'delete'],
            'rules' => [
                [
                    'actions' => ['index', 'create', 'update', 'view'],
                    'allow' => true,
                    'roles' => ['@'],
                    'matchCallback' => function ($rule, $action) {
                        return PermissionHelpers::requireMinimumRole('Admin')
                            && PermissionHelpers::requireStatus('Active');
                    }
                ],
                [
                    'actions' => [ 'delete'],
                    'allow' => true,
                ]
            ],
        ],
    ];
}
```

```

        'roles' => ['@'],
        'matchCallback' => function ($rule, $action) {
            return PermissionHelpers::requireMinimumRole('SuperUser')
                && PermissionHelpers::requireStatus('Active');
        }
    ],
],
],
],
];

'verbs' => [
    'class' => VerbFilter::className(),
    'actions' => [
        'delete' => ['post'],
    ],
],
];
}

```

One difference you can see is that I moved action update to admin level access. And, don't forget to pull in:

```
use common\models\PermissionHelpers;
```

Now we can work on the views. You can change a `textInput` field on `backend/views/status-message/_form.php` to the following:

```
<?= $form->field($model, 'body') ->
textArea(['maxlength' => 2025, 'rows' => 12]) ?>
```

Two lines to avoid the dreaded word-wrapping again. Use this as one line in your IDE.

So we changed it from `textInput` to `textArea` and also handed in the optional `rows` parameter, which lets you control the size of the text box. I also moved `status_message_description` and put it directly under `status_message_name` because that seems like a more logical order of fields.

And obviously we can get rid of created_at and updated_at fields, since we have behaviors handling those.

And last step, we need to add navigation to our views. Let's go to backend/views/layouts/main.php and add one more menu item:

```
$menuItems[] = ['label' => 'Status Messages', 'url' => ['status-message/index']] ;
```

For UI purposes, I made the label plural. Note the way the controller is referenced. Putting a dash between the two words is the convention here, even though the name of the controller file is StatusMessageController.php.

This UI is starting to get silly with so many items running across the nav. Don't worry, switching to drop down nav is our next section of bonus material, so we will be changing it.

Ok right now, if you save this, you can get to all the views and test everything. I will provide a screenshot of my first record, just to make sure we got it all right and understand what the fields mean, since we covered that a while back.

Home / Status Messages / Create Status Message

Create Status Message

Controller Name
site

Action Name
signup

Status Message Name
registration autoresponse

Status Message Description
confirmation message on signup

Subject
Thank you for Registering!

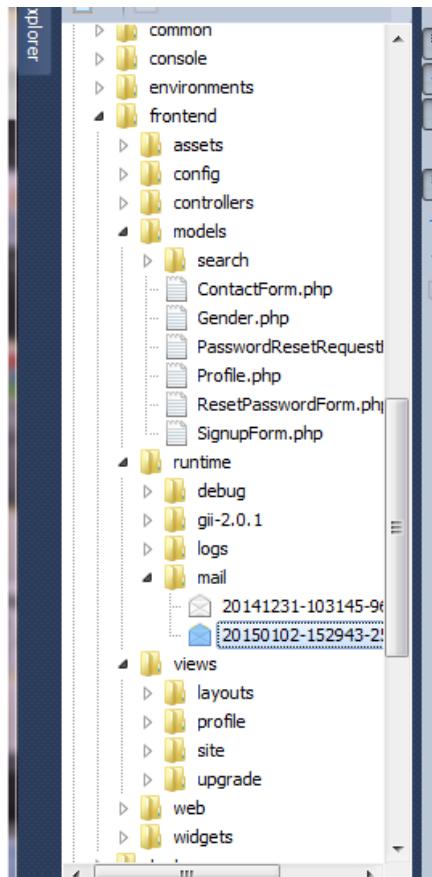
Body
Thanks for Registering!
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce a neque lobortis, facilisis neque et, ullamcorper quam. Sed velit dui, ultrices eu augue id, euismod volutpat massa. Integer purus felis, maximus et mollis nec, efficitur non diam. In sit amet elit et mauris finibus consequat eu sed magna. Proin vel lorem vitae elit finibus vehicula non sed massa. Quisque id mi vel nulla molestie interdum. Vivamus mattis, velit et viverra pretium, eros nunc semper ligula, vel volutpat orci diam sed nibh. Maecenas eget sapien augue. Sed cursus risus quis dolor dignissim, sed tempus arcu convallis. Morbi ornare tortor lacus. Sed nec tincidunt diam, vitae faucibus metus. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Suspendisse potenti. Donec pretium blandit leo, a facilisis massa congue id. Phasellus lobortis ullamcorper ex et viverra.

Create

Status Message Input

We start with the controller name. Then the action name, case sensitive, don't include the word action. Next comes the name of the status message, followed by a brief description. Then we have the subject and body. So go ahead and save this record.

Next, register a new user. Then check your frontend/runtime/mail folder and you will see your email in there.



Runtime Mail

I know I mentioned it before, but this is where all emails go in dev mode. Check that and you'll see that it indeed did send an email, with exactly the right content.

Now anywhere you want an autorepsonder in the application, just pop into the controller:

```
MailCall::onMailableAction($action_name, $controller_name);
```

And obviously, you have to use strings for the method signature, not variables. So just to reiterate for clarity, if you wanted to put that somewhere in the site controller's contact action:

```
MailCall::onMailableAction('contact', 'site');
```

If you didn't already have the use statement for MailCall included, you would have to include that as well. Then just create a matching status message record and it works. You can see how easy this will be to implement across the application.

And that's pretty much it. As with all code, it is not meant to be a final solution or grand solution, just one possibility among many. A lot of the choices you make in coding come down to personal preference and there is typically always room for improvement. Just do it your way and have fun coding!

Dropdown Navigation

It's time we used dropdown navigation for our top nav in the backend, otherwise we end up with way too much clutter. This should be the easiest task we face, right?

Unfortunately, it's not clear from the Yii 2 guide how their dropdown widget works, at least not to me. I wasn't satisfied with not knowing, so I looked around and googled like crazy. I saw an implementation on someone else's plugin and figured that the native Yii 2 Nav widget might work the same way, so I figured out a solution.

The actual implementation looks like this:

```
echo Nav::widget([
    'options' => ['class' => 'navbar-nav navbar-right'],
    'items' => [
        ['label' => 'Users', 'items' => [
            ['label' => 'Users', 'url' => ['user/index']],
            ['label' => 'Profiles', 'url' => ['profile/index']],
            ['label' => 'Something else here', 'url' => ['#']],
        ]],
        ['label' => 'Support', 'items' => [
            ['label' => 'Support Requests', 'url' => ['/content/index']],
            ['label' => 'Status Messages', 'url' => ['/status-message/index']],
            ['label' => 'FAQ', 'url' => ['/faq/index']],
            ['label' => 'FAQ Category', 'url' => ['/faq-category/index']],
        ]],
        ['label' => 'RBAC', 'items' => [
            ['label' => 'Roles', 'url' => ['/role/index']],
            ['label' => 'User Types', 'url' => ['/user-type/index']],
        ]],
    ],
]);
```

```

['label' => 'Statuses', 'url' => ['/status/index']],

]],

['label' => 'Content', 'items' => [
    ['label' => 'Content', 'url' => ['/content/index']],
    ['label' => 'Status Messages', 'url' => ['/status-message/index']],
    ['label' => 'FAQ', 'url' => ['/faq/index']],
    ['label' => 'FAQ Category', 'url' => ['/faq-category/index']],
]],

],
]);

```

I purposely did not provide the Gist. I will be giving you the entire file when I'm done explaining everything. You can see from the above, I've simply used nested arrays to create the dropdown.

As you know from previous chapters, I've wrapped the admin portions of the navigation in an if statement to determine if the user is logged in and has a minimum of admin status:

```
if (!Yii::$app->user->isGuest && $is_admin) {
```

So you would think I could just pop this into place where I previously had the nav, but no. We have to put the test for logged in or out above it:

```

if (Yii::$app->user->isGuest) {

$menuItemsLogOut[] = ['label' => 'Login', 'url' => ['site/login']];

} else {

$menuItemsLogOut[] =

['label' => 'Logout (' . Yii::$app->user->identity->username . ')',
 'url' => ['/site/logout'],
 'linkOptions' => ['data-method' => 'post']
];

}

```

```
echo Nav::widget([
    'options' => ['class' => 'navbar-nav navbar-right'],
    'items' => $menuItemsLogOut
]);
```

Because we are using:

```
'options' => ['class' => 'navbar-nav navbar-right'],
```

It pulls everything to the right. So each subsequent Nav::widget stacks to the left of the first one. So essentially they need to go in reverse order.

Also, you can see that I've used two different \$menuItems arrays. This is because we now have multiple Nav::Widget calls, so we can't use the same \$menuItems array for both of them.

In this if statement:

```
if (Yii::$app->user->isGuest) {

    $menuItemsLogOut[] = ['label' => 'Login', 'url' => ['site/login']];

} else {

    $menuItemsLogOut[] =

        ['label' => 'Logout (' . Yii::$app->user->identity->username . ')',
         'url' => ['/site/logout'],
         'linkOptions' => ['data-method' => 'post']
    ];

}

echo Nav::widget([
    'options' => ['class' => 'navbar-nav navbar-right'],
    'items' => $menuItemsLogOut
]);
```

First we are setting an element to the array, depending on whether or not the user is logged in. Then we call the array from within Nav::Widget.

This is a useful technique for testing for the state of the user, then deciding what to show them. Note that on the if statement where we decide whether or not to show the items at all, we don't need separate arrays because it's a choice between showing the items or nothing.

Like I said, this can be tricky to work with. I'm going to give the entire backend/views/layouts/-main.php file, so we can make sure we have everything in the correct order:

Gist:

Backend Layouts Main

from book:

```
<?php

use backend\assets\AppAsset;
use yii\helpers\Html;
use yii\bootstrap\Nav;
use yii\bootstrap\NavBar;
use yii\widgets\Breadcrumbs;
use common\models\PermissionHelpers;
use backend\assets\FontAwesomeAsset;

/**
 * @var \yii\web\View $this
 * @var string $content
 */

AppAsset::register($this);
FontAwesomeAsset::register($this);

?>

<?php $this->beginPage() ?>

<!DOCTYPE html>

<html lang="<?= Yii::$app->language ?>">

<head>
    <meta charset="<?= Yii::$app->charset ?>"/>

    <meta name="viewport"
        content="width=device-width,
        initial-scale=1">

    <?= Html::csrfMetaTags() ?>
```

```
<title><?= Html::encode($this->title) ?></title>

<?php $this->head() ?>

</head>

<body>
    <?php $this->beginBody() ?>

    <div class="wrap">

        <?php

        if (!Yii::$app->user->isGuest){

            $is_admin = PermissionHelpers::requireMinimumRole('Admin');

            NavBar::begin([
                'brandLabel' => 'Yii 2 Built <i class="fa fa-plug"></i> Admin',
                'brandUrl' => Yii::$app->homeUrl,
                'options' => [
                    'class' => 'navbar-inverse navbar-fixed-top',
                ],
            ]);
        }

        } else {

            NavBar::begin([
                'brandLabel' => 'Yii 2 Built <i class="fa fa-plug"></i>',
                'brandUrl' => Yii::$app->homeUrl,
                'options' => [
                    'class' => 'navbar-inverse navbar-fixed-top',
                ],
            ]);
        }

        if (Yii::$app->user->isGuest) {
```

```
$menuItemsLogOut[] = ['label' => 'Login', 'url' => ['site/login']];

} else {

$menuItemsLogOut[] = [
    'label' => 'Logout (' . Yii::$app->user->identity->username . ')',
    'url' => ['/site/logout'],
    'linkOptions' => ['data-method' => 'post']
];
}

echo Nav::widget([
    'options' => ['class' => 'navbar-nav navbar-right'],
    'items' => $menuItemsLogOut
]);

if (!Yii::$app->user->isGuest && $is_admin) {

echo Nav::widget([
    'options' => ['class' => 'navbar-nav navbar-right'],
    'items' => [

['label' => 'Users', 'items' => [
    ['label' => 'Users', 'url' => ['user/index']],
    ['label' => 'Profiles', 'url' => ['profile/index']],
    ['label' => 'Something else here', 'url' => ['#']],
]],

['label' => 'Support', 'items' => [
    ['label' => 'Support Requests', 'url' => ['content/index']],
    ['label' => 'Status Messages', 'url' => ['status-message/index']],
    ['label' => 'FAQ', 'url' => ['faq/index']],
    ['label' => 'FAQ Categories', 'url' => ['faq-category/index']],
]],

['label' => 'RBAC', 'items' => [
    ['label' => 'Roles', 'url' => ['role/index']],
    ['label' => 'User Types', 'url' => ['user-type/index']],
    ['label' => 'Statuses', 'url' => ['status/index']],
]],
```

```
[ 'label' => 'Content', 'items' => [
    [ 'label' => 'Content', 'url' => [ 'content/index' ] ],
    [ 'label' => 'Status Messages', 'url' => [ 'status-message/index' ] ],
    [ 'label' => 'FAQ', 'url' => [ 'faq/index' ] ],
    [ 'label' => 'FAQ Category', 'url' => [ 'faq-category/index' ] ],
],
],
]);
}

$menuItems = [ [ 'label' => 'Home', 'url' => [ 'site/index' ] ],
];

echo Nav::widget([
    'options' => [ 'class' => 'navbar-nav navbar-right' ],
    'items' => $menuItems
]);

NavBar::end();

?>

<div class="container">

<?= Breadcrumbs::widget([
    'links' => isset($this->params[ 'breadcrumbs' ]) ?
    $this->params[ 'breadcrumbs' ] : [],
])?>

<?= $content ?>

</div>
</div>
```

```
<footer class="footer">

    <div class="container">

        <p class="pull-left">&copy; Yii 2 Build <?= date('Y') ?></p>

        <p class="pull-right"><?= Yii::powered() ?></p>

    </div>

</footer>

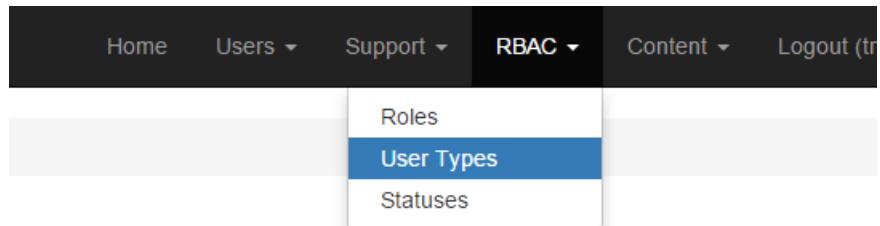
<?php $this->endBody() ?>

</body>
</html>

<?php $this->endPage() ?>
```

Something you may have noticed in the above file. We don't echo NavBar::begin. We do, however, need to echo the nav elements.

Also note, I have put some placeholders in certain spots in the dropdown for demonstration purposes. When you play around with this, you'll probably notice the cool highlight it gives for the selected dropdown. At first I thought that was a CSS mistake until I realized it was highlighting the dropdown element for the page we were on. When you are done, it should look like the image below:



Dropdown Nav

Well, that it's for the Nav dropdown, I hope you find that useful.

FAQ

Next we're going to set up an FAQ model and use some interesting elements that we have not used before, including use of Yii 2's `ArrayDataProvider`, a very handy iterator for use in sending data to the views.

When we think of FAQs, we think of simplicity, just questions and answers, and simplicity typically serves us well. But if we think about our demanding client and things that are important to them, the order of the questions, for example, might be very important. And obviously, the order they want will most likely not match the ascending or descending timestamp for creation or alphabetical order.

In most cases, the client won't even know what order they want the questions in until long after the project has been developed. So what we need is a way for them to determine the order via backend UI. Now this is really simple to do of course. We just add a `faq_value` column, with a data type of `int`, that we can sort either ascending or descending depending on our preference.

But thinking it over, `faq_value` as a name for the column might be too generic. I could call it something more descriptive like `faq_importance` or `faq_weight`. I think `faq_importance` is too much to type and prone to typos, but `faq_weight` seems like a good choice.

The reason why I just don't call it `weight`, is that I might use the `weight` concept on another table and I try to avoid having the same column name in different tables as much as I can. This helps avoid ambiguation problems with queries later on.

So the way our `faq_weight` would work would be that in an ascending sort, an FAQ with a `faq_weight` of 10 will be higher on the list than an FAQ with an `faq_weight` of 20. And as long as we give the client a method for changing the `faq_weight` of an FAQ from backend UI, the problem of the client needing to control the order is anticipated and solved before it ever becomes an issue.

So thinking along the same lines, what if the client wants to grab a group of specific questions and present them on different parts of the application? We can anticipate that they may well want do that, so we can create a boolean field on the db for `faq_is_featured`. That way if we need to call a group of featured FAQs for special presentation, we have an easy way to extract those FAQs, we just call FAQs where `faq_is_featured` is set to yes, in other words, a 1.

So let's get started and set up the data structure for FAQ. We will need the following columns on our `faq` table:

`id` (int, PK, NN, AI)

`faq_question` (VARCHAR(255), NN)

`faq_answer` (VARCHAR(1055), NN)

`faq_category_id` (INT)

`faq_is_featured` (BOOL, DEFAULT = 0)

`faq_weight` (INT, DEFAULT = 100)

created_by (INT)

updated_by (INT)

created_at (DATETIME)

updated_at (DATETIME)

Here is a screenshot:

The screenshot shows the 'faq' table definition in MySQL Workbench. The table has the following columns:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
faq_question	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
faq_answer	VARCHAR(1055)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
faq_category_id	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
faq_is_featured	BOOL	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'0'
faq_weight	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'100'
created_by	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
updated_by	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
created_at	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
updated_at	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Below the table definition are tabs for Columns, Indexes, Foreign Keys, Triggers, Partitioning, Options, Inserts, and Privileges.

faq table

We also need a table for faq_category. Here is the data structure:

id (INT, PK, NN, AI)

faq_category_name (VARCHAR(45))

faq_category_weight (INT, DEFAULT = 100)

faq_category_is_featured (Bool, DEFAULT = 0)

Here is a screenshot:

The screenshot shows the 'faq_category' table definition in MySQL Workbench. The table has the following columns:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT(11)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>					
faq_category_name	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
faq_category_weight	INT(11)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'100'
faq_category_is_featured	TINYINT(1)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'0'

faq table

Ok, so you can see that I've set up a separate faq_category table to hold the names of the categories. And just to be one step ahead of the game, I've built into the data structure the same type of ordering that I intend for faq. While we won't actually be using it for this chapter, it is there, should you decide to implement it.

If our pesky client falls in love with ordering his FAQs, he or she might just demand the same control over faq_categories.

The results in Php Myadmin should look like this:

The screenshot shows the 'Structure' tab of the PhpMyAdmin interface for the 'faq' table. The table has 10 columns:

#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	<code>id</code>	int(11)	latin1_general_ci		No	None	AUTO_INCREMENT	Change Drop Primary Unique Index Spatial Fulltext Distinct values
2	<code>faq_question</code>	varchar(255)	latin1_general_ci		No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
3	<code>faq_answer</code>	varchar(1055)	latin1_general_ci		No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
4	<code>faq_category_id</code>	int(11)			Yes	NULL		Change Drop Primary Unique Index Spatial Fulltext Distinct values
5	<code>faq_is_featured</code>	tinyint(1)			Yes	0		Change Drop Primary Unique Index Spatial Fulltext Distinct values
6	<code>faq_weight</code>	int(11)			Yes	100		Change Drop Primary Unique Index Spatial Fulltext Distinct values
7	<code>created_by</code>	int(11)			Yes	NULL		Change Drop Primary Unique Index Spatial Fulltext Distinct values
8	<code>updated_by</code>	int(11)			Yes	NULL		Change Drop Primary Unique Index Spatial Fulltext Distinct values
9	<code>created_at</code>	datetime			Yes	NULL		Change Drop Primary Unique Index Spatial Fulltext Distinct values
10	<code>updated_at</code>	datetime			Yes	NULL		Change Drop Primary Unique Index Spatial Fulltext Distinct values

Below the table, there are buttons for 'Check All', 'With selected:', and various actions like 'Browse', 'Change', 'Drop', 'Primary', 'Unique', and 'Index'. There are also links for 'Print view', 'Relation view', 'Propose table structure', 'Track table', and 'Move columns'. A 'Go' button is at the bottom right.

faq table

And for faq_category:

The screenshot shows the 'Structure' tab of the PhpMyAdmin interface for the 'faq_category' table. The table has 4 columns:

#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	<code>id</code>	int(11)	latin1_general_ci		No	None	AUTO_INCREMENT	Change Drop Browse distinct value
2	<code>faq_category_name</code>	varchar(45)	latin1_general_ci		No	None		Change Drop Browse distinct value
3	<code>faq_category_weight</code>	int(11)			Yes	100		Change Drop Browse distinct value
4	<code>faq_category_is_featured</code>	tinyint(1)			Yes	0		Change Drop Browse distinct value

Below the table, there are buttons for 'Check All / Uncheck All', 'With selected:', and various actions like 'Browse', 'Change', 'Drop', 'Primary', 'Unique', and 'Index'. There are also links for 'Print view', 'Relation view', 'Propose table structure', 'Track table', and 'Move columns'. A 'Go' button is at the bottom right.

At the bottom left, there is a 'Information' panel with 'Space usage' and 'Row Statistics' sections. The 'Space usage' section shows 'Data 16 KiB' and 'Index 0 B'. The 'Row Statistics' section shows 'Format Compact', 'Collation latin1_general_ci', and 'Next autoindex ?'.

faq category table

Something else you may have noticed in our data structure are the columns of `created_by` and `updated_by` on our `faq` table. These columns will allow us to implement a blameable behavior on the model that will automatically stamp it with the user who is creating or updating the record. Don't you just love the name blameable?

Sometimes it's important to know who created a record, there's a variety of reasons for this. It could be that we don't understand why it exists, and so we have to ask the team member that created it. That would be a backend scenario.

On the frontend, if someone posts something that doesn't otherwise visibly track the user doing it, this is a way to track the user, which can be useful for security reasons.

So those are a couple of scenarios where you would want to know who the user is, and obviously there are a lot more. Blameable is a really handy behavior that Yii 2 provides for us, so we will implement it here, and then you can decide where on the rest of the application you would like to use it.

Another thing we're going to do differently here, is take a backend model and create a separate frontend UI for it, this time utilizing the `ArrayDataProvider` that we have not used yet.

Let's go ahead and begin by using Gii to create the two models, `Faq` and `FaqCategory`, which are based on the two tables we just made. We will build these models into the backend, so make sure the namespace is for `backend\models`.

I'm not going to provide screenshots for this because by now you should know how to use Gii. If you need a refresher, please refer to one of the earlier chapters.

Now let's make the crud for both models. Remember we don't need to supply a view path, we are using the default.

Again, I'm not going to provide screenshots for this because by now you should know how to create the CRUD in Gii. If you need a refresher, please refer to one of the earlier chapters.

For each model, you should have a file for:

- model
- search model
- controller
- view
- update
- index
- create
- _search
- _form

Go ahead and check your folders in backend and make sure you have everything. If all is not good, retrace your steps to figure out what went wrong. If the files are missing, and yet you generated

them, they most likely went to the wrong place, which can happen if you make an error in the namespace entries.

At this point, I will assume all is good and move on to modifying files. So we need to make some changes, and we'll start with the model FaqCategory.php first.

Since I used a foreign key in MySql Workbench to create the table relationship between Faq and FaqCategory, Gii has automatically made the necessary relationship method. I will supply it here in case you skipped the foreign key:

Gist:

[Get Faqs](#)

From book:

```
public function getFaqs()
{
    return $this->hasMany(Faq::className(), ['faq_category_id' => 'id']);
}
```

The next thing we want to add to our model is a simple method to return the dropdown list options for faq_category_is_featured. We'll need this for our form view, so we can return yes or no in a dropdown list instead of having to enter 0 or 1 into a form field.

Now we covered this previously when the dropdown list was created from values in a related model, but this is the first time we are doing it this way. This is just formatting the data for the view, no relationship is necessary. But we are doing it in a way that is very consistent with how we do our other methods by placing it on the model, as opposed to inline in the form.

Anyway, here is the method:

Gist:

[GetFaqCategoryIsFeaturedList](#)

From book:

```
public static function getFaqCategoryIsFeaturedList()
{
    $options = [0 => "no", 1 => "yes"];
}
```

The Yii 2 guide shows an inline version of this in the form, but I prefer to create a reusable method. It avoids code duplication because it's very likely you could end up using this drop down list in more than one place.

Also, another benefit is that when you are creating your relationships on your models, you get into a habit of creating one for the dropdown list, which could typically be used in your rules, though we are not doing that this time.

Speaking of rules, we also need to do a little work there, so we get our default set the way we want it:

Gist:

FaqCategory Rules

From book:

```
public function rules()
{
    return [
        [['faq_category_name'], 'required'],
        [['faq_category_weight'], 'faq_category_is_featured'], 'integer'],
        ['faq_category_weight', 'default', 'value' => 100],
        [['faq_category_weight'], 'in', 'range'=>range(1,100)],
        [['faq_category_name']], 'string', 'max' => 45]
    ];
}
```

We added a rule for default value to make sure that gets set, and we are enforcing a range for category weight, from 1 to a 100, using built-in php function range.

Next, let's go to FaqCategoryController and change the behaviors method to make it similar to our other controllers, but this time we'll just have one array for rules:

Gist:

FaqCategoryController

From book:

```

public function behaviors()
{
    return [
        'access' => [
            'class' => \yii\filters\AccessControl::className(),
            'only' => ['index', 'view', 'create', 'update', 'delete'],
            'rules' => [
                [
                    'actions' => ['index', 'view', 'create', 'update', 'delete'],
                    'allow' => true,
                    'roles' => ['@'],
                    'matchCallback' => function ($rule, $action) {
                        return PermissionHelpers::requireMinimumRole('Admin')
                            && PermissionHelpers::requireStatus('Active');
                    }
                ],
            ],
        ],
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'delete' => ['post'],
            ],
        ],
    ];
}

```

Don't forget to add the use statement on the FaqCategory controller:

```
use common\models\PermissionHelpers;
```

So now we can work on our views. Let's change the _form view first:

Gist:

[FaqCategory Form View](#)

From book:

```

<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $model backend\models\FAQCategory */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="faq-category-form">

<?php $form = ActiveForm::begin(); ?>

<?= $form->field($model, 'faq_category_name')->
textInput(['maxlength' => 45]) ?>

<?= $form->field($model, 'faq_category_weight')->textInput() ?>

<?= $form->field($model, 'faq_category_is_featured')->
dropDownList($model->faqCategoryIsFeaturedList,
[ 'prompt' => 'Please Choose One' ]) ?>

<div class="form-group">

<?= Html::submitButton($model->isNewRecord ? 'Create' : 'Update',
[ 'class' => $model->isNewRecord ? 'btn btn-success' : 'btn btn-primary' ]) ?>

</div>

<?php ActiveForm::end(); ?>

</div>

```

So now that we have our form built, let's go ahead and add a record. Login to backend.yii2build.com and use the content dropdown to find FAQ Category. You'll notice that in the nav it's referring to the singular, but on the page, it's plural, so let's take a moment to fix the nav in main.php:

```
[ 'label' => 'FAQ Categories', 'url' => [ 'faq-category/index' ]],
```

We just made a simple change to the label.

After making the change navigate back to backend.yii2build.com and select FAQ Categories from the dropdown.

All the pages should work, even though we only modified _form.php. So from index.php, which is where the nav took you, just click on the create button and you should see our modified form. Add a couple of test records, you can use General and Specific as the test categories.

All the functions should work, including the defaults. Also, if you try entering a number greater than 100 into the Faq Category Weight form field, you will see it returns an error message, so we know our in range validator is working. Cool stuff.

Ok, so once you verified that's all working properly, let's punch our way through the rest of the view changes.

On _search, it's a one line change. We need to swap the text input for faq_category_is_featured with:

```
<?= $form->field($model, 'faq_category_is_featured')->
dropDownList($model->faqCategoryIsFeaturedList,
[ 'prompt' => 'Please Choose One' ])?>
```

Note the second use of the faqCategoryIsFeaturedList method, so it didn't take long for that theory to prove out.

Obviously the code is broken into 3 lines to avoid word wrap in PDF. You should make that a single line in your file.

Ok, moving on. There are no changes at this point to create and update, so now we can tackle view.php. Just a few minor changes:

Gist:

FaqCategory View.php

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\DetailView;

/* @var $this yii\web\View */
/* @var $model backend\models\FaqCategory */

$this->title = $model->faq_category_name;
$this->params['breadcrumbs'][] =
['label' => 'Faq Categories', 'url' => ['index']];
```

```

$this->params['breadcrumbs'][] = $this->title;
?>
<div class="faq-category-view">

    <h1>Faq Category: <?= Html::encode($this->title) ?></h1>

    <p>
        <?= Html::a('Update', ['update', 'id' => $model->id], [
            'class' => 'btn btn-primary']) ?>
        <?= Html::a('Delete', ['delete', 'id' => $model->id], [
            'class' => 'btn btn-danger',
            'data' => [
                'confirm' => 'Are you sure you want to delete this item?',
                'method' => 'post',
            ],
        ]) ?>
    </p>

    <?= DetailView::widget([
        'model' => $model,
        'attributes' => [
            'id',
            'faq_category_name',
            'faq_category_weight',
            ['attribute'=>'faq_category_is_featured',
            'format'=>'boolean'],
        ],
    ]) ?>
</div>

```

So 3 little changes here. We changed the title to \$model->faq_category_name instead of just the id, and we add the words “Faq Category:” to our h1.

The last change was to an attribute in the DetailView widget:

```
[ 'attribute'=>'faq_category_is_featured', 'format'=>'boolean' ],
```

Yii 2 allows us to set the format of the attribute, so in this case we set it to boolean, so now we get Yes and No instead of 0 or 1 for the faq_category_is_featured output.

If everything went well, your view page should look like this:

ID	3
Faq Category Name	Specific
Faq Category Weight	20
Faq Category Is Featured	Yes

Ok, let's move on to index.php.

Gist:

FaqCategory Index

From book:

```
<?php

use yii\helpers\Html;
use yii\grid\GridView;
use \yii\bootstrap\Collapse;

/* @var $this yii\web\View */
/* @var $searchModel backend\models\search\FaqCategorySearch */
/* @var $dataProvider yii\data\ActiveDataProvider */

$this->title = 'Faq Categories';
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="faq-category-index">

<h1><?= Html::encode($this->title) ?></h1>

<?php echo Collapse::widget([
    'items' => [
        // equivalent to the above
    ]
]) ?>
```

```

    [
        'label' => 'Search',
        'content' => $this->render('_search',
        ['model' => $searchModel]) ,
        // open its content by default
        // 'contentOptions' => ['class' => 'in']
    ],
]

]);
?>

<p>
    <?= Html::a('Create Faq Category', ['create'],
    ['class' => 'btn btn-success']) ?>
</p>

<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],

        'id',
        'faq_category_name',
        'faq_category_weight',
        ['attribute'=>'faq_category_is_featured',
        'format'=>'boolean'],

        ['class' => 'yii\grid\ActionColumn'],
    ],
]); ?>

</div>

```

Three simple changes. We pull in the Collapse widget in the use statement, and call our _search within the Collapse widget, just like we did on the other models. And then we formatted the boolean for our attribute faq_category_is_featured, like we did on the view page. And that should look like this:

The screenshot shows a web-based application interface for managing Faq Categories. At the top, there's a navigation bar with 'Home' and 'Faq Categories'. Below it is a search bar. A green button labeled 'Create Faq Category' is visible. The main area displays a table with two rows of data. The columns are labeled '#', 'ID', 'Faq Category Name', 'Faq Category Weight', and 'Faq Category Is Featured'. Row 1 (General) has weight 100 and is not featured. Row 2 (Specific) has weight 20 and is featured. Each row includes edit and delete icons.

#	ID	Faq Category Name	Faq Category Weight	Faq Category Is Featured
1	2	General	100	No
2	3	Specific	20	Yes

FaqCategory Index

Ok, cool, so that's it for FaqCategory, all built, neat and tidy.

Now we're ready to tackle the Faq model itself. We're going to start by adding the behaviors method, which will also contain the blameable behavior.

Gist:

Faq behaviors

From book:

```
public function behaviors()
{
    return [
        'timestamp' => [
            'class' => 'yii\behaviors\TimestampBehavior',
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
            ],
            'value' => new Expression('NOW()'),
        ],
        'blameable' => [
            'class' => BlameableBehavior::className(),
            'createdByAttribute' => 'created_by',
            'updatedByAttribute' => 'updated_by',
        ],
    ];
}
```

```
];
}
```

We need the following add to the use statement at the top of the file:

```
use yii\db\ActiveRecord;
use yii\db\Expression;
use yii\behaviors\BlameableBehavior;
```

You can see how we set up ‘blameable’ in the behaviors array and this syntax is very straightforward. All we have to do is map ‘createdByAttribute’ to ‘created_by’, which is the column name we used in our data structure, and do the same for updatedByAttribute, and we’re set.

Next we have to add our changes for our rules.

Gist:

Faq rules

From book:

```
/**
 * @inheritDoc
 */

public function rules()
{
    return [
        [['faq_question', 'faq_answer'], 'required'],
        [['faq_category_id', 'faq_is_featured', 'faq_weight', 'created_by',
        'updated_by'], 'integer'],
        [['faq_weight'], 'in', 'range'=>range(1,100)],
        ['faq_weight', 'default', 'value' => 100],
        [['created_at', 'updated_at'], 'safe'],
        [['faq_question'], 'string', 'max' => 255],
        [['faq_question'], 'unique'],
        [['faq_answer'], 'string', 'max' => 1055]
    ];
}
```

Notice we that besides the in range rule and default for faq_weight, we also have a unique rule for faq_question. This works great and stops you from repeating the question in Faq.

Next, let’s move on to our relationship and associated methods. We actually have quite a few for such a simple model, but they will all come in very handy.

Gist:

Faq Relationships

From book:

```
/**  
 * uses magic getFaqCategoryName on return statement  
 */  
  
public function getFaqCategoryName()  
{  
    return $this->faqCategory->faq_category_name;  
}  
  
/**  
 * get list of FaqCategory for dropdown  
 */  
public static function getFaqCategoryList()  
{  
  
    $droptions = FaqCategory::find()->asArray()->all();  
    return Arrayhelper::map($droptions, 'id', 'faq_category_name');  
}  
  
public static function getFaqIsFeaturedList()  
{  
    return $droptions = [0 => "no", 1 => "yes"];  
}  
  
public function getFaqIsFeaturedName()  
{  
    return $this->faq_is_featured == 0 ? "no" : "yes";  
}  
  
public function getCreatedByUser()  
{  
    return $this->hasOne(User::className(), ['id' => 'created_by']);  
}/*
```

```

* @getCreateUserName
*
*/
public function getCreatedByUsername()
{
    return $this->createdByUser ?
        $this->createdByUser->username : '- no user -';
}

public function getUpdatedByUser()
{
    return $this->hasOne(User::className(), ['id' => 'updated_by']);
}

/**
* @getUpdateUserName
*
*/
public function getUpdatedByUsername()
{
    return $this->updatedByUser ?
        $this->updatedByUser->username : '- no user -';
}

```

Since we have covered most of these previously, I'm only going to talk about the last 4. When we display created by and update by in our views, we need hook into the user table either by the getCreatedByUser method or the getUpdatedByUser method.

Then we use the relationship to be able to return the username in the next method, getCreatedByUsername, which is what we will use in our views.

And of course to have access to the User and FaqCategory model, as well as the helper classes, we need to add to our use statement.

Gist:

Use Statement

From book:

```

use backend\models\FaqCategory;
use yii\helpers\ArrayHelper;
use yii\helpers\Url;
use yii\helpers\Html;
use common\models\User;

```

And lastly, we need to add to replace our attributeLabels with the following.

Gist:

Faq Labels

From book:

```
public function attributeLabels()
{
    return [
        'id' => 'ID',
        'faq_question' => 'Question',
        'faq_answer' => 'Answer',
        'faq_category_id' => 'Category',
        'faq_weight' => 'Weight',
        'faq_is_featured' => 'Featured?',
        'created_by' => 'Created By',
        'updated_by' => 'Updated By',
        'created_at' => 'Created At',
        'updated_at' => 'Updated At',
        'faqCategoryName' => Yii::t('app', 'Category'),
        'faqCategoryList' => Yii::t('app', 'Category'),
        'faqIsFeaturedName' => Yii::t('app', 'Featured'),
        'createdByUserName' => Yii::t('app', 'Created By'),
        'updatedByUserName' => Yii::t('app', 'Updated By'),
    ];
}
```

Just for reference, so you have the complete and correct Faq model, I'm going to include the code for the entire model here.

Gist:

Faq Model

From book:

```
<?php

namespace backend\models;

use Yii;
use yii\db\ActiveRecord;
use yii\db\Expression;
use yii\behaviors\BlameableBehavior;
use backend\models\FaqCategory;
use yii\helpers\ArrayHelper;
use yii\helpers\Url;
use yii\helpers\Html;
use common\models\User;

/**
 * This is the model class for table "faq".
 *
 * @property integer $id
 * @property string $faq_question
 * @property string $faq_answer
 * @property integer $faq_category_id
 * @property integer $faq_is_featured
 * @property integer $faq_weight
 * @property integer $created_by
 * @property integer $updated_by
 * @property string $created_at
 * @property string $updated_at
 *
 * @property FaqCategory $faqCategory
 */

```

```
class Faq extends \yii\db\ActiveRecord
{
    /**
     * @inheritdoc
     */
    public static function tableName()
    {
        return 'faq';
    }

    public function behaviors()
```

```
{  
    return [  
        'timestamp' => [  
            'class' => 'yii\\behaviors\\TimestampBehavior',  
            'attributes' => [  
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],  
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],  
                ],  
            'value' => new Expression('NOW()'),  
            ],  
        'blameable' => [  
            'class' => BlameableBehavior::className(),  
            'createdByAttribute' => 'created_by',  
            'updatedByAttribute' => 'updated_by',  
            ],  
    ];  
}  
  
/**  
 * @inheritDoc  
 */  
public function rules()  
{  
    return [  
        [[ 'faq_question', 'faq_answer'], 'required'],  
        [[ 'faq_category_id', 'faq_is_featured', 'faq_weight',  
        'created_by', 'updated_by'], 'integer'],  
        [[ 'faq_weight'], 'in', 'range'=>range(1,100)],  
        ['faq_weight', 'default', 'value' => 100],  
        [[ 'created_at', 'updated_at'], 'safe'],  
        [[ 'faq_question'], 'string', 'max' => 255],  
        [[ 'faq_question'], 'unique'],  
        [[ 'faq_answer'], 'string', 'max' => 1055]  
    ];  
}  
  
/**  
 * @inheritDoc  
 */  
public function attributeLabels()  
{
```

```
        return [
            'id' => 'ID',
            'faq_question' => 'Question',
            'faq_answer' => 'Answer',
            'faq_category_id' => 'Category',
            'faq_weight' => 'Weight',
            'faq_is_featured' => 'Featured?',
            'created_by' => 'Created By',
            'updated_by' => 'Updated By',
            'created_at' => 'Created At',
            'updated_at' => 'Updated At',
            'faqCategoryName' => Yii::t('app', 'Category'),
            'faqCategoryList' => Yii::t('app', 'Category'),
            'faqIsFeaturedName' => Yii::t('app', 'Featured'),
            'createdByUserName' => Yii::t('app', 'Created By'),
            'updatedByUserName' => Yii::t('app', 'Updated By'),
        ];
    }

    /**
     * @return \yii\db\ActiveQuery
     */
    public function getFaqCategory()
    {
        return $this->hasOne(FaqCategory::className(),
['id' => 'faq_category_id']);
    }

    /**
     * uses magic getFaqCategoryName on return statement
     *
     */
    public function getFaqCategoryName()
    {
        return $this->faqCategory->faq_category_name;
    }

    /**
     * get list of FaqCategory for dropdown
     */
}
```

```
public static function getFaqCategoryList()
{
    $droptions = FaqCategory::find()->asArray()->all();
    return Arrayhelper::map($droptions, 'id', 'faq_category_name');

}

public static function getFaqlIsFeaturedList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}

public function getFaqlIsFeaturedName()
{
    return $this->faq_is_featured == 0 ? "no" : "yes";
}

public function getCreatedByUser()
{
    return $this->hasOne(User::className(),
['id' => 'created_by']);
}
/**
 * @getCreateUserName
 *
 */
public function getCreatedByUsername()
{
    return $this->createdByUser ?
$this->createdByUser->username : '- no user -';
}

public function getUpdatedByUser()
{
    return $this->hasOne(User::className(),
['id' => 'updated_by']);
}
/**
 * @getUpdateUserName
 *
```

```

/*
public function getUpdatedByUsername()
{
    return $this->updatedByUser ?
$this->updatedByUser->username : '- no user -';
}
}

```

And that does it for the Faq model. Now let's look at Faq Search. I'm not going to step you through the changes because this was covered in detail in chapter 11, how to modify the native search to use eager loading relationships.

Even though the FaqCategories table is likely to be small, it doesn't hurt to use eager loading for efficiency.

Gist:

Faq Search

From book:

```

<?php

namespace backend\models\search;

use Yii;
use yii\base\Model;
use yii\data\ActiveDataProvider;
use yii\data\ArrayDataProvider;
use yii\db\ActiveQuery;
use yii\db\Query;
use backend\models\Faq;

/**
 * FaqSearch represents the model behind the search form about `backend\models\FAQ`.
 */
class FaqSearch extends Faq
{
    public $faqCategoryName;
    public $faqCategoryList;
    public $faqIsFeaturedName;
    public $createdByUsername;
    public $updatedByUsername;
    public $faq_category;
}
```

```
    public $faq_weight;

    /**
     * @inheritDoc
     */
    public function rules()
    {
        return [
            ['id', 'faq_category_id', 'faq_weight', 'faq_is_featured',
                'created_by', 'updated_by'], 'integer'],
            [['faq_question', 'faq_answer', 'created_at', 'updated_at',
                'faqCategoryName', 'faqCategoryList', 'faqIsFeaturedName',
                'createdByUsername', 'updatedByUsername', 'faq_category',
                'faq_weight'], 'safe'],
        ];
    }

    /**
     * @inheritDoc
     */
    public function scenarios()
    {
        // bypass scenarios() implementation in the parent class
        return Model::scenarios();
    }

    /**
     * Creates data provider instance with search query applied
     *
     * @param array $params
     *
     * @return ActiveDataProvider
     */
    public function search($params)
    {
        $query = Faq::find();
        $dataProvider = new ActiveDataProvider([
            'query' => $query,
```

```
]);  
  
    /**  
     * Setup your sorting attributes  
     * Note: This is setup before the $this->load($params)  
     * statement below  
     */  
    $dataProvider->setSort([  
  
        'defaultOrder' => [  
            'faq_weight' => SORT_ASC,  
  
        ],  
  
        'attributes' => [  
            'id',  
            'faq_question' => [  
                'asc' => ['faq.faq_question' => SORT_ASC],  
                'desc' => ['faq.faq_question' => SORT_DESC],  
                'label' => 'Question'  
            ],  
            'faq_answer' => [  
                'asc' => ['faq.faq_answer' => SORT_ASC],  
                'desc' => ['faq.faq_answer' => SORT_DESC],  
                'label' => 'Answer'  
            ],  
  
            'faqCategoryName' => [  
                'asc' => ['faq_category.faq_category_name' => SORT_ASC],  
                'desc' => ['faq_category.faq_category_name' => SORT_DESC],  
                'label' => 'Category'  
            ],  
  
            'faq_weight' => [  
                'asc' => ['faq.faq_weight' => SORT_ASC],  
                'desc' => ['faq.faq_weight' => SORT_DESC],  
                'label' => 'Weight'  
            ],  
            'faqIsFeaturedName' => [  
                'asc' => ['faq.faq_is_featured' => SORT_ASC],  
                'desc' => ['faq.faq_is_featured' => SORT_DESC],  
            ],  
        ]  
    ]);  
}
```

```
'label' => 'Featured?'
] ,
]);

if (!($this->load($params) && $this->validate())) {

    $query->joinWith(['faqCategory']);

    return $dataProvider;
}

$this->addSearchParameter($query, 'id');
$this->addSearchParameter($query, 'faq_category_id');
$this->addSearchParameter($query, 'faq_weight');
$this->addSearchParameter($query, 'faq_is_featured');
$this->addSearchParameter($query, 'created_by');
$this->addSearchParameter($query, 'updated_by');
$this->addSearchParameter($query, 'faq_question', true);
$this->addSearchParameter($query, 'faq_answer', true);

// filter by category
$query->joinWith(['faqCategory' => function ($q) {
    $q->andFilterWhere(['=' , 'faq_category.faq_category_name',
$this->faqCategoryName]);
}]);

return $dataProvider;
}

protected function addSearchParameter($query, $attribute, $partialMatch = false)
{
    if (($pos = strpos($attribute, '.')) !== false) {
        $modelAttribute = substr($attribute, $pos + 1);
    } else {
        $modelAttribute = $attribute;
    }
}
```

```
        }

        $value = $this->$modelAttribute;

        if (trim($value) === '') {
            return;
        }

        /*
         * The following line is additionally added for right aliasing
         * of columns so filtering happen correctly in the self join
         */
        $attribute = "faq.$attribute";

        if ($partialMatch) {
            $query->andWhere(['like', $attribute, $value]);
        } else {
            $query->andWhere([$attribute => $value]);
        }
    }
}
```

You may notice that we have used statements on this search model that we have not used before:

```
use yii\data\ArrayDataProvider;  
use yii\db\ActiveQuery;  
use yii\db\Query;
```

We will need those later when we modify this class, so I included them now.

The other thing that is different is that we have specified a default order:

```
'defaultOrder' => [
    'faq_weight' => SORT_ASC,
],
```

This will set the default order of results to ascending on `faq_weight`, so to control the order we see them in, all we have to do is assign the `faq_weight` according through the UI. Please note that this method is for backend UI only. As promised, we will be doing it differently for the frontend, and that will involve using a different method. I will explain more on that when we get there.

Ok, let's make our typical change to behaviors on the controller:

Gist:

Faq Behaviors

From book:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => \yii\filters\AccessControl::className(),
            'only' => ['index', 'view', 'create', 'update', 'delete'],
            'rules' => [
                [
                    'actions' => ['index', 'view', 'create', 'update', 'delete'],
                    'allow' => true,
                    'roles' => ['@'],
                    'matchCallback' => function ($rule, $action) {
                        return PermissionHelpers::requireMinimumRole('Admin')
                            && PermissionHelpers::requireStatus('Active');
                    }
                ],
            ],
        ],
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'delete' => ['post'],
            ],
        ],
    ];
}
```

Don't forget to add the use statement:

```
use common\models\PermissionHelpers;
```

So let's change the views now, starting with _form.php.

Gist:

Faq Form

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $model backend\models\Faq */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="faq-form">

    <?php $form = ActiveForm::begin(); ?>

    <?= $form->field($model, 'faq_question')->
textInput(['maxlength' => 255]) ?>

    <?= $form->field($model, 'faq_answer')->
textArea(['maxlength' => 1055, 'rows' =>10]) ?>

    <?= $form->field($model, 'faq_category_id')->
dropDownList($model->faqCategoryList,
[ 'prompt' => 'Please Choose One' ]); ?>

    <?= $form->field($model, 'faq_is_featured')->
dropDownList($model->faqIsFeaturedList,
[ 'prompt' => 'Please Choose One' ]); ?>

    <?= $form->field($model, 'faq_weight')->textInput() ?>

    <div class="form-group">
        <?= Html::submitButton($model->isNewRecord ? 'Create' : 'Update',
[ 'class' => $model->isNewRecord ? 'btn btn-success' : 'btn btn-primary']) ?>
    </div>

    <?php ActiveForm::end(); ?>
```

```
</div>
```

Something new we did there was change faq_answer to a textArea. And with the textArea method, you can specify the number of rows, which we have set to 10.

Let's move on to the _search view.

Gist:

Faq Search Partial

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $model backend\models\search\FaqSearch */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="faq-search">

    <?php $form = ActiveForm::begin([
        'action' => ['index'],
        'method' => 'get',
    ]); ?>

    <?= $form->field($model, 'id') ?>

    <?= $form->field($model, 'faq_question') ?>

    <?= $form->field($model, 'faq_answer') ?>

    <?= $form->field($model, 'faq_category_id')->
dropDownList($model->getFaqCategoryList(),
[ 'prompt' => 'Please Choose One' ]);?>

    <?= $form->field($model, 'faq_is_featured')->
dropDownList($model->faqIsFeaturedList,
[ 'prompt' => 'Please Choose One' ]);?>

    <?= $form->field($model, 'faq_weight') ?>
```

```

<div class="form-group">
    <?= Html::submitButton('Search',
    ['class' => 'btn btn-primary']) ?>
    <?= Html::resetButton('Reset',
    ['class' => 'btn btn-default']) ?>
</div>

<?php ActiveForm::end(); ?>

</div>

```

Now let's move on to view.php.

Gist:

[Faq View](#)

From book:

```

<?php

use yii\helpers\Html;
use yii\widgets\DetailView;

/* @var $this yii\web\View */
/* @var $model backend\models\Faq */

$this->title = 'FAQ: ' . $model->faq_question;
$this->params['breadcrumbs'][] =
['label' => 'Faqs', 'url' => ['index']];
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="faq-view">

    <h1><?= Html::encode($this->title) ?></h1>

    <p>
        <?= Html::a('Update', ['update', 'id' => $model->id],
        ['class' => 'btn btn-primary']) ?>
        <?= Html::a('Delete', ['delete', 'id' => $model->id], [
            'class' => 'btn btn-danger',
            'data' => [

```

```

        'confirm' => 'Are you sure you want to delete this item?',
        'method' => 'post',
    ],
]) ?>
</p>

<?= DetailView::widget([
    'model' => $model,
    'attributes' => [
        'id',
        'faq_question',
        'faq_answer',
        'faqCategory.faq_category_name',
        'faq_weight',
        ['attribute'=>'faq_is_featured', 'format'=>'boolean'],
        ['attribute'=>'createdByUsername', 'format'=>'raw'],
        ['attribute'=>'updatedByUsername', 'format'=>'raw'],
        'created_at',
        'updated_at',
    ],
]) ?>

</div>

```

You can see we changed the title and made the attribute changes to DetailView widget so we can get the proper format when we view the record.

Now let's work on the index.php file for Faq.

Gist:

Faq Index

From book:

```

<?php

use yii\helpers\Html;
use yii\grid\GridView;
use \yii\bootstrap\Collapse;

/* @var $this yii\web\View */
/* @var $searchModel backend\models\search\FaqSearch */
/* @var $dataProvider yii\data\ActiveDataProvider */

```

```
$this->title = 'Faqs';
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="faq-index">

    <h1><?= Html::encode($this->title) ?></h1>
        <?php echo Collapse::widget([
            'items' => [
                // equivalent to the above
                [
                    'label' => 'Search',
                    'content' => $this->render('_search', ['model' => $searchModel]) ,
                    // open its content by default
                    //'contentOptions' => ['class' => 'in']
                ],
            ],
        ]);
    ?>

    <p>
        <?= Html::a('Create Faq', ['create'],
            ['class' => 'btn btn-success']) ?>
    </p>

    <?= GridView::widget([
        'dataProvider' => $dataProvider,
        'filterModel' => $searchModel,
        'columns' => [
            ['class' => 'yii\grid\SerialColumn'],

            'id',
            'faq_question',
            'faq_answer',
            ['attribute'=>'faqCategoryName', 'format'=>'raw'],
            'faq_weight',
            ['attribute'=>'faqIsFeaturedName', 'format'=>'raw'],

            ['class' => 'yii\grid\ActionColumn'],
        ],
    ]);
?>
```

```
</div>
```

That should look familiar. One difference however is that we used our `getFaqlsFeaturedName` method (via magic call) to display the yes or no for boolean. We could have used:

```
[ 'attribute'=>'faq_is_featured', 'format'=>'boolean' ],
```

But I thought it was best to use the magic call because we are using that method in our search method and it's best to be consistent, since our search model returns the results on index.

If you haven't already done so, create at least six Faq records, so you can see how all this looks.

So now we're going to move on to creating the frontend version of the Faqs. Although we could use the same dataprovider and widgets that we are using in the backend, I want to free us from that and at the same time, demonstrate the use of the `ArrayDataProvider` class, which returns results in an array, where you can easily set things like the order and which attributes to return.

We'll start by adding a method to our `FaqSearch` model named `frontendProvider`. I'm just calling it that so I can easily determine its use from the name.

Like I said, the purpose of this method is to provide the data for a list of FAQs sorted by `faq_weight`. This will allow the end user to control where the question appears on the list, simply by adjusting the weight up or down.

So go ahead and add this method to `FaqSearch.php`.

Gist:

Frontend Provider

From book:

```
public function frontendProvider()
{
    $query = new Query;
    $provider = new ArrayDataProvider([
        'allModels' => $query->from('faq')->all(),
        'sort' => [
            'defaultOrder' => [
                'faq_weight' => SORT_ASC,
            ],
            'attributes' => ['faq_question', 'faq_answer',
                'faq_weight'],
        ],
    ],
}
```

```
    'pagination' => [
        'pageSize' => 10,
    ],
]);

return $provider;

}
```

Ok, let's step through it. We start by creating an instance of Query, which will allow us to create a query within the instance of ArrayDataProvider. Please note that we need to pull in the following for this to work:

```
use yii\data\ArrayDataProvider;
use yii\db\ActiveQuery;
use yii\db\Query;
```

So make sure the use statements are in the appropriate place at the top of the file. Yii 2 does an excellent job of complaining, so if you do leave one out, it will let you know what's missing.

Ok, back to the code. I referenced for this example:

<http://www.yiiframework.com/doc-2.0/guide-output-data-providers.html>

So if you check the guide you will see it is exactly in that format.

The ArrayDataProvider is configured with the type of array config we see in many places on Yii 2. In this case, we have an 'allModels' key, which points to a query as value, which returns all the faq record results. And since we are using ArrrayDataProvider, we know we will be returning an array.

The guide doesn't show you how to set a default order, but I figured that out. We set the default order to faq_value, SORT_ASC, which means it will sort in ascending value. This means an FAQ with a value of 1 will come 1st on the list.

Then we tell it what attributes we want in the array. In this case, since we are not using this for backend record creation and maintenance, we can include fewer attributes. In this case, we only need:

```
'attributes' => ['faq_question', 'faq_answer', 'faq_weight'],
```

We are also setting page size for pagination to 10. You probably won't need that, but if you do, it's set. If you have more than 10 records, you also need to setup pagination, but we will do that later for another iteration, it's not necessary now. So this is pretty simple stuff.

Next we need to create a frontend controller for Faq, that uses the backend Faq model and backend FaqSearch model. We're only going to have 2 actions view and index. There is no need for create,

update or delete actions because they are handled in the backend by admin, not in the frontend by users. Because these are FAQs, we want them visible to all users, so we won't be restricting access, so this makes the controller fairly simple.

Here is the entire controller.

Gist:

Faq Frontend Controller

From book:

```
<?php

namespace frontend\controllers;

use Yii;
use backend\models\Faq;
use backend\models\search\FaqSearch;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;

/**
 * FaqController implements the CRUD actions for Faq model.
 */

class FaqController extends Controller
{
    public function behaviors()
    {
        return [
            'verbs' => [
                'class' => VerbFilter::className(),
                'actions' => [
                    'delete' => ['post'],
                ],
            ],
        ];
    }

    /**
     * Lists all Faq models.
     * @return mixed
     */
}
```

```
public function actionIndex()
{
    $searchModel = new FaqSearch();

    $provider = $searchModel->frontendProvider();

    return $this->render('index', [
        'searchModel' => $searchModel,
        'provider' => $provider,
    ]);
}

/**
 * Displays a single Faq model.
 * @param integer $id
 * @return mixed
 */

public function actionView($id)
{
    return $this->render('view', [
        'model' => $this->findModel($id),
    ]);
}

protected function findModel($id)
{
    if (($model = Faq::findOne($id)) !== null) {
        return $model;
    } else {
        throw new NotFoundHttpException
        ('The requested page does not exist.');
    }
}
```

It's really very simple. On the index method, we call a new instance of FaqSearch model, which allows us to set an instance of \$searchModel->frontendProvider() to \$provider, which we can pass

to the view. So now the view will have access to the ArrayDataProvider that we made, with the sort by ascending value on faq_weight.

In the view, to display this, we need the following in frontend/views/faq/index.php. Don't forget to create the faq folder in frontend/views.

Gist:

Frontend Faq Index

From book:

```
<?php
use yii\helpers\Html;
use yii\helpers\ArrayHelper;
use yii\helpers\Url;

$this->title = 'FAQs';
$this->params['breadcrumbs'][] = $this->title;

?>

<div class="site-about">
    <h1><?= Html::encode($this->title) ?></h1>
    </BR>
<div class="panel panel-default">
    <div class="panel-heading">
        <h3 class="panel-title">
            Questions
        </h3>
    </div>

<?php
$data = $provider->getModels();
$questions = ArrayHelper::map($data, 'faq_question', 'id');
foreach ($questions as $question => $id){

    $url = Url::to(['faq/view', 'id'=>$id]);
    $options = [];
    echo '<div class="panel-body">' .
Html::a($question, $url, $options) . '</div>';
}

}
```

```
?>  
</div>  
</div>
```

So now we're setting `$provider->getModels()` to `$data`. If you look at the guide, this is the method they use in their example, so that is exactly what I used. Then I created key value pairs out of question and Id by using the `ArrayHelper::map` method. If you try this with faq records that have different `faq_weight` settings, you will see that it respects the ordering by `faq_weight`. So you can absolutely hand control over the order to your client by providing backend UI that lets them set the `faq_weight`.

Next I'm using a foreach loop to do something with each question and it's corresponding id. In this case, the something is first creating a url that has `$id` as a parameter. Then I'm echoing out each line, using the `Html::a` method to create a link that uses `$question` as the link text and `$url` as the url to the view.

I threw in a little bootstrap styling on this page for fun. You can play with raw bootstrap at:

[Layoutit.com](#)

They feature drag and drop design, then you can copy your code, so it's perfect for designing little snippets like the above. The resulting index page should look like this:

The screenshot shows a web page titled "FAQs". At the top, there is a breadcrumb navigation bar with "Home / FAQs". Below the title, there is a section header "Questions". Underneath this, there is a list of five questions, each followed by a blue link: "Can I learn it?", "How much time does it take?", "Can I make money programming?", and "Is this site helpful?".

Getting back to the faq list, the links we created in our foreach loop will send a request to the view action on the frontend faq controller:

```
public function actionView($id)
{
    return $this->render('view', [
        'model' => $this->findModel($id),
    ]);
}
```

It takes in the \$id from the get variable, which was set in the url, and calls an instance of the view, using the correct instance of the model by using the last method on the controller:

```
protected function findModel($id)
{
    if (($model = Faq::findOne($id)) !== null) {
        return $model;
    } else {
        throw new NotFoundHttpException
            ('The requested page does not exist.');
    }
}
```

You can see how clean and simple all that is.

So for this to work, in frontend/views/faq/view.php, we need the following.

Gist:

Frontend Faq View

From book:

```
<?php
use yii\helpers\Html;

$this->title = 'FAQ: ' . $model->faq_question;

$this->params['breadcrumbs'][] = ['label' => 'FAQ', 'url' => ['index']];
$this->params['breadcrumbs'][] = $this->title;
?>

</br>
<div class="panel panel-default">
    <div class="panel-heading">
        <h3 class="panel-title">
```

```

<h1>    <?= $model->faq_question;?> </h1>

</h3>
</div>

<?= '<div class="panel-body"><h3>'.
$model->faq_answer . '</h3></div>';?>

</div>

```

The relevant parts here are \$model->faq_question and \$model->faq_answer, which are made available to us because we used findModel to find that instance of the Faq model and sent it to the view from the controller.

One last task, let's just pop it into our top nav in frontend/views/layouts/main.php. Add the new link to the first \$menuItems array:

```

$menuItems = [
    ['label' => 'Home', 'url' => ['/site/index']],
    ['label' => 'About', 'url' => ['/site/about']],
    ['label' => 'FAQs', 'url' => ['/faq/index']],
    ['label' => 'Contact', 'url' => ['/site/contact']],
];

```

That's a one line change, so no Gist for that. A question came up from a reader about whether or not to include the / before the controller. In the example that Yii 2 shows in the Guide, there is no forward slash. The code that comes out of the box with the template, includes the forward slash. Both ways work. So it's up to you which convention you want to follow.

And that's pretty much it. I'm sure you can make it prettier, since I'm not much of a UI designer, but you get the main point here, which is how to move the FAQ data around and keep control over the order, which your clients will love.

I purposely avoided using the ListView and GridView widgets in this example to show you how you can use things like ArrayDataProvider without them. You can customize your UI as you wish, you are not limited to the widgets.

Hopefully when you have a meeting with your client and you suggest to them how you can give them control over the order of their FAQs, they will be impressed, which is the desired result. We want to please those really difficult clients because that is how we get paid.

Although we didn't use FaqCategory on the presentation side, we could easily do so by making similar modifications to the FaqCategorySearch model and implementing from there. That's a benefit of putting some forethought into the data structure. So once again, if the client asks for that, you are one step ahead.

Test Controller

Sometimes in development, you want to isolate a block of code and play with it, without cluttering up your other code. Now version control should always be there if you made a huge mistake, but you really don't want to be relying on that to step backwards unless you absolutely have to.

So what we'll do is create a controller named test. We start by navigating to Gii, and clicking on Controller Generator. We don't have to worry about creating a model first, since we don't have one, our test controller will be used to play around with different bits of many models, and we will pull in the models via use statements as we need them.

From our controller generator on Gii, pop in test into the first input field to name the controller. We will create just one action for now, Index. Then check to see if the namespace is set to where you want it to go.

I've created a test controller for both frontend and backend. That's not really necessary, but it is convenient. It just helps me when I'm thinking of things to intuitively associate a backend test controller with things I want to try in the backend. Also, sometimes, I leave code in place in the test controller while I'm thinking about whether or not I want to use it.

Ok, so when you hit preview on Gii, it will show two files, the controller and the view file. Once we click generate, it will also create the directory to hold the view file. So let's do that now.

If all went well, you should have a TestController.php file in frontend/controllers and a frontend/views/test/index.php file.

Ok, so let's look at the TestController.php.

Gist:

[Test controller](#)

From book:

```
<?php

namespace frontend\controllers;

class TestController extends \yii\web\Controller
{
    public function actionIndex()
    {
        return $this->render('index');
    }
}
```

That's it. You can see it consists of very little except the actionIndex method, and all that is doing is rendering a view named index. Let's look at the view.

Gist:

Test Index View

From book:

```
<?php
/* @var $this yii\web\View */
?>
<h1>test/index</h1>

<p>
    You may change the content of this page by modifying
    the file <code><?= __FILE__; ?></code>.
</p>
```

Again, almost nothing here, so very easy to play with. Try `yii2build.com/index.php?r=test` and you should see the view.

Ok, so think of it as a blank canvass that you can use to try different snippets of code or to figure out how something works or what is contained in `var_dump()`.

You should probably create a test controller for both frontend and backend. I have a feeling this is going to come in very handy, very soon.

Note, you may have to specify the view path if it doesn't generate the view file, which is what happened when I did it:

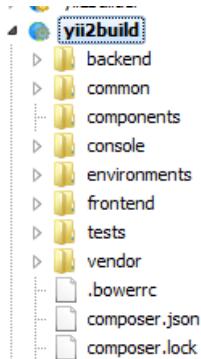
```
/var/www/yii2build/backend/views/test
```

Please keep in mind your path may differ if you path to the site folder is different. If you get your files in an unwanted location, either delete and try again or move the existing files. Just make sure you get the namespaces correct.

Components

We're going to setup a components folder for the purpose of holding our custom widget, which we have not yet made. In the Yii 2 guide, they use an example where they show the widget residing in a component folder, so we will follow as close to that as possible.

So we start by creating a new folder in our project directory named components. Your directory tree should look like this:



Components Folder Added

We will be testing our setup by creating a component, which is different than a widget, but will reside in the same folder. Widgets are used in views and display something to the user. Components typically work behind the scenes. You will get a better idea of this by example.

So let's create `MyComponent.php` inside the components folder.

Gist:

[MyComponent.php](#)

From book:

```
<?php

namespace components;

use Yii;
use yii\base\Component;
use yii\base\InvalidConfigException;

class MyComponent extends Component
{
    public function blastOff()
    {
        echo "Houston, we have ignition...";
    }
}
```

To create a component, we extend off of `Component`. Don't forget to include the necessary `use` statements.

Then we simply create the method we wish to call, in our case we are creating a `blastOff` method. I'm just using a simple `echo` statement for demonstration purposes.

Blastoff, for our international readers who might not understand that phrase in English, is what happens when a rocket launches. Houston we have ignition is what the ground controllers say to mission control when the rocket engine ignites. It's turned into a joke in English meaning things are underway, something has started, etc., usually something dramatic.

Anyway, back to the component. Now if you were to try to use this component from inside one of your test controllers, it would return an error because the autoloader can't see the file yet. We have to modify our common/config/bootstrap.php file to give our application visibility on the folder.

Change it to the following:

Gist:

[Common/Config/Bootstrap](#)

From book:

```
<?php  
Yii::setAlias('common', dirname(__DIR__));  
Yii::setAlias('frontend', dirname(dirname(__DIR__)) . '/frontend');  
Yii::setAlias('backend', dirname(dirname(__DIR__)) . '/backend');  
Yii::setAlias('components', dirname(dirname(__DIR__)) . '/components');  
Yii::setAlias('console', dirname(dirname(__DIR__)) . '/console');
```

This is using the setAlias method to create an alias to our folder. You can read about the setAlias method in the guide, they explain it better than I can:

[Yii 2 setAlias from Guide](#)

Now we need just one more step, we have to modify our components array in common/config/main.php to reference MyComponent.php. I'm going to give you the whole file because one little thing out of place and nothing works.

Please note, you will have enter your facebook app id and secret again because I'm using generic placeholders for those values.

Gist:

[Config/Main](#)

From book:

```
<?php
return [
    'vendorPath' => dirname(dirname(__DIR__)) . '/vendor',
    'extensions' => require(__DIR__ . '/../../vendor/yiisoft/extensions.php'),
    'modules' => [
        'social' => [
            // the module class
            'class' => 'kartik\social\Module',

            // the global settings for the disqus widget
            'disqus' => [
                'settings' => ['shortname' => 'DISQUS_SHORTNAME'] // default settings
            ],

            // the global settings for the facebook plugins widget
            'facebook' => [
                'appId' => 'your id',
                'secret' => 'your secret',
            ],

            // the global settings for the google plugins widget
            'google' => [
                'clientId' => 'GOOGLE_API_CLIENT_ID',
                'pageId' => 'GOOGLE_PLUS_PAGE_ID',
                'profileId' => 'GOOGLE_PLUS_PROFILE_ID',
            ],

            // the global settings for the google analytic plugin widget
            'googleAnalytics' => [
                'id' => 'TRACKING_ID',
                'domain' => 'TRACKING_DOMAIN',
            ],

            // the global settings for the twitter plugin widget
            'twitter' => [
                'screenName' => 'TWITTER_SCREEN_NAME'
            ],
        ],
        // your other modules
    ],
    'components' => [

```

```
'cache' => [
    'class' => 'yii\caching\FileCache',
],
'mycomponent' => [
    'class' => 'components\MyComponent',
],
];
];
```

Next we need to make sure we can blastOff as planned, so modify the index action on a test controller. Use frontend to follow my example exactly, to:

Gist:

[Test Controller blastOff](#)

From book:

```
<?php

namespace frontend\controllers;

use yii;

class TestController extends \yii\web\Controller
{
    public function actionIndex()
    {
        Yii::$app->mycomponent->blastOff();
    }
}
```

Don't forget the use statement. To test this simply point your browser to:

```
http://yii2build.com/index.php?r=test
```

If all went well, you were able to blastOff. And now you have a working components folder.

Creating a Custom Widget

Returning to our theme of the client who is demanding and difficult to satisfy, we can think about circumstances where the client needs flexibility in display of a partial view. In fact we can return to our FAQs and imagine the client wants to experiment with placement, and perhaps limit a special view of FAQs to featured, ordered by weight, so they can control the order.

In some ways, this is extremely trivial to us, but to the client, it helps them finesse their marketing message, so they will be as excited about this feature as any of the other more complicated features of the site.

The good news is that because we were planning for the client to expand their requirements, we start with a firm foundation already in place. For example, our data structure supports a featured column, which we've called `faq_is_featured`, and we already have been sorting according to `faq_weight`.

So it will be easy for us to make a new method in `FaqSearch` that will return the FAQs that have the `faq_is_featured` column set to 1, and order by weight. Let's go ahead and add the following method to `FaqSearch`.

Gist:

[FaqSearch Featured Provider](#)

From book:

```
public function featuredProvider()
{
    $query = new Query;
    $featuredProvider = new ArrayDataProvider([
        'allModels' => $query->from('faq')->
            where(['faq_is_featured' =>1])->all(),
        'sort' => [
            'defaultOrder' => [
                'faq_weight' => SORT_ASC,
            ],
            'attributes' => ['faq_question', 'faq_answer', 'faq_weight'],
        ],
        'pagination' => [
            'pageSize' => 10,
        ],
    ]);
}
```

```
    return $featuredProvider;

}
```

Now we need a way to test this, so let's use our backend test controller.

Modify backend/controllers/TestController.php.

Gist:

Backend Test Controller

From book:

```
<?php

namespace backend\controllers;

use Yii;
use backend\models\Faq;
use backend\models\search\FaqSearch;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;

class TestController extends \yii\web\Controller
{

    public function behaviors()
    {
        return [
            'verbs' => [
                'class' => VerbFilter::className(),
                'actions' => [
                    'delete' => ['post'],
                ],
            ],
        ];
    }

    /**
     * Lists all Faq models.
     * @return mixed
     */
    public function actionIndex()
```

```

{
    $searchModel = new FaqSearch();

    $provider = $searchModel->featuredProvider();

    return $this->render('index', [
        'searchModel' => $searchModel,
        'provider' => $provider,
    ]);
}

}

```

Really the only change here is which method we call from \$searchModel, in this case featuredProvider. We're not doing anything with behaviors because that is not what we are testing for.

One more step to test our method, and that is to setup backend/views/test/index.php.

Gist:

Backend Test Index

From book:

```

<?php
use yii\helpers\Html;
use yii\helpers\ArrayHelper;
use yii\helpers\Url;

$this->title = 'FAQs';
$this->params['breadcrumbs'][] = $this->title;

?>

<div class="site-about">
    <h1><?= Html::encode($this->title) ?></h1>
    </BR>
    <div class="panel panel-default">
        <div class="panel-heading">
            <h3 class="panel-title">
                Questions

```

```
</h3>
</div>

<?php
$data = $provider->getModels();
$questions = Arrayhelper::map($data, 'faq_question', 'id');
foreach ($questions as $question => $id){

    $url = Url::to(['faq/view', 'id'=>$id]);
    $options = [];
    echo '<div class="panel-body">' .
Html::a($question, $url, $options) . '</div>';

}

?>

</div>
</div>
```

Now that should look familiar, all we did was copy the frontend/views/faq/index.php file into backend/views/test/index.php.

So just make sure you have 5 featured FAQs created, with different weight for each, and a couple of FAQs that are not featured, so you can verify your results. Pay careful attention to the order to make sure it's right. It should work as planned.

I think this is a great opportunity to explore some alternatives on how we can do this. First, let's try using SqlDataProvider, which let's us use a sql statement. For those of us more used to working with SQL, that's actually easier to work with.

Gist:

[FaqSearch SqlDataProvider](#)

From book:

```
public function featuredProvider()
{
    $count = Yii::$app->db->createCommand('
        SELECT COUNT(*) FROM `faq` WHERE `faq_is_featured` =
:faq_is_featured', [':faq_is_featured' => 1])->queryScalar();

    $featuredProvider = new SqlDataProvider([
        'sql' => 'SELECT * FROM `faq` WHERE `faq_is_featured` =
:faq_is_featured ORDER BY `faq_weight` ASC',
        'params' => [':faq_is_featured' => 1],
        'totalCount' => $count,
        'sort' => [
            'attributes' => [
                'id',
                'faq_question'
            ],
            [
                'pagination' => [
                    'pageSize' => 10,
                ],
            ],
        ],
    ]);

    return $featuredProvider;
}
```

Let's not forget to add the use statement:

```
use yii\data\SqlDataProvider;
```

Here we are using SQL to tell the query what order to return the results in.

We start with a count query to use for pagination. Then we have the main query, where we just explicitly tell it in SQL that we want all results from faq where faq_is_featured = 1 and to order by faq_weight ascending.

This works perfectly, but because we are not using one of Yii 2's widgets, the pagination does not work.

This exposes a flaw in our approach because we might have enough results that would warrant use of pagination, and it's a shame not to take advantage of Yii 2's pagination object and LinkPager widget because they are very useful.

So looking over the docs, I found a more concise way to do all this and still get everything we want. We are going to ignore the search model method and go straight to our controller.

Gist:

Backend Test Controller FAQ

From book:

```
<?php

namespace backend\controllers;

use Yii;
use backend\models\Faq;
use backend\models\search\FaqSearch;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;
use yii\data\Pagination;

class TestController extends \yii\web\Controller
{

    public function behaviors()
    {
        return [
            'verbs' => [
                'class' => VerbFilter::className(),
                'actions' => [
                    'delete' => ['post'],
                ],
            ],
        ];
    }

    /**
     * Lists all Faq models.
     * @return mixed
     */
    public function actionIndex()
    {
        $query = Faq::find()->where(['faq_is_featured' => 1]);
        $query->orderBy(['faq_weight' => SORT_ASC]);
    }
}
```

```

    $countQuery = clone $query;
    $pages = new Pagination(['defaultPageSize' => 3,
'totalCount' => $countQuery->count()]);
$models = $query->offset($pages->offset)
    ->limit($pages->limit)
    ->all();

return $this->render('index', [
    'models' => $models,
    'pages' => $pages,
]);
}

}

```

Now that is really leveraging the power of Yii 2. Let's go through it carefully.

```
$query = Faq::find()->where(['faq_is_featured' => 1]);
```

We start by setting up our query, super easy syntax to understand at this point. Next we tell it how we want to order the query:

```
$query->orderBy(['faq_weight' => SORT_ASC]);
```

Yii 2's syntax is so simple, we don't need to explain it. Next we make a copy of the query object so we can use it to return a count, which we will feed into our pagination object.

```
$countQuery = clone $query;
```

Notice the use of clone, super efficient and easy to use. We used clone because we need a separate copy of our query to return a count.

Next we create the pagination object:

```
$pages = new Pagination(['defaultPageSize' => 3,
'totalCount' => $countQuery->count()]);
```

Notice we are handing the config for the pagination right in through the constructor. I set the page size to 3 , so I could easily test it. And of course we use \$countQuery to return the number of results, so our pagination can do its calculations.

Now we can set up the object \$models:

```
$models = $query->offset($pages->offset)
    ->limit($pages->limit)
    ->all();
```

You can see we have handed in the \$pages object into the offset method of query, which is how it sets the record limit for the page.

Then comes the render method:

```
return $this->render('index', [
    'models' => $models,
    'pages' => $pages,
]);
```

And we are passing along our two objects, \$models and \$pages into the view.

Now we need to set up the Index view.

Gist:

Test Index FAQ

From book:

```
<?php
use yii\helpers\Html;
use yii\helpers\Url;
use yii\widgets\LinkPager;

$this->title = 'FAQs';
$this->params['breadcrumbs'][] = $this->title;

?>

<div class="site-about">
    <h1><?= Html::encode($this->title) ?></h1>
    </BR>
    <div class="panel panel-default">
        <div class="panel-heading">
            <h3 class="panel-title">
                Questions
            </h3>
        </div>
```

```
<?php

foreach ($models as $model){

    $url = Url::to(['faq/view', 'id'=>$model->id]);
    $options = [];
    echo '<div class="panel-body">' .
Html::a($model->faq_question, $url, $options) . '</div>';

}

echo LinkPager::widget([
    'pagination' => $pages,
]);

?>

</div>
</div>
```

So you can see how much more concise this is. No need to setup an array, since we are working with the object. \$models goes into our foreach loop and \$pages goes into our LinkPager widget. And just like that, you have pagination that works.

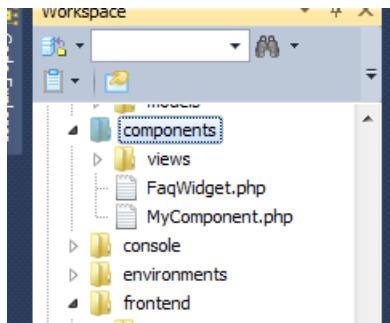
Now if you click on one of the Faq links, you'll see it goes to a backend view page. That's not what we want, but it's ok for now. Once we have created our widget, we will call it from within the frontend and the links will go to the frontend view pages.

It's clear that we got a lot of use out of our test controller. It allowed us to flush out what we were looking for in the logic of our widget. Now we know exactly how we are going to format the query.

Ok, so now we're ready to tackle the widget directly. Let's start by creating a folder inside of components named views. This will hold the view for our widget.

Then let's create a blank file for the widget named FaqWidget.php and put that in the components folder. Make sure to follow these instructions carefully, nothing will work otherwise.

So your directory tree should look like the following:



Directory Tree

Now we need to take the next step, which is to include FaqWidget in our common/config/main.php file. I'm only going to show the components array. You should have the complete file from the components section, so we will simply be adding to it here.

Gist:

Components Config

From book:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\FileCache',
    ],
    'mycomponent' => [
        'class' => 'components\MyComponent',
    ],
    'faqwidget' => [
        'class' => 'components\FaqWidget',
    ],
],
```

You can see we are supplying it with the classname lowercase, then the namespace, which identifies where the class resides. This allows the autoloader to map it correctly so we can use the class via use statement.

Before we can test anything, we need to create the FaqWidget class and the corresponding view.

Let's start with FaqWidget.php.

Gist:

FaqWidget.php

From book:

```
<?php

namespace components;

use yii\base\Widget;
use yii\helpers\Html;
use Yii;
use backend\models\Faq;
use backend\models\search\FaqSearch;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;
use yii\data\Pagination;

class FaqWidget extends Widget
{
    public $models;
    public $pages;

    public function init()
    {
        parent::init();

        $query = Faq::find()->where(['faq_is_featured' => 1]);
        $query->orderBy(['faq_weight' => SORT_ASC]);
        $countQuery = clone $query;
        $this->pages = new Pagination(['defaultPageSize' => 2,
'totalCount' => $countQuery->count()]);
        $this->models = $query->offset($this->pages->offset)
            ->limit($this->pages->limit)
            ->all();
    }

    public function run()
    {
        return $this->render('faq', [
            'models' => $this->models,
            'pages' => $this->pages,
        ]);
    }
}
```

```
}
```

So you can see all the necessary use statements:

```
namespace components;

use yii\base\Widget;
use yii\helpers\Html;
use Yii;
use backend\models\Faq;
use backend\models\search\FaqSearch;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;
use yii\data\Pagination;
```

Obviously we are using components as the namespace and pulling in everything we need via use statements.

Next we declare the class and two public properties:

```
class FaqWidget extends Widget
{
    public $models;
    public $pages;
```

I'll come back to why we need to declare the properties in a minute.

The rest of the class is comprised of just two methods, an init and a run method. The init method:

```
public function init()
{
    parent::init();

    $query = Faq::find()->where(['faq_is_featured' => 1]);
    $query->orderBy(['faq_weight' => SORT_ASC]);
    $countQuery = clone $query;
    $this->pages = new Pagination(['defaultPageSize' => 2,
        'totalCount' => $countQuery->count()]);
    $this->models = $query->offset($this->pages->offset)
        ->limit($this->pages->limit)
        ->all();
}
```

We start the method by calling parent::init. So what you need to know about init is that it acts like a constructor, it's going to run when the class is called. That means all the necessary logic will be performed and you can see the logic is simply what we built in our test controller, with one difference. We had to assign class properties \$pages and \$model using \$this->models and \$this->pages. That way we can access the values from our other method, run():

```
public function run()
{
    return $this->render('faq', [
        'models' => $this->models,
        'pages' => $this->pages,
    ]);
}
```

In this case, run is returning a render method to call our widget view faq. So just to recap, the init does the query, assigns the values to our class properties, which are then passed into our run method via \$this.

I found this to be a very intuitive flow. It's like a mini-controller calling a view.

So now let's get our view going. Inside of the component/views folder, create the following file, faq.php.

Gist:

[Faq View](#)

From book:

```
<?php
use yii\helpers\Html;
use yii\helpers\Url;
use yii\widgets\LinkPager;

?>

<div class="site-about">

    </BR>
    <div class="panel panel-default">
        <div class="panel-heading">
            <h3 class="panel-title">
                Featured Questions
            </h3>
        </div>
```

```
<?php

foreach ($models as $model){

    $url = Url::to(['faq/view', 'id'=>$model->id]);
    $options = [];
    echo '<div class="panel-body">' .
Html::a($model->faq_question, $url, $options) . '</div>';

}

echo LinkPager::widget([
    'pagination' => $pages,
]);


?>

</div>
```

We just copied from our Test index view page and chopped out the stuff we didn't need like the title and the h1.

Ok, so we're ready to make this work. Let's go to frontend/views/site/index.php and put this between the last and 2nd to last div tags like so:

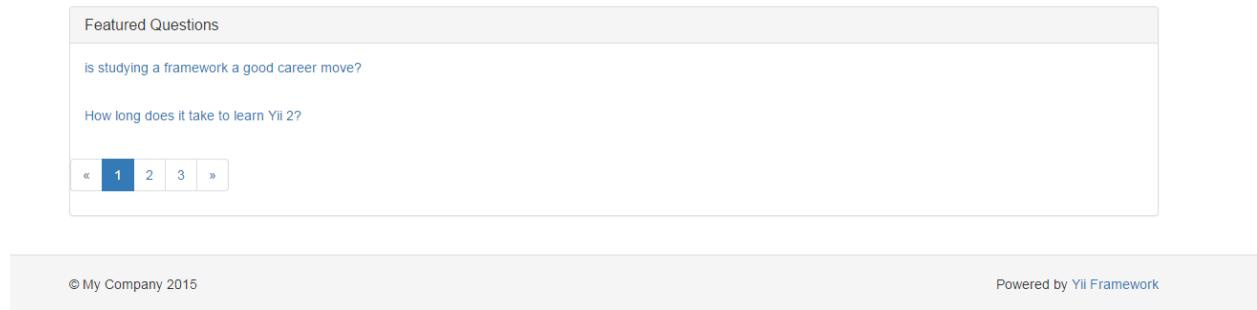
```
</div>

<?= FaqWidget::widget() ?>
</div>
```

No need to supply a gist for a single short line. How simple is that? Oh, but don't forget the use statement at the top of the file:

```
use components\FaqWidget;
```

And now if you save and refresh your index page, you should see the following at the bottom of the page:



So now you have a paginated widget for featured faq that you can embed anywhere on the site, which also respects the order according to weight, and only includes featured questions. It's also styled in Bootstrap, which means it resizes with device size. How cool is that?

It's pretty easy to add an optional parameter to the widget, so I'm going to cover that. Let's make the pagination page size a parameter that we can hand in.

We start in our FaqWidget class by declaring a new property:

```
public $pageSize;
```

This will hold the value of the page size when we hand it in or get set by our if statement, which we will add now:

```
parent::init();

if ($this->pageSize === null) {
    $this->pageSize = 2;
}
```

So if it's not set, set it to 2.

Then we just need to use the variable instead of the hard-coded number:

```
$this->pages = new Pagination(['defaultPageSize' => $this->pageSize,
```

Go ahead and save your changes.

Next we just modify the call to the widget like so:

```
<?= FaqWidget::widget(['pageSize' => 3]) ?>
```

If you set it to 5 and you only have 5 featured records, LinkPager will not show you the pagination count, so this is a very smart widget to play with.

Here is the entire FaqWidget.php file for reference.

Gist:

[FaqWidget](#)

From book:

```
<?php
namespace components;

use yii\base\Widget;
use yii\helpers\Html;
use Yii;
use backend\models\Faq;
use backend\models\search\FaqSearch;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;
use yii\data\Pagination;

class FaqWidget extends Widget
{
    public $models;
    public $pages;
    public $pageSize;

    public function init()
    {
        parent::init();

        if ($this->pageSize === null) {
            $this->pageSize = 2;
        }

        $query = Faq::find()->where(['faq_is_featured' => 1]);
        $query->orderBy(['faq_weight' => SORT_ASC]);
        $countQuery = clone $query;
        $this->pages = new Pagination(['defaultPageSize' => $this->pageSize,
'totalCount' => $countQuery->count()]);
        $this->models = $query->offset($this->pages->offset)
->limit($this->pages->limit)
->all();
    }
}
```

```
}

public function run()
{
    return $this->render('faq', [
        'models' => $this->models,
        'pages' => $this->pages,
    ]);
}
}
```

So I was really happy with that, but then I thought about our difficult-to-please client again. If they like the Featured Faq widget a lot, they might want a version of it for all FAQs. In fact they might change their mind several times about that. So what should we do? How can we make this easy on ourselves?

Well, instead of taking in a single parameter, why don't we take in an array of settings? Then we could test for something like 'featuredOnly' set to true or false, and depending on the answer, give them either just the featured FAQs or all of them.

If we look at it from the widget call and work our way backwards, it will be easy to see how this will work:

```
<?= FaqWidget::widget(['settings' => ['pageSize' => 3,
                                             'featuredOnly' => true]]) ?>
```

So now we're handing in an array named settings, with two key value pairs. As long as we have a class property to hold the value of settings we are good to go. We can then access it via keyword \$this to test the values and perform the appropriate logic based on the results.

Here is the updated FaqWidget.php file.

Gist:

[FaqWidget With Settings](#)

From book:

```
<?php

namespace components;

use yii\base\Widget;
use yii\helpers\Html;
use Yii;
use backend\models\Faq;
use backend\models\search\FaqSearch;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;
use yii\data\Pagination;

class FaqWidget extends Widget
{
    public $models;
    public $pages;
    public $settings = [];

    public function init()
    {
        parent::init();

        if (!isset($this->settings['pageSize'])) {
            $this->settings['pageSize'] = 2;
        }

        if ($this->settings['featuredOnly'] === true) {
            $query = Faq::find()->where(['faq_is_featured' => 1]);
        } else {

            $query = Faq::find();
        }
        $query->orderBy(['faq_weight' => SORT_ASC]);
        $countQuery = clone $query;
        $this->pages = new Pagination(['defaultPageSize' =>
$this->settings['pageSize'], 'totalCount' => $countQuery->count()]);
        $this->models = $query->offset($this->pages->offset)
->limit($this->pages->limit)
->all();
    }
}
```

```

    }

    public function run()
    {
        return $this->render('faq', [
            'models' => $this->models,
            'pages' => $this->pages,
            'settings' => $this->settings,
        ]);
    }
}

```

Ok, let's step through this. You can see we now have a settings array that is initialized to an empty array.

```

class FaqWidget extends Widget
{
    public $models;
    public $pages;
    public $settings = [];
}

```

The widget method of the Widget class will take in the values that we pass into the signature of the widget method and assign them to this array because it looks for a property with the same name. This is clever stuff and very powerful. It opens up a whole range of possibilities for you when you are making widgets.

In our case, we're only handing in two key value pairs, so this is still a relatively simple implementation. You can see how we then use the values in the array to test against.

```

if (!isset($this->settings['pageSize'])) {
    $this->settings['pageSize'] = 2;
}

if ($this->settings['featuredOnly'] === true) {
    $query = Faq::find()->where(['faq_is_featured' => 1]);
} else {

    $query = Faq::find();
}

```

So now we test against the array keys to perform our logic on the values. In the case of the second if statement, we are looking for a key ‘featuredOnly’ and if its value is set to true, then query only where featured is equal to one, otherwise get all records.

Then we just modify our render method accordingly:

```
public function run()
{
    return $this->render('faq', [
        'models' => $this->models,
        'pages' => $this->pages,
        'settings' => $this->settings,
    ]);
}
```

You’ll notice we are sending the \$settings array into the view. The reason for this is that we want to test against the ‘featuredOnly’ key, and depending on the results, show a different heading.

Gist:

[Faq View](#)

From book:

```
<?php
use yii\helpers\Html;
use yii\helpers\Url;
use yii\widgets\LinkPager;

?>

<div class="site-about">

    </BR>
    <div class="panel panel-default">
        <div class="panel-heading">
            <h3 class="panel-title">
                <?php
                    if ($settings['featuredOnly'] == true){
                        echo 'Featured Questions';
                    } else {
                        echo 'FAQs';
                    }
                ?>
            </h3>
        </div>
    </div>
</div>
```

```
</h3>
</div>

<?php

foreach ($models as $model){

    $url = Url::to(['faq/view', 'id'=>$model->id]);
    $options = [];
    echo '<div class="panel-body">'.
        Html::a($model->faq_question, $url, $options) . '</div>';

}

echo LinkPager::widget([
    'pagination' => $pages,
]);

?>

</div>
</div>
```

We test to see if we are only using ‘featuredOnly’, and if so, show the appropriate heading. Alternatively, you could use ternary syntax and that would be fine too.

Then to wrap this up, we just make the widget call, which, even though it got a little bit longer, is still a single line of code, if you are not avoiding word wrap of course:

```
<?= FaqWidget::widget(['settings' =>
    ['pageSize' => 3, 'featuredOnly' => true]]) ?>
```

And that’s it. You now have a configurable FAQ widget that you can use in any view, with a single line call. Just remember to include your use statements if you do place it elsewhere on the site.

CDN

Ok, so many of you may already know that using a CDN, a content delivery network, to pull in css, js, and jquery assets can dramatically speed up performance of your site.

The reason for this is that the CDN versions are typically already cached in the user's browser, so they don't have to pull down the library every time they visit a site. And because these assets are so common, chances are your site will not be the first they visit that utilizes these assets, so most likely they are already cached. This can make a huge difference in the speed of the page loading.

There's a really simple way to do this in common/config/main.php. For convenience, I'm going to give you the entire file, but don't forget I have generic placeholders for Facebook app id and secret.

Gist:

Main with CDN

From book:

```
<?php
return [
    'vendorPath' => dirname(dirname(__DIR__)) . '/vendor',
    'extensions' => require(__DIR__ . '/../../vendor/yiisoft/extensions.php'),
    'modules' => [
        'social' => [
            // the module class
            'class' => 'kartik\social\Module',

            // the global settings for the disqus widget
            'disqus' => [
                'settings' => ['shortname' => 'DISQUS_SHORTNAME']
            ],

            // the global settings for the facebook plugins widget
            'facebook' => [
                'appId' => 'your app id',
                'secret' => 'your secret',
            ],

            // the global settings for the google plugins widget
            'google' => [
                'clientId' => 'GOOGLE_API_CLIENT_ID',
                'pageId' => 'GOOGLE_PLUS_PAGE_ID',
                'profileId' => 'GOOGLE_PLUS_PROFILE_ID',
            ],

            // the global settings for the google analytic plugin widget
            'googleAnalytics' => [
                'id' => 'TRACKING_ID',
                'domain' => 'TRACKING_DOMAIN',
            ],
        ],
    ],
];
```

```
],  
  
    // the global settings for the twitter plugin widget  
    'twitter' => [  
        'screenName' => 'TWITTER_SCREEN_NAME'  
    ],  
],  
// your other modules  
],  
  
'components' => [  
  
    'assetManager' => [  
        'bundles' => [  
            // use bootstrap css from CDN  
            'yii\bootstrap\BootstrapAsset' => [  
                'sourcePath' => null, // do not use file from our server  
                'css' => [  
                    'https://maxcdn.bootstrapcdn.com/bootstrap/3.3.0/css/bootstrap.min.css'  
                ],  
                // use fontawesome css from CDN  
                'frontend\assets\FontAwesomeAsset' => [  
                    'sourcePath' => null, // do not use file from our server  
                    'css' => [  
                        'https://maxcdn.bootstrapcdn.com/font-awesome/4.2.0/css/font-awesome.min.css'  
                    ],  
                    // use fontawesome css from CDN  
                    'backend\assets\FontAwesomeAsset' => [  
                        'sourcePath' => null, // do not use file from our server  
                        'css' => [  
                            'https://maxcdn.bootstrapcdn.com/font-awesome/4.2.0/css/font-awesome.min.css'  
                        ],  
                        // use bootstrap js from CDN  
                        'yii\bootstrap\BootstrapPluginAsset' => [  
                            'sourcePath' => null, // do not use file from our server  
                            'js' =>  
                                'https://maxcdn.bootstrapcdn.com/bootstrap/3.3.0/js/bootstrap.min.js'  
                        ],  
                        // use jquery from CDN  
                        'yii\web\JqueryAsset' => [  
                            'sourcePath' => null, // do not publish the bundle  
                            'js' => [  
                                'https://code.jquery.com/jquery-2.1.4.min.js'  
                            ]  
                        ]  
                    ]  
                ]  
            ]  
        ]  
    ]  
]
```

```
'https://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js',
        ],
    ],
],
],
'cache' => [
    'class' => 'yii\caching\FileCache',
],
'mycomponent' => [
    'class' => 'components\MyComponent',
],
'faqwidget' => [
    'class' => 'components\FaqWidget',
],
];
];
```

By setting the source path to null, we are unsetting the path, then setting it to the CDN. The only other notable thing here is that we had to do font-awesome for both frontend and backend, since we created assets for font-awesome in both places.

Once these changes are in place, I think you'll find your application moves a lot quicker, which is always a plus.

Summary

Congratulations, you've completed the first bonus chapter. We've added some cool features to the template, and in the process of doing so, have learned much more about Yii 2.

That's about as much of a conclusion as I can draw right now because this isn't really the end of the book. I plan on adding new material on a go forward basis. Yii 2 is the primary PHP framework that I use personally, and I think so highly of it, I want to continue to share it with you.

I've come to understand that writing about a framework is more of a journey than a destination. For example, Yii 2 will be doing periodic version releases and things can change.

I'll do my best to stay on top of the changes and revise the book accordingly. For basic function updates from Yii 2, adjustments will be made to the core book.

The bonus chapters, on the other hand, will simply be added to successively, unless a method has to change due to a versioning change in Yii 2 itself. Successive additions gives us the sense that we are building on something.

Your purchase of *Yii 2 For Beginners* entitles you to free updates for the life of the book, so there is no reason not to benefit from that. Just look for update notices by email or check the leanpub.com website for the last update.

Now I'll take a moment to thank all the readers that wrote to me with positive comments and typo notifications. It really helps me in my pursuit to take *Yii 2 For Beginners* to the highest level of quality for a technical book. With your help, we can do it.

When I get reader feedback, it encourages me to push forward. As always, comments, links, reviews and word-of-mouth recommendations are greatly appreciated. Let's share this amazing framework with as many people as we can.

So thanks again everyone, see you soon.

Chapter 13: Bonus Material Pretty Urls & Slugs

Welcome back! We're ready for more bonus material. I've decided to make the bonus chapters smaller, so that I can release the material more quickly.

In this chapter we are going to set up our application to use pretty URLs and to use slugs. By pretty URL, we mean that we are going to replace:

```
http://www.yii2build.com/index.php?r=site/contact
```

And instead see the following:

```
http://www.yii2build.com/site/contact
```

You can see that we have two improvements. First we got rid of index.php, no reason to show that. Second, we got rid of the ?r= and replaced it with a search engine friendly alternative.

In this chapter, we are also going to implement slugs. For those who don't know, a slug is a string that describes the content of the page and is embedded in the url. We will be using slugs with our Faq model, and the url will look something like:

```
http://yii2start.com/faq/11/should-i-use-a-framework
```

You can see that not only do we not need 'view', but we also have the slug added directly to the url in a nice search engine friendly format. This format increases visibility to the search engine and this is important if you or the client wishes to have the page found by google and other search engines. I've never met anyone who didn't think that was important.

Anyway, all of this is fairly easy to set up, but does require some work.

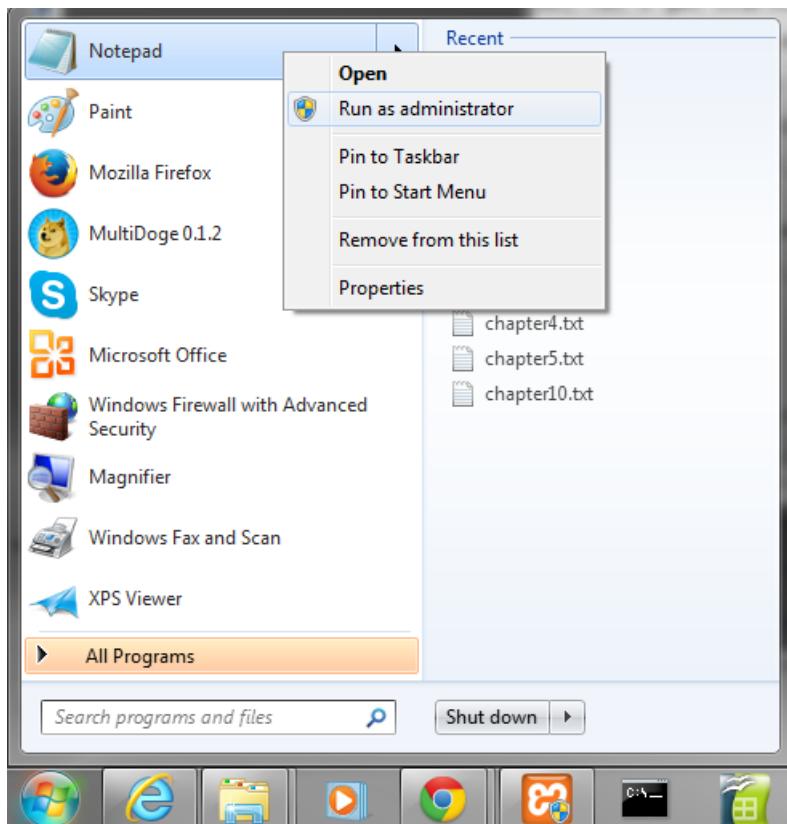
Pretty URLs

Let's start with the pretty URLs. The steps we need to take are:

- Create an htaccess file for both frontend and backend in the web folders.
- Modify our Vhost file for apache.
- Restart Apache
- Modify frontend/config/main.php and backend/config/main.php for urlManager

Apache Vhost

From a windows machine we will edit this from notepad, running in administrator mode. Find notepad from start button on taskbar. Right click and select run in administrator mode.

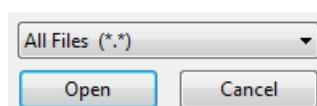


Notepad in Administrator Mode

Notepad will open. Select file open and the path to vhosts, in my case:

C:\xampp\apache\conf\extra\

Select all Files for file types:



Notepad All File Types

Then select:

httpd-vhosts.conf

Modify your host entry to the following.

Gist:

vHost Entry Apache

From book:

```
NameVirtualHost *
RewriteEngine on
<VirtualHost yii2build.com>
    DocumentRoot "C:\var\www\yii2build\frontend\web"
    ServerName localhost
    ServerAlias www.yii2build.com
</VirtualHost>
NameVirtualHost *
RewriteEngine on
<VirtualHost yii2build.com>
    DocumentRoot "C:\var\www\yii2build\backend\web"
    ServerName localhost
    ServerAlias backend.yii2build.com
</VirtualHost>
```

You can see all we did was add the instruction for RewriteEngine on. Go ahead and save and close.

Restart Apache

Make sure to restart apache so the changes take effect.

.htaccess

In the frontend/web, create a file named .htaccess. Don't forget to put the dot in front of the name.

Put the following in the file.

Gist:

.htaccess contents

From book:

```
# If a directory or a file exists, use it directly
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
# Otherwise forward it to index.php
RewriteRule . index.php
```

Now we need to go to frontend/config/main.php and add to the following to components array.

Gist:

[urlManager for components](#)

From book:

```
'urlManager' => [
    'class' => 'yii\web\UrlManager',
    // Disable index.php
    'showScriptName' => false,
    // Disable r= routes
    'enablePrettyUrl' => true,
    'rules' => [
        '<controller:\w+>/<id:\d+>' => '<controller>/view',
        '<controller:\w+>/<action:\w+>/<id:\d+>' => '<controller>/<action>',
        '<controller:\w+>/<action:\w+>' => '<controller>/<action>',
        '<controller:\w+-\w+>/<id:\d+>' => '<controller>/view',
    ],
],
```

For reference, I'm going to include a Gist to the entire file, in case you need to troubleshoot:

[frontend/config/main.php](#)

You can see that we set showScriptName to false, and this gets rid of index.php. The enablePrettyUrl gets rid of the ?r= syntax, so that is how the two combine to give us the urls we want.

You can also see that as part of the urlManager array, we have a rules section. The rules use regular expressions, such as w+ and d+ to represent words and numbers. So for example, when we pass the url a controller/action with id parameter, we account for that like so:

```
'<controller:\w+>/<action:\w+>/<id:\d+>' => '<controller>/<action>',
```

One thing to take note of is that when controllers have multiple words, such as UserType, the convention in url is user-type, so we have to account for that like so:

```
'<controller:\w+-\w+>/<id:\d+>' => '<controller>/view',
```

Remember that you have to configure backend/config/main.php the exact same way for this to work properly on the entire application. The easy way to do it is just copy the file.

Note, once implemented pretty URLs are implemented, the path to gii is now yii2build.com/gii.

So now that we got our pretty URLs enabled, we'll move on to creating slugs.

Slugs

We already mentioned what slugs are in the beginning of the chapter. Slugs are a common feature found on many websites, stackoverflow and yahoo news for example. They add context within the url and the search engines value this, but there is debate about how much value it adds.

One comment I read on stackoverflow stated that while they didn't see any increase in traffic from adding slugs, they did see a 300% click through increase.

My own view is that slugs are valuable and worth the time to implement. In certain scenarios, we can even automate this process.

Sluggable Behavior

Yii 2 has a ready-made behavior that we can use to create slugs on models. Our Faq model provides us a perfect example of how we can use slugs. There are a number of steps involved in implementation:

- Add sluggable behavior to Faq model
- Add slug column to faq table
- Delete old faq records and create new ones
- Add rule to backend/config/main.php
- Change view action on faq controller backend
- Change Create and Update actions on faq controller backend
- Change url on action column for gridview on backend faq index page
- Change url link on faq.php view file for widget
- Change frontend faq/index to use widget
- Change frontend FaqController/index to just render the page.

This sounds complicated, but once we work through it, you'll see how easy it is. Let's start by adding sluggable behavior to the Faq model:

Gist:

Sluggable Behavior

From book:

```

public function behaviors()
{
    return [
        'timestamp' => [
            'class' => 'yii\behaviors\TimestampBehavior',
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT =>
                ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE =>
                ['updated_at'],
            ],
            'value' => new Expression('NOW()'),
        ],
        'blameable' => [
            'class' => BlameableBehavior::className(),
            'createdByAttribute' => 'created_by',
            'updatedByAttribute' => 'updated_by',
        ],
        'sluggable' => [
            'class' => SluggableBehavior::className(),
            'attribute' => 'faq_question',
        ],
        // In case of attribute that contains slug has different name
        // 'slugAttribute' => 'alias',
    ],
];
}

```

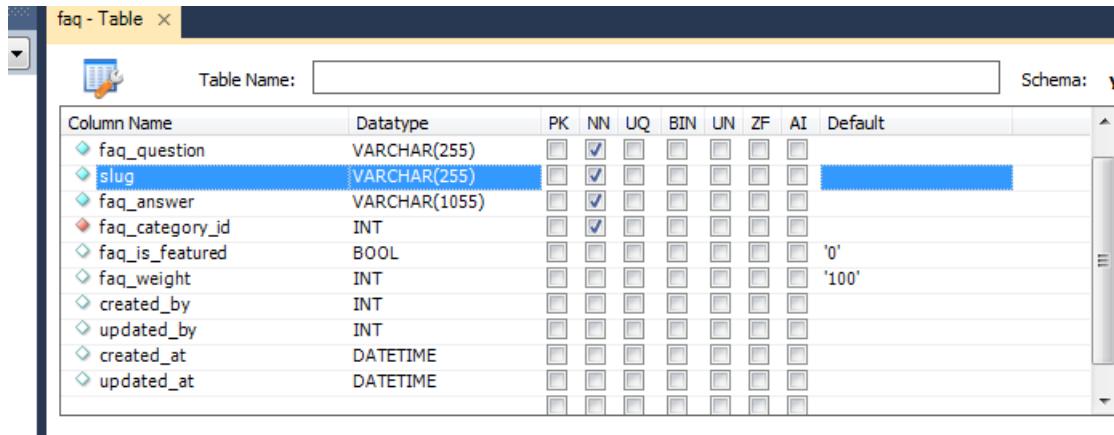
And let's not forget the use statement:

```
use yii\behaviors\SluggableBehavior;
```

So you can see in the sluggable behavior, we defined an attribute, faq_question. This is the attribute the behavior will use to create the slug. You can see we don't have to declare much. We just feed it the attribute and the behavior does the rest.

Slug Column

Of course for this to work, we need to add a column to the faq table:



Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
faq_question	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
slug	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
faq_answer	VARCHAR(1055)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
faq_category_id	INT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
faq_is_featured	BOOL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'0'
faq_weight	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'100'
created_by	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
updated_by	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
created_at	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
updated_at	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

slug column

You can see the slug column is a varchar(255), not null. Note that although the id column is not shown, it is there.

Once that's in place, you can create an faq to see if it works.

Drop old Faqs and Create New Ones

Once you successfully save a record, go ahead and delete the old FAQ's and create new ones.

Alternatively, if you have already populated your faq table with a lot of content that you want to keep, you can add the slugs manually via Php MyAdmin.

So if you do a few of these, you should end up with something that looks like this:

Show : Start row: 0 Number of rows: 30 Headers every 100 rows

Sort by key: None

+ Options

		Edit Copy Delete	id	faq_question	slug	faq_answer	faq_cat
<input type="checkbox"/>	Edit Copy Delete	7	do slugs work?	do-slugs-work	maybe		
<input type="checkbox"/>	Edit Copy Delete	8	Is programming fun?	is-programming-fun	It might be		
<input type="checkbox"/>	Edit Copy Delete	9	How much time does it take?	how-much-time-does-it-take	It depends		
<input type="checkbox"/>	Edit Copy Delete	10	How much money can we make?	how-much-money-can-we-make	A lot if you're good		
<input type="checkbox"/>	Edit Copy Delete	11	Should I use a framework?	should-i-use-a-framework	Probably, it's a good idea		
<input type="checkbox"/>	Edit Copy Delete	12	Am I going to finish?	am-i-going-to-finish	Yse!		
<input type="checkbox"/>	Edit Copy Delete	13	Am I still having fun?	am-i-still-having-fun	Probably		
<input type="checkbox"/>	Edit Copy Delete	14	Can this be this easy?	can-this-be-this-easy	Wouldnt' count on it		
<input type="checkbox"/>	Edit Copy Delete	15	What time is it?	what-time-is-it	Now		
<input type="checkbox"/>	Edit Copy Delete	16	is it similar to timestamp?	is-it-similar-to-timestamp	let's find out		

Slugs In Table

Hopefully the image is clear enough for you to see that it automatically dropped the ? from the faq_question records and put a dash between the words. How easy is that?

While we are successfully creating the slugs, we are not yet seeing them in the url.

Add Url Manager Rules

We will need to add the following line to both frontend and backend config/main.php:

```
'<controller:\w+>/<id:\d+>/<slug:[A-Za-z0-9 -_.]+>' => '<controller>/view',
```

Just pop that under the last existing rule and save.

Now we have given the urlManager a way to handle the slug in the url. We still need to modify the controller and the urls pointing at the view records.

Modify View Action on FaqController

Let's change the view action on the FaqController to the following:

Gist:

[View Action FaqController](#)

From book:

```

public function actionView($id, $slug = null)
{
    $model = $this->findModel($id);

    if ($slug == $model->slug){

        return $this->render('view', [
            'model' => $model,
            'slug' => $model->slug
        ]);
    } else {

        throw new NotFoundHttpException('The requested Faq does not exist.');
    }
}

```

Ok, so what we have different here is that we are taking in a second parameter and defaulting it to null. The reason why we are doing that is that is so we don't have to do a lot of error handling if the slug is not included in the url.

Next we call up the model record based on the id:

```
$model = $this->findModel($id);
```

Even though we allow the \$slug to be null in the signature, we require it to match to show the actual page:

```
if ($slug == $model->slug){
```

Otherwise, we throw the exception.

```
else {
```

```
    throw new NotFoundHttpException('The requested Faq does not exist.');
}
```

Then we simply pass along the \$model and \$model->slug to the view.

```
return $this->render('view', [
    'model' => $model,
    'slug' => $model->slug
]);
```

Modify Create and Update Actions on Backend Controller

We need to modify actionUpdate on the backend FaqController like so:

Gist:

[actionUpdate](#)

From book:

```
public function actionCreate()
{
    $model = new Faq();

    if ($model->load(Yii::$app->request->post()) && $model->save()) {
        $url = Url::toRoute('faq/' . $model->id . '/' . $model->slug);
        return $this->redirect($url);

    } else {
        return $this->render('create', [
            'model' => $model,
        ]);
    }
}
```

To use Url::, we need to include the use statement at the top:

```
use yii\helpers\Url;
```

The change to actionUpdate is similar.

Gist:

[actionUpdate FaqController Backend](#)

From book:

```

public function actionUpdate($id)
{
    $model = $this->findModel($id);

    if ($model->load(Yii::$app->request->post()) && $model->save()) {

        $url = Url::toRoute('faq/' . $model->id . '/' . $model->slug);
        return $this->redirect($url);

    } else {
        return $this->render('update', [
            'model' => $model,
        ]);
    }
}

```

In both cases, we had to format the redirect url to account for the slug. We used the Url::toRoute method to format the url, which allows us to specify a string.

We name the controller with / separator, concatenate the model id, add another / separator and then concatenate the slug. And that's it!

Change Gridview Action Column URL

In order to get the slug displayed in the URL window, we need to modify the links to the view record. Since we link to the individual via the Gridview on the index page, we will need to modify this.

Obviously, this is just for backend. Later, we will do some modifications for the frontend, giving you a complete solution.

So let's modify the GridView widget in the backend/views/faq/index.php to the following.

Gist:

[GridView](#)

From book:

```

<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'searchModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],

        'id',
        'faq_question',
        'faq_answer',
        ['attribute'=>'faqCategoryName', 'format'=>'raw'],
        'faq_weight',
        ['attribute'=>'faqIsFeaturedName', 'format'=>'raw'],

        [
            'class' => 'yii\grid\ActionColumn',
            'template' => '{view} {update} {delete}',
            'buttons' => [
                'view' => function ($url,$model) {
                    return Html::a(
                        '<span class="glyphicon glyphicon-eye-open"
                        aria-hidden="true"></span>',
                        $url.'/'.$model->slug);
                },
            ],
        ],
    ],
]); ?>

```

So you can see what is different. We are defining the template and the template tokens, {view}, for example, match up with the buttons as shown above.

In the buttons array, we define the view button only, the others will use default behavior. So we use an anonymous function to return the link via the Html::a method.

You can see we are concatenating \$model->slug onto the \$url:

```
$url.'/'.$model->slug
```

So now we have the complete solution for the backend, if you try clicking on your faq records from the index page, you will get the slug, which will give you the following format in the url:

```
http://backend.yii2start.com/faq/11/should-i-use-a-framework
```

If you decide you want a different separator for the words in the slug, you can accomplish that by modifying the behavior like so:

```
'sluggable' => [
    'class' => SluggableBehavior::className(),
    // 'attribute' => 'faq_question',
    // In case of attribute that contains slug has different name
    // 'slugAttribute' => 'alias',
    'value' => function ($event) {
        $question = rtrim($this->faq_question, '?');
        return str_replace(' ', '_', $question);
    }
],
```

We commented out the ‘attribute’ setting and added a value setting. For the value of the slug, we trim off the ? with rtrim. Then we do a simple str_replace to use an underscore instead of the default minus sign. You can use this format if you need to perform some other calculation, perhaps something more complicated.

Anyway, you can see how easy this is to work with. In the course of this book, we have used all four of Yii 2’s ready-made behaviors and they are incredibly useful and easy to use.

Ok, so to wrap up our implementation of slugs for Faq, we need to change the frontend.

Since we already have the rules copied into the frontend urlManager, we can move right to the view of the widget, faq.php.

Let’s make a simple change. In the foreach loop, remove the following line:

```
$url = Url::to(['faq/view', 'id'=>$model->id]);
```

And replace it with:

```
$url = Url::toRoute('faq/'.$model->id . '/' . $model->slug);
```

We are again using Yii 2’s Url::toRoute method, which allows us to pass in the url as a string.

That takes care of our friendly widget. But what about the Faq link in the header nav?

That’s actually not going to be so easy to change, we will have to change the controller as well as the index page.

Hmm. I wonder what could make this easier on us? If only we had a ready-made solution that we could just pop in. Wait. We do. Why don't we simply use the widget for this task, since it is already formatted?

It's true that we still have to change the frontend FaqController, but this is what we need to change the index action to:

```
public function actionIndex()
{
    return $this->render('index');
}
```

That one doesn't even require a gist. And here is the view action, exactly the same as the backend version:

```
public function actionView($id, $slug = null)
{
    $model = $this->findModel($id);

    if ($slug == $model->slug){
        return $this->render('view', [
            'model' => $model,
            'slug' => $model->slug
        ]);
    } else {
        throw new NotFoundHttpException('The requested Faq does not exist.');
    }
}
```

So again, no Gist necessary, you can copy your backend version if you like.

In the index.php view page, let's change it to the following.

Gist:

[Faq View Frontend](#)

From book:

```
use yii\helpers\Url;
use components\FaqWidget;

$this->title = 'FAQs';
$this->params['breadcrumbs'][] = $this->title;

?>

<div class="site-about">
    <h1><?= Html::encode($this->title) ?></h1>
</div>
<BR>

<?= FaqWidget::widget(['settings' =>
    ['pageSize' => 10,
     'featuredOnly' => false
    ]])?>
```

So you can see this actually got easier. Widgets are awesome, look at how much work we saved by not having to bother with another foreach loop.

One slight thing is not quite right. We are repeating the word Faq in the view page and this doesn't look right. Why don't we modify our widget to take in another parameter for heading, and then we can just pass it the value we want?

Because the settings array for our widget takes in the key/value pairs that we hand in from the method signature, we can just add another pair. All we have to do is change the logic on the widget view file.

So let's change Faq.php like so:

Gist:

[FaqWidget.php](#)

From book:

```
<?php

use yii\helpers\Html;

use yii\helpers\Url;

use yii\widgets\LinkPager;

?>

<div class="site-about">

    <div class="panel panel-default">

        <div class="panel-heading">

            <h3 class="panel-title">

                <?= $settings['heading']; ?>

            </h3>

        </div>

    </div>

<?php

foreach ($models as $model){

    $url = Url::toRoute('faq/' . $model->id . '/' . $model->slug);
```

```
$options = [];  
  
echo '<div class="panel-body">' .  
    Html::a($model->faq_question, $url, $options) . '</div>';  
  
}  
  
echo LinkPager::widget([  
  
    'pagination' => $pages,  
  
]);  
  
?>  
  
</div>  
  
</div>
```

You can see that now we are just echoing:

```
<?= $settings['heading'];?>
```

Ok, so now we just need to hand in the parameter in the two widget calls. First let's do the one on site index.php:

```
<?= FaqWidget::widget(['settings' => [
    'pageSize' => 3,
    'featuredOnly' => true,
    'heading' => 'Featured FAQs'
])
]) ?>
```

And on the Faq index view:

```
<?= FaqWidget::widget(['settings' => [
    'pageSize' => 3,
    'featuredOnly' => true,
    'heading' => 'Questions'
])
]) ?>
```

And that does it. Now we have more control over our output. Just don't forget to hand in a heading setting or you will get an error in your Faq.php file.

Alternatively, you could wrap it in an if statement to only display if it's set:

```
<h3 class="panel-title">

<?php

if(isset($settings['heading'])){
    echo $settings['heading'];
}

?>

</h3>
```

Summary

By taking control of the URLs and the slugs, we made our application more visible to the search engines. And since Yii 2 provides sluggable behavior, we can use it to automatically add slugs to our models.

This makes it easy and convenient to provide slugs for view pages.

Thanks again to all the readers who wrote in to help with typos and other suggestions. Also, we are getting a lot of great reviews on:

[GoodReads.com Yii 2 For Beginners Reviews](#)

Please take a moment if you can to share your thoughts, everyone will appreciate it. Let's share this amazing framework with as many people as we can.

Just to remind readers who may be reading this chapter as part of their initial purchase of the book, you get free updates of Yii 2 For Beginners for the life of the book. Just login to your leanpub.com account and grab the latest version.

I have a lot more bonus material planned, so look for the email announcements and take full advantage of that.

We will have a new bonus chapter on Yii 2 Authclient shortly, so we can integrate Facebook login and signup.

Thanks again for continuing to share the Yii 2 journey, see you soon.

Chapter 14: Bonus Material Social Login and Register

Welcome back to another round of bonus material in Yii 2 For Beginners. We are going to continue to add features to our template, and the focus of this chapter is social login and registration through social networks Facebook and Github.

Most clients will want automatic login and registration with Facebook, it's a feature that has become so commonplace that a web application doesn't seem modern without it.

Yii2 - AuthClient

Fortunately for us, the team at Yii 2 has a ready made extension, Yii2-authclient, which gives us everything we need to get started. I'm going to link to the section in the Yii Guide for reference:

[Yii2-AuthClient](#)

Since this is a fairly new entry into the guide, please keep in mind that it is subject to change, so it might not match our implementation perfectly. I do my best to stay on top of changes, but this can easily be missed, so it's best for you to reference the guide as well as this book.

At any rate, if it is different, you should be able to work it out. I'm going on with the assumption it's the same, and that this solution will work perfectly.

Yii2-AuthClient actually supports multiple social networks, including, Facebook, Twitter, Github, Google and more. For our purposes, we are going to focus on Facebook and Github, but you can use what you learn here for the other networks as well.

Ok, so what we're going to do:

- Add the Yii2-authclient extension to our composer.json and run composer update.
- Create an auth table.
- Create Auth model.
- Add the Yii2-authclient config to our components array in common/config/main.php
- Create our auth applications on each provider.
- Modify our actions method on the site controller.
- Add onAuthSuccess method to the site controller.
- Add navigation and links to Facebook and Github signup and sync.
- Add a social sync feature for existing accounts to sync with social networks.
- Refactor for maintainability.

Install yii2authclient via Composer

Go to your composer.json file and add the following line to your require block:

```
"yiisoft/yii2-authclient": "*",
```

Note that you don't need the comma if it's the last line. Then entire block at this point should look like:

```
"require": {  
    "php": ">=5.4.0",  
    "yiisoft/yii2": "*",  
    "yiisoft/yii2-bootstrap": "*",  
    "yiisoft/yii2-swiftmailer": "*",  
    "kartik-v/yii2-social": "dev-master",  
    "yiisoft/yii2-authclient": "*",  
    "fortawesome/fontawesome": "4.2.0"  
},
```

Then from your command line, run composer update:

```
C:\var\www\yii2build>composer update
```

Composer Update

Configuration

The next step is to configure the component. Go to common/config/main.php and add the following to your components array.

Gist:

[Auth Collection](#)

From book:

```
'authClientCollection' => [
    'class' => 'yii\authclient\Collection',
    'clients' => [
        'facebook' => [
            'class' => 'yii\authclient\clients\Facebook',
            'clientId' => 'your client id',
            'clientSecret' => 'your client secret',
        ],
        'github' => [
            'class' => 'yii\authclient\clients\GitHub',
            'clientId' => 'your client id',
            'clientSecret' => 'your client secret',
        ],
        'twitter' => [
            'class' => 'yii\authclient\clients\Twitter',
            'consumerKey' => 'your consumer key',
            'consumerSecret' => 'your consumer secret',
        ],
        'google' => [
            'class' => 'yii\authclient\clients\GoogleOAuth',
            'clientId' => 'your client id',
            'clientSecret' => 'your client secret',
        ],
        'linkedin' => [
            'class' => 'yii\authclient\clients\LinkedIn',
            'clientId' => 'your client id',
            'clientSecret' => 'your client secret',
        ],
    ],
],
```

You can see that we added settings for 5 providers. Obviously, replace the placeholders for ‘your client id’ and ‘your client secret’ with your actual settings.

As of this writing, Facebook, Linkedin, Google and Github work beautifully.

Twitter Issue

Twitter does not return the email address of the user, see the following issue:

Twitter Dev Support

After four years of developers complaining, they seem to be finally ready to supply the email, but as of this writing, and despite the promises, they have not done so. Without the email address, we would have to create a separate process for the email to be supplied by the end user, which at this point, doesn't make sense to write, if they are going to make the email available via api soon.

Provider Applications

Now we need to create our provider applications. For reference, I'm going to give you a list of provider urls to create the apps:

Facebook:

<https://developers.facebook.com/>

GitHub:

<https://github.com/settings/applications>

Twitter:

<https://apps.twitter.com/>

Google:

<https://console.developers.google.com/project>

LinkedIn:

<https://www.linkedin.com/secure/developer>

Yii2authclient also supports:

- Microsoft Live
- VKontakte
- Yandex

At this point, I won't be implementing those into the template.

Facebook App

Since we have a Facebook app already, let's start with that, it will only need a simple setting added to it. Make sure you are logged in to Facebook and go to:

<https://developers.facebook.com/>

Select your app from the My Apps dropdown in the header nav. Then select settings.

We have to make sure that we define the App Domains:

You are currently editing a test version of **yii 2 start**

Basic **Advanced** **Migrations**

App ID: [REDACTED] App Secret: [REDACTED] **Show**

Display Name: yii2build Namespace: [REDACTED]

App Domains: yii2build.com

Website Quick Start

Site URL: http://www.yii2build.com/

Mobile Site URL: http://www.yii2build.com/

+ Add Platform

App Domains

Obviously, I scratched out my App Id. Your App Id and Secret should be there from when we created it in chapter 10. Also, make sure you have added your id and secret to the config in components. And that's it, we should be good.

Github App

Next we'll make an app for Github. Login to Github and go to:

Search GitHub

Explore Gist Blog Help

evercode1

Personal settings Developer applications Register new application

Profile

These are applications you have registered to use the GitHub API.

Create Github App

Click on the Register new application button. Then fill in your details as follows:

Personal settings
Profile
Account settings
Emails
Notification center
Billing
SSH keys
Security
Applications
Repositories
Organizations

Applications / **Register a new OAuth application**

Application name
Yii2Build
Something users will recognize and trust

Homepage URL
http://www.yii2build.com
The full URL to your application homepage

Application description
Demo from Yii 2 For Beginners
This is displayed to all potential users of your application

Authorization callback URL
http://www.yii2build.com
Your application's callback URL. Read our OAuth documentation for more information

Register application

Github App Details

Once you submit, it will return your client id and client secret:

Applications / **Yii2Build**

 evercode1 owns this application. [Transfer ownership.](#)

0 users

Client ID
'your id'
Client Secret
'your secret'

Revoke all user tokens **Reset client secret**

Application name
Yii2Build
Something users will recognize and trust

Homepage URL
Drag & drop

Github App Id and Secret

Obviously, yours will display the actual id and secret, not the placeholder text. Also, make sure you have added your id and secret to the config in components. Ok, so by now you can see how this works and what info you need to supply.

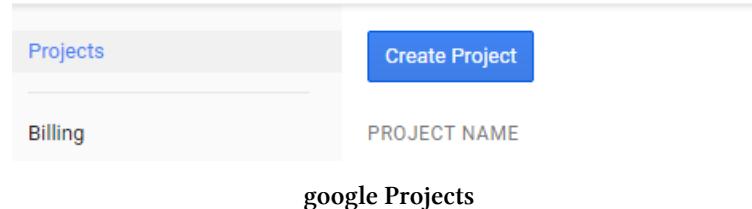
Google App

I'm going to provide screenshots, but please note that things may change over time. If what you see on Google is different than what you see here, this should still be enough to point you in the right direction. When you go to Google:

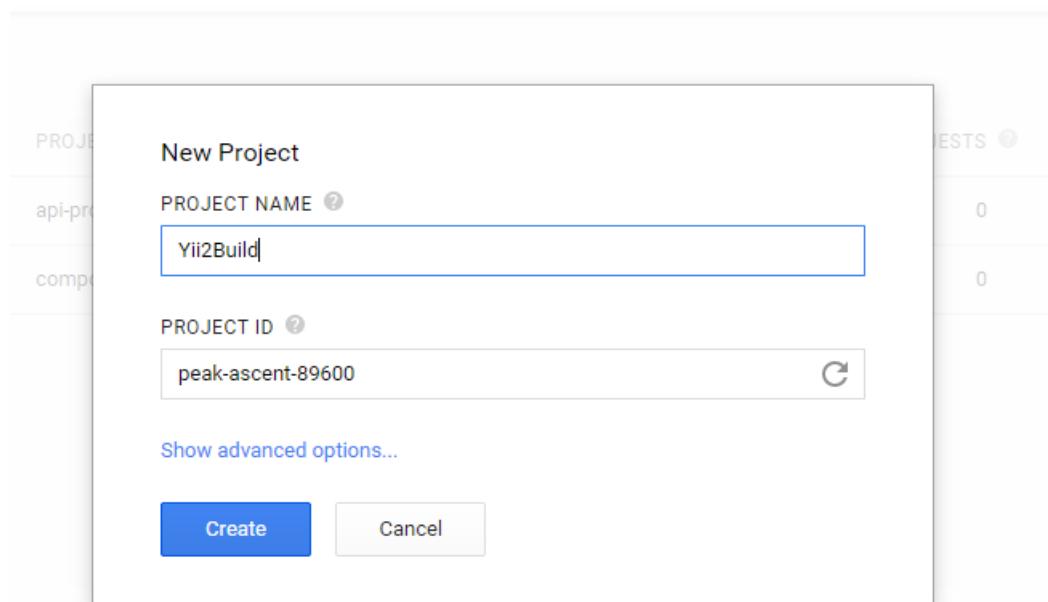
<https://console.developers.google.com/project>

You will see:

Google Developers Console

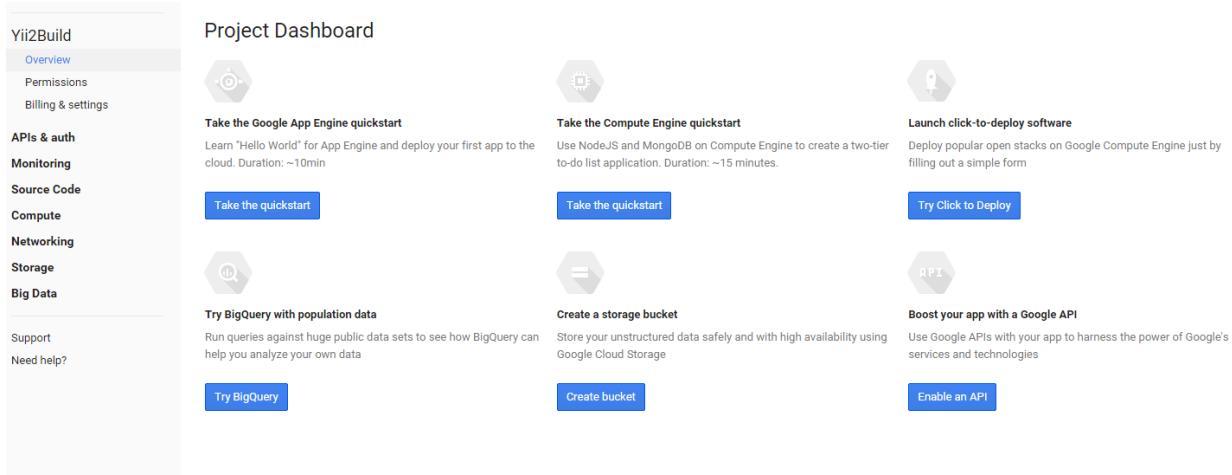


Then create your project:



Create Google Project

You will land on:



Google Project Dash

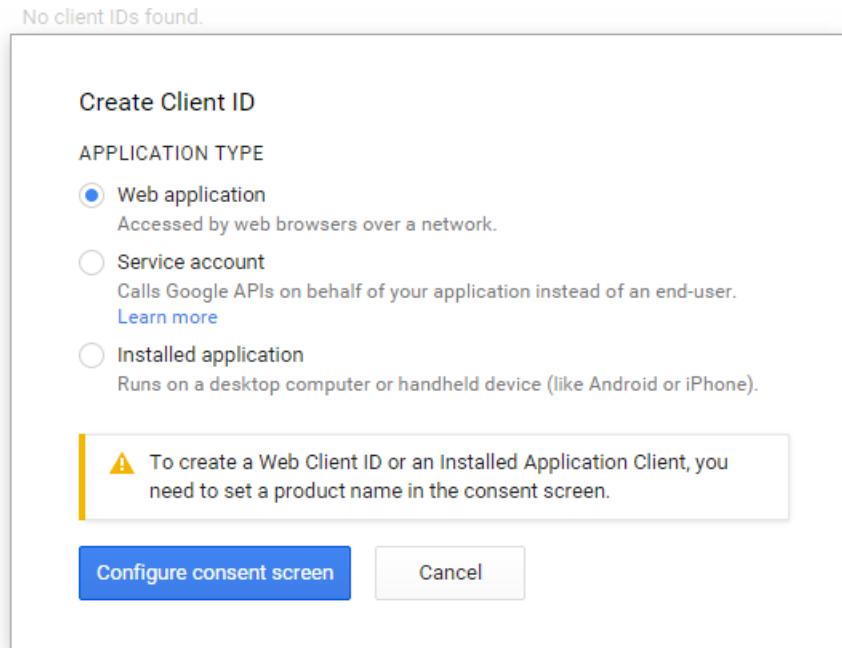
Then select Credentials under APIs & auth on the left menu. It will bring up:

The screenshot shows the 'Credentials' section of the Google APIs & auth page for the 'Yii2Build' project. The left sidebar includes links for Overview, Permissions, Billing & settings, APIs & auth (selected), APIs, Consent screen, Push, Monitoring, Source Code, Compute, and Networking. The main content area has two sections:

- OAuth**: Describes OAuth 2.0 and provides a 'Create new Client ID' button. Text: 'No client IDs found.' Below it is a 'Learn more' link.
- Public API access**: Describes Public API access and provides a 'Create new Key' button. Text: 'No keys found.' Below it is a 'Learn more' link.

Create OAuth Credentials

Click on Create new Client ID, you will get:



Configure Consent

Continue to:

< Projects

Yii2Build

- Overview
- Permissions
- Billing & settings

APIs & auth

- APIs
- Credentials
- Consent screen**
- Push

Monitoring

Source Code

Compute

Networking

Storage

Big Data

Support

Need help?

Consent screen

The consent screen will be shown to users whenever you request access to their private data using your client ID.

Note: This screen will be shown for all of your applications registered in this project

EMAIL ADDRESS

PRODUCT NAME

HOME PAGE URL (Optional)

PRODUCT LOGO (Optional) ⓘ

This is how your logo will look to end users.
Max size: 120x120 px

PRIVACY POLICY URL (Optional)

TERMS OF SERVICE URL (Optional)

GOOGLE+ PAGE (Optional) ⓘ

Save **Cancel**

Consent Screen

Next we come to:

Create Client ID

APPLICATION TYPE

Web application
Accessed by web browsers over a network.

Service account
Calls Google APIs on behalf of your application instead of an end-user.
[Learn more](#)

Installed application
Runs on a desktop computer or handheld device (like Android or iPhone).

AUTHORIZED JAVASCRIPT ORIGINS
Cannot contain a wildcard (`http://*.example.com`) or a path
(`http://example.com/subdir`).

`http://www.yii2build.com`

AUTHORIZED REDIRECT URIS
One URI per line. Needs to have a protocol, no URL fragments, and no relative paths. Can't be a non-private IP Address.

`http://www.yii2build.com/site/auth?authclient=google|`

Create Client ID

Create Client Id

That will bring you to:

The screenshot shows the OAuth settings page in the Yii2Build dashboard. On the left, there's a sidebar with 'Yii2Build' and various navigation links like 'Overview', 'Permissions', 'Billing & settings', 'APIs & auth' (which is expanded), 'APIs', 'Credentials' (which is selected and highlighted in grey), 'Consent screen', and 'Push'. The main content area has a heading 'OAuth' with a sub-section about OAuth 2.0. It includes a 'Learn more' link and a prominent blue button labeled 'Create new Client ID'. To the right, there's a form for creating a client ID for a web application, with fields for 'CLIENT ID', 'EMAIL ADDRESS', 'CLIENT SECRET', 'REDIRECT URIS', and 'JAVASCRIPT ORIGINS'. At the bottom right of the form are two buttons: 'Edit settings' and 'Reset se'.

I purposely left out the actual Client Id, but you will see it when you land on this page. Please note that the email address is in between ClientId and Client Secret, so don't get confused and grab the wrong id.

Next click on APIs, you will get:

The screenshot shows the APIs page in the Yii2Build dashboard. The sidebar on the left is identical to the previous screenshot, with 'APIs & auth' expanded and 'APIs' selected. The main content area features a search bar with the placeholder 'Search all 100+ APIs'. Below it, there's a section titled 'Popular APIs' with a list of Google Cloud APIs. Each API is represented by a small icon and a link. The 'Google Cloud APIs' section includes: Compute Engine API, BigQuery API, Cloud Storage API, Cloud Datastore API, Cloud Deployment Manager API, and Cloud DNS API. There's also a 'More' link. To the right of this, under 'Social APIs', there's a red square icon with a white 'g+' symbol. The 'Social APIs' section includes: Google+ API, Blogger API, Google+ Pages API, and Google+ Domains API.

Google APIs

Select Google+ Api.. Enable the API and you will get:

The screenshot shows the Google Developers Console interface. On the left, there's a sidebar for the 'Yii2Build' project with sections for Overview, Permissions, Billing & settings, APIs & auth (which is currently selected), APIs, Credentials, Consent screen, and Push. The main content area is titled 'Google+ API' with tabs for OVERVIEW (selected), USAGE, and QUOTAS. Below the tabs, it says 'The Google+ API enables developers to build on top of th...' and provides links to Learn more, Explore this API, and View reports in API Console.

Google APIs

Go back to the APIs screen and enable Google contacts API, it's under the Google Apps APIs heading. And you should be good to go with Google.

LinkedIn App

This provider has significantly less steps than Google. Start by going to:

LinkedIn:

<https://www.linkedin.com/secure/developer>

You will land on:

The screenshot shows the LinkedIn Developer Network 'List of Applications' page. It features a table with columns for Company and Application Name. One entry is visible: 'Company' is 'Yii 2 For Beginners' and 'Application Name' is 'Yii 2 Start'. There is also a 'View API Usage' link. At the bottom of the table, there's a link to 'Add New Application'. Below the table, a link leads back to the LinkedIn Developer Network.

Linked App Dash

Click on Add New Application and you will get the form:

Create a New Application

Company Name:*

Yii 2 For Beginners ▾

Name:*

Description:*

Application Logo URL:*

Application Use:*

Select One... ▾

Website URL:*

Business Email:*

Business Phone:*

I have read and agree to the [LinkedIn API Terms of Use](#).

Submit

Cancel

Linked App Form

A couple of things to note. You can only select one default scope, it will return a cryptic error if you select more than one.

Also note that the redirect url is the same format as the one for Google, but obviously with linkedin as the authclient.

You will go to the success page. You want the following:

Consumer Key / API Key maps to clientId

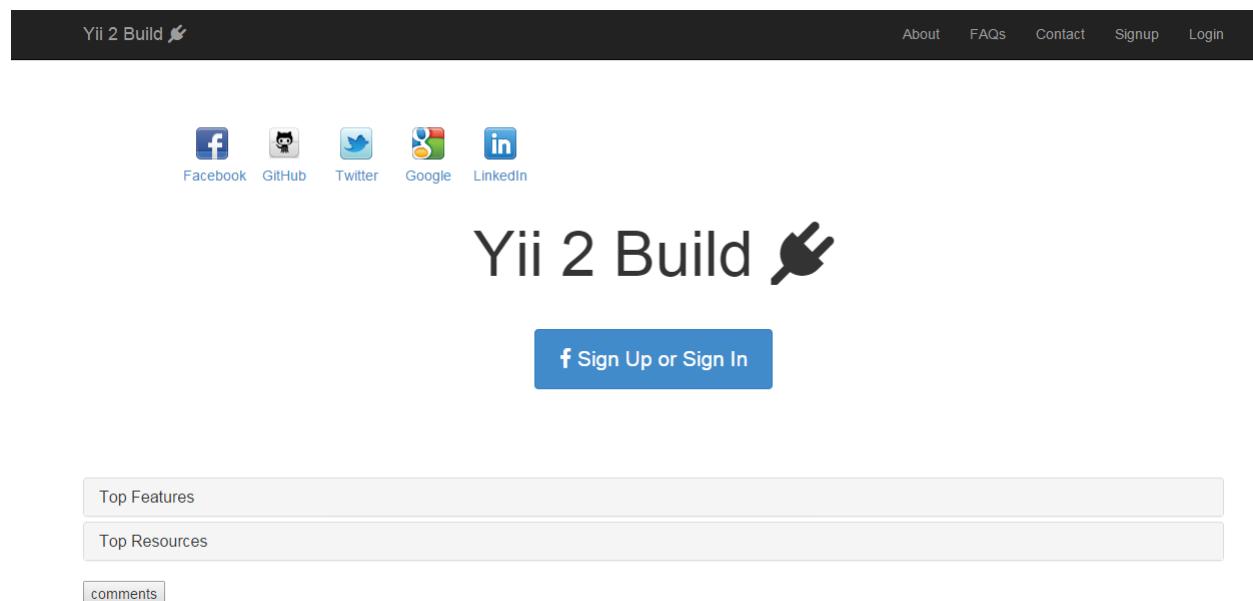
Consumer Secret / Secret Key maps to clientSecret

It's important to use clientId and clientSecret in your common/config/main.php settings. And that's it, you should be good to go.

To get a sense of how we will use social login and register, I will provide some screenshots, followed by Gists of the related code.

Index View Change

Let's start with frontend/view/site/index.php:



New index.php

Ok, so you can see we have some nice icons for the providers. These are appearing because we have used the AuthChoice::widget:

```
echo yii\authclient\widgets\AuthChoice::widget([
    'baseAuthUrl' => ['site/auth'],
    'popupMode' => false,
])
```

This widget will call as many icons as you have configured in your components array. Since we included 5 there, we are getting that many in our view. If you want less icons to appear, remove the providers from the config. Even though not all are working now, I'm keeping them because I will include them once they are working.

I also made a big redundant Facebook Signup or Login button. Obviously it's up to you to make this look the way you want it to. I put the large facebook button in as well as the widgets to demonstrate both.

Here is what we have changed in the index.php view file, replacing everything up to the opening php tag of the first collapse widget:

Gist:

New Index.php code

From book:

```
<?php
use \yii\bootstrap\Modal;
use kartik\social\FacebookPlugin;
use \yii\bootstrap\Collapse;
use \yii\bootstrap\Alert;
use yii\helpers\Html;
use components\FaqWidget;

/* @var $this yii\web\View */
$this->title = 'My Yii Application';
?>
<div class="site-index">

<div class="jumbotron">

<?php

if (Yii::$app->user->isGuest) {

    echo yii\authclient\widgets\AuthChoice::widget([
        'baseAuthUrl' => ['site/auth'],
        'popupMode' => false,
    ]);
}

?>

<h1>Yii 2 Start <i class="fa fa-plug"></i></h1><br>
<?php

if (!Yii::$app->user->isGuest) {

    echo '<p class="lead">Use this Yii 2 Template to start Projects.</p>';
}

} else {
```

```
echo '<h4>' . Html::a(' <i class="fa fa-facebook"></i>
    Sign Up or Sign In',
    ['auth', 'authclient' => 'facebook'],
    ['class' => 'btn btn-primary ']).'</h4>';

}

?>

</div>
```

You'll note that we wrapped our widget in an if statement to test to see if the user is a guest. Then we do another if statement to check for guest and if the user is a guest, show them the facebook sign up or sign in button.

Login View Change

We also need to add our widget to our login and signup pages. Here's what it should look like:

The screenshot shows a login page with a header navigation bar containing 'Home' and 'Login'. Below the header is a large 'Login' heading. Underneath the heading are five social login icons: Facebook, GitHub, Twitter, Google, and LinkedIn. A subtext below the icons reads 'Please fill out the following fields to login:'. There are two input fields labeled 'Username' and 'Password'. Below the password field is a checked checkbox labeled 'Remember Me'. A link 'If you forgot your password you can [reset it.](#)' is present. At the bottom is a blue 'Login' button. The entire page has a clean, modern design with a light gray background.

New index.php

You can see that all we did was strategically place the widget. No need to review the code, but I will supply the gist for the entire file for reference:

Gist:

[Login](#)

Signup View Change

Next up is the Signup view:

The screenshot shows the Yii 2 Build application interface. At the top, there is a dark header bar with the text "Yii 2 Build" and a small icon. Below the header, a navigation bar contains "Home" and "Signup". The main content area has a large title "Signup". Below the title are five social media icons with their respective names: Facebook, GitHub, Twitter, Google, and LinkedIn. A note below the icons says "Otherwise please fill out the following fields to signup:". There are three input fields labeled "Username", "Email", and "Password". At the bottom is a blue "Signup" button.

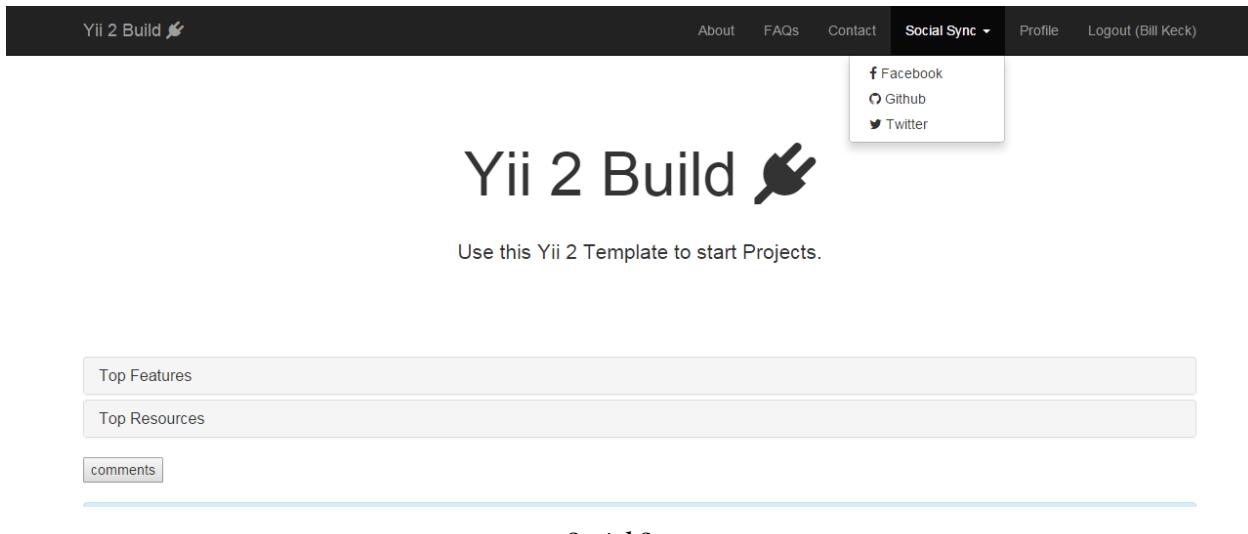
New index.php

Again, such a simple change we don't need to review, but here is the Gist for reference:

[New Signup View](#)

Social Sync Dropdown

Also, we will need a change to frontend/views/layouts/main.php. We want a dropdown list of social providers that allow the user to sync their existing account to their social provider, if they wish to do so. That will look like this:



Here is the entire frontend/views/layouts/main.php code:

Gist:

New Main

You can see in the following block that we are using font-awesome for our social icons:

```
$menuItems[] = [ 'label' => 'Social Sync', 'items' => [
    [ 'label' => '<i class="fa fa-facebook"></i> Facebook',
      'url' => [ 'site/auth', 'authclient' => 'facebook' ] ],
    [ 'label' => '<i class="fa fa-github"></i> Github',
      'url' => [ 'site/auth', 'authclient' => 'github' ] ],
    [ 'label' => '<i class="fa fa-twitter"></i> Twitter',
      'url' => [ 'site/auth', 'authclient' => 'twitter' ] ],
]];
```

We are specifying in the 'url' to go to site controller, auth action, and then handing it a get parameter of authclient='facebook' for example.

We have no auth action yet, but we will take care of that soon.

In order for font-awesome to work correctly, we have to specify encodeLabels to false in the Nav::widget:

```
echo Nav::widget([
    'options' => ['class' => 'navbar-nav navbar-right'],
    'items' => $menuItems,
    'encodeLabels' => false,
]);
```

Auth Data Structure

Now that we got all the cosmetic stuff out of the way, we build the data structure to support it. We'll start by adding a table to our database. The guide gives us some SQL to work with but it doesn't work as is for MySql:

```
CREATE TABLE auth (
    id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    user_id int(11) NOT NULL,
    source string(255) NOT NULL,
    source_id string(255) NOT NULL
);
```

```
ALTER TABLE auth ADD CONSTRAINT fk-auth-user_id-user-id
FOREIGN KEY user_id REFERENCES user(id);
```

Obviously, string is not a data type that MySql will recognize. Plus, using dashes in the foreign key seemed to create a problem for me, so I had to use MySql Workbench to get this syntax correct for that and other issues. This is what it gave me:

Gist:

[SQL for Auth table](#)

From book:

```
CREATE TABLE IF NOT EXISTS `yii2build`.`auth` (
    `id` INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
    `user_id` INT(11) UNSIGNED NOT NULL,
    `source` VARCHAR(255) NOT NULL,
    `source_id` VARCHAR(255) NOT NULL,
    PRIMARY KEY (`id`),
    INDEX `fk_auth_user_id_user_id` (`user_id` ASC),
    CONSTRAINT `fk_auth_user1`
        FOREIGN KEY (`user_id`)
        REFERENCES `yii2build`.`user` (`id`)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION)
```

I thought it would be wise to have an example in SQL that I could refer to in the future, if I needed to. If you are collaborating with someone or need to post code on Github or elsewhere, you will end up writing the SQL or creating a migration.

Auth Model

Now that we have the table, the next step is to create an Auth model. We going to do this using Gii:

The screenshot shows the Yii Gii Model Generator interface. On the left, there's a sidebar with several generator options: Model Generator (selected), CRUD Generator, Controller Generator, Form Generator, Module Generator, and Extension Generator. The main area is titled "Model Generator" and contains the following fields:

- Table Name:** auth
- Model Class:** Auth
- Namespace:** common\models
- Base Class:** yii\db\ActiveRecord
- Database Connection ID:** db
- Options:**
 - Use Table Prefix
 - Generate Relations
 - Generate Labels from DB Comments
 - Enable I18N

The title at the bottom of the form is "Auth Model".

We're going to put the model in common, so the namespace field is:

```
common\models
```

Then we just generate the file and we're good to go. Gii will automatically include the relationship we need:

```
public function getUser()
{
    return $this->hasOne(User::className(), ['id' => 'user_id']);
}
```

Ok, now we need to work on our site controller.

Site Controller Actions Method

The first thing is we need to add to our actions method.

Gist:

[New Actions Method](#)

From book:

```
public function actions()
{
    return [
        'error' => [
            'class' => 'yii\web\ErrorAction',
        ],
        'captcha' => [
            'class' => 'yii\captcha\CaptchaAction',
            'fixedVerifyCode' => YII_ENV_TEST ? 'testme' : null,
        ],
        'auth' => [
            'class' => 'yii\authclient\AuthAction',
            'successCallback' => [$this, 'onAuthSuccess'],
        ],
    ];
}
```

You can see that ‘auth’ is specifying the class and a successCallback parameter, essentially telling the actions that when the url is site/auth on a callback to use the onAuthSuccess method.

OnAuth Success

We will add that method to the site controller shortly. First let’s look at the onAuthSuccess method from the guide:

```
public function onAuthSuccess($client)
{
    $attributes = $client->getUserAttributes();

    /** @var Auth $auth */
    $auth = Auth::find()->where([
        'source' => $client->getId(),
        'source_id' => $attributes['id'],
    ])->one();

    if (Yii::$app->user->isGuest) {
        if ($auth) { // login
            $user = $auth->user;
            Yii::$app->user->login($user);
        } else { // signup
            if (isset($attributes['email']))
                && isset($attributes['username'])
                && User::find()->where([
                    'email' => $attributes['email']]])->exists()) {
                Yii::$app->getSession()->setFlash('error', [
                    Yii::t('app', "User with the same email as in {client}
                        account already exists but isn't linked to it.
                        Login using email first to link it.",
                    ['client' => $client->getTitle()]),
                ]);
            } else {
                $password = Yii::$app->security->generateRandomString(6);
                $user = new User([
                    'username' => $attributes['login'],
                    'email' => $attributes['email'],
                    'password' => $password,
                ]);
                $user->generateAuthKey();
                $user->generatePasswordResetToken();
                $transaction = $user->getDb()->beginTransaction();
                if ($user->save()) {
                    $auth = new Auth([
                        'user_id' => $user->id,
                        'source' => $client->getId(),
                        'source_id' => (string)$attributes['id'],
                    ]);
                    if ($auth->save()) {

```

```
        $transaction->commit();
        Yii::$app->user->login($user);
    } else {
        print_r($auth->getErrors());
    }
} else {
    print_r($user->getErrors());
}
}

} // user already logged in
if (!$auth) { // add auth provider
    $auth = new Auth([
        'user_id' => Yii::$app->user->id,
        'source' => $client->getId(),
        'source_id' => $attributes['id'],
    ]);
    $auth->save();
}
}
}
```

If you simply find:

```
'username' => $attributes['login'],
```

And replace it with:

```
'username' => $attributes['name'],
```

That will work to some degree for Facebook. You can test the basic signup and login, it should work. But it's not quite right for our purposes.

I'm going to step us through the `onAuthSuccess` method in detail, but there are enough changes in my version that I think we should just get straight to it.

Updated OnAuth Success

The updated onAuthSuccess method.

Gist:

onAuthSuccess Revision 1

From book:

```
public function onAuthSuccess($client)
{
    $attributes = $client->getUserAttributes();

    // $attributes['email'] = null;
    // var_dump($attributes);
    // die();

    // if provider didn't supply email

    if (!isset($attributes['email'])){

        return Yii::$app->getSession()->setFlash('error', [
            Yii::t('app', "Unable to finish, {client} did not
            provide us with an email. Please check your settings on
            {client}.", ['client' => $client->getTitle()]),
        ]);
    }

    $source = $client->getId();

    // set $username to correct $attribute from each provider

    switch ($source){

        case $source == 'facebook' :
            $username = 'name';
            break;

        case $source == 'github' :
            $username = 'login';
            break;

        case $source == 'twitter' :
            $username = 'screen_name';
            break;

        default:
```

```
$username = 'name';

}

$auth = Auth::find()->where([
    'source' => $source,
    'source_id' => $attributes['id'],
])->one();

if (Yii::$app->user->isGuest) {

    if ($auth) { // login

        $user = $auth->user;
        Yii::$app->user->login($user);

    } else { // signup

        if (isset($attributes['email']))
            && User::find()->where
                ([ 'email' => $attributes['email']])->exists()) {
            return Yii::$app->getSession()->setFlash('error', [
                Yii::t('app', "User with the same email as in {client} account
already exists but isn't synced. Login with username and password
and click the {client} sync link to sync accounts.",
                ['client' => $client->getTitle()]),
            ]);
        }

    } else {

        $password = Yii::$app->security->generateRandomString(6);
        $user = new User([
            'username' => $attributes[$username],
            'email' => $attributes['email'],
            'password' => $password,
        ]);
        $user->generateAuthKey();
        // $user->generatePasswordResetToken();
        $transaction = $user->getDb()->beginTransaction();
        if ($user->save()) {

    }
}
```

```
$auth = new Auth([
    'user_id' => $user->id,
    'source' => $client->getId(),
    'source_id' => (string)$attributes['id'],
]);

if ($auth->save()) {
    $transaction->commit();
    Yii::$app->user->login($user);
    MailCall::onMailableAction('signup', 'site');

} else {
    return Yii::$app->getSession()->setFlash('error', [
        Yii::t('app', "We were unable to complete the
process and sync {client}.",
        ['client' => $client->getTitle()]),
    ]);
}

} else {

    return Yii::$app->getSession()->setFlash('error', [
        Yii::t('app', "We were unable to complete the process
and sync {client}.",
        ['client' => $client->getTitle()]),
    ]);
}

}

} else { // user already logged in

if (!$auth && $this->attributes['email'] ==
    Yii::$app->user->identity->email) { // add auth provider

$auth = new Auth([
    'user_id' => Yii::$app->user->id,
    'source' => $source,
    'source_id' => (string)$attributes['id'],
]);

$auth->save();
```

```

        Yii::$app->getSession()->setFlash('success', [
            Yii::t('app', "Your {client} account is successfully
            synced.", ['client' => $client->getTitle()]),
        ]);

    } else {//emails don't match

        if($attributes['email'] != Yii::$app->user->identity->email){

            Yii::$app->getSession()->setFlash('error', [
                Yii::t('app', "Your {client} account could not be
                synced.", ['client' => $client->getTitle()]),
            ]);

        } else {// account was already synced

            Yii::$app->getSession()->setFlash('success', [
                Yii::t('app', "Your {client} account is already
                synced.", ['client' => $client->getTitle()]),
            ]);

        }
    }

}

```

Note that I have not added Google and LinkedIn yet. I will do this in the final version.

Login and Registration Scenarios

With this solution, Facebook and Github should be fully operational in all scenarios, including:

1. User signup when no existing email is found in the database.
2. User signup cannot complete because email already in database.
3. User signup cannot complete because social provider did not supply email address.
4. Login with synced account.
5. Sync social account with existing user record from logged in state.

Scenario 1 User Signup

This is for a new registration on our application by using the social provider. The email address is the unique identifier that ties everything together. We check for an existing email in our user table, and if we find one, we return a message instructing the user to login first in order to sync their social provider for future login.

Scenario 2 Email Already In Use

When the user tries to signup via social provider, we check for the existence of the email in the system. If it already belongs to a user, we return a message instructing the registrant to login first, then sync their account.

Scenario 3 Provider Did Not Provide Email

In some cases, the provider does not provide the email address, such as in the case of twitter. Also, a user might have the wrong settings in their account, for example, Github has two places to set email permission, under profile and under email, so that can be tricky. The user must make sure they have given permission to make the email available.

In case of missing email, our application simply returns a message suggesting they check their social settings.

Scenario 4 Login with Provider

This scenario allows users to login in with their social provider. The user has to have already created an account through social provider or synced their existing account for this to work.

Scenario 5 Sync Existing Account

In this scenario, the user already has a user record, but it is not synced with their social provider. So in this case, they need to login and click the Social Sync link we created in main.php.

When their account is successfully synced, we return a message stating this. If they already have synced their account and decide to click the button again, we send them a message stating that their account is already synced.

So now that we know how it works, we can go over the onAuthSuccess method to see how we implemented this.

Let's start with the signature:

```
public function onAuthSuccess($client)
```

It's taking an object of the auth provider, which will hold all the attributes, which we can access like so:

```
$attributes = $client->getUserAttributes();
```

So we now have an array of the attributes to play with.

You may have noticed the debug lines:

```
// debug stuff
//$attributes['email'] = null;
//var_dump($this->attributes);
// die();
```

If you want to test what happens if ‘email’ is not set in \$attributes, uncomment:

```
$attributes['email'] = null;
```

If you want to see what the provider is returning, uncomment:

```
var_dump($this->attributes);
die();
```

Just make sure you comment out the rest of the method or it will not pause for you to see the results.

So Facebook, for example, returns the following:

```
array(11) { ["id"]=> string(15) "11111111111111111"
["email"]=> string(17) "some@email.com"
["first_name"]=> string(4) "Bill"
["gender"]=> string(4) "male"
["last_name"]=> string(4) "Keck"
["link"]=> string(60)
"https://www.facebook.com/app_scoped_user_id
/11111111111111111/" ["locale"]=> string(5) "en_US"
["name"]=> string(9) "Bill Keck"
["timezone"]=> int(-0)
["updated_time"]=> string(24) "2015-02-18T02:05:22+0000"
["verified"]=> bool(true) }
```

Obviously I have changed some of the values not to expose personal data.

Ok, comment out the debug and uncomment the rest of the method.

The first thing we do after creating \$attributes to hold our values is check to see if they have provided an email:

```
if (!isset($attributes['email'])){

    return Yii::$app->getSession()->setFlash('error', [
        Yii::t('app', "Unable to finish, {client}
            did not provide us with an email.
            Please check your settings on {client}.",
        ['client' => $client->getTitle()]),
    ]);
}

}
```

By using the Yii::t method, we can create a client token, using the brackets, and set the token value, in this case:

```
['client' => $client->getTitle()])
```

Next we set up a switch statement to allow us to set attributes according to provider, since they have different names for things:

```
$source = $client->getId();

//set $username to correct $attribute from each provider

switch ($source){

    case $source == 'facebook' :

        $username = 'name';

        break;

    case $source == 'github' :
```

```
$username = 'login';

break;

case $source == 'twitter' :

$username = 'screen_name';

break;

default:

$username = 'name';

}
```

`$client->getId()` gives us the name of the provider, so we assign this to a local variable named `$source` and then switch on that.

We are also using `$username` as variable name that will hold the attribute value we are looking for. You can see that with our 3 providers listed, we have 3 different original field names that we convert to `$username`.

We are using `$source` and `$username` later in the code.

Next we try to grab an auth record for the user if it exists:

```
$auth = Auth::find()->where([
    'source' => $source,
    'source_id' => $attributes['id'],
])->one();
```

If you recall our data structure for the auth table, we record a source_id, which is the id record the social provider gives them. So using, Facebook as an example, if there is an auth record with a source_id that matches the id from the \$attributes array, then we have a matching record for that user.

Next it will try to login in the user:

```
if (Yii::$app->user->isGuest) {

    if ($auth) { // login

        $user = $auth->user;

        Yii::$app->user->login($user);
    }
}
```

If we have successfully returned a record from the previous code block, we log the user in. If not, we go to the else statement. Let's look at the first part:

```
// signup

if (isset($attributes['email']) && User::find()->where(
(['email' => $attributes['email']])->exists()) {

    return Yii::$app->getSession()->setFlash('error', [
        Yii::t('app', "User with the same email as in {client}
account already exists but isn't synced. Login with
username and password and click the {client} sync
link to sync accounts.", ['client' => $client->getTitle()]),
    ]);
}
```

We do a check to see if the email is already taken. We're using the handy exists method, so take note of that for future reference. If it does already exist, we return a flash message with the appropriate tokens.

If we do not already have the email in our db, then we execute the else statement, which will create the user's account. We start by generating a password:

```
else {  
  
    $password = Yii::$app->security->generateRandomString(6);  
  
    $user = new User([  
        'username' => $attributes[$username],  
        'email' => $attributes['email'],  
        'password' => $password,  
    ]);  
  
    $user->generateAuthKey();  
  
    // $user->generatePasswordResetToken();
```

Then you can see we create a new instance of User and set the column values we need for the user record from the \$attributes array and the newly created \$password. You can see I commented out:

```
// $user->generatePasswordResetToken();
```

I'm just keeping that for reference since it was in the method in the guide. It's completely unnecessary for our implementation, so you can remove it if you want.

Since we are saving records into two tables, one for auth and one for user, we will use a transaction. We start like this:

```
$transaction = $user->getDb()->beginTransaction();
```

Then we move on to saving the user and creating a new auth record and saving that as well:

```

if ($user->save()) {
    $auth = new Auth([
        'user_id' => $user->id,
        'source' => $client->getId(),
        'source_id' => (string)$attributes['id'],
    ]);
    if ($auth->save()) {
        $transaction->commit();
        Yii::$app->user->login($user);
        MailCall::onMailableAction('signup', 'site');

    }
}

```

You can see that if we save both the \$user and \$auth, we commit the transaction. Then we login the user and send them an autoresponder.

I'm basing this use of a transaction on the original example of this method that was provided in the guide. Normally, I would expect to see a try/catch block and a rollback statement, but that doesn't seem to be necessary here.

I did play around with it to see if I could break it and get one of the pieces of the transaction to save, but from what I can tell it actually enforces the transaction as written.

I checked the guide:

Yii 2 Database

But I couldn't find any reference to the way we are doing it here. So, since it works, it came from another part of the guide, and it allows me to write error messages with tokens, I'm going to use it.

That brings us to the next part. If the transaction does not complete, we send them a message:

```

else {

    return Yii::$app->getSession()->setFlash('error', [
        Yii::t('app', "We were unable to complete the
        process and sync {client}.",
        ['client' => $client->getTitle()]),
    ]);
}

} else {

    return Yii::$app->getSession()->setFlash('error', [
        Yii::t('app', "We were unable to complete the process
        and sync {client}.",
        ['client' => $client->getTitle()]),
    ]);
}

```

```

        ]);
    }
}
}

```

You would think we're done, but of course we're not. Now we execute on the else statement when we checked on whether or not the user was a guest. So, if they are not a guest, they are already logged in and we move on:

```

else { // user already logged in

    if (!$auth && $this->attributes['email'] ==
        Yii::$app->user->identity->email) { // add auth provider

        $auth = new Auth([
            'user_id' => Yii::$app->user->id,
            'source' => $source,
            'source_id' => (string)$attributes['id'],
        ]);

        $auth->save();

        Yii::$app->getSession()->setFlash('success', [
            Yii::t('app', "Your {client} account
                is successfully synced.",
            ['client' => $client->getTitle()]),
        ]);
    }
}

```

Since we have previously tried to return an instance of \$auth where source_id and \$attributes['id'] match, we can use an if statement set the condition to create an auth record:

```
if (!$auth && $this->attributes['email'] == Yii::$app->user->identity->email)
```

So if that evaluates true, then create the new auth record. Note that the email values have to match. You'll notice (string) here:

```
'source_id' => (string)$attributes['id'],
```

That's there because we need to tell it we want a string, which is what we need to pass validation. Remember, we set that column up as a varchar, and this gives us flexibility to accommodate the different formats from the providers.

Then finally, we have:

```
else {//emails don't match

    if($attributes['email'] != Yii::$app->user->identity->email){

        Yii::$app->getSession()->setFlash('error', [
            Yii::t('app', "Your {client} account could not be synced.",
            ['client' => $client->getTitle()]),
        ]);

    } else {// account was already synced

        Yii::$app->getSession()->setFlash('success', [
            Yii::t('app', "Your {client} account is already synced.",
            ['client' => $client->getTitle()]),
        ]);

    }

}

}
```

So we're covered in case the emails don't match or if the account was already synced.

This is a working solution and you should be able to test it successfully. But there's a problem, which might not seem so obvious at first. As subtle as it is, we are running into code duplication.

Notice that we used:

```
MailCall::onMailableAction('signup', 'site');
```

In the signature, ‘signup’ designates the action, but that is not the correct action. So we would either have to make another status message for this action, which makes no sense, or we have to use this `onMailableAction` in a way that was not intended.

That workaround is easy enough, and just to get up and running and test things, it's fine. But maintaining an application over time, this could be a real problem. If you made changes to how and when you want to call the same autoresponder, you would have to update in 2 places. That's just a horrible practice if it can be avoided.

And the autoresponder isn't the only code duplication. Login and signup should be handled by their respective controller actions. By creating methods independent of those actions, you leave the door open to coding errors and omissions. If you want to do things like add session data to the user login

process or block certain ip address from registration, you would again have to do it in 2 separate places. Not good.

Wouldn't it be better to move the sign up and sign in portions of onAuthSuccess into those respective actions? My view is that we should. It will make the actionLogin and actionSignup more complicated, but there is no way to avoid that if we are going to have it all in one method. This approach is entirely optional however and you are free to do as you choose.

Refactor For Maintainability and Extensibility

I decided that since this is for my template, and I plan to use it often for many projects, that I would do some refactoring for both readability and to avoid duplication. In the process, I ended up creating 7 new methods:

- createUser()
- createAuth(\$user)
- findExistingAuth()
- emailPresent()
- matchEmail()
- formatProviderResponse(\$source)
- emailAlreadyInUse()

Some of those methods are just for cosmetic readability and some are to cut down on duplication. I wanted to make it easy to come back in the future and understand the code.

Don't worry, we are going to work on this incrementally, so you will understand it fully.

In order to move the values around to more than one method, we need to set them as class properties.

New Class Properties

We do this by declaring empty properties at the top of the class. Place the following inside the SiteController class:

```
private $attributes = [];
private $username;
private $source;
private $socialUser;
```

Then we modify onAuthSuccess to set and use those properties.

Gist:

[onAuthSuccess Refactor 2](#)

From book:

```
public function onAuthSuccess($client)
{
    $this->attributes = $client->getUserAttributes();

    // $this->attributes['email'] = null;  for example in book.
    // var_dump($this->attributes);
    // die();

    // if provider didn't supply email

    if (!isset($this->attributes['email'])){

        return Yii::$app->getSession()->setFlash('error', [
            Yii::t('app', "Unable to finish, {client} did not provide
            us with an email. Please check your settings on {client}.",
            ['client' => $client->getTitle()]),
        ]);
    }

    $this->source = $client->getId();

    // set $username to correct $attribute from each provider

    switch ($this->source){

        case $this->source == 'facebook' :
            $this->username = 'name';
            break;

        case $this->source == 'github' :
            $this->username = 'login';
            break;

        case $this->source == 'twitter' :
            $this->username = 'screen_name';
            break;
    }
}
```

```
default:

$this->username = 'name';

}

$auth = Auth::find()->where([
    'source' => $this->source,
    'source_id' => $this->attributes['id'],
])->one();

if (Yii::$app->user->isGuest) {

    if ($auth) { // login

        $user = $auth->user;
        Yii::$app->user->login($user);

    } else { // signup

        if (isset($this->attributes['email']))
            && User::find()->where
            ([ 'email' => $this->attributes['email']] )->exists()) {

            return Yii::$app->getSession()->setFlash('error', [
                Yii::t('app', "User with the same email as in {client}
                account already exists but isn't synced. Login with
                username and password and click the {client} sync link
                to sync accounts.", ['client' => $client->getTitle()]),
            ]);

        } else {
            $password = Yii::$app->security->generateRandomString(6);
            $user = new User([
                'username' => $this->attributes[$this->username],
                'email' => $this->attributes['email'],
                'password' => $password,
            ]);
            $user->generateAuthKey();

            // $user->generatePasswordResetToken();
        }
    }
}
```

```

$transaction = $user->getDb()->beginTransaction();

if ($user->save()) {
    $auth = new Auth([
        'user_id' => $user->id,
        'source' => $client->getId(),
        'source_id' => (string)$this->attributes['id'],
    ]);

    if ($auth->save()) {
        $transaction->commit();
        Yii::$app->user->login($user);
        MailCall::onMailableAction('signup', 'site');

    } else {
        return Yii::$app->getSession()->setFlash('error', [
            Yii::t('app', "We were unable to complete the process
            and sync {client}.", ['client' => $client->getTitle()]),
        ]);
    }
}

} else {
    return Yii::$app->getSession()->setFlash('error', [
        Yii::t('app', "We were unable to complete the process
        and sync {client}.", ['client' => $client->getTitle()]),
    ]);
}
}

} else { // user already logged in

if (!$auth && $this->attributes['email'] ==
Yii::$app->user->identity->email) { // add auth provider

$auth = new Auth([
    'user_id' => Yii::$app->user->id,
    'source' => $this->source,
    'source_id' => (string)$this->attributes['id'],
]);
}
}

```

```
$auth->save();

Yii::$app->getSession()->setFlash('success', [
    Yii::t('app', "Your {client} account is
    successfully synced."),
    ['client' => $client->getTitle()],
]);

} else { //emails don't match

    if($this->attributes['email'] != Yii::$app->user->identity->email){

        Yii::$app->getSession()->setFlash('error', [
            Yii::t('app', "Your {client} account could not be synced."),
            ['client' => $client->getTitle()],
        ]);

    } else { // account was already synced

        Yii::$app->getSession()->setFlash('success', [
            Yii::t('app', "Your {client} account is already synced."),
            ['client' => $client->getTitle()],
        ]);

    }
}

}
```

So now you can test that and it should all be good. Making incremental modifications makes it easier to make the changes we need.

New Helper Methods

Let's move on by making the 7 helper methods. These methods will reside in the SiteController class, so we will always have access to them via \$this.

Here is the createUser Method:

Gist:

[createUser](#)

From book:

```
private function createUser()
{
    $password = Yii::$app->security->generateRandomString(6);
    $user = new User([
        'username' => $this->attributes[$this->username],
        'email' => $this->attributes['email'],
        'password' => $password,
    ]);

    $user->generateAuthKey();

    return $user;
}
```

So you can see that we simply chopped out the code from the onAuthSuccess method and made it a private function. We generate a password, create a new instance of User, and generate an auth key, and then return.

Now we can call it like so:

```
$user = $this->createUser();
```

You will see that in action shortly and hopefully you will like what it does to the readability of the code.

Next we have createAuth:

gist:

[createAuth](#)

From book:

```

private function createAuth($user)
{
    $auth = new Auth([
        'user_id' => $user->id,
        'source' => $this->source,
        'source_id' => (string)$this->attributes['id'],
    ]);

    return $auth;
}

```

Again we simply chopped it out of our onAuthSuccess and made it a private function. Since we need to set ‘user_id’ ⇒ \$user->id, we need to hand in an instance of User in the signature.

Next we have findExistingAuth:

Gist:

[findExistingAuth](#)

From book:

```

private function findExistingAuth()
{
    $auth = Auth::find()->where([
        'source' => $this->source,
        'source_id' => $this->attributes['id'],
    ])->one();

    return $auth;
}

```

This one simply finds the existing auth record, if one exists. If it doesn’t exist, it returns null.

Next is emailPresent:

Gist:

[emailPresent](#)

From book:

```
private function emailPresent()
{
    return isset($this->attributes['email']) ? true : false;
}
```

This one just tells us if the provider has passed along the email.

Next we have matchEmail:

Gist:

[matchEmail](#)

From book:

```
private function matchEmail()
{
    return $this->attributes['email'] ==
        Yii::$app->user->identity->email ? true : false;
}
```

Obviously should just be one line. This just tells us if the email returned by the provider matches the one of the current application user.

Next is formatProviderResponse:

Gist:

[formatProviderResponse](#)

```
private function formatProviderResponse($source)
{
    switch ($source){
        case $source == 'facebook' :
            $this->username = 'name';
            break;

        case $source == 'github' :

```

```
$this->username = 'login';
break;

case $source == 'twitter' :

$this->username = 'screen_name';
break;

case $source == 'linkedin' :

$this->username = 'fullName';

$fullName = $this->attributes['first_name'] . ' ' .
$this->attributes['last_name'];

$this->attributes['fullName'] = $fullName;

break;

case $source == 'google' :

$this->username = 'displayName';

$emails = $this->attributes['emails'];

foreach ($emails as $email){

    foreach ($email as $k => $v) {

        if ($k == 'value'){

            $this->attributes['email'] = $v;

        }

    }

}

break;

default:
```

```

    $this->username = 'name';

}

}

```

This method formats the provider's response. As our application grows more sophisticated and we want to work with more attributes, we can format them here. You can also see how easy it is to add another provider, just add another case statement as I did for Google and LinkedIn.

Of course both of the providers made it more complicated to format the response. In the case of LinkedIn, I had to create a name out of first name and last name and I called it fullName. I then set it to the class property, so we could use it later.

In the case of Google, they apparently can give us more than one email, so they did not have a value named email. That meant I had to extract it from a multidimensional array using nested foreach loops.

Next we have emailAlreadyInUse:

Gist:

[emailAlreadyInUse](#)

From book:

```

private function emailAlreadyInUse()
{
    return User::find()
        ->where(['email' => $this->attributes['email']])
        ->exists() ? true : false;
}

```

It simply tells us if the email returned by the provider is already in use by a registered user, we use before syncing to determine if the user should login first or not.

Now as we move on, we will use these methods and you will see how they make the code easier to follow.

OnAuthSuccess Method

Let's start with the onAuthSuccess method, which has been modified drastically:

Gist:

onAuthSuccess

From book:

```
public function onAuthSuccess($client)
{
    $this->attributes = $client->getUserAttributes();

    $this->source = $client->getId();

    $this->formatProviderResponse($this->source);

    if (!(!$this->emailPresent())){

        return Yii::$app->getSession()->setFlash('error', [
            Yii::t('app', "Unable to finish, {source} did not provide us
with an email. Please check your settings on {source}.",
            ['source' => $this->source]),
        ]);
    }

    $existingAuth = $this->findExistingAuth();

    if (Yii::$app->user->isGuest) {

        if ($existingAuth) { // login steps

            $this->socialUser = $existingAuth->user;

            $viaSocial = true;

            $this->actionLogin($viaSocial);

        } else { // signup steps

            $viaSocial = true;

            $this->actionSignup($viaSocial);

        }
    }
}
```

```
    } else { // user already logged in, require email match

        if (!$existingAuth && $this->matchEmail()) { // add auth provider

            $auth = $this->createAuth(Yii::$app->user);

            $auth->save();

            Yii::$app->getSession()->setFlash('success', [
                Yii::t('app', "Your {source} account is successfully synced.",
                    ['source' => $this->source]),
            ]);
        }

    } else { // emails don't match

        if (!$this->matchEmail()){

            Yii::$app->getSession()->setFlash('error', [
                Yii::t('app', "Your {source} account could not be synced.",
                    ['source' => $this->source]),
            ]);
        }

    } else { // account was already synced

        Yii::$app->getSession()->setFlash('success', [
            Yii::t('app', "Your {source} account is already synced.",
                ['source' => $this->source]),
        ]);
    }

}

}

}

} //
```

Yes, it's still a beast, but a lot more readable. Let's look at the first part:

```

public function onAuthSuccess($client)
{
    $this->attributes = $client->getUserAttributes();

    $this->source = $client->getId();

    $this->formatProviderResponse($this->source);

    if (!$this->emailPresent()){

        return Yii::$app->getSession()->setFlash('error', [
            Yii::t('app', "Unable to finish, {source} did not provide us
            with an email. Please check your settings on {source}.",
            ['source' => $this->source]),
        ]);
    }
}

```

So, just like before, we are pulling in the UserAttributes from the \$client object which has been handed in, and we set them to our class property \$this->attributes, which is an array which will hold the values.

Then we set the \$source property from \$client->getId(). This is followed by the formatProviderResponse method, which will look at the \$source and set \$this->username accordingly.

Then comes the if statement to see if the email is present:

```
if (!$this->emailPresent()){
```

That's just way easier to understand, so when I come back to it a year from now, I will save time by being able to comprehend it more quickly.

If that statement evaluates true and we don't have an email, we return a flash message informing the user that they need to check their provider's settings.

Since we have \$this->source available to us as a class property and it has already been set, we are using that instead of \$client->getTitle().

Ok, moving on:

```
$existingAuth = $this->findExistingAuth();
```

This will return null if there is no existing auth record or it will return the right record if it exists. We are going to use this in a check in our next block:

```
if (Yii::$app->user->isGuest) {  
  
    if ($existingAuth) { // login steps  
  
        $this->socialUser = $existingAuth->user;  
  
        $viaSocial = true;  
  
        $this->actionLogin($viaSocial);  
  
    } else { // signup steps  

```

So we check to see if the user is a guest, then test to see if the \$existingAuth evaluates true, if so, we want to login, if not, we want to signup.

Since \$existingAuth is an instance of Auth, and since we have a relation to User in the Auth model, we can access the user via:

```
$existingAuth->user;
```

So we set this user to the class property \$socialUser, so we will have access to it in actionLogin:

```
$this->socialUser = $existingAuth->user;
```

Then we set \$viaSocial to true, which as you'll see in a moment, has meaning in the actionLogin method, which we call:

```
$this->actionLogin($viaSocial);
```

You can see we're handing in \$viaSocial as a parameter.

Action Login

Now let's take a look at the actionLogin method:

Gist:

```
actionLogin
```

From book:

```

public function actionLogin($viaSocial = false)
{
    if (!\Yii::$app->user->isGuest) {
        return $this->goHome();
    }

    if($viaSocial){

        Yii::$app->user->login($this->socialUser);

    } else {

        $model = new LoginForm();
        if ($model->load(Yii::$app->request->post()) && $model->login()) {
            return $this->goBack();
        } else {
            return $this->render('login', [
                'model' => $model,
            ]);
        }
    }

}

```

This actually didn't change too much. We obviously have set \$viaSocial=null as a default in the signature. This allows us to not hand in a value for \$viaSocial when we login in through the normal login form.

Then first thing after checking to see if the user is already logged in, we test for \$viaSocial:

```

if ($viaSocial){

    Yii::$app->user->login($this->socialUser);

} else

```

So if \$viaSocial is true, we login the user by handing in \$this->socialUser into the login method of the user class, which is not the same as actionLogin on the controller. We covered that earlier in the book.

We've already set \$this->socialUser as a class property earlier in the onAuthSuccess method, so it is available to us here.

Everything after the else statement is exactly what was there before, so you can see our login method hasn't changed too drastically.

The actionSignup method, however, is quite a bit bigger.

Action Signup

Gist:

[actionSignup](#)

From book:

```
public function actionSignup($viaSocial=false)
{
    if ($viaSocial){

        if ($this->emailPresent() && $this->emailAlreadyInUse()) {

            return Yii::$app->getSession()->setFlash('error', [
                Yii::t('app', "User with the same email as in {source} account
already exists but isn't synced. Login with username and
password and click the {source} sync link to sync accounts.",
                ['source' => $this->source]),
            ]);

        } else {

            $user = $this->createUser();

            $transaction = $user->getDb()->beginTransaction();

            if ($user->save()) {

                $auth = $this->createAuth($user);

                if ($auth->save()) {

                    $transaction->commit();

                    Yii::$app->user->login($user);

                    MailCall::onMailableAction('signup', 'site');
                }
            }
        }
    }
}
```

```
        } else {

            return Yii::$app->getSession()->setFlash('error', [
                Yii::t('app', "We were unable to complete the process
                and sync {source}.", ['source' => $this->source]),
            ]);

        }

    } else {

        if( User::find()->where(['username' => $this->username])){

            return Yii::$app->getSession()->setFlash('error', [
                Yii::t('app', "Username already taken, please signup
                through the site Signup form and use a different
                username, thanks."),
            ]);

        }

        else {

            return Yii::$app->getSession()->setFlash('error', [
                Yii::t('app', "We were unable to complete the process
                and sync {source}.", ['source' => $this->source]),
            ]);

        }

    }

}

else {

    $model = new SignupForm();

    if ($model->load(Yii::$app->request->post())) {

        if ($user = $model->signup()) {

            if (Yii::$app->getUser()->login($user)) {
```

```

        MailCall::onMailableAction('signup', 'site');

        return $this->goHome();
    }
}

return $this->render('signup', [
    'model' => $model,
]);
}

}

```

Breaking this down into smaller chunks makes it easier to digest. Let's take the first part:

```

public function actionSignup($viaSocial=false)
{
    if ($viaSocial){

        if ($this->emailPresent() && $this->emailAlreadyInUse()) {

            return Yii::$app->getSession()->setFlash('error', [
                Yii::t('app', "User with the same email as in {source} account
already exists but isn't synced. Login with username and
password and click the {source} sync link to sync accounts."),
                ['source' => $this->source]),
            ]);
        } else
    }
}

```

So we use the same technique of setting \$viaSocial to false as a default so that we can use the regular signup form if we wish to without having to pass a value in. Then we test for:

```
if ($this->emailPresent() && $this->emailAlreadyInUse()) {
```

You can see that's easy to understand. If we evaluate true there, we return a flash message, else:

```
$user = $this->createUser();

$transaction = $user->getDb()->beginTransaction();

if ($user->save()) {

    $auth = $this->createAuth($user);

    if ($auth->save()) {

        $transaction->commit();

        Yii::$app->user->login($user);

        MailCall::onMailableAction('signup', 'site');

    } else {

        return Yii::$app->getSession()->setFlash('error', [
            Yii::t('app', "We were unable to complete the process
            and sync {source}.", ['source' => $this->source]),
        ]);
    }
} else {

    if( User::find()->where(['username' => $this->username])){

        return Yii::$app->getSession()->setFlash('error', [
            Yii::t('app', "Username already taken, please signup
            through the site Singup form and use a different
            username, thanks."),
        ]);
    } else {

        return Yii::$app->getSession()->setFlash('error', [
            Yii::t('app', "We were unable to complete the process
            and sync {source}.", ['source' => $this->source]),
        ]);
    }
}
```

```
    }  
}
```

Note in the above, we do a comparison to check and see if the registration is failing due the username already being in use, since validation requires it to be unique. If it is already in use, the user gets a nicely formatted message instructing them on what to do.

We start the else statement by creating a user via our handy createUser method:

```
$user = $this->createUser();
```

Then we start our transaction:

```
$transaction = $user->getDb()->beginTransaction();  
  
if ($user->save()) {  
  
    $auth = $this->createAuth($user);  
  
    if ($auth->save()) {  
  
        $transaction->commit();  
    }  
}
```

You can see we use our createAuth method with \$user handed in, so the code here is a bit more concise than it would otherwise be.

We follow that with 2 else statements that return a flash message if something goes wrong and the transaction does not complete.

That brings us the final else statement of the method, and that is simply everything we had before we made the change, so we can either process form input or render the form.

Since we're in the signup action for example, you can see that now the following call makes sense:

```
MailCall::onMailableAction('signup', 'site');
```

That was a lot of work for a small effect. However, imagine that you are also doing other things nearby:

```
// save user data to session
// log login time and ip address to db

MailCall::onMailableAction('signup', 'site');
```

Those are just a couple of examples as comments only, but you get the idea. As your application grows, you will want a single centralized method to add these requirements to.

Ok, that was a huge, but necessary detour. Now we are ready to return to onAuthSuccess. If we are not signing up or logging in, then we are already logged in:

```
else { // user already logged in, require email match

    if (!$existingAuth && $this->matchEmail()) { // add auth provider

        $auth = $this->createAuth(Yii::$app->user);

        $auth->save();

        Yii::$app->getSession()->setFlash('success', [
            Yii::t('app', "Your {source} account is successfully synced.", [
                ['source' => $this->source]
            ]),
        ]);
    }
}
```

We check to see if there is no \$existingAuth and that the email matches. If we're good, we createAuth, handing in the currently logged in user and save. We send them a nice confirmation message.

If we're not good, we have the else statement:

```
else { //emails don't match

    if (!$this->matchEmail()){

        Yii::$app->getSession()->setFlash('error', [
            Yii::t('app', "Your {source} account could not be synced.", [
                ['source' => $this->source]
            ]),
        ]);
    }

    } else { // account was already synced

        Yii::$app->getSession()->setFlash('success', [
```

```

        Yii::t('app', "Your {source} account is already synced.", [
            'source' => $this->source]),
    ]);

}

}

```

So now we are just handling if the emails don't match or if the account is already synced. Some people just love pushing a button just to see what happens, so we have to handle that scenario.

Now all of that should be working and testable. If you go over the code, you will find it much more readable, and therefore, it should be more maintainable.

So one negative effect of all this is that our site controller is a lot more crowded with code than it used to be. The inclusion of social auth required 8 new methods and a rewrite of two existing methods.

For my taste, this is a little too much in one controller, if such a thing can be avoided. So I decided to move actionIndex, actionContact and actionAbout to a new controller, which I'm going to name PagesController.

Any additional static pages, such as terms of service, privacy, etc. will be controlled by the PagesController.

Pages Controller

I'm not going to cover every detail in this modification to the template because you already know how to do most of it. However I will make a list of steps, along with any key details, so that you don't miss anything and I also provide Gists at the end of the chapter.

Steps to implement:

1. Create frontend pages controller via Gii with index, about, and contact actions.
2. Include the following on the pages controller:

```

namespace frontend\controllers;

use Yii;
use frontend\models\ContactForm;
use yii\filters\AccessControl;

```

1. Move index view, about view, and contact view from site views folder to pages views folder.
2. Change facebook button link on pages index to point to 'site/auth'.
3. Configure captcha correctly. You will need to put in the following behaviors method on the pages controller:

```

public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'only' => ['captcha'],
            'rules' => [
                [
                    'actions' => ['captcha'],
                    'allow' => true,
                    'roles' => ['?', '@'],
                ],
            ],
        ],
    ];
}

```

Also, specify captcha action on ContactForm model in rules:

```
[ 'verifyCode', 'captcha', 'captchaAction' => 'pages/captcha' ],
```

And also specify it in the widget in the form view:

```

<?= $form->field($model, 'verifyCode')->
    widget(Captcha::className(), [
        'captchaAction' => 'pages/captcha',
        'template' => '<div class="row">
            <div class="col-lg-3">{image}</div>
            <div class="col-lg-6">{input}</div></div>',
    ]) ?>

```

1. Configure default controller/action for application because the new site controller will have the following actions when the change is done:

- actions
- login
- logout
- signup
- RequestPasswordReset

- ResetPassword

Additionally, it has the following methods:

- behaviors
- onAuthSuccess
- createUser
- createAuth
- findExistingAuth
- emailPresent
- matchEmail
- formatProviderResponse
- emailAlreadyInUse

Obviously, if we removed index from the site controller, the application will need to know where to send users. We have some choices. We need to set the following:

```
'defaultRoute' => 'pages/index',
```

We can put that either in common/config/main.php or in frontend/config/main.php. For my template, I chose to place it in frontend/config/main, like so:

...

```
'bootstrap' => ['log'],
'controllerNamespace' => 'frontend\controllers',
'defaultRoute' => 'pages/index',
'components' => [
    'user' => [
```

...

Just pop it in between controllerNamespace and components.

The backend application has different requirements, since we are enforcing admin level access and there aren't any static pages. So I'm going to keep that as is.

Then the question is, do we want to include social auth on the backend? This is a security issue. It means if you left a computer or device unattended and were logged into Facebook for example, anyone jumping on the computer would be able to gain access to your site's admin area. This could actually be quite devastating if the wrong person were to gain access.

So you have to weigh the convenience vs. the potential security issue. In my case, I chose not to implement social auth on the backend. That is subject to change of course, but that's how I'm doing it for now.

Ok, so for reference, and in case you need it for debugging, I'm going to provide the Gists for the new site controller and pages controller.

Gist:

[Site Controller](#)

Gist:

[Pages Controller](#)

Summary

We implemented one-click Facebook login and registration, covering every scenario we could think of. We used Yii 2's authclient extension, the authclient widget, and made significant changes to the site controller.

It was important for us to make sure `actionLogin` and `actionSignup` on the site controller would be maintained properly, so we had to integrate our social auth into those methods as well.

Then finally we moved the rest of the site controller actions, like about, contact, privacy, etc. to a new controller, pages. Our pages controller will take care of all the site pages that don't otherwise have their own controllers, including index.

We decided that in our application, we would not make the same changes to the backend application. This is a personal choice and you may choose to do otherwise if you wish.

Once again I'd like to thank all the readers for their contributions and suggestions, help in finding typos and bugs, and of course for positive reviews and recommendations.

Please feel free to contact me at leanpub.com through the contact the author link:

[Email Bill Keck](#)

Thanks again for supporting the book.

Chapter 15: Template Migration guide.

On March 15, 2015, I released a new version of this book that included a heavily refactored version of the helper classes and a change to the relationships in the User, Role, Status, and UserType models.

If you purchased the book after the above date, you can ignore this chapter entirely.

For those who have purchased the book prior to this date and would like to update to the newer template, you can either start from chapter 5 and rebuild everything or follow the instructions here. I will provide a list of changed files and their Gists, so you can easily replace what you need.

I highly recommend using the newer code, this will keep your template current with all the bonus material, both existing bonus chapters and future chapters.

If you decide to rebuild from chapter 5, you will benefit from the additional material that I've added concerning using Active Record and it will make things clearer, so there are some good reasons for doing that.

Not every chapter is different. Mostly the changes are in chapters 5,6,7,11, and 12. A lot of the changes resulted from a relationship change. Previously, we had the following relationship:

```
public function getRole()
{
    return $this->hasOne(Role::className(), ['role_value' => 'role_id']);
}
```

Although this worked and did not create problems in the code, a number of programmers were confused by it, since the more typical relationship would look like:

```
public function getRole()
{
    return $this->hasOne(Role::className(), ['id' => 'role_id']);
}
```

After receiving a letter from a University student in Germany asking about the relationship, I decided to review the code with some ninja programmers that I know. They too were uncomfortable with the existing relationship.

So I realized that this would impact maintainability of the template because it's not as intuitive as it should be. It all worked fine, but was probably not the best practice. And because I'm building this template for long term use, I couldn't ignore this fact.

Rather than be a negative, the situation produced a number of upsides. First, we get cleaner, more intuitive code, which makes the template stronger.

It also gave me a chance to revisit old code and an opportunity to go deeper into Yii 2's Active Record implementation, which is very powerful and made the rewrite enjoyable. I got to demonstrate how easy Yii 2 makes queries, whereas before I was simply relying on raw SQL.

The following models are changed:

User

Gist:

[New User Model](#)

Note: This changes assumes you have a status record in your DB with a `status_name` of 'Active' and is also dependent on you making the rest of the changes in this chapter. This version of User is used from chapter 7 on. If you want the earlier version, please refer to the Gist at the end of chapter 5.

Role

Gist:

[New Role Model](#)

UserType:

Gist:

[New UserType Model](#)

Status

Gist:

[New Status Model](#)

ValueHelpers

Gist:

[New ValueHelpers Class](#)

PermissionHelpers

Gist:

[New PermissionHelpers Class](#)

RecordHelpers

Gist:

[New RecordHelpers Class](#)

Database Changes

We need to change the defaults on the user table. Also, in the DB itself, the columns role_id, user_type_id, and status should default to 1.

This assumes that in your DB, the first record with an id of 1 in your role table is role_name ‘User.’ Also, in the status table, the first record with an id of 1, is ‘Active.’ And finally in the user_type table, the first record with an id of 1, is ‘Free.’

Alternatively, you can drop the defaults in the db, and in the rules method, use:

```
[ 'status_id', 'default', 'value' => ValueHelpers::getStatusId('Active')],  
[ 'role_id', 'default', 'value' => ValueHelpers::getRoleId('User')],  
[ 'user_type_id', 'default', 'value' => ValueHelpers::getUserTypeId('Free')],
```

Note that using the ValueHelpers method to set the default for status is something we do in chapter 7, which allows us to remove the constant at the top of the User model. Again make sure you have a record with a name of ‘Active’ in the status table before making this change.

Extra ValueHelpers

I don’t supply the getUserId and getRoleId methods out of the box, so you will have to add to ValueHelpers, if you wish to remove the hardcoded defaults from the db:

Gist:

[Extra ValueHelpers](#)

From book:

```

public static function getRoleId($role_name)
{
    $role = Role::find('id')
        ->where(['role_name' => $role_name])
        ->one();

    return isset($role->id) ? $role->id : false;
}

public static function getUserId($user_type_name)
{
    $userType = UserType::find('id')
        ->where(['user_type_name' => $user_type_name])
        ->one();

    return isset($userType->id) ? $userType->id : false;
}

```

It might be a good idea to add the above methods just to have them, even if you don't immediately need them.

LoginForm Model

In chapter 7, we change the loginAdmin method on the LoginForm model located in common/models to the following:

Gist:

[New LoginForm Model](#)

PasswordResetRequestForm

We also made changes to the frontend/models/PasswordResetRequestForm.php:

Gist:

[New PasswordResetRequestForm Model](#)

In chapter 11, we make a correction to the UserSearch model and the ProfileSearch model, which corrects a bug.

UserSearch

Gist:

[UserSearch Model](#)

ProfileSearch

Gist:

[ProfileSearch Model](#)

Also, in chapter 11, we make changes to:

backend/views/layouts/main.php

and to:

backend/views/site/index.php.

The versions I'm going to provide here are the final versions as of chapter 14. If you wish to pull in the versions from chapter 11, please refer to the gists in that chapter.

Main.php

Gist:

[Main.php](#)

Note: This is the current version as of chapter 14. If you want the earlier version, refer to chapter 11.

Gist:

Index.php

[Index.php](#)

Note: This is the current version as of chapter 14. If you want the earlier version, refer to chapter 11.

Troubleshooting

1. Errors due to method not found. This could be a namespace problem, so make sure you have your use statements that are necessary for the class. It could also come from referencing a method whose name has been changed.

2. Errors due to missing records from DB. In order for the application to operate correctly, you need records in the role, status, and user_type tables as outlined in the instructions above.
3. Errors due to view permissions. Since we changed the backend layout/main and site/index views, these changes will need to be in place before you can use the backend.

It's entirely possible that I missed something, please contact me if that is the case. You can email me at:

[Email Bill Keck](#)

The good news is that I know the template works and the code has gone directly from my IDE to the Gists, so you should be able to correct anything by following the instructions in the book.

Summary

I can't tell you how many technical books I've read with broken useless code, but it's a lot. It's such a waste of a programmer's time. I'm committed to making this book different. The template we build in this book is a long-term project that will evolve over time, especially as I add more bonus material.

That's why I made this chapter to insure that all readers of the book have an easy way to keep the template up-to-date. It's important to me that you get maximum value out of the book and that we achieve a high level of excellence with this work.

Thanks again for sharing this journey with me.

Chapter 16: Images and File Uploads

Welcome back to another bonus chapter. I know it's kind of crazy if you are reading the chapters successively to constantly welcome you back, but keep in mind that I'm creating these additions and publishing them after the core book has been finished, one chapter at a time.

I want to keep going in this style because it feels more like the actual journey that I'm taking with Yii 2 and in building the template, sort of a blog style approach. I hope you enjoy it too.

Ok, so let's jump into the material for this chapter, which is all about uploading and managing images. The scenario that we are going to work with ties directly to the template, but you can use the skills you learn in the chapter to cover a wide range of photo or file management, including user content.

We don't cover user content directly, however, because it is not specific to the template. So instead, we are going to develop a MarketingImage model that will allow us to manage our marketing images.

We will use our marketing images to populate a carousel widget on the frontend index page. A carousel on the index page is such a common feature in sites these days that I thought it made a lot of sense to include one in the template.

If for some reason the client doesn't want a carousel, then it will be easy to remove it, and that is because as widget, it will simply be one line of code.

In addition to having the carousel display the images, we will also have full control over creating, updating, and deleting the files, both as model instances and as actual files on the server in our admin area.

And with the weight attribute, they will be able to control the order of the images, regardless of what order they are created in.

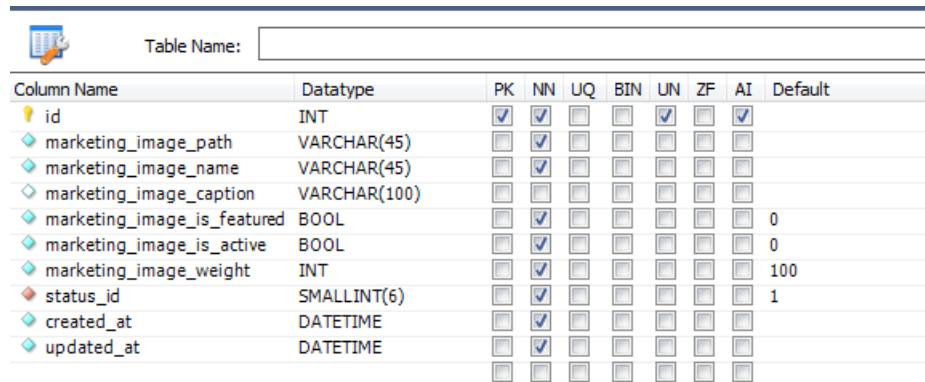
This will enable you to give full control of the marketing images over to the client, and they can finesse their carousel exactly how they want it.

The Uploads Folder

Let's start our work by creating an uploads folder(use lowercase), located within our backend web folder.

Marketing Image Table

Let's start by defining our marketing_image table.



The screenshot shows a table definition window in MySQL Workbench. The table name is 'marketing_image'. The columns and their properties are:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0
marketing_image_path	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
marketing_image_name	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
marketing_image_caption	VARCHAR(100)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
marketing_image_is_featured	BOOL	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0
marketing_image_is_active	BOOL	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0
marketing_image_weight	INT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	100
status_id	SMALLINT(6)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	1
created_at	DATETIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
updated_at	DATETIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Marketing Images Table

Marketing Image SQL

Here is the SQL

Gist:

[Marketing Image SQL](#)

From book:

```
CREATE TABLE IF NOT EXISTS `yii2build`.`marketing_image` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `marketing_image_path` VARCHAR(45) CHARACTER SET 'utf8'
    COLLATE 'utf8_unicode_ci' NOT NULL,
  `marketing_image_name` VARCHAR(45) CHARACTER SET 'utf8'
    COLLATE 'utf8_unicode_ci' NOT NULL,
  `marketing_image_caption` VARCHAR(100) CHARACTER SET 'utf8'
    COLLATE 'utf8_unicode_ci' NULL DEFAULT NULL,
  `marketing_image_is_featured` TINYINT(1) NOT NULL DEFAULT 0,
  `marketing_image_is_active` TINYINT(1) NOT NULL DEFAULT 0,
  `marketing_image_weight` INT(11) NOT NULL DEFAULT 100,
  `status_id` SMALLINT(6) NOT NULL DEFAULT 1,
  `created_at` DATETIME NOT NULL,
  `updated_at` DATETIME NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_marketing_image_status1_idx` (`status_id` ASC),
  CONSTRAINT `fk_marketing_image_status1`
    FOREIGN KEY (`status_id`)
    REFERENCES `yii2build`.`status` (`id`)
```

```
ON DELETE NO ACTION  
ON UPDATE NO ACTION)
```

Note that we're sticking with the convention of singular for the table name. You can also see that we have a foreign key, status_id pointing at id on the status table. Because there are so few records in the status table, I didn't bother making id unsigned, so that's why status_id is not unsigned as well.

A couple of other notes about the data structure. We are using marketing_image_is_active, set as a boolean, to determine if the image should be the active item in the carousel. The marketing_image_caption will be displayed in the carousel.

We are using status_id to determine the status of the image because the status table holds the values active, pending, and retired. So this gives us a chance to reuse existing data structure and avoid data duplication.

This data structure is for our template, but you are absolutely free to change it if you have other requirements. If you want more options to designate different types of images, feel free to do so, but personally, I would do so after working through the rest of the chapter, so you know how it is supposed to work first.

Marketing Image Model

The next step is to build our model using Gii. We're going to locate this model in the backend/models folder, since working with the images will be an admin's job.

Obviously we're setting the namespace to:

```
backend\models;
```

By this point, you should be familiar enough with Gii for me not to have to repeat all the instructions. So I will just assume you have correctly created the MarketingImage model.

Next we need to create the CRUD. Since we already have our search folder in backend/models, we don't need to create it.

So go ahead and create the CRUD with Gii.

Don't you just love this workflow? Every time I use it I'm reminded about how much I enjoy working with Yii 2.

Modify MarketingImage Model

Ok, so now that we got our basic model/crud setup, we need to modify our model.

Gist:

Modified MarketingImage Model

From book:

```
<?php

namespace backend\models;

use Yii;
use yii\db\ActiveRecord;
use yii\db\Expression;
use yii\helpers\ArrayHelper;
use yii\behaviors\TimestampBehavior;

/**
 * This is the model class for table "marketing_image".
 *
 * @property string $id
 * @property string $marketing_image_path
 * @property string $marketing_image_name
 * @property integer $marketing_image_is_featured
 * @property integer $marketing_image_is_active
 * @property integer $status_id
 * @property string $created_at
 * @property string $updated_at
 *
 * @property Status $status
 */
class MarketingImage extends \yii\db\ActiveRecord
{

    public $file;

    /**
     * @inheritdoc
     */

    public static function tableName()
    {
        return 'marketing_image';
    }
}
```

```
public function behaviors()
{
    return [
        'timestamp' => [
            'class' => TimeStampBehavior::className(),
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
                ],
            ],
        'value' => new Expression('NOW()'),
        ],
    ];
}

/**
 * @inheritDoc
 */

public function rules()
{
    return [
        [['marketing_image_path', 'marketing_image_name',
            'marketing_image_weight', 'file'], 'required'],

        ['marketing_image_weight', 'default', 'value' => 100 ],
        ['marketing_image_is_featured', 'default', 'value' => 0 ],
        ['marketing_image_is_active', 'default', 'value' => 0 ],

        [['marketing_image_is_featured'], 'in',
            'range'=>array_keys($this->getMarketingImageIsFeaturedList())],
        [['marketing_image_is_active'], 'in',
            'range'=>array_keys($this->getMarketingImageIsActiveList())],

        [['marketing_image_name', 'marketing_image_path'], 'trim'],
        [['marketing_image_caption'], 'string', 'max' => 100],
        [['marketing_image_is_featured', 'marketing_image_is_active',
            'marketing_image_weight', 'status_id'], 'integer'],
        [['file'], 'file', 'extensions' => ['png', 'jpg', 'gif'],
            'maxSize' => 1024*1024],
        [['marketing_image_path', 'marketing_image_name']],
    ];
}
```

```
        'string', 'max' => 45]
    ];
}

public function beforeValidate()
{
    $this->marketing_image_name = preg_replace('/\s+/', '',
    $this->marketing_image_name);

    $this->marketing_image_path = preg_replace('/\s+/', '',
    $this->marketing_image_path);

    return parent::beforeValidate();
}

/**
 * @inheritDoc
 */

public function attributeLabels()
{
    return [
        'id' => 'ID',
        'marketing_image_path' => 'Marketing Image Path',
        'marketing_image_name' => 'Marketing Image Name',
        'marketing_image_caption' => 'Caption',
        'marketing_image_is_featured' => 'Marketing Image Is Featured',
        'marketing_image_is_active' => 'Marketing Image Is Active',
        'marketing_image_weight' => 'Marketing Image Weight',
        'status_id' => 'Status ID',
        'created_at' => 'Created At',
        'updated_at' => 'Updated At',
        'file' => 'Marketing Image'
    ];
}

public static function getMarketingImageIsFeaturedList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}
```

```
public static function getMarketingImageIsActiveList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}

/**
 * @return \yii\db\ActiveQuery
 */

public function getStatus()
{
    return $this->hasOne(Status::className(), ['id' => 'status_id']);
}

/**
 * * get status name
 *
 */

public function getStatusName()
{
    return $this->status ? $this->status->status_name : '- no status -';
}

/**
 * get list of statuses for dropdown
 */

public static function getStatusList()
{
    $droptions = Status::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'status_name');
}

}
```

We're going to step this rather quickly and just talk about what's different from the boiler plate. Let's start with the use statements:

```
use Yii;
use yii\db\ActiveRecord;
use yii\db\Expression;
use yii\helpers\ArrayHelper;
use yii\behaviors\TimestampBehavior;
```

We pulled in the necessary classes to support TimestampBehavior and the ArrayHelper for our dropdown lists.

Next we added a public property:

```
public $file;
```

The reason we have to add it like this is because the model can't pull it from the table via reflection like it does all the other properties. In this case, \$file will hold the value of the file, not a db table value.

Next we added our timestamp behavior in the behaviors method:

```
public function behaviors()
{
    return [
        'timestamp' => [
            'class' => TimeStampBehavior::className(),
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
            ],
            'value' => new Expression('NOW()'),
        ],
    ];
}
```

We've done this a number of times, so not much to say there.

Now we move onto the rules:

```

public function rules()
{
    return [
        [['marketing_image_path', 'marketing_image_name',
            'marketing_image_weight', 'file'], 'required',

        ['marketing_image_weight', 'default', 'value' => 100 ],
        ['marketing_image_is_featured', 'default', 'value' => 0 ],
        ['marketing_image_is_active', 'default', 'value' => 0 ],

        [['marketing_image_is_featured'], 'in',
            'range'=>array_keys($this->getMarketingImageIsFeaturedList())],
        [['marketing_image_is_active'], 'in',
            'range'=>array_keys($this->getMarketingImageIsActiveList())],

        [['marketing_image_name', 'marketing_image_path'], 'trim'],
        [['marketing_image_caption']], 'string', 'max' => 100],
        [['marketing_image_is_featured', 'marketing_image_is_active',
            'marketing_image_weight', 'status_id']], 'integer'],
        [['file']], 'file', 'extensions' => ['png', 'jpg', 'gif'],
        'maxSize' => 1024*1024],
        [['marketing_image_path', 'marketing_image_name'],
            'string', 'max' => 45]
    ];
}

```

We are using a couple of validators that we haven't used before. the 'trim' validator gets rid of white space, but not space in between words. We will write a beforeValidation method for that, since we want to strictly enforce filenames.

You can also see that we added a validation rule for file:

```

[['file']], 'file', 'extensions' => ['png', 'jpg', 'gif'],
'maxSize' => 1024*1024],

```

This sets the allowable types and max size of the file, very simple stuff.

Unfortunately, when I tried to save a photo the first time, I got a nasty error about PHP fileInfo not being enabled.

PHP FileInfo

For whatever reason, my PHP ini file had it commented out, so I had to get rid of the semicolon:

```
; If you only provide the name
; default extension directory.

; Windows Extensions
; Note that ODBC support is bu
; Note that many DLL files are
; extension folders as well as
; Be sure to appropriately set

extension=php_bz2.dll
extension=php_curl.dll
extension=php_mbstring.dll
extension=php_exif.dll
extension=php_fileinfo.dll
extension=php_gd2.dll
extension=php_gettext.dll
;extension=php_gmp.dll
extension=php_intl.dll
;extension=php_imap.dll
;extension=php_interbase.dll
;extension=php_ldap.dll
;extension=php_mssql.dll
;extension=php_mbstring.dll
;extension=php_exif.dll
```

PHP FileInfo

You can see it in the highlighted line above. If you do need to make this change, also remember to restart Apache so the change takes effect.

Ok, moving on. Let's look at our beforeValidate method:

```
public function beforeValidate()
{
    $this->marketing_image_name =
        preg_replace('/\s+/', ' ', $this->marketing_image_name);

    $this->marketing_image_path =
        preg_replace('/\s+/', ' ', $this->marketing_image_path);

    return parent::beforeValidate();
}
```

We're doing formatting here to remove spaces from the marketing_image_path and the marketing_image_name. On the controller, we will do the same for the actual file that's being saved to the server. That way we don't end up with filenames with spaces in them.

Obviously we're just using a preg_replace to replace a regular expression:

```
/\s+/
```

That gets rid of the spaces between words.

Next we add a label for file and getStatusName in our attribute labels method:

```
'file' => 'Marketing Image',
'statusName' => Yii::t('app', 'Status'),
```

That will show nicely on the form, when we modify it to accept a file upload. Also note, we use the magic name for getStatusName.

Next we have a couple of methods that format the values for the dropdown lists we will need in our form:

```
public static function getMarketingImageIsFeaturedList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}

public static function getMarketingImageIsActiveList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}
```

And finally, we added 2 new status relationships, so we can use status as a dropdown list as well:

```
public function getStatusName()
{
    return $this->status ? $this->status->status_name : '- no status -';
}

/**
 * get list of statuses for dropdown
 */

public static function getStatusList()
{
    $droptions = Status::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'status_name');
}
```

We did not need to add getStatus because that was autogenerated for us by Gii due to the foreign Key definition.

Modify MarketingImage Search Model

Ok, so now we need to modify our search model. There's nothing new in this file, we are just using the other search models as an example. That said, I find when I'm doing this that I always seem to trip over this part, so after giving you the file, I will point out some sticking points.

Gist:

[MarketingImage Search Model](#)

From book:

```
<?php

namespace backend\models\search;

use Yii;
use yii\base\Model;
use yii\data\ActiveDataProvider;
use backend\models\MarketingImage;

class MarketingImageSearch extends MarketingImage
{
    public $statusName;

    /**
     * @inheritdoc
     */

    public function rules()
    {
        return [
            [['id', 'marketing_image_is_featured',
                'marketing_image_is_active', 'status_id'], 'integer'],
            [['marketing_image_path', 'marketing_image_name',
                'marketing_image_caption', 'marketing_image_weight',
                'created_at', 'statusName',
                'updated_at'], 'safe'],
        ];
    }

    /**
     * @inheritdoc
     */
}
```

```
public function scenarios()
{
    // bypass scenarios() implementation in the parent class
    return Model::scenarios();
}

/**
 * Creates data provider instance with search query applied
 *
 * @param array $params
 *
 * @return ActiveDataProvider
 */
public function search($params)
{
    $query = MarketingImage::find();

    $dataProvider = new ActiveDataProvider([
        'query' => $query,
    ]);

    $dataProvider->setSort([
        'attributes' => [
            'id',
            'marketing_image_name',
            'marketing_image_path',
            'marketing_image_caption',
            'marketing_image_is_featured',
            'marketing_image_is_active',
            'marketing_image_weight',
            'statusName' => [
                'asc' => ['status.status_name' => SORT_ASC],
                'desc' => ['status.status_name' => SORT_DESC],
                'label' => 'Status'
            ],
        ],
    ]);
}
```

```
if (!($this->load($params) && $this->validate())) {  
  
    $query->joinWith(['status']);  
  
    return $dataProvider;  
}  
  
$this->addSearchParameter($query, 'id');  
$this->addSearchParameter($query, 'marketing_image_name', true);  
$this->addSearchParameter($query, 'marketing_image_path', true);  
$this->addSearchParameter($query, 'marketing_image_caption', true);  
$this->addSearchParameter($query, 'marketing_image_is_featured');  
$this->addSearchParameter($query, 'marketing_image_is_active');  
$this->addSearchParameter($query, 'marketing_image_weight');  
$this->addSearchParameter($query, 'status_id');  
  
// filter by gender name  
  
$query->joinWith(['status' => function ($q) {  
  
    $q->andFilterWhere(['=', 'status.status_name', $this->statusName]);  
  
}]);  
  
return $dataProvider;  
}  
  
protected function addSearchParameter($query, $attribute, $partialMatch = false)  
{  
    if (($pos = strpos($attribute, '.')) !== false) {  
        $modelAttribute = substr($attribute, $pos + 1);  
    } else {  
        $modelAttribute = $attribute;  
    }  
  
    $value = $this->$modelAttribute;  
    if (trim($value) === '') {  
        return;  
    }  
}
```

```
/*
 * The following line is additionally added for right aliasing
 * of columns so filtering happen correctly in the self join
 */

$attribute = "marketing_image.$attribute";

if ($partialMatch) {
    $query->andWhere(['like', $attribute, $value]);
} else {
    $query->andWhere([$attribute => $value]);
}

}

}
```

So the sticking points are as follows:

- Make sure to set a property for the magic call to relation public \$statusName
- Make sure to set a rule for \$statusName
- Make sure \$attribute = “marketing_image.\$attribute”; is set correctly
- Make sure the correct attributes have the true option on addSearchParameter
- Don’t expect sort on status to work until we modify gridview in index

Hopefully those tips will save you time.

Modify Index View

Now we’ll work on the views. We will start with index and finish with _form, so we have a perfect transition into the file upload process and the controller.

Here is the modification for index.php:

Gist:

[Index.php](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\grid\GridView;
use \yii\bootstrap\Collapse;

/* @var $this yii\web\View */
/* @var $searchModel backend\models\search\MarketingImageSearch */
/* @var $dataProvider yii\data\ActiveDataProvider */

$this->title = 'Marketing Images';
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="marketing-image-index">

    <h1><?= Html::encode($this->title) ?></h1>
    <?php echo Collapse::widget([
        'encodeLabels' => false,
        'items' => [
            // equivalent to the above
            [
                'label' => '<i class="fa fa-caret-square-o-down"></i> Search',
                'content' => $this->render('_search', ['model' => $searchModel]),
                // open its content by default
                // 'contentOptions' => ['class' => 'in']
            ],
            // 'encodeLabels' => false,
        ]
    ]);
?>

    <p>
        <?= Html::a('Create Marketing Image', ['create'],
            ['class' => 'btn btn-success']) ?>
```

```

</p>

<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        'id',
        'marketing_image_path',
        'marketing_image_name',
        'marketing_image_caption',
        ['attribute'=>'marketing_image_is_featured', 'format'=>'boolean'],
        ['attribute'=>'marketing_image_is_active', 'format'=>'boolean'],
        'marketing_image_weight',
        'statusName',
        // 'created_at',
        // 'updated_at',

        ['class' => 'yii\grid\ActionColumn'],
    ],
]); ?>

</div>

```

So by now this should look familiar. We put the search partial inside a collapse widget. One thing new there is that we set:

```
'encodeLabels' => false,
```

This allows us to put a font-awesome icon in the label:

```
'label' => '<i class="fa fa-caret-square-o-down"></i> Search',
```

This adds a nice touch to the UI. So I went around the application to all the places where I could make that change to be consistent. Here is the list:

- user index
- profile index
- faq index
- faq category Index

The other index pages in the backend don't have the search partial visible. However if you wish to change that, you can see how easy it would be.

Modify View

Ok, moving on. The only other thing to note in this view is that we are using \$statusName as one of the attributes in the GridView widget and this gives us the eager-loaded sortable attribute.

Let's look at the modified view file:

Gist:

[view.php](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\DetailView;

/* @var $this yii\web\View */
/* @var $model backend\models\MarketingImage */

$this->title = $model->id;

$this->params['breadcrumbs'][] =
['label' => 'Marketing Images', 'url' => ['index']];

$this->params['breadcrumbs'][] = $this->title;
?>
<div class="marketing-image-view">

    <p>
        <?= Html::a('Update', ['update', 'id' => $model->id],
                    ['class' => 'btn btn-primary']) ?>
        <?= Html::a('Delete', ['delete', 'id' => $model->id], [
                    'class' => 'btn btn-danger',
                    'data' => [
                        'confirm' => 'Are you sure you want to delete this item?',
                        'method' => 'post',
                    ],
                ]) ?>
    </p>

    <h1><?= Html::encode($model->marketing_image_name) ?></h1>
    <br>
```

```
<div>
<?php

echo Html::img('/'. $model->marketing_image_path . '?'. 'time()',
    ['width' => '600px']);

?>

</div>
<br>

<?= DetailView::widget([
    'model' => $model,
    'attributes' => [
        'id',
        'marketing_image_caption',
        'marketing_image_path',
        //marketing_image_name',
        ['attribute' => 'marketing_image_is_featured',
            'format' => 'boolean'],
        ['attribute' => 'marketing_image_is_active',
            'format' => 'boolean'],
        'marketing_image_weight',
        'status.status_name',
        'created_at',
        'updated_at',
    ],
]) ?>

</div>
```

A couple of minor changes in the DetailView widget and in the H1. Notice the use of:

```
echo Html::img('/'. $model->marketing_image_path . '?'. 'time()',
    ['width' => '600px']);
```

We are using the Html::img helper to provide the path, then appending a get variable ?time=the-current-time. The reason we are appending the get variable is to prevent caching, so we always see the correct image.

Sometimes where you update the image, you might see the old image if you have it cached in your browser. Finally, we are enforcing a width to the image, so that we can have all the images set at a certain size.

For our carousel, we might need a specific size, and I will come back to this if that is the case.

Obviously, our view can't display an image yet because we haven't uploaded any, but we will be doing that shortly.

Modify Update View

The only change we are making to the update.php view is that we are also echoing the image there for reference.

Gist:

[update.php](#)

From book:

```
<?php

use yii\helpers\Html;

/* @var $this yii\web\View */
/* @var $model backend\models\MarketingImage */

$this->title = 'Update Marketing Image: ' . ' ' . $model->id;
$this->params['breadcrumbs'][] =
['label' => 'Marketing Images', 'url' => ['index']];
$this->params['breadcrumbs'][] =
['label' => $model->id, 'url' => ['view', 'id' => $model->id]];
$this->params['breadcrumbs'][] = 'Update';
?>
<div class="marketing-image-update">

<h1><?= Html::encode($this->title) ?></h1>

<br>
<div>
<?php
echo Html::img('/' . $model->marketing_image_path,
['width' => '600px']);

?>
</div>
<br>
```

```
<?= $this->render('_form', [  
    'model' => $model,  
) ?>  
  
</div>
```

There is no change to the create view.

Modify _search Partial

Gist:

[_search view](#)

From book:

```
<?php  
  
use yii\helpers\Html;  
use yii\widgets\ActiveForm;  
  
/* @var $this yii\web\View */  
/* @var $model backend\models\search\MarketingImageSearch */  
/* @var $form yii\widgets\ActiveForm */  
?  
  
<div class="marketing-image-search">  
  
    <?php $form = ActiveForm::begin([  
        'action' => ['index'],  
        'method' => 'get',  
    ]); ?>  
  
    <?= $form->field($model, 'id') ?>  
  
    <?= $form->field($model, 'marketing_image_path') ?>  
  
    <?= $form->field($model, 'marketing_image_name') ?>  
  
    <?= $form->field($model, 'marketing_image_caption') ?>
```

```
<?= $form->field($model, 'marketing_image_is_featured')
    ->dropDownList($model->marketingImageIsFeaturedList,
    ['prompt' => 'Please Choose One']);?>

<?= $form->field($model, 'marketing_image_is_active')
    ->dropDownList($model->marketingImageIsActiveList,
    ['prompt' => 'Please Choose One']);?>

<?= $form->field($model, 'status_id')->dropDownList($model->statusList,
    ['prompt' => 'Please Choose One' ]);?>

<?php // echo $form->field($model, 'created_at') ?>

<?php // echo $form->field($model, 'updated_at') ?>

<div class="form-group">
    <?= Html::submitButton('Search', ['class' => 'btn btn-primary']) ?>
    <?= Html::resetButton('Reset', ['class' => 'btn btn-default']) ?>
</div>

<?php ActiveForm::end(); ?>

</div>
```

You can see we are using model methods to populate the dropdown lists. We did not include marketing_image_weight on the search form, but obviously you can if you want to.

Modify _form Partial

Gist:

_form Partial

From book:

```
use yii\widgets\ActiveForm;
use yii\helpers\Html;

/* @var $this yii\web\View */
/* @var $model backend\models\MarketingImage */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="marketing-image-form">

    <?php $form = ActiveForm::begin(['options'=>
        ['enctype' => 'multipart/form-data']])); ?>

    <?= $form->field($model, 'marketing_image_name')
        ->textInput(['maxlength' => 45]) ?>

    <?= $form->field($model, 'marketing_image_caption') ?>

    <?= $form->field($model, 'marketing_image_is_featured')
        ->dropDownList($model->marketingImageIsFeaturedList,
        ['prompt' => 'Please Choose One']); ?>

    <?= $form->field($model, 'marketing_image_is_active')
        ->dropDownList($model->marketingImageIsActiveList,
        ['prompt' => 'Please Choose One']); ?>

    <?= $form->field($model, 'marketing_image_weight') ?>

    <?= $form->field($model, 'status_id')->dropDownList($model->statusList); ?>

    <?= $form->field($model, 'file')->fileInput(); ?>

</div>

<?php ActiveForm::end(); ?>
```

```
</div>
```

First thing to note is:

```
<?php $form = ActiveForm::begin(['options'=>
    ['enctype' => 'multipart/form-data']])); ?>
```

Setting the options like this allows us to upload a file.

The other important thing to note is:

```
<?= $form->field($model, 'file')->fileInput(); ?>
```

This adds the button to upload the file. And if you remember in the model, we set the label to 'Marketing Image.'

So your form should look like this:

Home / Marketing Images / Create Marketing Image

Create Marketing Image

Marketing Image Name

Caption

Marketing Image Is Featured

Marketing Image Is Active

Marketing Image Weight

Status ID

Marketing Image

 No file chosen

Create

File Form

Modifying the Controller

Gist:

[MarketingImageController](#)

From book:

```
<?php

namespace backend\controllers;

use Yii;
use backend\models\MarketingImage;
use backend\models\search\MarketingImageSearch;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use yii\web\ForbiddenHttpException;
use yii\filters\VerbFilter;
use common\models\PermissionHelpers;
use yii\web\UploadedFile;

/**
 * MarketingImageController implements the CRUD
 * actions for MarketingImage model.
 */

class MarketingImageController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => \yii\filters\AccessControl::className(),
                'only' => ['index', 'view', 'create', 'update', 'delete'],
                'rules' => [
                    [
                        'actions' => ['index', 'view', 'create', 'update', 'delete'],
                        'allow' => true,
                        'roles' => ['@'],
                        'matchCallback' => function ($rule, $action) {
                            return PermissionHelpers::requireMinimumRole('Admin')
                                && PermissionHelpers::requireStatus('Active');
                        }
                    ],
                ],
            ],
        ],
    }
}
```

```
'verbs' => [
    'class' => VerbFilter::className(),
    'actions' => [
        'delete' => ['post'],
    ],
],
];
}

/**
 * Lists all MarketingImage models.
 * @return mixed
 */

public function actionIndex()
{
    $searchModel = new MarketingImageSearch();
    $dataProvider = $searchModel->search(Yii::$app->request->queryParams);

    return $this->render('index', [
        'searchModel' => $searchModel,
        'dataProvider' => $dataProvider,
    ]);
}

/**
 * Displays a single MarketingImage model.
 * @param string $id
 * @return mixed
 */

public function actionView($id)
{
    return $this->render('view', [
        'model' => $this->findModel($id),
    ]);
}

/**
 * Creates a new MarketingImage model.
 * If creation is successful, the browser will be redirected to the

```

```
* 'view' page.
* @return mixed
*/
public function actionCreate()
{
    $model = new MarketingImage();

    if ($model->load(Yii::$app->request->post())) {

        $imageName = $model->marketing_image_name;

        $model->file = UploadedFile::getInstance($model, 'file');

        $fileName = 'uploads/' . $imageName . '.' . $model->file->extension;
        $fileName = preg_replace('/\s+/', '', $fileName);

        $model->marketing_image_path = $fileName;
        $model->save();

        $model->file->saveAs($fileName);

        //Save the path in the DB

        return $this->redirect(['view', 'id' => $model->id, 'model' => $model,]);
    } else {
        return $this->render('create', [
            'model' => $model,
        ]);
    }
}

/**
 * Updates an existing MarketingImage model.
 * If update is successful, the browser will be
 * redirected to the 'view' page.
 * @param string $id
 * @return mixed
*/
```

```
public function actionUpdate($id)
{
    $model = $this->findModel($id);

    if ($model->load(Yii::$app->request->post())) {

        $imageName = $model->marketing_image_name;

        $oldImage = MarketingImage::find('marketing_image_name')
            ->where(['id' => $id])
            ->one();

        if ($oldImage->marketing_image_name != $imageName){

            throw new ForbiddenHttpException
                ('You cannot change the name, you must delete instead.');

        }

        $model->file = UploadedFile::getInstance($model, 'file');

        $model->save();

        $model->file->saveAs('uploads/' . $imageName . '.' . $model->file->extension);

        return $this->redirect(['view', 'id' => $model->id]);
    } else {
        return $this->render('update', [
            'model' => $model,
        ]);
    }
}

/**
 * Deletes an existing MarketingImage model.
 * If deletion is successful, the browser will be
 * redirected to the 'index' page.
 * @param string $id
 * @return mixed
 */
```

```
public function actionDelete($id)
{
    $model = $this->findModel($id);

    if(unlink($model->marketing_image_path)){
        $model->delete();

        return $this->redirect(['index']);
    }

    throw new NotFoundHttpException('We were unable to Delete');
}

/**
 * Finds the MarketingImage model based on its primary key value.
 * If the model is not found, a 404 HTTP exception will be thrown.
 * @param string $id
 * @return MarketingImage the loaded model
 * @throws NotFoundHttpException if the model cannot be found
 */

protected function findModel($id)
{
    if (($model = MarketingImage::findOne($id)) !== null) {
        return $model;
    } else {
        throw new NotFoundHttpException('The requested page does not exist.');
    }
}
```

Ok, so let's discuss what else is different. You can see we added some new use statements and the following is very important:

```
use yii\web\UploadedFile;
```

This gives us access to the UploadedFile class, which we will need to work with our file upload.

Also, we obviously modified behaviors to make it compliant with the other backend controllers, so that it requires admin or greater for access.

After that, it's just 3 methods that are different, actionCreate, actionUpdate, and actionDelete.

The Create Action

Our create action starts just as we expect:

```
public function actionCreate()
{
    $model = new MarketingImage();

    if ($model->load(Yii::$app->request->post())) {
```

We create a new instance of MarketingImage, then load the post data if we have any.

Next we create a local variable \$imageName to make things easier to work with and assign it like so:

```
$imageName = $model->marketing_image_name;
```

Next we assign the \$model->file property like so:

```
$model->file = UploadedFile::getInstance($model, 'file');
```

So now we are working with an instance of the actual file and we will be able to assign it the name we want. But before we save it, we do some formatting to make sure we are stripping any spaces out of the filename.

So just to make this perfectly clear, we are dealing with an instance of the actual file, which will be stored in our uploads folder, and also an instance of the model, which we will store in our DB. These are two separate things.

Once we have made sure the fileName is formatted correctly, we assign it to our \$model->marketing_image_path property, then save the model:

```
$model->marketing_image_path = $fileName;
$model->save();
```

It's important that we save the model, and therefore perform validation, before we save the file, which comes next:

```
$model->file->saveAs($fileName);
```

Just a tip, if you do that out of order, the instance of the file is no longer available in a temp directory and it messes up the validation, and you will get an error. So make sure to save model first, then the file.

The validation I'm referring to is in the model:

```
[['file'], 'file', 'extensions' => ['png', 'jpg', 'gif'],
'maxSize' => 1024*1024],
```

Yii 2 gives us a simple way to define the allowable extensions and the max size of the file.

So, returning to the controller. After saving, we go to the view file:

```
return $this->redirect(['view', 'id' => $model->id, 'model' => $model,]);
```

Otherwise, if we don't have a valid post, show the form:

```
else {
    return $this->render('create', [
        'model' => $model,
    ]);
}
```

The Update Action

The update action has notable differences. We need to know which instance of the model we are working on, so we take in the \$id of the model in the method signature and then find that instance of the model:

```
public function actionUpdate($id)
{
    $model = $this->findModel($id);
```

If we have valid post data, we set a local variable \$imageName to the value in the post:

```
$imageName = $model->marketing_image_name;
```

Then we need to lookup the existing image name and compare the two:

```
$oldImage = MarketingImage::find('marketing_image_name')
    ->where(['id' => $id])
    ->one();

if ($oldImage->marketing_image_name != $imageName){
    throw new ForbiddenHttpException
    ('You cannot change the name, you must delete instead.');
}
```

If they don't match, we throw an exception.

The reason I'm doing this is so we don't end up with a bunch of orphan files on the server that have no corresponding reference in the DB. You could also accomplish this by creating a custom validator or by doing a more robust implementation that covers that scenario.

I preferred to keep things simple.

So if we're good with the image name:

```
$model->file = UploadedFile::getInstance($model, 'file');

$model->save();
```

We get an instance of the file from post, then save the model. This is useful if we are just changing something like marketing_image_is_featured from no to yes.

To make an update, you will be required to upload the photo again because the validation rule requires it.

So we save the new instance of the file and redirect to view:

```
$model->file->saveAs('uploads/' . $imageName . '.' . $model->file->extension);

return $this->redirect(['view', 'id' => $model->id]);
```

Else, we show the form:

```

else {
    return $this->render('update', [
        'model' => $model,
    ]);
}

```

The Delete Action

Ok, one last method, actionDelete. We take in the id and look up the appropriate model:

```

public function actionDelete($id)
{
    $model = $this->findModel($id);

    if(unlink($model->marketing_image_path)){
        $model->delete();

        return $this->redirect(['index']);
    }

    throw new NotFoundHttpException('We were unable to Delete');
}

```

We call PHP's unlink method to delete the file and if successful, we also call \$model->delete to delete the model instance from the db.

Then we return to index or we throw an exception because we were unable to complete the unlink. This method probably should use a try/catch block to return a more useful error. I will make that improvement in the next section.

And that's it for basic image management. It's not a bad implementation, but we can do better.

For example, when we create an image, we should create a thumbnail to go along with it. Then we could list that thumbnail in the Gridview and in the other views.

Also, while we're able to update a record, the way we have it setup now is that we also have to re-upload the image or validation fails. But what if you just want to require the file on create, but not update? That would work out well because it would give you the option of not uploading the file again if you just wanted to change something like marketing_image_is_featured.

Our solution for that will utilize yii 2's scenario solution, which is pretty cool. I think you will like it.

The downside is that our update method gets a bit more complicated. The upside is that we get a much more robust solution for updating images.

Ok, let's get started.

Image Thumbnails with Imagine

We are going to take advantage of Yii 2's Imagine plugin to create our thumbnails. This is an extension created by the Yii 2 core team, and it's really easy to use.

Install Yii 2 Imagine Extension

To use this library, we need to add to our composer.json in the require section:

```
"yiisoft/yii2-imagine": "~2.0.0"
```

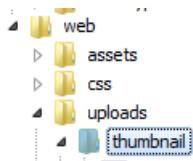
Then run composer update on the command line.

```
C:\var\www\yii2build>composer update
```

Composer Update

Create Thumbnail Folder

Inside of the backend/web/uploads folder, create a thumbnail folder:



Thumbnail Folder

Alter Marketing Image Table

We will also need to modify our marketing_image table to hold the thumbnail path.

The screenshot shows a table named 'marketing_image' in MySQL Workbench. The table has 11 columns:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
marketing_image_path	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
marketing_image_name	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
marketing_image_caption_title	VARCHAR(100)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
marketing_image_caption	VARCHAR(100)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
marketing_thumb_path	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
marketing_image_is_featured	BOOL	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0
marketing_image_is_active	BOOL	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0
marketing_image_weight	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	100
status_id	SMALLINT(6)	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	1
created_at	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
updated_at	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

New Marketing Image Table

Here is the SQL:

```
ALTER TABLE `yii2build`.`marketing_image`
ADD COLUMN `marketing_image_caption_title` VARCHAR(100)
CHARACTER SET 'utf8' COLLATE 'utf8_unicode_ci'
NULL DEFAULT NULL AFTER `marketing_image_name`,
ADD COLUMN `marketing_thumb_path` VARCHAR(45)
CHARACTER SET 'utf8' COLLATE 'utf8_unicode_ci'
NOT NULL AFTER `marketing_image_caption`;
```

Note: I added marketing_image_caption_title to give us more elements to work with in the future, should we decide to expand our options.

Modify MarketingImage Model

Gist:

[Modify MarketingImage Model](#)

From book:

```
<?php

namespace backend\models;

use Yii;
use yii\db\ActiveRecord;
use yii\db\Expression;
use yii\helpers\ArrayHelper;
use yii\behaviors\TimestampBehavior;
use yii\helpers\Html;

/**
 * This is the model class for table "marketing_image".
 *
 * @property string $id
 * @property string $marketing_image_path
 * @property string $marketing_image_name
 * @property integer $marketing_image_is_featured
 * @property integer $marketing_image_is_active
 * @property integer $status_id
 * @property string $created_at
 * @property string $updated_at
 *
 * @property Status $status
 */

```

```
class MarketingImage extends \yii\db\ActiveRecord
{

    public $file;

    /**
     * @inheritdoc
     */

    public static function tableName()
    {
        return 'marketing_image';
    }

    public function behaviors()
    {
```

```
    return [
        'timestamp' => [
            'class' => TimeStampBehavior::className(),
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
            ],
            'value' => new Expression('NOW()'),
        ],
    ];
}

/**
 * @inheritDoc
 */
public function rules()
{
    return [
        [['marketing_image_path', 'marketing_image_name',
            'marketing_thumb_path', 'marketing_image_weight'],
            'required'],
        ['marketing_image_weight', 'default', 'value' => 100 ],
        ['marketing_image_is_featured', 'default', 'value' => 0 ],
        ['marketing_image_is_active', 'default', 'value' => 0 ],
        ['file', 'required', 'message' =>
            '{attribute} can\'t be blank', 'on'=>'create'],
        [['marketing_image_name', 'marketing_image_path'], 'trim'],
        [['marketing_image_is_featured',
            'marketing_image_is_active', 'marketing_image_weight',
            'status_id'], 'integer'],
        [['marketing_image_is_featured'], 'in',
            'range'=>array_keys($this->getMarketingImageIsFeaturedList())],
        [['marketing_image_is_active'], 'in',
            'range'=>array_keys($this->getMarketingImageIsActiveList())],
        [['file'], 'file', 'extensions' => ['png', 'jpg', 'gif'],
            'maxSize' => 1024*1024],
        [['marketing_image_path', 'marketing_image_name'],
            'string', 'max' => 45],
        [['marketing_image_caption', 'marketing_image_caption_title'],
            'string', 'max' => 100],
    ];
}
```

```
    ];
}

public function scenarios()
{
    $scenarios = parent::scenarios();
    $scenarios['create'] = ['file', 'marketing_image_path',
        'marketing_image_name', 'marketing_thumb_path',
        'marketing_image_is_featured', 'marketing_image_is_active',
        'marketing_image_caption', 'marketing_image_caption_title',
        'marketing_image_weight'];

    return $scenarios;
}

public function beforeValidate()
{
    $this->marketing_image_name =
        preg_replace('/\s+/', '', $this->marketing_image_name);
    $this->marketing_image_path =
        preg_replace('/\s+/', '', $this->marketing_image_path);

    return parent::beforeValidate();
}

/**
 * @inheritDoc
 */

public function attributeLabels()
{
    return [
        'id' => 'ID',
        'marketing_image_path' => 'Marketing Image Path',
        'marketing_image_name' => 'Marketing Image Name',
        'marketing_thumb_path' => 'Marketing thumb Path',
        'marketing_image_caption' => 'Caption',
        'marketing_image_caption_title' => 'Caption Title',
        'marketing_image_is_featured' => 'Marketing Image Is Featured',
        'marketing_image_is_active' => 'Marketing Image Is Active',
        'marketing_image_weight' => 'Marketing Image Weight',
        'status_id' => 'Status ID',
    ];
}
```

```
'created_at' => 'Created At',
'updated_at' => 'Updated At',
'file' => 'Marketing Image',
'statusName' => Yii::t('app', 'Status'),
];
}

public static function getMarketingImageIsFeaturedList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}

public static function getMarketingImageIsActiveList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}

/**
 * @return \yii\db\ActiveQuery
 */

public function getStatus()
{
    return $this->hasOne(Status::className(),
        ['id' => 'status_id']);
}

/**
 * * get status name
 *
 */

public function getStatusName()
{
    return $this->status ? $this->status->status_name : '- no status -';
}

/**
 * get list of statuses for dropdown
 */

```

```

public static function getStatusList()
{
    $droptions = Status::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'status_name');
}

public function getThumb()
{
    $image = Html::img('/'.\$this->marketing_thumb_path);
    return Html::a($image, ['view', 'id' => \$this->id]);
}

}

```

So let's talk about what's new. We added a use statement for Html:

```
use yii\helpers\Html;
```

We'll need that when we want to return the url for the thumbnail image in Gridview.

We have a couple of different lines in the rules. We also have a rule for marketing_image_caption_title, but it's not shown here:

```

[[['marketing_image_path', 'marketing_image_name',
  'marketing_thumb_path'], 'required'],

['file', 'required', 'message' => '{attribute} can\'t be blank',
 'on'=>'create'],

```

You can see we added the new attribute, marketing_thumb_path to the required rule, and moved required rule for file to a separate line. This is because we've added a condition to the rule via the 'on' attribute.

```
'on'=>'create'
```

This tells the rule to apply only to the create method. As we mentioned, we want the file to be not required on update because we might only be updating different attributes of the model and not the image file itself.

Scenarios

So logically, the scenario method comes next.

```
public function scenarios()
{
    $scenarios = parent::scenarios();
    $scenarios['create'] = [
        'file', 'marketing_image_path',
        'marketing_image_name', 'marketing_thumb_path',
        'marketing_image_is_featured', 'marketing_image_is_active',
        'marketing_image_caption', 'marketing_image_caption_title',
        'marketing_image_weight' ];

    return $scenarios;
}
```

I'm moving through this quickly, but you should take note of scenarios, this is a powerful feature that allows us to finesse the rules.

The format here is fairly intuitive. We're setting a scenario for the create method, and we list all the attributes that will be validated under that scenario. When we want to use this in the controller, we use it when we call the model:

```
$model = new MarketingImage();
$model->scenario = 'create';
```

All in all, very simple. We will see that in action when we work on the controller.

Next we make the additions to our attributeLabels:

```
'marketing_thumb_path' => 'Marketing thumb Path',
'marketing_image_caption_title' => 'Caption Title',
```

Lastly, we will add a method to return an image link for our thumbnail, which we will use on our Gridview:

```

public function getThumb()
{
    $image = Html::img('/'.$this->marketing_thumb_path);
    return Html::a($image, ['view','id' => $this->id]);
}

```

You can see we first get the image and set it to \$image using Html::img, then we return it as a link, using Html::a.

You will love how easy this is to use in Gridview when we come to that part.

Modify MarketingImageSearch

We need to modify MarketingImageSearch, since marketing_image_caption will be a searchable field.

Gist:

[Modified MarketingImageSearch](#)

From book:

```

<?php

namespace backend\models\search;

use Yii;
use yii\base\Model;
use yii\data\ActiveDataProvider;
use backend\models\MarketingImage;

/**
 * MarketingImageSearch represents the model behind the search form about `backend\models\MarketingImage`.
 */

class MarketingImageSearch extends MarketingImage
{
    public $statusName;
    /**
     * @inheritdoc

```

```
*/\n\npublic function rules()\n{\n    return [\n        [['id', 'marketing_image_is_featured',\n            'marketing_image_is_active', 'status_id'], 'integer'],\n        [['marketing_image_path', 'marketing_image_name',\n            'marketing_image_caption',\n            'marketing_image_caption_title', 'marketing_image_weight',\n            'created_at', 'statusName', 'updated_at'], 'safe'],\n    ];\n}\n\n/**\n * @inheritDoc\n */\n\npublic function scenarios()\n{\n    // bypass scenarios() implementation in the parent class\n    return Model::scenarios();\n}\n\n/**\n * Creates data provider instance with search query applied\n *\n * @param array $params\n *\n * @return ActiveDataProvider\n */\n\npublic function search($params)\n{\n    $query = MarketingImage::find();\n\n    $dataProvider = new ActiveDataProvider([ \n        'query' => $query,\n    ]); \n\n    $dataProvider->setSort([
```

```
'attributes' => [
    'id',
    'marketing_image_name',
    'marketing_image_path',
    'marketing_image_caption',
    'marketing_image_caption_title',
    'marketing_image_is_featured',
    'marketing_image_is_active',
    'marketing_image_weight',
    'statusName' => [
        'asc' => ['status.status_name' => SORT_ASC],
        'desc' => ['status.status_name' => SORT_DESC],
        'label' => 'Status'
    ],
    ],
]);
};

if (!($this->load($params) && $this->validate())) {
    $query->joinWith(['status']);
    return $dataProvider;
}

$this->addSearchParameter($query, 'id');
$this->addSearchParameter($query, 'marketing_image_name', true);
$this->addSearchParameter($query, 'marketing_image_path', true);
$this->addSearchParameter($query, 'marketing_image_caption', true);
$this->addSearchParameter($query, 'marketing_image_caption_title', true);
$this->addSearchParameter($query, 'marketing_image_is_featured');
$this->addSearchParameter($query, 'marketing_image_is_active');
$this->addSearchParameter($query, 'marketing_image_weight');
$this->addSearchParameter($query, 'status_id');

// filter by gender name

$query->joinWith(['status' => function ($q) {
    $q->andFilterWhere(['=' , 'status.status_name', $this->statusName]);
}]);
```

```
        }]);

        return $dataProvider;

    }

protected function addSearchParameter($query, $attribute, $partialMatch = false)
{
    if (($pos = strpos($attribute, '.')) !== false) {
        $modelAttribute = substr($attribute, $pos + 1);
    } else {
        $modelAttribute = $attribute;
    }

    $value = $this->$modelAttribute;
    if (trim($value) === '') {
        return;
    }

    /*
     * The following line is additionally added for right aliasing
     * of columns so filtering happen correctly in the self join
     */
}

$attribute = "marketing_image.$attribute";

if ($partialMatch) {
    $query->andWhere(['like', $attribute, $value]);
} else {
    $query->andWhere([$attribute => $value]);
}
}
```

Modifying Marketing Image Controller

Gist:

[Marketing Image Controller](#)

From book:

```
<?php

namespace backend\controllers;

use Yii;
use backend\models\MarketingImage;
use backend\models\search\MarketingImageSearch;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use yii\web\ForbiddenHttpException;
use yii\filters\VerbFilter;
use common\models\PermissionHelpers;
use yii\web\UploadedFile;
use yii\imagine\Image;

/**
 * MarketingImageController implements the CRUD actions for
 * MarketingImage model.
 */

class MarketingImageController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => \yii\filters\AccessControl::className(),
                'only' => ['index', 'view', 'create', 'update', 'delete'],
                'rules' => [
                    [
                        'actions' => ['index', 'view', 'create',
                            'update', 'delete'],
                        'allow' => true,
                        'roles' => ['@'],
                        'matchCallback' => function ($rule, $action) {
                            return PermissionHelpers::requireMinimumRole('Admin')
                                && PermissionHelpers::requireStatus('Active');
                        }
                    ],
                ],
            ],
        ];
    }
}
```

```
],  
  
    'verbs' => [  
        'class' => VerbFilter::className(),  
        'actions' => [  
            'delete' => ['post'],  
        ],  
    ],  
];  
}  
  
/**  
 * Lists all MarketingImage models.  
 * @return mixed  
 */  
  
public function actionIndex()  
{  
  
    $searchModel = new MarketingImageSearch();  
    $dataProvider = $searchModel->search(Yii::$app->request->queryParams);  
  
    return $this->render('index', [  
        'searchModel' => $searchModel,  
        'dataProvider' => $dataProvider,  
    ]);  
}  
  
/**  
 * Displays a single MarketingImage model.  
 * @param string $id  
 * @return mixed  
 */  
  
public function actionView($id)  
{  
    return $this->render('view', [  
        'model' => $this->findModel($id),  
    ]);  
}
```

```
/**  
 * Creates a new MarketingImage model.  
 * If creation is successful, the browser will be redirected to  
 * the 'view' page.  
 * @return mixed  
 */  
  
public function actionCreate()  
{  
    $model = new MarketingImage();  
    $model->scenario = 'create';  
  
    if ($model->load(Yii::$app->request->post())) {  
  
        $imageName = $model->marketing_image_name;  
  
        $model->file = UploadedFile::getInstance($model, 'file');  
  
        $fileName = 'uploads/' . $imageName . '.' . $model->file->extension;  
        $fileName = preg_replace('/\s+/', '', $fileName);  
  
        $thumbName = 'uploads/' . 'thumbnail/' . $imageName  
            . 'thumb.' . $model->file->extension;  
  
        $thumbName = preg_replace('/\s+/', '', $thumbName);  
  
        $model->marketing_image_path = $fileName;  
        $model->marketing_thumb_path = $thumbName;  
        $model->save();  
  
        $model->file->saveAs($fileName);  
  
        Image::thumbnail( $fileName , 60, 60)  
            ->save($thumbName, [ 'quality' => 50]);  
  
    return $this->redirect(['view', 'id' => $model->id, 'model' => $model,]);  
}  
else {  
    return $this->render('create', [  
        'model' => $model,
```

```
        ]);
    }
}

/**
 * Updates an existing MarketingImage model.
 * If update is successful, the browser will be
 * redirected to the 'view' page.
 * @param string $id
 * @return mixed
 */

public function actionUpdate($id)
{
    $model = $this->findModel($id);

    if ($model->load(Yii::$app->request->post())) {

        $imageName = $model->marketing_image_name;

        $oldImage = MarketingImage::find('marketing_image_name')
            ->where(['id' => $id])
            ->one();

        if ($oldImage->marketing_image_name != $imageName){

            throw new ForbiddenHttpException
                ('You cannot change the name, you must delete instead.');
        }

        if( $model->file = UploadedFile::getInstance($model, 'file')){

            $thumbName = 'uploads/' . 'thumbnail/' . $imageName . 'thumb.'
                . $model->file->extension;

            $model->save();

        } else {

            $model->save();
        }
    }
}
```

```
    }

    if ($model->file) {

        $fileName = 'uploads/' . $imageName . '.' . $model->file->extension;

        $model->file->saveAs($fileName);

        Image::thumbnail( $fileName , 60, 60)
            ->save($thumbName, [ 'quality' => 50]);

    }

    return $this->redirect(['view', 'id' => $model->id]);
}

} else {
    return $this->render('update', [
        'model' => $model,
    ]);
}

}

/** 
 * Deletes an existing MarketingImage model.
 * If deletion is successful, the browser will be redirected
 * to the 'index' page.
 * @param string $id
 * @return mixed
 */
public function actionDelete($id)
{
    $model = $this->findModel($id);

    try {
        unlink($model->marketing_image_path);
    }
}
```

```
        unlink($model->marketing_thumb_path);

        $model->delete();

        return $this->redirect(['index']);;

    }

    catch(\Exception $e) {

        throw new NotFoundHttpException($e->getMessage());

    }

}

/**
 * Finds the MarketingImage model based on its primary key value.
 * If the model is not found, a 404 HTTP exception will be thrown.
 * @param string $id
 * @return MarketingImage the loaded model
 * @throws NotFoundHttpException if the model cannot be found
 */

protected function findModel($id)
{
    if (($model = MarketingImage::findOne($id)) !== null) {
        return $model;
    } else {
        throw new NotFoundHttpException
            ('The requested page does not exist.');
    }
}
```

So let's just discuss what's new. We start with a new use statement:

```
use yii\imagine\Image;
```

Modify Create Action

The next new part is actionCreate. We start it a little differently:

```
public function actionCreate()
{
    $model = new MarketingImage();
    $model->scenario = 'create';
```

You can see we are setting our \$model->scenario to ‘create.’

Alternatively, we could:

```
$model = new MarketingImage(['scenario' => 'create']);
```

We will also have to account for the marketing_thumb_path attribute and for the actual thumbnail images themselves.

So after we set \$fileName, we will assign a local variable to \$thumbName and then do the same kind of formatting to make sure we don’t have spaces in the filename:

```
$thumbName = 'uploads/' . 'thumbnail/' . $imageName . 'thumb.'
. $model->file->extension;

$thumbName = preg_replace('/\s+/', '', $thumbName);
```

Besides stripping out the spaces, if there are any, we are also appending ‘thumb’ onto \$imageName for the \$thumbName. If we didn’t do that, your thumbnail images would have the same filename as your primary image. I’m not a fan of that. So even though they are in separate folders, I like to give them different names.

\$thumbName will result in a path that we can use to set the \$model->marketing_thumb_path attribute. We have to set that one manually because we are not doing it in the form.

The reason for this is that the thumbnails are auto-generated from our primary image, so we don’t want the user to be able to set the value. It’s too easy for them to make a typo.

Once we’ve set that attribute, we can do a \$model->save():

```
$model->marketing_image_path = $fileName;
$model->marketing_thumb_path = $thumbName;
$model->save();
```

That takes care of the DB, but we still need to save our primary image and create a thumbnail from it. Fortunately, our imagine Image class makes this a snap:

```
$model->file->saveAs($fileName);

Image::thumbnail( $fileName , 60, 60)
->save($thumbName, ['quality' => 50]);
```

In this case, \$fileName is the file we are creating the thumbnail from. 60, 60 are dimensions, \$thumbName is the path to save to, and ‘quality’ is set to 50. It’s really just one line of code, very simple indeed.

And the rest is the same as we had before. You can test this and it should work perfectly, however we have not added the thumbnail in a view anywhere yet, so you won’t see it. You can check it by checking in the thumbnail folder.

Modify Update Action

Ok, moving on to update. This one got a little more complicated, but don’t worry, I will take you through it step by step.

This part is exactly as it was before:

```
public function actionUpdate($id)
{
    $model = $this->findModel($id);

    if ($model->load(Yii::$app->request->post())) {

        $imageName = $model->marketing_image_name;

        $oldImage = MarketingImage::find('marketing_image_name')
            ->where(['id' => $id])
            ->one();

        if ($oldImage->marketing_image_name != $imageName){

            throw new ForbiddenHttpException
                ('You cannot change the name, you must delete instead.');
        }
    }
}
```

Next we check to see if there is a new file uploaded, and if so, set the \$thumbName, which we will use to create the new thumb, and \$model->save():

```

if ( $model->file = UploadedFile::getInstance($model, 'file')){

    $thumbName = 'uploads/' . 'thumbnail/' . $imageName . 'thumb.'
    . $model->file->extension;

    $model->save();

}

```

If we don't have file uploaded, which is now possible because we used a scenario to only enforce the require rule on create, then we simply save the model:

```

else {

    $model->save();

}

```

Next, if we do have a file uploaded, we need to set the \$fileName, then save the file and create the thumbnail:

```

if ($model->file) {

    $fileName = 'uploads/' . $imageName . '.'
    . $model->file->extension;

    $model->file->saveAs($fileName);

    Image::thumbnail( $fileName , 60, 60)
    ->save($thumbName, ['quality' => 50]);

}

```

So we are only saving a primary image and thumbnail image if there is an image uploaded, otherwise we are happily redirected to the view with the \$model->id that we handed in, same as we had before

Modify Delete Action

Ok, so we need to account for the thumbnail in our delete action and we also wanted to wrap this in a try/catch block:

```
public function actionDelete($id)
{
    $model = $this->findModel($id);

    try {
        unlink($model->marketing_image_path);
        unlink($model->marketing_thumb_path);
        $model->delete();
        return $this->redirect(['index']);
    }

    catch(\Exception $e) {
        throw new NotFoundHttpException($e->getMessage());
    }
}
```

Nothing too difficult there. We obviously have to unlink both files to get rid of the image completely. Notice the \ in the catch:

```
catch(\Exception $e) {
```

When you are using built-in PHP classes such as Exception, you have to put a \ in front or Yii 2 will not find it.

The other thing I did was throw an exception:

```
throw new NotFoundHttpException($e->getMessage());
```

So what that does is if one of the files is missing, or it can't complete the unlink, it will return the message formatted gracefully, which means it will still have the page layout instead of a raw message on a white background.

Modify Views

Modify View

We're just adding the thumbnail, but I'll give you the entire file for reference.

Gist:

[view.php](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\DetailView;

/* @var $this yii\web\View */
/* @var $model backend\models\MarketingImage */

$this->title = $model->id;
$this->params['breadcrumbs'][] =
['label' => 'Marketing Images', 'url' => ['index']];

$this->params['breadcrumbs'][] = $this->title;
?>

<div class="marketing-image-view">

    <p>
        <?= Html::a('Update', ['update', 'id' => $model->id],
            ['class' => 'btn btn-primary']) ?>

        <?= Html::a('Delete', ['delete', 'id' => $model->id], [
            'class' => 'btn btn-danger',
            'data' => [
                'confirm' => 'Are you sure you want to delete this item?',
                'method' => 'post',
            ],
        ]) ?>
    </p>

    <h1><?= Html::encode($model->marketing_image_name) ?></h1>
```

```
<br>
<div>
<?php

echo Html::img('/'. $model->marketing_image_path . '?' . 'time='
    . time() , ['width' => '600px']);

?>

</div>
<br>
<div>
<?php

echo Html::img('/'. $model->marketing_thumb_path . '?' . 'time=' . time());

?>

</div>
<br>

<?= DetailView::widget([
    'model' => $model,
    'attributes' => [
        'id',
        'marketing_image_caption_title',
        'marketing_image_caption',
        'marketing_image_path',
        'marketing_thumb_path',
        'marketing_image_weight',
        // 'marketing_image_name',
        ['attribute' => 'marketing_image_is_featured',
            'format' => 'boolean'],
        ['attribute' => 'marketing_image_is_active',
            'format' => 'boolean'],
        'status.status_name',
        'created_at',
        'updated_at',
    ],
]) ?>

</div>
```

Modify Update

Again just adding the thumb and providing the entire file for reference.

Gist:

[update.php](#)

From book:

```
<?php

use yii\helpers\Html;

/* @var $this yii\web\View */
/* @var $model backend\models\MarketingImage */

$this->title = 'Update Marketing Image: ' . ' ' . $model->id;

$this->params['breadcrumbs'][] =
['label' => 'Marketing Images', 'url' => ['index']];

$this->params['breadcrumbs'][] =
['label' => $model->id, 'url' => ['view', 'id' => $model->id]];

$this->params['breadcrumbs'][] = 'Update';
?>
<div class="marketing-image-update">

<h1><?= Html::encode($this->title) ?></h1>

<br>
<div>
<?php
echo Html::img('/'. $model->marketing_image_path, ['width' => '600px']);

?>
</div>
<br>
<div>
<?php

echo Html::img('/'. $model->marketing_thumb_path . '?' . 'time=' . time());
```

```
?>

</div>
<br>

<?= $this->render('_form', [
    'model' => $model,
]) ?>

</div>
```

Modify _form

Just add the one line:

```
<?= $form->field($model, 'marketing_image_caption_title') ?>
```

Modify _search

Again, we just modify one line:

```
<?= $form->field($model, 'marketing_image_caption_title') ?>
```

Modify Index

We only have a small change here, but again I'm providing the entire file for reference. Doing this is a way for me to make sure I haven't missed something, since this code comes directly from my IDE.

Gist:

[index.php](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\grid\GridView;
use \yii\bootstrap\Collapse;

/* @var $this yii\web\View */
/* @var $searchModel backend\models\search\MarketingImageSearch */
/* @var $dataProvider yii\data\ActiveDataProvider */

$this->title = 'Marketing Images';
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="marketing-image-index">

    <h1><?= Html::encode($this->title) ?></h1>
    <?php echo Collapse::widget([
        'encodeLabels' => false,
        'items' => [
            // equivalent to the above
            [
                'label' => '<i class="fa fa-caret-square-o-down"></i> Search',
                'content' => $this->render('_search', ['model' => $searchModel]) ,
                // open its content by default
                //'contentOptions' => ['class' => 'in']
            ],
        ]
    ]);
?>

    <p>
        <?= Html::a('Create Marketing Image', ['create'],
```

```

        ['class' => 'btn btn-success']) ?>
</p>

<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        'id',
        'marketing_image_path',
        'marketing_image_name',
        'marketing_image_caption_title',
        'marketing_image_caption',
        ['attribute'=>'marketing_image_is_featured',
         'format'=>'boolean'],
        ['attribute'=>'marketing_image_is_active',
         'format'=>'boolean'],
        'marketing_image_weight',
        'statusName',
        // 'created_at',
        // 'updated_at',
        'thumb:html',
        ['class' => 'yii\grid\ActionColumn'],
    ],
]); ?>

</div>

```

So obviously the change here is the additions of:

```
'thumb:html',
'marketing_image_caption_title',
```

What a powerful little line ‘thumb:html’ is. When we call ‘thumb’, we are calling the model method, using magic get syntax, so that is actually calling getThumb from the model. The part after the colon tells gridview it’s returning html. And that’s it! You get a clickable thumbnail in your grid. You also get the column heading.

There are a lot of ways to appreciate the power of Yii 2 and this is certainly one of them.

Ok, so now that we got our image management up and running, we want to put our images to good use in the frontend.

URL Manager

Because we are using Yii 2's advanced template, using the images from the backend in the frontend takes some configuration. This is because the frontend and backend are two separate applications.

So what we need to do is add to communicate with the backend path, and we do that by setting up a key in our components array. I should note that I got this solution from Skworden in the forum after I spent quite a few hours being stumped by it. Yii 2 has a great community of developers and that can really help you out when you need it.

Anyway, what we need to do is add the following to your frontend/config/main.php file in your components array:

```
//can name it whatever you want as it is custom
```

```
'urlManagerBackend' => [
    'class' => 'yii\web\urlManager',
    'baseUrl' => 'http://backend.yii2build.com',
    'enablePrettyUrl' => true,
    'showScriptName' => false,
],
```

I should also note that this is still a bit of a workaround. I would prefer not to use an absolute url here if I don't have to because it's not as efficient as a relative link.

If I can't find any other solution, I will eventually reverse it so that the images and thumbnails are saved to the frontend and referenced by absolute link on the backend. This would be more efficient in terms of server resource.

Every once in a while you come across something you can't instantly solve, and this is especially true with a big framework like Yii 2. It takes a long time to learn every nuance of the framework, so everyone, including me, has to have patience for that.

Anyway, once you have that in place, you can reference images that are stored in the backend, like so:

```
echo Html::img(Yii::$app->urlManagerBackend->baseUrl. '/uploads/your-image.png');
```

I'm going to give you a Gist for reference on the entire file, in case you need to debug. Remember, the path is frontend/config/main.php:

Gist:

[main.php](#)

Carousel Widget

Ok, our objective will be to have a carousel widget that we can embed on the index page of the site, which just as a reminder, is controlled by the pages controller.

The idea will be to be able to call our carousel with a single line of code, like so:

```
<?= CarouselWidget::widget([ 'settings' => [
    'height' => '300px',
    'width' => '700px'])?>
```

You can see that we will have the option to hand in settings, in this case it will be height and width of the images. We will also build our widget to have default settings, so we could call it like so:

```
<?= CarouselWidget::widget()?>
```

We will create the widget so that it sets default values if we choose to call it without settings.

When we created our MarketingImages model, we made three attributes that will be used to determine if the image will be shown in the carousel. One is the marketing_image_is featured boolean column, which if true, means the image is displayed.

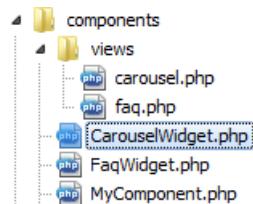
The second is the marketing_image_is_active boolean column, which if true, means the image is the active image, which means it is the one that is first loaded in the carousel. So when you are creating images, you should only have one image set to true.

The last column we test for is the status column, and we only want the images that are active.

So this data structure is nice and flexible, while giving the admin total control via UI to display or not to display images in the carousel.

Before we code the actual widget, we need to setup the files.

If you remember from chapter 12, when we introduced you to widgets, the basic structure is two files, a widget file directly under the components directory and a view file, which is in the views folder inside the components directory.



Carousel Widget in Components

So you can see the two files we added there. I will give you the files in a moment. First, let's set up our component. Go to common\config\main.php and add this to your components array:

```
'carouselwidget' => [  
    'class' => 'components\CarouselWidget',  
],
```

Now our application will be able to see the CarouselWidget class.

CarouselWidget.php

Gist:

[CarouselWidget.php](#)

from book:

```
<?php  
namespace components;  
  
use yii\base\Widget;  
use yii\helpers\Html;  
use Yii;  
use backend\models\MarketingImage;  
use backend\models\search\MarketingImageSearch;  
use yii\web\NotFoundHttpException;  
use yii\filters\VerbFilter;  
  
class CarouselWidget extends Widget  
{  
  
    public $activeImage;  
    public $images;  
    public $count;  
    public $settings = [];  
  
    public function init()  
    {  
        parent::init();
```

```
$this->activeImage = MarketingImage::find('marketing_image_path')
    ->where(['marketing_image_is_active' => 1])
    ->andWhere(['marketing_image_is_featured' => 1])
    ->andWhere(['status_id' => 1])
    ->one();

$this->images = MarketingImage::find()
    ->where(['marketing_image_is_active' => 0])
    ->andWhere(['marketing_image_is_featured' => 1])
    ->andWhere(['status_id' => 1])
    ->orderBy('marketing_image_weight')
    ->all();

$this->count = MarketingImage::find()
    ->where(['marketing_image_is_active' => 0])
    ->andWhere(['marketing_image_is_featured' => 1])
    ->andWhere(['status_id' => 1])
    ->count();

$this->setDefaults();

$this->validateSize();

}

public function setDefaults()
{
    if(!isset($this->settings['height'])){
        $this->settings['height'] = '300px';
    }

    if(!isset($this->settings['width'])){
        $this->settings['width'] = '700px';
    }

    if(!isset($this->settings['autoplay'])){
        $this->settings['autoplay'] = true;
    }
}
```

```
    }

}

public function validateSize()
{
    if (!preg_match("/px/", $this->settings['width'])
        or !preg_match("/px/", $this->settings['height'])) {
        throw new NotFoundHttpException('You Must Use px
            and number for size, example 300px');
    }

    $height = (int) preg_replace("/[^0-9]/","", $this->settings['height']);

    switch ($height){
        case $height < 40 :
            throw new NotFoundHttpException('You Must Stay within
                40 to 1000 px and use px and number for size, example 300px');
            break;

        case $height > 1000 :
            throw new NotFoundHttpException('You Must Stay within
                40 to 1000 px and use px and number for size, example 300px');
            break;
    }

    $width = (int) preg_replace("/[^0-9]/","", $this->settings['width']);

    switch ($width){
        case $width < 40 :
            throw new NotFoundHttpException('You Must Stay within
```

```

        40 to 1000 px and use px and number for size, example 300px');
    break;

    case $width > 1000 :
        throw new NotFoundHttpException('You Must Stay within
        40 to 1000 px and use px and number for size, example 300px');
    break;

}

}

public function run()
{
    return $this->render('carousel',
        ['activeImage' => $this->activeImage,
         'images' => $this->images,
         'count' => $this->count,
         'settings' => $this->settings]);
}

}

```

Since we covered the creation of a custom widget in chapter 12, I'm going to step through this quickly. Obviously, we include everything we need in the use statements, then move on to declaring the class properties:

```

public $activeImage;
public $images;
public $count;
public $settings = [];

```

The \$activeImage property will hold the ActiveRecord instance with the image that has marketing_image_is_active set to a value of 1. Please remember to keep only one record with that value.

The \$images property will be an instance of ActiveRecord with multiple results, all the images that are set to featured.

Lastly, the \$settings array will hold our settings values when they are handed in.

So moving on to init():

```

public function init()
{
    parent::init();

    $this->activeImage = MarketingImage::find('marketing_image_path')
        ->where(['marketing_image_is_active' => 1])
        ->andWhere(['marketing_image_is_featured' => 1])
        ->andWhere(['status_id' => 1])
        ->one();

    $this->images = MarketingImage::find()
        ->where(['marketing_image_is_active' => 0])
        ->andWhere(['marketing_image_is_featured' => 1])
        ->andWhere(['status_id' => 1])
        ->orderBy('marketing_image_weight')
        ->all();

    $this->count = MarketingImage::find()
        ->where(['marketing_image_is_active' => 0])
        ->andWhere(['marketing_image_is_featured' => 1])
        ->andWhere(['status_id' => 1])
        ->count();

    $this->setDefaultSize();

    $this->validateSize();
}

}

```

You can see we call the parent::init() and then we use ActiveRecord to return the results we are looking for and set them to the class properties via \$this.

Note in the \$this->images instance we are ordering by marketing_image_weight. This is really simple stuff and it gives us complete control over the order that the images will be displayed in the carousel.

We use the \$count property in the view to determine the correct number of carousel-indicators to display.

Next we call a couple of methods I created to set the default size of the images if none is provided and also to do just a little validation on the size input. Let's look at the setDefaults first:

```

public function setDefaults()
{
    if(!isset($this->settings['height'])){
        $this->settings['height'] = '300px';
    }

    if(!isset($this->settings['width'])){
        $this->settings['width'] = '700px';
    }

    if(!isset($this->settings['autoplay'])){
        $this->settings['autoplay'] = true;
    }
}

```

This allows us to call the widget like so:

```
<?= CarouselWidget::widget() ?>
```

Next I decided to do a little validation on the size input, so that we can have a hint if something in the settings has a typo:

```

public function validateSize()
{
    if (!preg_match("/px/", $this->settings['width'])
        or !preg_match("/px/", $this->settings['height'])) {
        throw new NotFoundHttpException
        ('You Must Use px and number for size, example 300px');
    }

    $height = (int) preg_replace("/[^0-9]/", "", $this->settings['height']);
}

```

```
switch ($height){

    case $height < 40 :
        throw new NotFoundHttpException('You Must Stay within
40 to 1000 px and use px and number for size, example 300px');
        break;

    case $height > 1000 :
        throw new NotFoundHttpException('You Must Stay within
40 to 1000 px and use px and number for size, example 300px');
        break;

}

$width = (int) preg_replace("/[^0-9]/","", $this->settings['width']);

switch ($width){

    case $width < 40 :
        throw new NotFoundHttpException('You Must Stay within
40 to 1000 px and use px and number for size, example 300px');
        break;

    case $width > 1000 :
        throw new NotFoundHttpException('You Must Stay within
40 to 1000 px and use px and number for size, example 300px');
        break;

}

}
```

Ok, so in the first part, I'm just looking to see if we included 'px':

```
if (!preg_match("/px/", $this->settings['width'])
or !preg_match("/px/", $this->settings['height'])) {

    throw new NotFoundHttpException
('You Must Use px and number for size, example 300px');

}
```

Then I set minimum and maximum values for size. But first I had to extract the number from the string:

```
$height = (int) preg_replace("/[^0-9]/","", $this->settings['height']);
```

Then we get the switch statement to handle the ranges:

```
switch ($height){

    case $height < 40 :
        throw new NotFoundHttpException('You Must Stay within
40 to 1000 px and use px and number for size, example 300px');
        break;

    case $height > 1000 :
        throw new NotFoundHttpException('You Must Stay within
40 to 1000 px and use px and number for size, example 300px');
        break;

}
```

I probably could have written a stronger set of validation rules, but I felt it would have been overkill, since we're just looking for a hint of what could be wrong if the carousel does not display properly.

Moving on, we get to the run method:

```
public function run()
{
    return $this->render('carousel',
        ['activeImage' => $this->activeImage,
         'images' => $this->images,
         'count' => $this->count,
         'settings' => $this->settings]);
}
```

We just render carousel.php and pass in the properties we will need in the view.

carousel.php

Now we are ready to look at the widget view:

Gist:

[carousel.php](#)

From book:

```
<?php

use yii\helpers\Html;

?>

<div id="carouselMain" class="carousel slide"

<?php

if($settings['autoplay'] == false ){

echo 'data-interval="false"';
}

?>

data-ride="carousel">

<!-- Indicators -->

<ol class="carousel-indicators">
```

```
<li data-target="#carouselMain" data-slide-to="0"
    class="active"></li>

<!-- dynamic indicator data -->

<?php

foreach (range(1, $count) as $number) {

echo '<li data-target="#carouselMain" data-slide-to="'. $number .'"></li>';

}

?>

<!-- end dynamic indicator data -->

</ol>

<!-- Wrapper for slides ?>

<div class="carousel-inner" role="listbox">

<!-- dynamic slide data -->

<?php

$width = $settings['width'];
$height = $settings['height'];

//active item first

echo '<div class="item active">
<center>' .
Html::img(Yii::$app->urlManagerBackend->baseUrl. '/' .
$activeImage['marketing_image_path'],
['width' => $width, 'height' => $height ])
.</center>
<div class="carousel-caption">' . $activeImage['marketing_image_caption'] .'

</div>
```

```
</div>';

//all other images

foreach ($images as $image){

    echo '<div class="item">
        <center>' .
            Html::img(Yii::$app->urlManagerBackend->baseUrl . '/' .
                $image['marketing_image_path'],
                ['width' => $width, 'height' => $height ])
        . '</center>
        <div class="carousel-caption">' . $image['marketing_image_caption'] . '
    </div>
</div>';

}

?>

<!-- end dynamic slide data -->

</div>

<!-- Controls -->

<a class="left carousel-control" href="#carouselMain" role="button"
    data-slide="prev">
    <span class="glyphicon glyphicon-chevron-left" aria-hidden="true"></span>
    <span class="sr-only">Previous</span>
</a>
<a class="right carousel-control" href="#carouselMain" role="button"
    data-slide="next">
    <span class="glyphicon glyphicon-chevron-right" aria-hidden="true"></span>
    <span class="sr-only">Next</span>
</a>
</div>
```

Looks like a giant plate of spaghetti, but this just a straight copy and paste from Bootstrap 3, with a few dynamic elements.

When working with PHP and HTML together, things get ugly quickly. Sometimes I think it's actually easier to work with this on your own than to read someone else's code.

You have to very careful to use single quotes and place them very precisely. Don't worry, we will step through this.

So you can see what changes we made, here is the original bootstrap code for reference:

Bootstrap 3 Carousel

We named our element carouselMain instead of carousel-example-generic:

```
<div id="carouselMain" class="carousel slide"
```

Then we get an if statement to see if we want the carousel to autoplay, which we can set in our settings, when we call the widget. It defaults to true, but if autoplay is set to false, we echo the following:

```
<?php  
  
    if($settings['autoplay'] == false ){  
  
        echo 'data-interval="false"';  
    }  
  
?>
```

That will stop the slides from moving on their own.

Next we need to dynamically determine the number of indicators, which are the clickable circles in the carousel that allow you to jump to any slide. The first indicator is set to 0 for the active slide.

We need to figure out how many rows and what number to place in the subsequent tags. So we do a foreach loop using PHP's built-in range function to set a range between 1 and the total number held in \$count. Then our handy foreach loop echoes out the correct number of lines with the appropriate number in each line:

```
<?php  
  
    foreach (range(1, $count) as $number) {  
  
        echo '<li data-target="#carouselMain" data-slide-to="'.$number.'"></li>';  
  
    }  
  
?>
```

Next we have the section that echoes the active item and the other items:

```
<!-- dynamic slide data -->

<?php

    $width = $settings['width'];
    $height = $settings['height'];

    //active item first

    echo '<div class="item active">
        <center>' .
        Html::img(Yii::$app->urlManagerBackend->baseUrl . '/' .
        $activeImage['marketing_image_path'],
        ['width' => $width, 'height' => $height ])
        . '</center>
    <div class="carousel-caption">' . $activeImage['marketing_image_caption'] . '

        </div>
    </div>';

    //all other images

    foreach ($images as $image){

        echo '<div class="item">
            <center>' .
            Html::img(Yii::$app->urlManagerBackend->baseUrl . '/' .
            $image['marketing_image_path'],
            ['width' => $width, 'height' => $height ])
            . '</center>
        <div class="carousel-caption">' . $image['marketing_image_caption'] . '

        </div>
    </div>';

    }

?>

<!-- end dynamic slide data -->
```

To make the code easier to read, I'm assigning the value of \$width and \$height, but you could easily just pop \$settings['height'] into Html image tag.

In the case of the items, we use a foreach loop to carefully echo out the exact syntax need for the carousel, while injecting our dynamic data.

For the image tag, we are using:

```
Html::img(Yii::$app->urlManagerBackend->baseUrl . '/'  
. $image['marketing_image_path'], ['width' => $width, 'height' => $height ])
```

You can see we are using our urlManagerBackend to get the baseUrl that allows us to reference an image from the backend.

Pages Index

I'm giving you the entire file for reference, but it's only the top area that has changed. You can see where we put the widget. I set the autoplay to false:

```
<div class="jumbotron">  
  
<br>  
<div>  
  
<?= CarouselWidget::widget(['settings' =>  
    ['height' => '300px', 'width' => '700px'  
    'autoplay'=>false])?>  
  
</div>  
  
</div>
```

Gist:

[Pages Index](#)

From book:

```
<?php

use \yii\bootstrap\Modal;
use kartik\social\FacebookPlugin;
use \yii\bootstrap\Collapse;
use \yii\bootstrap\Alert;
use yii\helpers\Html;
use components\FaqWidget;
use components\CarouselWidget;

/* @var $this yii\web\View */
$this->title = 'My Yii Application';
?>
<div class="site-index">

<?php

    if (Yii::$app->user->isGuest) {
        echo yii\authclient\widgets\AuthChoice::widget([
            'baseAuthUrl' => ['site/auth'],
            'popupMode' => false,
        ]);
    }
?>

<div class="jumbotron">

<br>
<div>

    <?= CarouselWidget::widget(['settings' =>
        ['height' => '300px', 'width' => '700px'
        'autoplay'=>false]])?>

</div>

</div>

<?php

echo Collapse::widget([
```

```
'items' => [  
    [  
        'label' => 'Top Features',  
        'content' => FacebookPlugin::widget([  
  
            'type'=>FacebookPlugin::SHARE,  
            'settings' => ['href'=>'http://www.yii2build.com','width'=>'350']  
  
        ]),  
  
        // open its content by default  
        // 'contentOptions' => ['class' => 'in']  
  
    ],  
  
    // another group item  
    [  
  
        'label' => 'Top Resources',  
        'content' => FacebookPlugin::widget([  
  
            'type'=>FacebookPlugin::SHARE,  
            'settings' => ['href'=>'http://www.yii2build.com','width'=>'350']  
  
        ]),  
  
        // 'contentOptions' => [],  
        // 'options' => [],  
  
    ],  
  
]);  
  
Modal::begin([
```

```
'header' => '<h2>Latest Comments</h2>',
'toggleButton' => [ 'label' => 'comments' ],
]);


echo FacebookPlugin::widget([
    'type'=>FacebookPlugin::COMMENT,
    'settings' => [ 'href'=>'http://www.yii2build.com', 'width'=>'350' ]
]);


Modal::end();


?>

<br/>
<br/>

<?Php

echo Alert::widget([
    'options' => [
        'class' => 'alert-info',
    ],
    'body' => 'Launch your project like a rocket...',
]);
?>

<div class="body-content">

<div class="row">
    <div class="col-lg-4">
        <h2>Free</h2>

        <p>
<?php

if (!Yii::$app->user->isGuest) {
```

```
echo Yii::$app->user->identity->username . ' is doing cool stuff.';  
}
```

```
?>
```

Starting with this free, open source Yii 2 template and it will save you a lot of time. You can deliver projects to the customer quickly, with a lot of boilerplate already taken care of for you, so you can concentrate on the complicated stuff.</p>

```
<p>
```

```
<a class="btn btn-default"  
href="http://www.yiiframework.com/doc-2.0/guide-index.html">  
Yii Documentation &raquo;</a>
```

```
</p>
```

```
<?php
```

```
echo FacebookPlugin::widget([  
  
'type'=>FacebookPlugin::LIKE,  
'settings' => []  
]);
```

```
?>
```

```
</div>  
<div class="col-lg-4">  
    <h2>Advantages</h2>
```

```
<p>
```

Ease of use is a huge advantage. We've simplified RBAC and given you Free/Paid user type out of the box. The Social plugins are so quick and easy to install, you will love it!

```
</p>
```

```
<p>
```

```
<a class="btn btn-default"
    href="http://www.yiiframework.com/forum/">
    Yii Forum &raquo;</a>

</p>

<?php

echo FacebookPlugin::widget([
    'type'=>FacebookPlugin::COMMENT,
    'settings' => [ 'href'=>'http://www.yii2build.com', 'width'=>'350' ]
]);

?>
</div>
<div class="col-lg-4">
    <h2>Code Quick, Code Right!</h2>
```

<p>

Leverage the power of the awesome Yii 2 framework with this enhanced template.
Based Yii 2's advanced template, you get a full frontend and backend
implementation that features rich UI for backend management.

</p>

<p>

```
<a class="btn btn-default"
    href="http://www.yiiframework.com/extensions/">
    Yii Extensions &raquo;</a>
```

</p>

```
    </div>
</div>
```

```
</div>
<br>
```

```

<?= FaqWidget::widget(['settings' =>
  ['pageSize' => 3, 'featuredOnly' => true,
  'heading' => 'Featured FAQs']] ) ?>

</div>

```

Carousel Settings

Once I had all that done, I thought, wouldn't it be cool if you could hand all control of the carousel to admin via the backend? Clients would love that. That shouldn't be too hard if you think about it, we have everything we need in place, except for the settings model.

This seemed like fun, so I took it up as a challenge to see how many of the carousel settings I could manage from backend UI and I'm fairly satisfied with the result. It turns out there are more settings that we have previously played with.

Let's start by setting up a `CarouselSettings` model. This is where we will store all of our carousel settings. And we will have access to them via the backend, just like we can access roles and statuses, etc.

`carousel_settings` table

Here is a screenshot of what we need:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT(11)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>					
carousel_name	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
image_height	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
image_width	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
carousel_autoplay	TINYINT(1)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	'1'				
show_indicators	TINYINT(1)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	'1'				
show_caption_title	TINYINT(1)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1				
show_captions	TINYINT(1)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	'1'				
show_caption_background	TINYINT(1)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1				
caption_font_size	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
show_controls	TINYINT(1)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1				
status_id	SMALLINT(6)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
created_at	DATETIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
updated_at	DATETIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					

Carousel Settings Table

SQL:

```

CREATE TABLE IF NOT EXISTS `yii2build`.`carousel_settings` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `carousel_name` VARCHAR(45) CHARACTER SET 'utf8'
    COLLATE 'utf8_unicode_ci' NOT NULL,
  `image_height` VARCHAR(45) CHARACTER SET 'utf8'
    COLLATE 'utf8_unicode_ci' NOT NULL,
  `image_width` VARCHAR(45) CHARACTER SET 'utf8'
    COLLATE 'utf8_unicode_ci' NOT NULL,
  `carousel_autoplay` TINYINT(1) NOT NULL DEFAULT '1',
  `show_indicators` TINYINT(1) NOT NULL DEFAULT '1',
  `show_caption_title` TINYINT(1) NOT NULL DEFAULT 1,
  `show_captions` TINYINT(1) NOT NULL DEFAULT '1',
  `show_caption_background` TINYINT(1) NOT NULL DEFAULT 1,
  `caption_font_size` VARCHAR(45) NOT NULL,
  `show_controls` TINYINT(1) NOT NULL DEFAULT 1,
  `status_id` SMALLINT(6) NOT NULL,
  `created_at` DATETIME NOT NULL,
  `updated_at` DATETIME NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_carousel_settings_status1_idx` (`status_id` ASC),
  CONSTRAINT `fk_carousel_settings_status1`
    FOREIGN KEY (`status_id`)
    REFERENCES `yii2build`.`status` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)

```

Go ahead and create the table.

Basically, the table and model will supply the attributes for our settings array, so we can call the widget like so:

```

<?= CarouselWidget::widget(['settings' => [
    'height' => $carouselSettings['image_height'],
    'width' => $carouselSettings['image_width'],
    'autoplay' => $carouselSettings['carousel_autoplay'],
    'show_indicators' => $carouselSettings['show_indicators'],
    'show_captions' => $carouselSettings['show_captions'],
    'show_controls' => $carouselSettings['show_controls'],
    'show_caption_title' => $carouselSettings['show_caption_title'],
]])?>

```

Note that in the above, I created friendlier settings names when I could use a single word, like height, for example. For multiple words, I kept it exactly as it is in the DB.

CarouselSettings Model

Let's start by using Gii to create the model.

Gist:

[CarouselSettings Model](#)

From book:

```
<?php

namespace backend\models;

use Yii;
use yii\db\ActiveRecord;
use yii\db\Expression;
use yii\helpers\ArrayHelper;
use yii\behaviors\TimestampBehavior;
use yii\helpers\Html;

/**
 * This is the model class for table "carousel_settings".
 *
 * @property integer $id
 * @property string $carousel_name
 * @property string $image_height
 * @property string $image_width
 * @property integer $carousel_autoplay
 * @property integer $show_indicators
 * @property integer $status_id
 * @property string $created_at
 * @property string $updated_at
 *
 * @property Status $status
 */

class CarouselSettings extends \yii\db\ActiveRecord
{
    /**
     * @inheritdoc
     */
    public static function tableName()
```

```
{  
    return 'carousel_settings';  
}  
  
public function behaviors()  
{  
    return [  
        'timestamp' => [  
            'class' => TimeStampBehavior::className(),  
            'attributes' => [  
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],  
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],  
                ],  
            'value' => new Expression('NOW()'),  
            ],  
        ],  
    ];  
}  
  
/**  
 * @inheritDoc  
 */  
  
public function rules()  
{  
    return [  
        [['carousel_name', 'image_height',  
            'image_width', 'caption_font_size', 'status_id'], 'required'],  
        [['carousel_autoplay', 'show_indicators', 'show_captions',  
            'status_id', 'show_controls'], 'integer'],  
        [['carousel_autoplay'], 'in',  
            'range'=>array_keys($this->getCarouselAutoplayList())],  
        [['show_indicators'], 'in',  
            'range'=>array_keys($this->getShowIndicatorsList())],  
        [['show_captions'], 'in',  
            'range'=>array_keys($this->getShowCaptionsList())],  
        [['show_caption_background'], 'in',  
            'range'=>array_keys($this->getShowCaptionBackgroundList())],  
        [['show_caption_title'], 'in',  
            'range'=>array_keys($this->getShowCaptionTitleList())],  
        [['show_controls'], 'in',  
            'range'=>array_keys($this->getShowControlsList())],  
    ];  
}
```

```
        [[ 'status_id'], 'in', 'range'=>array_keys($this->getStatusList())],
        [[ 'created_at', 'updated_at'], 'safe'],
        [[ 'carousel_name', 'image_height', 'image_width'],
         'string', 'max' => 45]
    ];
}

/**
 * @inheritDoc
 */

public function attributeLabels()
{
    return [
        'id' => 'ID',
        'carousel_name' => 'Name',
        'image_height' => 'Height',
        'image_width' => 'Width',
        'carousel_autoplay' => 'Autoplay',
        'show_indicators' => 'Show Indicators',
        'show_captions' => 'Show Captions',
        'show_caption_background' => 'Show Caption Background',
        'show_caption_title' => 'Show Caption Title',
        'show_controls' => 'Show Controls',
        'caption_font_size' => 'Caption Size',
        'status_id' => 'Status',
        'created_at' => 'Created At',
        'updated_at' => 'Updated At',
        'statusName' => Yii::t('app', 'Status'),
    ];
}

/**
 * @return \yii\db\ActiveQuery
 */

public function getStatus()
{
    return $this->hasOne(Status::className(),
        ['id' => 'status_id']);
}
```

```
}

public function getStatusName()
{
    return $this->status ? $this->status->status_name : '- no status -';

}

/**
 * get list of statuses for dropdown
 */

public static function getStatusList()
{
    $droptions = Status::find()->asArray()->all();
    return ArrayHelper::map($droptions, 'id', 'status_name');
}

public static function getCarouselAutoplayList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}

public static function getShowIndicatorsList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}

public static function getShowCaptionsList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}

public static function getShowCaptionTitleList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}

public static function getShowCaptionBackgroundList()
{
    return $droptions = [0 => "no", 1 => "yes"];
}
```

```
}
```

There really isn't anything on the model that we haven't seen before. I did shorten some of the attribute labels so we could get all the columns without scrolling.

CarouselSettingsSearch Model

Gist:

[CarouselSettingsSearch Model](#)

From book:

```
<?php

namespace backend\models\search;

use Yii;
use yii\base\Model;
use yii\data\ActiveDataProvider;
use backend\models\CarouselSettings;

/**
 * CarouselSettingsSearch represents the model
 * behind the search form about `backend\models\CarouselSettings`.
 */
class CarouselSettingsSearch extends CarouselSettings
{

    public $statusName;

    /**
     * @inheritdoc
     */

    public function rules()
    {
        return [
            [['id', 'carousel_autoplay', 'show_indicators',
            'show_captions', 'show_controls'],

```

```
        'show_caption_background', 'show_caption_title',
        'status_id'], 'integer'],
    [['carousel_name', 'image_height', 'image_width',
      'caption_font_size', 'statusName', 'created_at',
      'updated_at'], 'safe'],
];
}

/***
 * @inheritDoc
 */

public function scenarios()
{
    // bypass scenarios() implementation in the parent class
    return Model::scenarios();
}

/***
 * Creates data provider instance with search query applied
 *
 * @param array $params
 *
 * @return ActiveDataProvider
 */
public function search($params)
{
    $query = CarouselSettings::find();

    $dataProvider = new ActiveDataProvider([
        'query' => $query,
    ]);

    $dataProvider->setSort([
        'attributes' => [
            'id',
            'carousel_name',
            'image_height',
            'image_width',
            'carousel_autoplay',
            'show_indicators',
        ]
    ]);
}
```

```
'show_captions',
'show_caption_title',
'show_caption_background',
'caption_font_size',
'show_controls',
'statusName' => [
    'asc' => ['status.status_name' => SORT_ASC],
    'desc' => ['status.status_name' => SORT_DESC],
    'label' => 'Status'
],
'updated_at',

],
]);


if (!($this->load($params) && $this->validate())) {

    $query->joinWith(['status']);

    return $dataProvider;
}

$this->addSearchParameter($query, 'id');
$this->addSearchParameter($query, 'carousel_name', true);
$this->addSearchParameter($query, 'image_height');
$this->addSearchParameter($query, 'image_width');
$this->addSearchParameter($query, 'carousel_autoplay');
$this->addSearchParameter($query, 'show_indicators');
$this->addSearchParameter($query, 'show_captions');
$this->addSearchParameter($query, 'show_caption_title');
$this->addSearchParameter($query, 'show_caption_background');
$this->addSearchParameter($query, 'caption_font_size');
$this->addSearchParameter($query, 'show_controls');
$this->addSearchParameter($query, 'status_id');

// filter by gender name

$query->joinWith(['status' => function ($q) {
    $q->andFilterWhere(['=' , 'status.status_name', $this->statusName]);
}]);
```

```
        }]);

        return $dataProvider;

    }

protected function addSearchParameter($query, $attribute, $partialMatch = false)
{
    if (($pos = strrpos($attribute, '.')) !== false) {
        $modelAttribute = substr($attribute, $pos + 1);
    } else {
        $modelAttribute = $attribute;
    }

    $value = $this->$modelAttribute;
    if (trim($value) === '') {
        return;
    }

    /*
     * The following line is additionally added for right aliasing
     * of columns so filtering happen correctly in the self join
     */
}

$attribute = "carousel_settings.$attribute";

if ($partialMatch) {
    $query->andWhere(['like', $attribute, $value]);
} else {
    $query->andWhere([$attribute => $value]);
}
}

public static function getCarouselSettings($carousel_name)
{
    $carouselSettings = CarouselSettings::find()
        ->where(['carousel_name' => $carousel_name])
        ->one();

    return $carouselSettings;
```

```
    }  
}
```

Most of this is boilerplate, but we do have the `getCarouselSettings` method to return an AR instance holding our settings. You'll note that we pass `$carousel_name` into the method signature and also that it's static, so we could call it like so:

```
$carouselSettings = CarouselSettingsSearch::getCarouselSettings('Front Page');
```

We could then pass `$carouselSettings` into a view via a controller and have access to all the settings.

CarouselSettingsController

Gist:

[CarouselSettings Controller](#)

From book:

```
<?php  
  
namespace backend\controllers;  
  
use Yii;  
use backend\models\CarouselSettings;  
use backend\models\search\CarouselSettingsSearch;  
use yii\web\Controller;  
use yii\web\NotFoundHttpException;  
use yii\filters\VerbFilter;  
use common\models\PermissionHelpers;  
  
/**  
 * CarouselSettingsController implements the CRUD actions for CarouselSettings model.  
 */  
class CarouselSettingsController extends Controller  
{  
    public function behaviors()  
    {  
        return [  
    }
```

```
'access' => [
    'class' => \yii\filters\AccessControl::className(),
    'only' => ['index', 'view', 'create', 'update', 'delete'],
    'rules' => [
        [
            'actions' => ['index', 'view', 'create', 'update', 'delete'],
            'allow' => true,
            'roles' => ['@'],
            'matchCallback' => function ($rule, $action) {
                return PermissionHelpers::requireMinimumRole('Admin')
                    && PermissionHelpers::requireStatus('Active');
            }
        ],
    ],
],
];

'verbs' => [
    'class' => VerbFilter::className(),
    'actions' => [
        'delete' => ['post'],
    ],
],
];
};

/**
 * Lists all CarouselSettings models.
 * @return mixed
 */

public function actionIndex()
{
    $searchModel = new CarouselSettingsSearch();
    $dataProvider = $searchModel->search(Yii::$app->request->queryParams);

    return $this->render('index', [
        'searchModel' => $searchModel,
        'dataProvider' => $dataProvider,
    ]);
}
```

```
}

/**
 * Displays a single CarouselSettings model.
 * @param integer $id
 * @return mixed
 */
public function actionView($id)
{
    return $this->render('view', [
        'model' => $this->findModel($id),
    ]);
}

/**
 * Creates a new CarouselSettings model.
 * If creation is successful, the browser will be
 * redirected to the 'view' page.
 * @return mixed
 */
public function actionCreate()
{
    $model = new CarouselSettings();

    if ($model->load(Yii::$app->request->post()) && $model->save()) {

        return $this->redirect(['view', 'id' => $model->id]);
    } else {
        return $this->render('create', [
            'model' => $model,
        ]);
    }
}

/**
 * Updates an existing CarouselSettings model.
 * If update is successful, the browser will be
 * redirected to the 'view' page.
 * @param integer $id
 * @return mixed
 */
public function actionUpdate($id)
```

```
{  
    $model = $this->findModel($id);  
  
    if ($model->load(Yii::$app->request->post()) && $model->save()) {  
        return $this->redirect(['view', 'id' => $model->id]);  
    } else {  
        return $this->render('update', [  
            'model' => $model,  
        ]);  
    }  
}  
  
/**  
 * Deletes an existing CarouselSettings model.  
 * If deletion is successful, the browser will be  
 * redirected to the 'index' page.  
 * @param integer $id  
 * @return mixed  
 */  
public function actionDelete($id)  
{  
    $this->findModel($id)->delete();  
  
    return $this->redirect(['index']);  
}  
  
/**  
 * Finds the CarouselSettings model based on its primary key value.  
 * If the model is not found, a 404 HTTP exception will be thrown.  
 * @param integer $id  
 * @return CarouselSettings the loaded model  
 * @throws NotFoundHttpException if the model cannot be found  
 */  
protected function findModel($id)  
{  
    if (($model = CarouselSettings::findOne($id)) !== null) {  
        return $model;  
    } else {  
        throw new NotFoundHttpException('The requested page does not exist.');  
    }  
}
```

All we're doing there is checking for admin permission, the rest is straight from Gii.

CarouselSettings _form View

Gist:

[CarouselSettings _form view](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $model backend\models\CarouselSettings */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="carousel-settings-form">

    <?php $form = ActiveForm::begin(); ?>

    <?= $form->field($model, 'carousel_name')->textInput(['maxlength' => 45]) ?>

    <?= $form->field($model, 'image_height')->textInput(['maxlength' => 45]) ?>

    <?= $form->field($model, 'image_width')->textInput(['maxlength' => 45]) ?>

    <?= $form->field($model, 'carousel_autoplay')
        ->dropDownList($model->carouselAutoplayList,
        ['prompt' => 'Please Choose One']);?>

    <?= $form->field($model, 'show_indicators')
        ->dropDownList($model->showIndicatorsList,
        ['prompt' => 'Please Choose One']);?>

    <?= $form->field($model, 'show_caption_title')
        ->dropDownList($model->showCaptionTitleList,
        ['prompt' => 'Please Choose One']);?>

    <?= $form->field($model, 'show_captions') ?>
```

```
->dropDownList($model->showCaptionsList,
['prompt' => 'Please Choose One']);?>

<?= $form->field($model, 'show_caption_background')
->dropDownList($model->showCaptionBackgroundList,
['prompt' => 'Please Choose One']);?>

<?= $form->field($model, 'caption_font_size')
->textInput(['maxlength' => 45]) ?>

<?= $form->field($model, 'show_controls')
->dropDownList($model->showControlsList,
['prompt' => 'Please Choose One']);?>

<?= $form->field($model, 'status_id')
->dropDownList($model->statusList);?>

<div class="form-group">
    <?= Html::submitButton($model->isNewRecord ? 'Create' : 'Update',
['class' => $model->isNewRecord ? 'btn btn-success' : 'btn btn-primary']) ?>
</div>

<?php ActiveForm::end(); ?>

</div>
```

We just cleaned up the form with dropdown lists, like we always do.

CarouselSettings view.php View

Gist:

CarouselSettings view.php

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\DetailView;

/* @var $this yii\web\View */
/* @var $model backend\models\CarouselSettings */

$this->title = $model->carousel_name;
$this->params['breadcrumbs'][] = ['label' => 'Carousel Settings',
    'url' => ['index']];
$this->params['breadcrumbs'][] = $this->title . ' Carousel';
?>
<div class="carousel-settings-view">

    <h1><?= Html::encode($model->carousel_name) ?> Carousel</h1>

    <p>
        <?= Html::a('Update', ['update', 'id' => $model->id],
            ['class' => 'btn btn-primary']) ?>
        <?= Html::a('Delete', ['delete', 'id' => $model->id], [
            'class' => 'btn btn-danger',
            'data' => [
                'confirm' => 'Are you sure you want to delete this item?',
                'method' => 'post',
            ],
        ]) ?>
    </p>

    <?= DetailView::widget([
        'model' => $model,
        'attributes' => [
            'id',
            'carousel_name',
            'image_height',
            'image_width',
            ['attribute' => 'carousel_autoplay', 'format' => 'boolean'],
            ['attribute' => 'show_indicators', 'format' => 'boolean'],
            ['attribute' => 'show_caption_title', 'format' => 'boolean'],
            ['attribute' => 'show_captions', 'format' => 'boolean'],
            ['attribute' => 'show_caption_background', 'format' => 'boolean'],
            'caption_font_size',
        ],
    ]) ?>
</div>
```

```
[ 'attribute' => 'show_controls', 'format' => 'boolean'] ,  
    'status.status_name',  
    'created_at',  
    'updated_at',  
],  
]) ?>  
</div>
```

Again, this should be familiar to us at this point. I'm providing it for reference, but you shouldn't need it.

CarouselSettings index.php View

Gist:

[CarouselSettings index.php](#)

From book:

```
<?php  
  
use yii\helpers\Html;  
use yii\grid\GridView;  
  
/* @var $this yii\web\View */  
/* @var $searchModel backend\models\search\CarouselSettingsSearch */  
/* @var $dataProvider yii\data\ActiveDataProvider */  
  
$this->title = 'Carousel Settings';  
$this->params['breadcrumbs'][] = $this->title;  
?>  
<div class="carousel-settings-index">  
  
    <h1><?= Html::encode($this->title) ?></h1>  
    <?php // echo $this->render('_search', ['model' => $searchModel]); ?>  
  
    <p>  
        <?= Html::a('Create Carousel Settings', ['create'],  
            ['class' => 'btn btn-success']) ?>  
    </p>
```

```
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],

        'id',
        'carousel_name',
        'image_height',
        'image_width',
        ['attribute' => 'carousel_autoplay', 'format' => 'boolean'],
        ['attribute' => 'show_indicators', 'format' => 'boolean'],
        ['attribute' => 'show_caption_title', 'format' => 'boolean'],
        ['attribute' => 'show_captions', 'format' => 'boolean'],
        ['attribute' => 'show_caption_background', 'format' => 'boolean'],
        'caption_font_size',
        ['attribute' => 'show_controls', 'format' => 'boolean'],

        'statusName',
        //'created_at',
        'updated_at',

        ['class' => 'yii\grid\ActionColumn'],
    ],
]); ?>

</div>
```

This has the same minor formatting changes that we always have on index.php.

CarouselSettings _search View

Gist:

[CarouselSettings _search View](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $model backend\models\search\CarouselSettingsSearch */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="carousel-settings-search">

    <?php $form = ActiveForm::begin([
        'action' => ['index'],
        'method' => 'get',
    ]); ?>

    <?= $form->field($model, 'id') ?>

    <?= $form->field($model, 'carousel_name')->textInput(['maxlength' => 45]) ?>

    <?= $form->field($model, 'image_height')->textInput(['maxlength' => 45]) ?>

    <?= $form->field($model, 'image_width')->textInput(['maxlength' => 45]) ?>

    <?= $form->field($model, 'carousel_autoplay')
        ->dropDownList($model->carouselAutoplayList,
            ['prompt' => 'Please Choose One']);?>

    <?= $form->field($model, 'show_indicators')
        ->dropDownList($model->showIndicatorsList,
            ['prompt' => 'Please Choose One']);?>

    <?= $form->field($model, 'show_caption_title')
        ->dropDownList($model->showCaptionTitleList,
            ['prompt' => 'Please Choose One']);?>

    <?= $form->field($model, 'show_captions')
        ->dropDownList($model->showCaptionsList,
            ['prompt' => 'Please Choose One']);?>

    <?= $form->field($model, 'show_caption_background')
```

```
->dropDownList($model->showCaptionBackgroundList,
['prompt' => 'Please Choose One']);?>

<?= $form->field($model, 'caption_font_size')
->textInput(['maxlength' => 45]) ?>

<?= $form->field($model, 'show_controls')
->dropDownList($model->showControlsList,
['prompt' => 'Please Choose One']);?>

<?= $form->field($model, 'status_id')->dropDownList($model->statusList);?>

<?php echo $form->field($model, 'updated_at') ?>

<div class="form-group">
    <?= Html::submitButton('Search', ['class' => 'btn btn-primary']) ?>
    <?= Html::resetButton('Reset', ['class' => 'btn btn-default']) ?>
</div>

<?php ActiveForm::end(); ?>

</div>
```

We're not using search, especially since we will only have one record, but I formatted it anyway, in case things change in the future and we find ourselves searching for carousels.

PagesController

Gist:

[Pages Controller](#)

From book:

```
<?php

namespace frontend\controllers;

use Yii;
use frontend\models\ContactForm;
use yii\filters\AccessControl;
use backend\models\search\CarouselSettingsSearch;

class PagesController extends \yii\web\Controller
{

    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::className(),
                'only' => ['captcha'],
                'rules' => [
                    [
                        'actions' => ['captcha'],
                        'allow' => true,
                        'roles' => ['?', '@'],
                    ],
                    [
                        'allow' => true,
                    ],
                ],
            ],
        ];
    }

    public function actions()
    {
        return [
            'error' => [
                'class' => 'yii\web\ErrorAction',
            ],
            'captcha' => [
                'class' => 'yii\captcha\CaptchaAction',
                'fixedVerifyCode' => YII_ENV_TEST ? 'testme' : null,
            ],
        ];
    }
}
```

```
        ];
    }

    public function actionIndex()
    {

$carouselSettings = CarouselSettingsSearch::getCarouselSettings('Front Page');

    return $this->render('index', ['carouselSettings' => $carouselSettings]);
}

public function actionAbout()
{
    return $this->render('about');
}

public function actionContact()
{
    $model = new ContactForm();
    if ($model->load(Yii::$app->request->post()) && $model->validate()) {
        if ($model->sendEmail(Yii::$app->params['adminEmail'])) {
            Yii::$app->session->setFlash('success', 'Thank you for contacting
us. We will respond to you as soon as possible.');
        } else {
            Yii::$app->session->setFlash('error', 'There was an error sending email.');
        }
        return $this->refresh();
    } else {
        return $this->render('contact', [
            'model' => $model,
        ]);
    }
}

public function actionPrivacy()
{
```

```

        return $this->render('privacy');
    }

    public function actionTermsService()
    {
        return $this->render('terms-service');
    }

}

```

The main thing to take note of here is the index action:

```

public function actionIndex()
{
    $carouselSettings = CarouselSettingsSearch::getCarouselSettings('Front Page');

    return $this->render('index', ['carouselSettings' => $carouselSettings]);
}

```

Here we are calling our getCarouselSettings method, so we can pass it to Pages index view.

Pages Index View

Since at this point we're only modifying the widget, I'm just going to supply you with that change.

Gist:

[Pages Index View Change](#)

From book:

```

<?= CarouselWidget::widget(['settings' =>
    ['height' => $carouselSettings['image_height']],
    ['width' => $carouselSettings['image_width']],
    ['autoplay' => $carouselSettings['carousel_autoplay']],
    ['show_indicators' => $carouselSettings['show_indicators']],
    ['show_captions' => $carouselSettings['show_captions']],
    ['show_controls' => $carouselSettings['show_controls']],
    ['show_caption_title' => $carouselSettings['show_caption_title']],
])?>

```

That takes us to the widget itself.

CarouselWidget

Gist:

[CarouselWidget](#)

From book:

```
<?php

namespace components;

use yii\base\Widget;
use yii\helpers\Html;
use Yii;
use backend\models\MarketingImage;
use backend\models\search\MarketingImageSearch;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;

class CarouselWidget extends Widget
{

    public $activeImage;
    public $images;
    public $count;
    public $settings = [];

    public function init()
    {
        parent::init();

        $this->activeImage = MarketingImage::find('marketing_image_path')
            ->where(['marketing_image_is_active' => 1])
            ->andWhere(['marketing_image_is_featured' => 1])
            ->andWhere(['status_id' => 1])
            ->one();

        $this->images = MarketingImage::find()
            ->where(['marketing_image_is_active' => 0])
            ->andWhere(['marketing_image_is_featured' => 1])
            ->andWhere(['status_id' => 1])
    }
}
```

```
->orderBy('marketing_image_weight')
->all();

$this->count = MarketingImage::find()
->where(['marketing_image_is_active' => 0])
->andWhere(['marketing_image_is_featured' => 1])
->andWhere(['status_id' => 1])
->count();

$this->setDefaults();

$this->validateSize();

}

public function setDefaults()
{

    if(!isset($this->settings['height'])){
        $this->settings['height'] = '300px';
    }

    if(!isset($this->settings['width'])){
        $this->settings['width'] = '700px';
    }

    if(!isset($this->settings['caption_font_size'])){
        $this->settings['caption_font_size'] = '15px';
    }

    if(!isset($this->settings['autoplay'])){
        $this->settings['autoplay'] = true;
    }

    if(!isset($this->settings['show_indicators'])){
        $this->settings['show_indicators'] = true;
    }
}
```

```
    if(!isset($this->settings['show_captions'])){

        $this->settings['show_captions'] = false;
    }

    if(!isset($this->settings['show_caption_background'])){

        $this->settings['show_caption_background'] = false;
    }

    if(!isset($this->settings['show_caption_title'])){

        $this->settings['show_caption_title'] = false;
    }

    if(!isset($this->settings['show_controls'])){

        $this->settings['show_controls'] = true;
    }

}

public function validateSize()
{

    if (!preg_match("/px/", $this->settings['width'])
    or !preg_match("/px/", $this->settings['height'])) {

        throw new NotFoundHttpException
        ('You Must Use px and number for size, example 300px');

    }

    $height = (int) preg_replace("/[^0-9]/","", $this->settings['height']);

    switch ($height){

        case $height < 40 :
```

```
        throw new NotFoundHttpException('You Must Stay within
40 to 1000 px and use px and number for size, example 300px');
break;

case $height > 1000 :
throw new NotFoundHttpException('You Must Stay within
40 to 1000 px and use px and number for size, example 300px');
break;

}

$width = (int) preg_replace("/[^0-9]/","", $this->settings['width']);

switch ($width){

case $width < 40 :
throw new NotFoundHttpException('You Must Stay within
40 to 1000 px and use px and number for size, example 300px');
break;

case $width > 1000 :
throw new NotFoundHttpException('You Must Stay within
40 to 1000 px and use px and number for size, example 300px');
break;

}

}

public function run()
{

return $this->render('carousel', ['activeImage' => $this->activeImage,
                                'images' => $this->images,
                                'count' => $this->count,
                                'settings' => $this->settings]);
}

}
```

Obviously we have more defaults in our setDefaults method. Also, a reminder that some of the

settings are shortened, like in the case of height and width. Their names are different in the model and DB, but you can call them what you want in the widget, as long as you remember what you named them.

carousel.php

That brings us to carousel.php.

Gist:

[carousel.php](#)

From book:

```
<?php

use yii\helpers\Html;

?>

<div id="carouselMain" class="carousel slide"

<?php

if($settings['autoplay'] == false ){

    echo 'data-interval="false"';
}

?>

data-ride="carousel">

<!-- Indicators --> <?php

if ($settings['show_indicators']){
    echo '<ol class="carousel-indicators">
        <li data-target="#carouselMain" data-slide-to="0"
            class="active"></li>';
    foreach (range(1, $count) as $number) {
        echo '<li data-target="#carouselMain" data-slide-to="'
            . $number . '"';
    }
}<?php

?>
```

```
echo '<li data-target="#carouselMain" data-slide-to="'.  
    $number.'"></li>';  
  
}  
  
echo '</ol> ';  
  
}  
  
?  
  
<!-- Wrapper for slides -->  
  
<div class="carousel-inner" role="listbox">  
  
<!-- dynamic slide data -->  
  
<?php  
  
$width = $settings['width'];  
$height = $settings['height'];  
  
//active item first  
  
echo '<div class="item active">  
    <center>' .  
  
        Html::img(Yii::$app->urlManagerBackend->baseUrl. '/' .  
            $activeImage['marketing_image_path'],  
            ['width' => $width, 'height' => $height ])  
  
    . '</center>';  
  
if($settings['show_captions']){  
  
    echo '<div class="carousel-caption">';  
  
    if ($settings['show_caption_title']){  
  
        echo '<div><h1>' .  
            $activeImage['marketing_image_caption_title'] .
```

```
' </h1></div>';

}

echo $activeImage['marketing_image_caption'] . '</div>';

}

echo '</div>';

//all other images

foreach ($images as $image){

echo '<div class="item">
<center>' .
Html::img(Yii::$app->urlManagerBackend->baseUrl .
'/' . $image['marketing_image_path'],
['width' => $width, 'height' => $height ]) .
'</center>';

if($settings['show_captions']){

echo '<div class="carousel-caption">';

if ($settings['show_caption_title']){

echo    '<div><h1>' .
$image['marketing_image_caption_title'] .
'</h1></div>';

}

echo $image['marketing_image_caption']. '</div>';

}

echo '</div>';

}

?>
```

```
<!-- end dynamic slide data -->

</div>

<!-- Controls -->

<?php

if ($settings['show_controls']){

    echo '<a class="left carousel-control"
        href="#carouselMain" role="button"
        data-slide="prev">
        <span class="glyphicon glyphicon-chevron-left"
        aria-hidden="true"></span>
        <span class="sr-only">Previous</span>
    </a>

    <a class="right carousel-control"
        href="#carouselMain" role="button"
        data-slide="next">
        <span class="glyphicon glyphicon-chevron-right"
        aria-hidden="true"></span>
        <span class="sr-only">Next</span>
    </a>' ;

}

?>

</div>
```

The really simple explanation to that code is we simply added a bunch of if statements that rely on our settings and dynamic data. For example:

```
if($settings['show_captions']){
    // do something
}
```

So if ‘show_captions’ is true, we get to see the captions. Then nested further down, we test for the caption title:

```
if ($settings['show_caption_title']){
```

So the if statements in this code control:

- autoplay
- show_indicators
- show_captions
- show_caption_title
- show_controls

So how do we control caption_font_size and show_caption_background?

Those are CSS elements. There may be more than one way to dynamically control CSS, but the way I chose to do it was to place a small amount of style code in frontend/views/layouts/main.php:

Main

I'll give you the entire file first, then we'll focus on the relevant bits.

Gist:

[layouts/main.php](#)

Froom book:

```
<?php
use yii\helpers\Html;
use yii\bootstrap\Nav;
use yii\bootstrap\NavBar;
use yii\widgets\Breadcrumbs;
use frontend\assets\AppAsset;
use frontend\widgets\Alert;
use frontend\assets\FontAwesomeAsset;
use backend\models\search\CarouselSettingsSearch;

/* @var $this \yii\web\View */
/* @var $content string */

AppAsset::register($this);
FontAwesomeAsset::register($this);

?>
```

```
<?php $this->beginPage() ?>
<!DOCTYPE html>
<html lang="= Yii::$app-&gt;language ?&gt;"&gt;
&lt;head&gt;
    &lt;meta charset="<?= Yii::$app-&gt;charset ?&gt;"/&gt;
    &lt;meta name="viewport" content="width=device-width, initial-scale=1"&gt;
    &lt;?= Html::csrfMetaTags() ?&gt;
    &lt;title&gt;&lt;?= Html::encode($this-&gt;title) ?&gt;&lt;/title&gt;

&lt;?php

$carouselSettings = CarouselSettingsSearch::getCarouselSettings('Front Page');

if($carouselSettings['caption_font_size']){
    echo '&lt;style&gt;.carousel-caption{';
    if ($carouselSettings['show_caption_background']){
        echo 'background: rgba(0,0,0,0.5);';
    }
    echo 'font-size:' . $carouselSettings['caption_font_size'] . ';
    }
    '&lt;/style&gt;';
}

?&gt;

&lt;?php $this-&gt;head() ?&gt;
&lt;/head&gt;
&lt;body&gt;
    &lt;?php $this-&gt;beginBody() ?&gt;

    &lt;div class="wrap"&gt;
        &lt;?php

            NavBar::begin([
                'brandLabel' =&gt; 'My Yii Application',
                'brandUrl' =&gt; Yii::$app-&gt;homeUrl,
                'options' =&gt; [
                    'class' =&gt; 'navbar-inverse navbar-fixed-top',
                ],
            ]);
        &lt;?php
            NavBar::end();
        &lt;?php
    </pre
```

```
'brandLabel' => 'Yii 2 Start <i class="fa fa-plug"></i>',
'brandUrl' => Yii::$app->homeUrl,
'options' => [
    'class' => 'navbar-inverse navbar-fixed-top',
],
]);

$menuItems = [
//['label' => 'Home', 'url' => ['/site/index']],
['label' => 'About', 'url' => ['/pages/about']],
['label' => 'FAQs', 'url' => ['/faq/index']],
['label' => 'Contact', 'url' => ['/pages/contact']],
];

if (Yii::$app->user->isGuest) {
    $menuItems[] = ['label' => 'Signup', 'url' => ['/site/signup']];
    $menuItems[] = ['label' => 'Login', 'url' => ['/site/login']];
}

} else {
    $menuItems[] = ['label' => 'Social Sync', 'items' => [
        ['label' => '<i class="fa fa-facebook"></i> Facebook',
         'url' => ['site/auth', 'authclient' => 'facebook']],
        ['label' => '<i class="fa fa-github"></i> Github',
         'url' => ['site/auth', 'authclient' => 'github']],
        ['label' => '<i class="fa fa-twitter"></i> Twitter',
         'url' => ['site/auth', 'authclient' => 'twitter']],
        ['label' => '<i class="fa fa-linkedin"></i> LinikedIn',
         'url' => ['site/auth', 'authclient' => 'linkedin']],
        ['label' => '<i class="fa fa-google"></i> Google+',
         'url' => ['site/auth', 'authclient' => 'google']],
    ]];
}

$menuItems[] = ['label' => 'Profile', 'url' => ['/profile/view']];
$menuItems[] = [
    'label' => 'Logout (' . Yii::$app->user->identity->username . ')',
    'url' => ['/site/logout'],
];
```

```

    'linkOptions' => [ 'data-method' => 'post' ]
];
}

echo Nav::widget([
    'options' => [ 'class' => 'navbar-nav navbar-right' ],
    'items' => $menuItems,
    'encodeLabels' => false,
]);

NavBar::end();
?>

<div class="container">
<?= Breadcrumbs::widget([
    'links' => isset($this->params['breadcrumbs']) ?
        $this->params['breadcrumbs'] : [],
]) ?>
<?= Alert::widget() ?>
<?= $content ?>
</div>
</div>

<footer class="footer">
    <div class="container">
        <p class="pull-left">&copy; My Company <?= date('Y') ?></p>
        <p class="pull-right"><?= Yii::powered() ?></p>
    </div>
</footer>

<?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>

```

So to take control of the desired CSS elements, we first have to include our use statement:

```
use backend\models\search\CarouselSettingsSearch;
```

Next we added a block of PHP that will echo out the style tags with our if statements controlling the scenario:

```
<?php

$carouselSettings = CarouselSettingsSearch::getCarouselSettings('Front Page');

if($carouselSettings['caption_font_size']){

echo '<style>.carousel-caption{';

if ($carouselSettings['show_caption_background']){
echo 'background: rgba(0,0,0,0.5);';

}

echo 'font-size:' . $carouselSettings['caption_font_size'] . ';

}
</style>';
}

?>
```

So obviously, we are getting the relevant settings:

```
$carouselSettings = CarouselSettingsSearch::getCarouselSettings('Front Page');
```

Then if a caption_font_size has been set, we enforce that as well as check for the show_caption_background setting. If there is no caption_font_size selected, there will be no background.

Note that the caption title is controlled by an h1 tag, and you could make a setting for that and control it in a similar fashion.

And that's pretty much it for our control over the carousel. Play around with the different settings and have fun.

Summary

This chapter turned out to be extensive. We built our basic image management that lets us upload, edit and delete images and their corresponding thumbnails.

Along the way we learned about scenarios and we used yii2-imagine extension to make thumbnail creation as simple as it can be. But since we're building a template, we wanted to immediately put our image management capability to good use. So we decided to build a dynamically-driven carousel that would display marketing images.

It's often very important for our clients to have control over things like their marketing message, so we also made a full UI for carousel settings that are respected by the custom widget we built. This gives a lot of control over the carousel directly to the client and they will love that.

While I was writing this chapter, some of our readers made nice comments, along with positive reviews and ratings on Goodreads.com. That really inspired me to work as hard as I could to deliver all the features in the carousel, which took multiple revisions on my part. So please keep all the comments and reviews coming, I really appreciate it.

Thanks once again for taking the Yii 2 journey with me. More will be coming. I hope to see you soon.

Chapter 17: Bonus Material Ratings Widget

We're back again for more fun with Yii 2. I hope you are enjoying the bonus material.

I was looking at Kartik's widgets at:

[Krajee.com](#)

And I thought his big yii2-widgets extension would come in very handy for future projects.

I counted 24 widgets on the Git page, all available in one install, so this is like hitting the jackpot. And because you have them all installed at once, you don't have to install them individually, which saves you a lot of time.

If you are unfamiliar with that package, I suggest you take a look:

[Yii2 Widgets](#)

I actually had my eye on several of Kartik's widgets to implement for the template, but I wanted to start with something simple to get started. So I chose to implement the StarRating widget.

This provides a nice bootstrap rating scale that we can implement on our Faq model, since that is the only list of records in our application that faces the public and is something they could rate.

It's totally up to you if you would use this on your actual template, so this is optional of course. But this is a good way to get familiar with it.

I thought this would be super-easy to implement since I had such good luck with Kartik's social widget. Well, it got a little complicated, not because of the widget itself, which is a snap to install, but because we end up with a slightly more complex scenario than what we have been used to.

And because of that, it makes for a perfect lesson.

So let's start by adding the following to your composer.json file:

```
"kartik-v/yii2-widgets": "*",
```

Then run composer update on the command line:

```
C:\var\www\yii2build>composer update
```

composer update

Next we need to decide on a data structure. I decided to call my new table faq_rating:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>					
user_id	INT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
faq_id	INT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
faq_rating	DOUBLE	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
created_at	DATETIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					
updated_at	DATETIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					

faq_rating table

Here is the SQL:

```
CREATE TABLE IF NOT EXISTS `yii2build`.`faq_rating` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `user_id` INT(11) NOT NULL,
  `faq_id` INT(11) NOT NULL,
  `faq_rating` DOUBLE NOT NULL,
  `created_at` DATETIME NOT NULL,
  `updated_at` DATETIME NOT NULL,
  PRIMARY KEY (`id`)
```

You'll note that we chose double as our datatype for 'faq_rating' and this allows us to have the decimal in the rating such as 2.5 etc.

Next we need to create the model from Gii. I'm going to put the model in the backend. Lately I've decided to put models in the backend, unless I have good reason to do otherwise.

You are of course free to structure it how you wish. As long as you namespace everything properly, you have a lot of flexibility in your choices.

You may have noticed that our table references to other primary keys, user_id and faq_id. This is also known as a junction table or a pivot table. If we needed to reference the relationships, they would be many to many relationships.

But with a rating system, at least our implementation of it, we don't really need to reference it that way, so we didn't bother creating foreign keys or setting up the relationships.

If you wanted to get more complex, for example, if you wanted to write a report that showed all the faq_ratings made by a specific user, then the relationships would be important. But we are not anticipating that use case for this, so that's just something to keep in mind if you do a more complex integration with another model.

For our purposes here, we only need to store the ratings so we can calculate an average, store the faq_id the rating applies to, and store the user_id so that people don't rate more than once.

We will also allow people to update their ratings. Alternatively, you could prevent them from doing so if you wish. I will point out where you could do that later if you want to go in that direction.

So now we have to imagine our implementation. How does the rating fit in with the Faqs?

Ultimately, we want it to look like this:

The screenshot shows a Yii 2 Build application interface. At the top, there is a dark header bar with the text "Yii 2 Build" and a small icon. Below the header, a breadcrumb navigation bar shows "Home / FAQ / FAQ: Should I use a framework?". The main content area has a title "Should I use a framework?" followed by a subtitle "Probably, it's a good idea". Below the subtitle is a "Faq Rating" section. This section includes a "Remove" button (-), five yellow stars, and a green button labeled "5 Stars". At the bottom of the rating section is a button labeled "Add Your Rating".

Faq with Rating

So if you land on an Faq view page, you see the rating below the faq.

And if you click the Add Your Rating button:

The screenshot shows a Yii 2 Build FAQ page titled "Should I use a framework?". The main content area contains the text "Probably, it's a good idea". Below this, there is a "Rate This FAQ" section. It features a star rating system with five stars, where the first four are filled and the fifth is empty. To the left of the stars is a minus sign button (-) and to the right is a "Not Rated" button. A green "Rate It!" button is located below the rating controls. The entire interface is contained within a light gray box.

Add Your Rating

So once we click on Add Your Rating, we get the above screen. If you mouse over the stars, they fill in, then you push the rate it button and the rating gets recorded.

So this seems incredibly easy, until you realize we are combining models, using both the Faq model and the FaqRating model.

That raises all kinds of interesting questions about what controller is going to do what, as well as how to mix the models onto a single view page, since the StarRating widget is going to pass us a value that we need move along via a form.

Taking a clue from Yii 2 itself, it seems like forms are best created as partials, so our plan will be to create a form partial for our FaqRating model and control form submission through our soon-to-be-made FaqRatingController.

But first, let's look at the model.

FaqRatings Model

Gist:

[FaqRatings Model](#)

From book:

```
<?php

namespace backend\models;

use Yii;
use yii\db\ActiveRecord;
use yii\db\Expression;
use yii\helpers\ArrayHelper;
use kartik\widgets\StarRating;

/**
 * This is the model class for table "faq_rating".
 *
 * @property integer $id
 * @property integer $user_id
 * @property integer $faq_id
 * @property double $faq_rating
 * @property string $created_at
 * @property string $updated_at
 */
class FaqRating extends \yii\db\ActiveRecord
{
    /**
     * @inheritdoc
     */
    public static function tableName()
    {
        return 'faq_rating';
    }

    public function behaviors()
    {
        return [
            'timestamp' => [
                'class' => 'yii\behaviors\TimestampBehavior',
                'attributes' => [
                    ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                    ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
                ],
                'value' => new Expression('NOW()'),
            ],
        ];
    }
}
```

```
}

public function beforeValidate()
{
    $this->faq_rating = (double)$this->faq_rating;
    $this->faq_id = (int)$this->faq_id;

    return parent::beforeValidate();
}

public function rules()
{
    return [
        [['user_id', 'faq_id', 'faq_rating'], 'required'],
        [['user_id', 'faq_id'], 'integer'],
        [['faq_rating'], 'double'],
        [['created_at', 'updated_at'], 'safe']
    ];
}

public function attributeLabels()
{
    return [
        'id' => 'ID',
        'user_id' => 'User ID',
        'faq_id' => 'Faq ID',
        'faq_rating' => 'Faq Rating',
        'created_at' => 'Created At',
        'updated_at' => 'Updated At',
    ];
}

public function showAverageRating($faq_id)
{
    $averageRating = $this->getAverageRating($faq_id);
```

```
echo StarRating::widget([
    'name' => 'rating_' . $averageRating,
    'value' => $averageRating,
    'disabled' => true,
    'pluginOptions' => [
        'size' => 'sm',
        'stars' => 5,
        'min' => 0,
        'max' => 5,
        'step' => 0.5,
        // 'symbol' => html_entity_decode('&#xe005;', ENT_QUOTES, "utf-8"),
        // 'defaultCaption' => '{rating} hearts',
        'starCaptions'=>[]
    ]
]);

}

public function getAverageRating($faq_id)
{
    $ratings = FaqRating::find('faq_rating')->asArray()
        ->where(['faq_id' => $faq_id])
        ->all();

    $ratings = ArrayHelper::map($ratings, 'id', 'faq_rating');

    $ratingsSum = array_sum($ratings);

    $ratingsCount = count($ratings);

    if($ratingsCount){
        $averageRating = $ratingsSum/$ratingsCount;
    } else {

```

```

    $averageRating = 0;
}

return $averageRating;
}
}

```

So that is our change from the boilerplate. Obviously, we added timestamp behavior and the appropriate use statements to use it. We also modified our rules, removing ‘created_at’ and ‘updated_at’ from being required.

If you don’t remove the required rule for those columns, you get hard-to-diagnose validation errors. This happens when validation fails behind the scenes, not in an obvious way related to the form.

For example, if you don’t fill out a required field on a form, you don’t get to submit, and it makes it fairly obvious what the problem is. However, when the problem is not directly related to the form, things can get tricky.

Let’s say for example, you forget to make the table auto-increment and then you use Gii to create the Model and Crud. Gii will put the id as an input field because it’s not done automatically. Let’s say you see the id field, not realizing your mistake and you remove that field from the form, thinking it would be a simple fix.

What would happen in that scenario is that when you submit the form, you would get a blank white screen and no record inserted. More importantly, you would get no error message telling you what’s wrong.

\$model->getErrors()

Most programmers know how to use var_dump() to help figure out problems, but these validation errors can be a real pain to sort out. Fortunately, Yii 2 gives us a model method that returns the errors that you can use like so:

```

var_dump($model->getErrors());
die();

```

You can test the save method like so:

```

if (!$companyrecord->save()) {
    var_dump($companyrecord->getErrors());
}

```

In this example, \$companyrecord is the model.

In some cases where validation fails on saving a record, you could still have errors that don’t show up in that var_dump, so there is another method.

Overwrite Save Method on Model

You can overwrite the save method on the model to see if you can find the errors. I've had to use this several times and it's really handy:

```
public function save($runValidation = true, $attributeNames = NULL)
{
    if(!parent::save()){

        echo 'something is wrong';
        die();

        //var_dump($this->getErrors());


    }
}
```

You would add this method to whichever model you are trying to troubleshoot.

Just remember that after you use this to diagnose a problem that you should remove it in order to restore full functionality.

Anyway, those are some tips on how to overcome validation errors that are not otherwise visible to you.

Ok, let's get back to our new model, FaqRatings. In addition to the changes we already talked about, we have two new methods, showAverageRating and getAverageRating. Let's discuss the second one first.

```
public function getAverageRating($faq_id)
{
    $ratings = FaqRating::find('faq_rating')->asArray()
        ->where(['faq_id' => $faq_id])
        ->all();

    $ratings = ArrayHelper::map($ratings, 'id', 'faq_rating');

    $ratingsSum = array_sum($ratings);
```

```
$ratingsCount = count($ratings);

if($ratingsCount){

    $averageRating = $ratingsSum/$ratingsCount;

} else {

    $averageRating = 0;
}

return $averageRating;
}
```

You can see we need the id of the faq in the signature, then we use ActiveRecord to get all ratings that have an faq_id equal to the faq_id that we handed in. We're taking the results as an array because its easy to work with these values in this format.

Since we have all our ratings stored in our \$ratings array, we use ArrayHelper::map to filter everything out except the two values we want, id and faq_rating. So now we can use:

```
$ratingsSum = array_sum($ratings);
```

This will add up the total value of all ratings, which we can then divide by the number of ratings, for which we need to count the ratings array:

```
$ratingsCount = count($ratings);
```

Then we check to see if there is a count, and if so do the calculation to return the average:

```
if($ratingsCount){

    $averageRating = $ratingsSum/$ratingsCount;

} else {

    $averageRating = 0;
}

return $averageRating;
```

So that is how we get our average rating.

We've also built a method to show our average rating:

```
public function showAverageRating($faq_id)
{
    $averageRating = $this->getAverageRating($faq_id);

    echo StarRating::widget([
        'name' => 'rating_' . $averageRating,
        'value' => $averageRating,
        'disabled' => true,
        'pluginOptions' => [
            'size' => 'sm',
            'stars' => 5,
            'min' => 0,
            'max' => 5,
            'step' => 0.5,
            'starCaptions'=>[]
        ]
    ]);

}
```

In order to show the average rating, we need to get the average rating first, which is why we built the other method first. Once we have the average rating, we can simply plug it into the widget in the appropriate places, as shown above.

I don't usually echo a widget within a model method, but I did so this time to make things easy on us. From the Faq controller, I'll pass an instance of the FaqRatings model, which will make this method available to the view.

Some things to note about the widget. We can set the size to small via 'sm', the number of stars, in this case 5, and the step, which indicates a partial value on the star. You can reference Krajee.com if you want to know more about what settings you can use.

Faq Controller

We need to modify the view action as follows.

Gist:

[Frontend Faqcontroller View Action](#)

From book:

```

public function actionView($id, $slug = null)
{
    $model = $this->findModel($id);

    $faqRating = new FaqRating();

    if ($slug == $model->slug){

        return $this->render('view', [
            'model' => $model,
            'slug' => $model->slug,
            'faqRating' => $faqRating,
        ]);
    } else {
        throw new NotFoundHttpException('The requested Faq does not exist.');
    }
}

```

You can see all we did was make a new FaqRating instance available to the view.

Frontend Faq view.php

So let's look at the view.

Gist:

[Faq View](#)

From book:

```

<?php
use yii\helpers\Html;
use kartik\widgets\Growl;

$this->title = 'FAQ: ' . $model->faq_question;

$this->params['breadcrumbs'][] = ['label' => 'FAQ', 'url' => ['index']];
$this->params['breadcrumbs'][] = $this->title;
?>

</br>
<div class="panel panel-default">

```

```
<div class="panel-heading">
    <h3 class="panel-title">

        <h1>    <?= $model->faq_question;?> </h1>

    </h3>
</div>

<?= '<div class="panel-body"><h3>' . $model->faq_answer . '</h3></div>';?>

</div>
<?php

if (Yii::$app->getSession()->hasFlash('success')){

    echo Growl::widget([
        'type' => Growl::TYPE_SUCCESS,
        'title' => 'Thank you!',
        'icon' => 'glyphicon glyphicon-ok-sign',
        'body' => Yii::$app->session->setFlash('success'),
        'showSeparator' => true,
        'delay' => 0,
        'pluginOptions' => [
            'placement' => [
                'from' => 'top',
                'align' => 'right',
            ]
        ]
    ]);
}

Yii::$app->getSession()->removeFlash('success');

?>
<div id="showAverage">
    <strong> Faq Rating </strong>
<?php

$faqRating->showAverageRating($model->id);

?>
<br>
```

```

<button type="button" id="rateMe" class="btn btn-default">
    Add Your Rating
</button>
</div>

<div id="rateIt">
<?php
echo $this->render('rating-form', ['model'=> $model,
    'faqRating' => $faqRating]);
?>
</div>
<?Php
$script = <<< JS
$(document).ready(function(){
    $("#rateIt").hide();
    $("#rateMe").click(function(){
        $("#showAverage").hide();
        $("#rateIt").show();
    });
});

$);
JS;

$this->registerJs($script);

?>

```

Ok, so we added some new things here. Right below the faq answer, we added a growl widget:

```

<?php

if (Yii::$app->getSession()->hasFlash('success')){

    echo Growl::widget([
        'type' => Growl::TYPE_SUCCESS,
        'title' => 'Thank you!',
        'icon' => 'glyphicon glyphicon-ok-sign',
        'body' => Yii::$app->session->setFlash('success'),
        'showSeparator' => true,
        'delay' => 0,
        'pluginOptions' => [
            'placement' => [

```

```

        'from' => 'top',
        'align' => 'right',
    ]
]
]);
}
}

Yii::$app->getSession()->removeFlash('success');

?>

```

What the Growl::widget does is format a flash message into an animated message. I thought it added a nice touch to the UI and the widget is already part of the Kartik widget extension, so we already had everything we need for it. We will discuss it more in detail when we work on the controller.

Next we have the call to the showAverageRatings method:

```

<div id="showAverage">
    <strong> Faq Rating </strong>
    <?php
        $faqRating->showAverageRating($model->id);

    ?>
    <br>
    <button type="button" id="rateMe" class="btn btn-default">
        Add Your Rating
    </button>
</div>

```

You can see that we added a div id of showAverage. We also have a button within the div with an id of rateMe.

Next we have a call to a form partial, which will allow us to submit the rating:

```

<div id="rateIt">
    <?php
    echo $this->render('_rating-form', ['model'=> $model,
        'faqRating' => $faqRating]);
    ?>
</div>

```

Even without seeing the form yet, you can see that it should render the form, which is odd because we are also displaying the average rating. And yet from our screenshot, we know that it is only supposed to show the form when the add your rating button is clicked.

That brings us to our next section. We manage this by using show/hide in jquery:

```
<?Php
$script = <<< JS
$(document).ready(function(){
    $("#rateIt").hide();
    $("#rateMe").click(function(){
        $("#showAverage").hide();
        $("#rateIt").show();
    });
});

JS;

$this->registerJs($script);

?>
```

We're using heredoc notation to setup our javascript:

```
<?Php
$script = <<< JS
```

You can google heredoc if you are unfamiliar with that notation. It's a very convenient way to integrate PHP and javascript. At the end, we use:

```
$this->registerJs($script);
```

This allows our view to use the asset.

In the script itself, you can see it's pretty simple. As a default, on document ready, we hide anything in the rateIt div.

```
$(document).ready(function(){
    $("#rateIt").hide();
```

Then when someone clicks the Add Your Rating button, which has an id of rateMe, then we hide showAverage and we show the rateIt div:

```
$( "#rateMe" ).click( function(){
    $( "#showAverage" ).hide();
    $( "#rateIt" ).show();
});
```

You don't have to worry about getting back to showAverage from this point because the controller that processes our _rating-form will take care of that.

So this is a demonstration of how to embed javascript directly in the page. The other way to go is to put the file in a js folder under frontend/web and reference it in the AppAsset.php file.

_rating-form.php

Since we're calling _rating-form within this view, create that now. This file should go in the frontend/views/faq folder.

Gist:

[_rating-form.php](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;
use kartik\widgets\StarRating;

/* @var $this yii\web\View */
/* @var $model backend\models\FaqRating */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="faq-rating-form">

    <?php $form = ActiveForm::begin([
        'method' => 'post',
        'action' => ['faq-rating/rating'],
    ]); ?>

    <?= Html::activeHiddenInput($faqRating, 'faq_id',
        [ 'value' => $model->id]) ?>

    <?= $form->field($faqRating, 'faq_rating') ?>
```

```

->label('Rate This FAQ')->widget(StarRating::classname(), [
'pluginOptions' => [
'size' => 'sm',
'stars' => 5,
'min' => 0,
'max' => 5,
'step' => 0.5,
// 'symbol' => html_entity_decode('&#xe005;', ENT_QUOTES, "utf-8"),
// 'defaultCaption' => '{rating} hearts',
'starCaptions'=>[]
]
])?>

<div class="form-group">
<?= Html::submitButton('Rate It!', ['class' => 'btn btn-success']) ?>
</div>

<?php ActiveForm::end(); ?>

</div>

```

Ok, just a few things to note. We are setting the action:

```

<?php $form = ActiveForm::begin([
'method' => 'post',
'action' => ['faq-rating/rating'],
]); ?>

```

Most of the forms we have done so far did not require us to set the action. But this one is not being processed by the FaqController, even though the form is sitting in the faq view folder.

Instead, we are sending this form to the FaqRatings Controller, which we have not created yet.

Ok, back to the form. After setting the action, we are using the StarRating widget for the form field:

```
<?= $form->field($faqRating, 'faq_rating')
    ->label('Rate This FAQ')->widget(StarRating::classname(), [
        'pluginOptions' => [
            'size' => 'sm',
            'stars' => 5,
            'min' => 0,
            'max' => 5,
            'step' => 0.5,
            // 'symbol' => html_entity_decode('&#xe005;', ENT_QUOTES, "utf-8"),
            // 'defaultCaption' => '{rating} hearts',
            'starCaptions'=>[]
        ]
    ])
])?>
```

This will take in the value the user gives it when they select their star rating and submit via the form when the submit button is pressed.

```
<div class="form-group">
    <?= Html::submitButton('Rate It!', ['class' => 'btn btn-success']) ?>
</div>

<?php ActiveForm::end(); ?>
```

We're using the Html helper submitButton method, and we give it a label, 'Rate It!', and a style. Note that this is different from what we have seen in most of our forms, something like:

```
<?= Html::submitButton($model->isNewRecord ? 'Create' :
    'Update', ['class' => $model->isNewRecord ?
    'btn btn-success' : 'btn btn-primary']) ?>
```

In the above example, we use the \$model->isNewRecord method to determine if it's a new record or not, and then show the appropriate button.

I couldn't get \$model->isNewRecord to work correctly in this case, and once I started to work around it, I realized this would be a good example for how to do it when that method is not available. This has implications in the controller, as we will see shortly.

FaqRatings Controller

We need to create the controller, and in this case, Gii is not going to be helpful, so you can just copy this file.

Gist:

FaqRatings Controller

From book:

```
<?php

namespace frontend\controllers;

use Yii;
use backend\models\FaqRating;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use yii\filters\VerbFilter;
use backend\models\Faq;
use yii\helpers\Html;
use yii\helpers\Url;

/**
 * FaqRatingController implements the CRUD actions
 * for FaqRating model.
 */

class FaqRatingController extends Controller
{
    public function behaviors()
    {
        return [
            'verbs' => [
                'class' => VerbFilter::className(),
                'actions' => [
                    'delete' => ['post'],
                ],
            ],
        ];
    }

    /**
     * Creates a new FaqRating model.
     * If creation is successful, the browser will
     * be redirected to
     * the 'view' page.
     * @return mixed
    
```

```
*/\n\npublic function actionRating()\n{\n    if(Yii::$app->user->isGuest){\n\n        return $this->redirect(['site/login']);\n    }\n\n    $model = new FaqRating();\n    $model->user_id = (int) Yii::$app->user->identity->id;\n\n\n    if ($model->load(Yii::$app->request->post())) {\n\n        $existingRating = FaqRating::find()\n            ->where(['user_id' => $model->user_id])\n            ->andWhere(['faq_id' => $model->faq_id])\n            ->one();\n\n        if(isset($existingRating->id)){\n\n            $existingRating->faq_rating = $model->faq_rating;\n\n            $existingRating->update();\n\n\n            $slug = Faq::find('slug')\n                ->where(['id' => $existingRating->faq_id])\n                ->one();\n\n            Yii::$app->session->setFlash('success',\n                'Thank you for updating this Faq to '\n                . $existingRating->faq_rating.\n                ' stars. Your result is factored into the\n                average.');
```

```
    } else {

        if($model->save()) {

            $slug = Faq::find('slug')
                ->where(['id' => $model->faq_id])
                ->one();

            Yii::$app->session->setFlash('success',
                'Thank you for rating this Faq ' . $model->faq_rating.
                ' stars. Your result is factored into the average.');

            return $this->redirect(['faq/view',
                'id' => $model->faq_id, 'slug' => $slug->slug
            ]);

        }

    }

} else {

    throw new NotFoundHttpException('There was a problem');

}

}
```

We do start with some boilerplate on the behaviors method:

```

class FaqRatingController extends Controller
{
    public function behaviors()
    {
        return [
            'verbs' => [
                'class' => VerbFilter::className(),
                'actions' => [
                    'delete' => ['post'],
                ],
            ],
        ];
    }
}

```

By only allowing delete on post, we put up another defense against delete via url manipulation. If you don't specify this, someone might be able to manipulate a get variable in the url to delete records, which is obviously not good.

In this case, specifying post probably isn't necessary, since we don't have a delete action, but I'll leave it in just as a precaution.

The only other method we have in the controller is the actionRating method. Let's look at the first part:

```

public function actionRating()
{
    if(Yii::$app->user->isGuest){

        return $this->redirect(['site/login']);
    }
}

```

This simply requires login by the user in order to submit a rating. By doing so, we don't allow a user create multiple rating records for a single Faq, since the user must be logged in and we can check to see if they have already rated it.

Next instantiate a new FaqRating model, so we can assign the user_id property to the currently logged in user, since we're not handing that in with the form:

```

$model = new FaqRating();

$model->user_id = (int) Yii::$app->user->identity->id;

```

The (int) specifies that we want an int out of Yii::\$app->user->identity->id. I did that because when I var_dumped to test, I got string as an answer. The validation requires an int however.

Next we load the post data and check to see if there is a matching record that already exists:

```
if ($model->load(Yii::$app->request->post())) {

    $existingRating = FaqRating::find()
        ->where(['user_id' => $model->user_id])
        ->andWhere(['faq_id' => $model->faq_id])
        ->one();
}
```

If it does already exist, we assign the rating from the form to this instance of the model, which is \$existingRating, and then we run update:

```
if(isset($existingRating->id)){

    $existingRating->faq_rating = $model->faq_rating;

    $existingRating->update();
}
```

Note that we can't just run save because it will create a new record for us. This is a byproduct of not being able to use \$model->isNewRecord in the form.

Next, because our Faq view pages require a slug to resolve, we need to look up the appropriate slug:

```
$slug = Faq::find('slug')
    ->where(['id' => $existingRating->faq_id])
    ->one();
```

After that, we set a flash message for the view:

```
Yii::$app->session->setFlash('success', 'Thank you for updating this Faq to '
    . $existingRating->faq_rating.
    ' stars. Your result is factored into the
    average.');
```

We're making this flash message dynamic so it will pass back the updated rating, which we will include in the message.

Finally, we redirect to the correct view, passing the id of the faq and the slug:

```
return $this->redirect(['faq/view',
    'id' => $existingRating->faq_id,
    'slug' => $slug->slug
]);
```

If it was not an existing record, we have a simpler path. We just save, find the slug, set the flash, and redirect:

```
} else {

    if($model->save()) {

        $slug = Faq::find('slug')
            ->where(['id' => $model->faq_id])
            ->one();

        Yii::$app->session->setFlash('success',
            'Thank you for rating this Faq ' . $model->faq_rating.
            ' stars. Your result is factored into the average.');

        return $this->redirect(['faq/view',
            'id' => $model->faq_id, 'slug' => $slug->slug
        ]);
    }
}
```

You should be able to test that at this point and it all should work.

Let's return to the view for a minute to discuss the Growl widget:

```
<?php

if (Yii::$app->getSession()->hasFlash('success')){

    echo Growl::widget([
        'type' => Growl::TYPE_SUCCESS,
        'title' => 'Thank you!',
        'icon' => 'glyphicon glyphicon-ok-sign',
        'body' => Yii::$app->session->getFlash('success'),
        'showSeparator' => true,
```

```
'delay' => 0,  
'pluginOptions' => [  
    'placement' => [  
        'from' => 'top',  
        'align' => 'right',  
    ]  
],  
]);  
}  
  
Yii::$app->getSession()->removeFlash('success');  
?>
```

We wrap it in an if statement, so it only fires if:

```
if (Yii::$app->getSession()->hasFlash('success')){
```

In the body setting, we pull in our specific flash message, which has been set in the controller:

```
'body' => Yii::$app->session->setFlash('success'),
```

Obviously you can change this to handle error messages as well. I didn't feel it was necessary because if there is an error, we throw an exception in the controller. You can check Kartik's widget documentation for more details on settings.

You'll also note that we add a separate line after the widget fires:

```
Yii::$app->getSession()->removeFlash('success');
```

This prevents the default behavior, which would be a Bootstrap alert element containing the flash message, which would fire in addition to the growl.

If you wanted to code this where you didn't want someone to be able to update their rating, you would code that logic in here:

```
if(isset($existingRating->id)){  
  
    // redirect with flash message  
  
    // sorry, you are not allowed to update your rating
```

So we have a nice frontend implementation of our FaqRatings model, but we probably want to know the ratings for individual Faqs in the backend too.

Since we've already done all the heavy lifting, this will be easy to implement. We have just 3 changes.

Faq Model

We need to add to the model a method to return the ratings of each Faq. Let's add the following:

```
public function getFaqRatings($id)  
{  
  
    $rating = new FaqRating;  
  
    return $rating->getAverageRating($id) ?  
        $rating->getAverageRating($id) : 'Not Rated' ;  
  
}
```

We already have a method for getting the average rating on the FaqRating model, so we just use that to return the average rating of our Faq. If it's null or 0, we'll set it to 'Not Rated.'

Troubleshooting tip: Please make sure you added that to the Faq model. We have a number of models with Faq in the name, so watch out for that.

Faq Index View

Next we modify the index view by adding the following into the columns array in the Gridview widget:

```
[ 'attribute'=>'Rating', 'value' => function($model){  
    return $model->getFaqRatings($model->id);  
},
```

Faq View

Finally, we add one line to the DetailView widget in view.php:

```
[ 'attribute'=>'Rating', 'format'=>'raw',  
    'value' => $model->getFaqRatings($model->id)  
,
```

And that's it. I don't have a solution yet for making the rating column sortable, but I'm working on it.

It's complicated because we are using the setSort method of ActiveDataProvider, and that is set up to order DB columns and our ratings column is obviously not that. I will probably have to use a fairly complicated query with either raw SQL or by creating a custom query class to get what I need. It's a subject worthy of its own chapter, but I didn't want to hold up publication for that, so I will update the book when I have that solution in place.

Ok, let's move on.

When users join a website, they typically have to agree to the terms of service. This is often represented as a checkbox, and if you don't agree, you don't get to signup.

Typically, we also see the terms of service in a scrollable box near the check box, like so:

Signup



Otherwise please fill out the following fields to signup:

Username

Email

Password

Terms Of Service

We will put all the terms of service here.

1. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis at

Agree To Terms

Signup

This is easy to set up, just 5 parts to it:

- Modify SignupForm model with the new checkbox attribute and validation
- Modify signup.php view file to contain the checkbox and render terms
- Create terms.php view file to hold the terms content
- Create overflow css style to define scroll bars
- Modify AppAsset.php to hold the new css style.

Signup Form Model

Let's look at the modified SignupForm model.

Gist:

[SignupForm.php](#)

From book:

```
<?php  
namespace frontend\models;  
  
use common\models\User;  
use yii\base\Model;  
use Yii;  
  
/**  
 * Signup form  
 */  
  
class SignupForm extends Model  
{  
    public $username;  
    public $email;  
    public $password;  
    public $agreeToTerms;  
  
    /**  
     * @inheritdoc  
     */  
    public function rules()  
    {  
        return [  
            ['username', 'filter', 'filter' => 'trim'],  
            ['username', 'required'],  
            ['username', 'unique', 'targetClass' => '\common\models\User',  
                'message' => 'This username has already been taken.'],  
            ['username', 'string', 'min' => 2, 'max' => 255],  
  
            ['email', 'filter', 'filter' => 'trim'],  
            ['email', 'required'],  
            ['email', 'email'],  
            ['email', 'unique', 'targetClass' => '\common\models\User',  
                'message' => 'This email address has already been taken.'],  
  
            ['password', 'required'],  
            ['password', 'string', 'min' => 6],  
        ];  
    }  
}
```

```

        ['agreeToTerms', 'boolean'],
        ['agreeToTerms', 'compare', 'compareValue'=>true,
         'message' =>'You must agree to our Terms of Service.'],
    ];
}

/**
 * Signs user up.
 *
 * @return User|null the saved model or null if saving fails
 */
public function signup()
{
    if ($this->validate()) {
        $user = new User();
        $user->username = $this->username;
        $user->email = $this->email;
        $user->setPassword($this->password);
        $user->generateAuthKey();
        if ($user->save()) {
            return $user;
        }
    }

    return null;
}
}

```

We added the class property:

```
public $agreeToTerms;
```

And then we set up validation for it:

```

['agreeToTerms', 'boolean'],
['agreeToTerms', 'compare', 'compareValue'=>true,
 'message' =>'You must agree to our Terms of Service.'],

```

So we're defining it as a boolean. Then we use the compare validator to make sure the value is true, which means the checkbox has been checked. We also set the error message if the compare value fails.

signup.php

Gist:

[signup.php](#)

From book:

```
<?php
use yii\helpers\Html;
use yii\bootstrap\ActiveForm;

/* @var $this yii\web\View */
/* @var $form yii\bootstrap\ActiveForm */
/* @var $model \frontend\models\SignupForm */

$this->title = 'Signup';
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="site-signup">
    <h1><?= Html::encode($this->title) ?></h1>
    <br>

    <?= yii\authclient\widgets\AuthChoice::widget([
        'baseAuthUrl' => ['site/auth'],
        'popupMode' => false,
    ]) ?>

    <p>Otherwise please fill out the following fields to signup:</p>

    <div class="row">
        <div class="col-lg-5">
            <?php $form = ActiveForm::begin(['id' => 'form-signup']); ?>
            <?= $form->field($model, 'username') ?>
            <?= $form->field($model, 'email') ?>
            <?= $form->field($model, 'password')->passwordInput() ?>

            <div class="col-lg-5-offset-4" id="terms">

                <?php
                echo \Yii::$app->view->renderFile('@app/views/pages/terms.php');
                ?>
```

```
<br>
</div>

<br>
<div class="row">
<div class="col-sm-4">
<?= $form->field($model, 'agreeToTerms')->checkbox() ?>
<div class="form-group">
<?= Html::submitButton('Signup', ['class' =>
    'btn btn-primary', 'name' => 'signup-button']) ?>
</div>
</div>
</div>
<?php ActiveForm::end(); ?>
</div>
</div>
</div>
```

After the password field, we make a div for the terms:

```
<div class="col-lg-5-offset-4" id="terms">

<?php
echo \Yii::$app->view->renderFile('@app/views/pages/terms.php');
?>

<br>
</div>

<br>
```

You can see how we are rendering the terms.php view, which we have not created yet. @app is a path alias that we need so the renderFile method knows where to look for the file.

Next we have:

```
<div class="row">
<div class="col-sm-4">

<?= $form->field($model, 'agreeToTerms')->checkbox() ?>
    <div class="form-group">
<?= Html::submitButton('Signup', ['class' => 'btn btn-primary',
    'name' => 'signup-button']) ?>

</div>
</div>
</div>
```

We are getting ‘Agree To Terms’ as a default label generated by the form field. If you want to change it, use the label method. For example:

```
<?= $form->field($model, 'agreeToTerms')
    ->label('I Agree To Terms')->checkbox() ?>
```

Also note:

```
<div class="row">
<div class="col-sm-4">
```

We need that to format the css. The scroll bars will not be added, however, until we define a style for them. We will do that after the next step.

terms.php

Let’s create terms.php in the frontend/views/pages folder. The reason we are putting it here is that so later on we can add it in other places on the application, perhaps a footer, to make it more accessible.

Gist:

[terms.php](#)

From book:

```
<h1>Terms Of Service</h1>
```

```
We will put all the terms of service here.
```

```
<br>
```

```
<br>
```

```
<p><strong>1.</strong> Lorem ipsum dolor sit amet, consectetur  
adipiscing elit. Duis at condimentum tortor. Nulla vestibulum ultricies  
urna. Lorem ipsum dolor sit amet, consectetur adipiscing elit.  
Fusce consequat lorem nec enim vehicula vulputate. Ut lacus  
quam, semper sit amet urna ac, pharetra porta enim. Cras et  
ex convallis, interdum nunc volutpat, consequat massa. Phasellus  
aliquet euismod nulla vel euismod. Integer dolor diam, fringilla  
quis elementum vehicula, egestas sit amet tellus. Aliquam convallis  
viverra auctor.</p>
```

```
<p><strong>2.</strong> Nunc eu neque vitae augue pharetra posuere.  
Donec et ligula in mauris mollis sodales. Maecenas neque diam, dictum  
consectetur nunc vel, euismod blandit mauris. Praesent ornare, mi nec  
lobortis elementum, felis dolor elementum metus, nec placerat quam velit  
ut nisi. Donecfringilla iaculis ligula, sit amet interdum augue ultricies ultric  
es.
```

```
Aenean bibendum ante vitae malesuada efficitur. Duis tincidunt ante quam,  
non lobortis metus varius at.</p>
```

```
<p><strong>3.</strong> Integer quis rhoncus purus, quis imperdiet enim.  
Vestibulum venenatis, erat at consequat dapibus, sem lorem venenatis dui,  
eget tempus diam erat non urna. Nullam at magna nec nulla sagittis luctus  
eget sit amet odio. Vivamus vel condimentum quam, eu fringilla nibh. Aliquam  
venenatis, elit id vulputate consectetur, elit erat aliquam lectus, id finibus  
augue ipsum eget enim. Nullam efficitur tincidunt condimentum. In luctus, nibh  
sit amet imperdiet pretium, ligula magna malesuada velit, vitae pharetra quam  
ex sit amet ligula. Cras malesuada, justo vel dictum dignissim, magna ligula  
porttitor nisi, in dapibus mi erat eget elit. Proin tristique interdum enim, id \  
luctus  
dolor mollis in.</p>
```

```
<p><strong>4. </strong> Praesent tempus mauris convallis, ultricies lorem  
elementum, pharetra nulla. Pellentesque in felis sed magna sollicitudin fermentum  
sit amet in lectus. Etiam pharetra dolor id lectus faucibus tristique. Vivamus
```

```
malesuada eu risus vitae tristique. Mauris at dolor blandit, ullamcorper ante eu\\
', mollis elit. Fusce ultrices ligula ut tellus semper, ut ullamcorper sem tristiqu\\
e.
```

```
Mauris blandit, urna at pulvinar efficitur, elit eros hendrerit urna, nec fringi\\
lla
```

```
leo metus sit amet odio. Donec eu ultrices nisl. Quisque eu enim porta,
fringilla metus non, bibendum tortor. Cras eu egestas enim. Aenean interdum,
eros sed pretium rutrum, nisi augue hendrerit enim, dictum feugiat est augue
vitae est. Donec vitae sagittis eros. Duis congue, eros vel pellentesque
molestie, justo turpis facilisis felis, mollis semper velit sem a nunc.
Integer eleifend, tellus sit amet auctor bibendum, quam leo maximus est,
eget feugiat velit nulla eget mauris.</p>
```

```
<p><strong>5.</strong> Fusce vel dolor eget ex varius porta quis ac libero.
Proin eget metus egestas, pretium mi non, dictum nisi. Cras fringilla varius
massa vitae convallis. Nunc ut blandit odio. Sed vehicula felis in neque
auctor hendrerit. Nunc a magna eget felis interdum auctor. Aliquam
accumsan metus justo, eget iaculis diam pharetra nec. Donec mauris lacus,
lacinia non venenatis in, interdum at purus. Donec ac mi id tortor mattis
convallis id sed tellus. Nullam volutpat justo pretium odio eleifend, ac
vestibulum lectus tempus. Quisque ut sem viverra lorem facilisis facilisis.
Aliquam erat volutpat.
</p>
```

Obviously not much to that. I will mention that I used the lorem ipsum generator at:

[Lorem Ipsum Generator](#)

termsoverflow.css

To get the scroll bars to work correctly, we need to create termsoverflow.css and add it to the frontend/web/css folder.

This class consists of the following:

```
#terms {  
    height: 150px;  
    overflow: scroll;  
}
```

Frontend AppAsset.php

Now that we have our style class to show the scroll bars, we need to tell our app to use it. We need to modify the our frontend AppAsset.php

Gist:

[Frontend AppAsset.php](#)

From book:

```
<?php  
/**  
 * @link http://www.yiiframework.com/  
 * @copyright Copyright (c) 2008 Yii Software LLC  
 * @license http://www.yiiframework.com/license/  
 */  
  
namespace frontend\assets;  
  
use yii\web\AssetBundle;  
  
/**  
 * @author Qiang Xue <qiang.xue@gmail.com>  
 * @since 2.0  
 */  
class AppAsset extends AssetBundle  
{  
    public $basePath = '@webroot';  
    public $baseUrl = '@web';  
    public $css = [  
        'css/site.css',  
        'css/termsoverflow.css'  
    ];  
    public $js = [  
    ];
```

```
public $depends = [
    'yii\web\YiiAsset',
    'yii\bootstrap\BootstrapAsset',
];
}
```

That was just a simple addition, adding our new style to the public \$css array.

That should be everything you need for this implementation.

Improving The Carousel

In the last chapter, we built a carousel for marketing images that we could control from the our backend admin UI. It works well, except that the photos don't scale properly when they are scaled down to a mobile-sized browser.

This is happening for two reasons.

1. We are specifying height and width of the image, which defeats the mobile response.
2. The shape of the photo we have been working with is rectangular, while the mobile shape is square.

If we remove the height and width attributes, the image scales, but we still have the problem of the shapes not matching, and therefore scaling is imprecise.

You could solve it by making sure you have a square image in the carousel to start with, but this is a serious limitation on the space. Not good.

I imagined showing the current carousel to a client and them complaining about it not scaling the way they want it to. That makes all the hard work we've done on it previously worthless, and we can't settle for that.

So now we will be working with a mobile image that the carousel uses when we detect the device is using a smaller browser window. This will hand complete control of the image over to the client or admin.

None of the steps are too difficult to make this happen, but we impact our existing code and we have to do some revising:

- Modify marketing_image Table with a new field for mobile image path.
- Modify MarketingImage model to account for the new property, rules and scenario.
- Modify various views including the form to allow the upload of the additional image.
- Modify MarketingImageController create, update, and delete actions for new image type.
- Modify defaults in CaouselWidget.php to allow for no width or height to be set
- Modify carousel.php to show the appropriate image based on browser size via jquery.
- Modify CarouselSettings model to make height and width of the image not required.

Modify marketing_imageTable

Ok, let's jump in. Here is the SQL to alter the marketing_image table:

```
ALTER TABLE `yii2build`.`marketing_image`
ADD COLUMN `marketing_mobile_path` VARCHAR(45)
CHARACTER SET 'utf8' COLLATE 'utf8_unicode_ci'
NOT NULL AFTER `marketing_thumb_path`;
```

Modify MarketingImage Model

Now let's make the modifications on the model. We'll start by adding a public property:

```
public $mobileFile;
```

We need that to be able to upload the mobile image.

Next we'll add rules for 'marketing_mobile_path.' I'll give you the entire method for reference.

Gist:

[MarketingImage Rules](#)

From book:

```
public function rules()
{
    return [
        [['marketing_image_path', 'marketing_image_name',
            'marketing_thumb_path', 'marketing_mobile_path',
            'marketing_image_weight'], 'required'],
        ['marketing_image_weight', 'default', 'value' => 100 ],
        ['marketing_image_is_featured', 'default', 'value' => 0 ],
        ['marketing_image_is_active', 'default', 'value' => 0 ],
        ['file', 'required', 'message' =>
            '{attribute} can\'t be blank', 'on'=>'create'],
        [['marketing_image_name', 'marketing_thumb_path',
            'marketing_mobile_path','marketing_image_path'], 'trim'],
        [['marketing_image_is_featured', 'marketing_image_is_active',
            'marketing_image_weight', 'status_id'], 'integer'],
        [['marketing_image_is_featured'],'in',
            'range'=>array_keys($this->getMarketingImageIsFeaturedList())],
        [['marketing_image_is_active'],'in',
```

```

        'range'=>array_keys($this->getMarketingImageIsActiveList())],
        [['file'], 'file', 'extensions' => ['png', 'jpg',
            'gif', 'jpeg'], 'maxSize' => 1024*1024],
        [['mobileFile'], 'file', 'extensions' => ['png', 'jpg',
            'gif', 'jpeg'], 'maxSize' => 250*250],
        [['marketing_image_path', 'marketing_image_name'],
            'string', 'max' => 45],
        [['marketing_image_caption', 'marketing_image_caption_title'],
            'string', 'max' => 100],

    ];
}

```

You should note that I set the size limit on the mobileFile to 250 x250. You can change this if you prefer a different size in the mobile image.

Next, add into scenarios ‘marketing_mobile_path’.

Gist:

[scenarios](#)

From book:

```

public function scenarios()
{
    $scenarios = parent::scenarios();
    $scenarios['create'] = ['file', 'marketing_image_path',
        'marketing_image_name', 'marketing_thumb_path',
        'marketing_mobile_path', 'marketing_image_is_featured',
        'marketing_image_is_active', 'marketing_image_caption',
        'marketing_image_caption_title', 'marketing_image_weight'];

    return $scenarios;
}

```

We also need to add ‘marketing_mobile_path’, to attribute labels.

```
'mobileFile' => 'Mobile Image',
```

For reference, I’m going to give you a Gist for the complete file:

Gist:

[MarketingImage.php](#)

MarketingImage Views

Ok, moving on to the views. Let's start with _form.php. We just need to add a single line:

```
<?= $form->field($model, 'mobileFile')->label('Mobile Image')->fileInput(); ?>
```

Marketing Image view.php

Our view.php file had a few changes:

Gist:

[view.php](#)

From book:

```
<?php

use yii\helpers\Html;
use yii\widgets\DetailView;

/* @var $this yii\web\View */
/* @var $model backend\models\MarketingImage */

$this->title = $model->id;
$this->params['breadcrumbs'][] = ['label' =>
    'Marketing Images', 'url' => ['index']]
];
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="marketing-image-view">

<p>
    <?= Html::a('Update', ['update', 'id' => $model->id], [
        'class' => 'btn btn-primary']) ?>
    <?= Html::a('Delete', ['delete', 'id' => $model->id], [
        'class' => 'btn btn-danger',
        'data' => [
            'confirm' => 'Are you sure you want to delete this item?',
            'method' => 'post',
        ],
    ]) ?>
</p>
```

```
<h1><?= Html::encode($model->marketing_image_name) ?></h1>
<br>
<div>

<h3>Primary Image:</h3>
<?php

echo Html::img('/'. $model->marketing_image_path .
    '?'. 'time=' . time() , ['width' => '600px',
    'alt' => $model->marketing_image_name]);


?>

</div>
<br>
<div>

<h3>Mobile Image:</h3>
<?php

echo Html::img('/'. $model->marketing_mobile_path .
    '?'. 'time=' . time(), [
    'alt' => $model->marketing_image_name]);


?>
</div>
<br>
<div>

<h3>Thumbnail Image:</h3>
<?php

echo Html::img('/'. $model->marketing_thumb_path .
    '?'. 'time=' . time(), [
    'alt' => $model->marketing_image_name]);


?>

</div>
<br>
```

```

<?= DetailView::widget([
    'model' => $model,
    'attributes' => [
        'id',
        'marketing_image_caption_title',
        'marketing_image_caption',
        'marketing_image_path',
        'marketing_thumb_path',
        'marketing_mobile_path',
        'marketing_image_weight',
        ['attribute' => 'marketing_image_is_featured',
            'format' => 'boolean'],
        ['attribute' => 'marketing_image_is_active',
            'format' => 'boolean'],
        'status.status_name',
        'created_at',
        'updated_at',
    ],
]) ?>

</div>

```

In addition to adding ‘marketing_mobile_path’ to the attributes in the DetailView widget, we also added h3 titles to the images, like so:

```
<h3>Mobile Image:</h3>
```

And of course we added the image itself, using our get variable to prevent caching:

```

<?php

echo Html::img('/'. $model->marketing_mobile_path .
'?'. 'time=' . time(), [
'alt' => $model->marketing_image_name]);


?>

```

Update View

We also add the mobile image and titles to update.php.

Gist:

Update View

From book:

```
<?php

use yii\helpers\Html;

/* @var $this yii\web\View */
/* @var $model backend\models\MarketingImage */

$this->title = 'Update Marketing Image: ' . ' ' . $model->id;
$this->params['breadcrumbs'][] = ['label' =>
    'Marketing Images', 'url' => ['index']];
$this->params['breadcrumbs'][] = ['label' => $model->id,
    'url' => ['view', 'id' => $model->id]];
$this->params['breadcrumbs'][] = 'Update';
?>
<div class="marketing-image-update">

    <h1><?= Html::encode($this->title) ?></h1>

    <br>
    <div>
        <h3>Primary Image:</h3>
        <?php
        echo Html::img('/'. $model->marketing_image_path,
            ['width' => '600px']);
    ?>
    </div>
    <br>
    <div>
        <h3>Mobile Image:</h3>
        <?php
        echo Html::img('/'. $model->marketing_mobile_path .
            '?'. 'time=' . time());
    ?>
    </div>
```

```

<br>
<div>
<h3>Thumbnail Image:</h3>
<?php

echo Html::img('/'. $model->marketing_thumb_path .
'?'. 'time=' . time());

?>

</div>
<br>

<?= $this->render('_form', [
    'model' => $model,
]) ?>

</div>

```

MarketingImage Controller

On MarketingImageController, we modified create, update and delete to account for our new \$mobileFile. Let's look at these one at a time.

Create Action

Gist:

[create action](#)

From book:

```

public function actionCreate()
{
    $model = new MarketingImage();
    $model->scenario = 'create';

    if ($model->load(Yii::$app->request->post())) {

        $imageName = $model->marketing_image_name;

        $model->file = UploadedFile::getInstance($model, 'file');
    }
}

```

```

$model->mobileFile = UploadedFile::getInstance($model, 'mobileFile');

$fileName = 'uploads/' . $imageName . '.' . $model->file->extension;
$fileName = preg_replace('/\s+/', '', $fileName);

$thumbName = 'uploads/' . 'thumbnail/' . $imageName .
'-thumb.' . $model->file->extension;
$thumbName = preg_replace('/\s+/', '', $thumbName);

$mobileName = 'uploads/' . 'mobile/' . $imageName . '-mobile.' .
$model->mobileFile->extension;
$mobileName = preg_replace('/\s+/', '', $mobileName);

$model->marketing_image_path = $fileName;
$model->marketing_thumb_path = $thumbName;
$model->marketing_mobile_path = $mobileName;
$model->save();

$model->file->saveAs($fileName);
$model->mobileFile->saveAs($mobileName);

Image::thumbnail( $fileName , 60, 60)
->save($thumbName, ['quality' => 50]);

return $this->redirect(['view', 'id' => $model->id, 'model' => $model,]);
}

} else {
    return $this->render('create', [
        'model' => $model,
    ]);
}
}

```

You can see that we have used getInstance set the uploaded file to \$model->mobileFile.

We also have a block to set the name and remove spaces:

```

$mobileName = 'uploads/' . 'mobile/' . $imageName . '-mobile.' .
$model->mobileFile->extension;
$mobileName = preg_replace('/\s+/', '', $mobileName);

```

We make sure we have the name set correctly:

```
$model->marketing_mobile_path = $mobileName;
```

After saving the model, we use saveAs for the file:

```
$model->mobileFile->saveAs($mobileName);
```

Update Action

Gist:

[update Action](#)

From book:

```
public function actionUpdate($id)
{
    $model = $this->findModel($id);

    if ($model->load(Yii::$app->request->post())){
        $imageName = $model->marketing_image_name;

        $oldImage = MarketingImage::find('marketing_image_name')
            ->where(['id' => $id])
            ->one();

        if ($oldImage->marketing_image_name != $imageName){
            throw new ForbiddenHttpException
                ('You cannot change the name, you must delete instead.');
        }

        if ( $model->file = UploadedFile::getInstance($model, 'file')){
            $thumbName =  'uploads/' . 'thumbnail/' . $imageName .
                '-thumb.' . $model->file->extension;
        }
    }

    if ($model->mobileFile = UploadedFile::getInstance($model, 'mobileFile')){
```

```
$mobileName = 'uploads/mobile/' . $imageName .
'-mobile.' . $model->mobileFile->extension;

}

$model->save();

if ($model->file) {

$fileName = 'uploads/' . $imageName . '.' . $model->file->extension;

$model->file->saveAs($fileName);

Image::thumbnail( $fileName , 60, 60)
->save($thumbName, ['quality' => 50]);

}

if ($model->mobileFile) {

$model->mobileFile->saveAs($mobileName);

}

return $this->redirect(['view', 'id' => $model->id]);

} else {

return $this->render('update', [
'model' => $model,
]);
}

}
```

Just two spots to note. The first is before we save the model:

```
if ($model->mobileFile = UploadedFile::getInstance($model, 'mobileFile')){  
  
    $mobileName = 'uploads/mobile/' . $imageName .  
    '-mobile.' . $model->mobileFile->extension;  
  
}
```

This is just like what we do for the thumbnail image, so we have already covered this in the previous chapter.

Next we add:

```
if ($model->mobileFile) {  
  
    $model->mobileFile->saveAs($mobileName);  
  
}
```

Note that both of these statements are wrapped in if statements because updating the mobile image is not required, whereas when we create the record, the mobile image is required.

Delete Action

Gist:

[delete Action](#)

From book:

```
public function actionDelete($id)  
{  
  
    $model = $this->findModel($id);  
  
    try {  
  
        unlink($model->marketing_image_path);  
  
        unlink($model->marketing_thumb_path);  
  
        unlink($model->marketing_mobile_path);  
  
        $model->delete();  
    }  
}
```

```
    return $this->redirect(['index']);  
}  
  
catch(\Exception $e) {  
    throw new NotFoundHttpException($e->getMessage());  
}  
}
```

Ok, that's simple enough. We just added the extra unlink for the mobile image.

Entire File

I'm supplying a gist of the entire file for reference, should you need it:

Gist:

[MarketingImage Controller](#)

CarouselSettings Model Rules

So the first thing we can do is remove image_height and image_width from the first rule because they are no longer required. While we're here, I did decide that the carousel_name had to be unique. This will prevent an admin from accidentally creating two records of settings for one carousel. So here we have the entire rules method:

Gist:

[rules](#)

From book:

```

public function rules()
{
    return [
        [['carousel_name', 'caption_font_size', 'status_id'], 'required'],
        [['carousel_name'], 'unique'],
        [['carousel_autoplay', 'show_indicators', 'show_captions',
            'status_id', 'show_controls'], 'integer'],
        [['carousel_autoplay'], 'in',
            'range'=>array_keys($this->getCarouselAutoplayList())],
        [['show_indicators'], 'in',
            'range'=>array_keys($this->getShowIndicatorsList())],
        [['show_captions'], 'in',
            'range'=>array_keys($this->getShowCaptionsList())],
        [['show_caption_background'], 'in',
            'range'=>array_keys($this->getShowCaptionBackgroundList())],
        [['show_caption_title'], 'in',
            'range'=>array_keys($this->getShowCaptionTitleList())],
        [['show_controls'], 'in',
            'range'=>array_keys($this->getShowControlsList())],
        [['status_id'], 'in', 'range'=>array_keys($this->getStatusList())],
        [['created_at', 'updated_at'], 'safe'],
        [['carousel_name', 'image_height', 'image_width'],
            'string', 'max' => 45]
    ];
}

```

CarouselWidget

Moving on to the CarouselWidget class, we are going to change the setDefaults method to set height and width to null if no other value is provided. This default will be the typical implementation.

I don't need to give you a Gist, just change the value to null as below:

```

if (!isset($this->settings['height'])){

    $this->settings['height'] = null;
}

if (!isset($this->settings['width'])){

    $this->settings['width'] = null;
}

```

validateSize Method

Next we wrap everything in the validateSize method in an if statement, so that we are only validating size if height and width are not empty:

```
if (!empty($this->settings['width']) && !empty($this->settings['height'])){

}
```

Entire CarouselWidget File

For reference, I am providing the entire CarouselWidget file below:

Gist:

[CarouselWidget.php](#)

carousel.php

The last step in this is to modify `carousel.php`. Ultimately, we are going to do something similar to what we did with the faq rating, where we show/hide based, in this case, on the size of the browser.

Unfortunately, that means we have to add a second carousel to the view, which takes a giant plate of spaghetti and doubles it. There's not much I can do about that. I tried simply replacing part of the carousel, but that didn't work.

What I did do, that actually worked, is wrap each carousel in a div, one named `big` and the other named `small`. So hopefully that will keep everything clear.

The other big difference is the path to the image in the small carousel. All that is followed by the javascript at the bottom of the file. Here is the entire file:

Gist:

[carousel.php](#)

From book:

```
<?php

use yii\helpers\Html;

?>

<div id="big">
<div id="carouselMain" class="carousel slide"

<?php

if($settings['autoplay'] == false ){

    echo 'data-interval="false"';
}

?>

data-ride="carousel">

<!-- Indicators --> <?php

if ($settings['show_indicators']){
    echo '<ol class="carousel-indicators">
<li data-target="#carouselMain" data-slide-to="0"
    class="active"></li>';

    foreach (range(1, $count) as $number) {

        echo '<li data-target="#carouselMain"
            data-slide-to="'.$number.'"></li>';

    }

    echo '</ol> ';
}

?>

<!-- Wrapper for slides -->
```

```
<div class="carousel-inner" role="listbox">

<!-- dynamic slide data -->

<?php

$width = $settings['width'];
$height = $settings['height'];
//active item first

echo '<div class="item active">
<center>'.
Html::img(Yii::$app->urlManagerBackend->baseUrl . '/' .
$activeImage['marketing_image_path'], ['width' => $width,
'height' => $height ])
.</center>';

if($settings['show_captions']){

echo '<div class="carousel-caption">';

if ($settings['show_caption_title']){

echo '<div><h1>' . $activeImage
['marketing_image_caption_title'] . '</h1></div>';

}

echo $activeImage['marketing_image_caption'] . '</div>';
}

echo '</div>';

//all other images

foreach ($images as $image){

echo '<div class="item">
<center>'.
Html::img(Yii::$app->urlManagerBackend->baseUrl . '/' .
$image['marketing_image_path'], ['width' => $width,
'height' => $height ])
```

```
.  '</center>';

if($settings['show_captions']){
    echo '<div class="carousel-caption">';
    if ($settings['show_caption_title']){
        echo  '<div><h1>' .
        $image['marketing_image_caption_title'] .
        '</h1></div>';
    }
    echo $image['marketing_image_caption'] . ' </div>';
}
echo '</div>';

}

?>

<!-- end dynamic slide data -->

</div>

<!-- Controls -->
<?php
if ($settings['show_controls']){
    echo '<a class="left carousel-control" href="#carouselMain" role="button" data-slide="prev">
        <span class="glyphicon glyphicon-chevron-left" aria-hidden="true"></span>
        <span class="sr-only">Previous</span>
    </a>
    <a class="right carousel-control" href="#carouselMain" role="button" data-slide="next">
        <span class="glyphicon glyphicon-chevron-right" aria-hidden="true"></span>
        <span class="sr-only">Next</span>
    </a>
}
?>
```

```
<span class="glyphicon glyphicon-chevron-right"
      aria-hidden="true"></span>
<span class="sr-only">Next</span>
</a>';

}

?>
</div>

</div>

<div id="small">
<div id="carouselSmall" class="carousel slide"

<?php

if($settings['autoplay'] == false ){

    echo 'data-interval="false"';
}

?>

data-ride="carousel">

<!-- Indicators --> <?php

if ($settings['show_indicators']){
    echo '<ol class="carousel-indicators">
<li data-target="#carouselSmall" data-slide-to="0"
    class="active"></li>';

    foreach (range(1, $count) as $number) {

        echo '<li data-target="#carouselSmall"
            data-slide-to="'. $number .'"></li>';
    }
}
```

```
echo '</ol> ';
```

```
}
```

```
?>
```

```
<!-- Wrapper for slides -->
<div class="carousel-inner" role="listbox">
```

```
<!-- dynamic slide data -->
```

```
<?php
```

```
$width = $settings['width'];
$height = $settings['height'];
```

```
//active item first
```

```
// use the path for mobile image
```

```
echo '<div class="item active">
<center>'.
Html::img(Yii::$app->urlManagerBackend->baseUrl. '/' .
$activeImage['marketing_mobile_path']. '?' . 'time=' . time())
.</center>';
```

```
if($settings['show_captions']){
    echo '<div class="carousel-caption">';
    if ($settings['show_caption_title']){
        echo '<div><h1>' . $activeImage
        ['marketing_image_caption_title'] . '</h1></div>';
    }
    echo $activeImage['marketing_image_caption'] . '</div>';
}
```

```
echo '</div>';
```

```
//all other images

// use the path for mobile image

foreach ($images as $image){

    echo '<div class="item">
    <center>' .
    Html::img(Yii::$app->urlManagerBackend->baseUrl . '/' .
    $image['marketing_mobile_path'] . '?' . 'time=' . time())
    . '</center>';

    if ($settings['show_captions']){

        echo '<div class="carousel-caption">';

        if ($settings['show_caption_title']){
            echo    '<div><h1>' .
            $image['marketing_image_caption_title']. '</h1></div>';

        }

        echo $image['marketing_image_caption']. '</div>';

    }

    echo '</div>';

}

?>

<!-- end dynamic slide data -->

</div>

<!-- Controls -->
<?php
if ($settings['show_controls']){

```

```
echo '<a class="left carousel-control" href="#carouselSmall" role="button"
      data-slide="prev">
  <span class="glyphicon glyphicon-chevron-left" aria-hidden="true"></span>
  <span class="sr-only">Previous</span>
</a>
<a class="right carousel-control" href="#carouselSmall" role="button"
      data-slide="next">
  <span class="glyphicon glyphicon-chevron-right" aria-hidden="true"></span>
  <span class="sr-only">Next</span>
</a>';

}

?>
</div>

</div>

<?Php
$script = <<< JS
if ($(window).width() <= 800){
  $("#big").hide();
  $("#small").show();
  $('#carouselSmall').carousel({
    interval: 1000
  });
} else{
  if ($(window).width() >800){
    $("#small").hide();
    $("#big").show();
    $('#carouselMain').carousel();
    interval: 1000
  }
}
$(window).resize(function(){
  if ($(window).width() <= 800){
    $("#big").hide();
    $("#small").show();
  } else{
    $("#small").hide();
    $("#big").show();
  }
});
```

```

        $("#small").show();
        $('#carouselSmall').carousel({
            interval: 1000
        });
    }else{

        if ($(window).width() >800){
            $("#small").hide();
            $("#big").show();
            $('#carouselMain').carousel();
            interval: 1000
        }

    }
});

JS;

$this->registerJs($script);

?>

```

So let's hit the highlights.

- We have two carousels, each in a separate div, one named big, the other small.
- The carousels also have div ids as carouselMain and carouselSmall
- The path to the image in carouselSmall is obviously different than that of carouselMain.
- The javascript at the bottom of the file determines which carousel will be visible, depending on browser size.

Summary

In this chapter, we got to build a ratings system for our Faqs, using Kartik's widget extension. That worked out really well for us because now we know how to create a rating system for any type of model that users might be willing to rate.

We also implemented the Growl widget, which gave us a nice UI experience, whenever there is a save or update to the ratings. You can use the Growl widget often, whenever you are setting flash messages, and it would make the application act in a very consistent way.

We also had our first use of a checkbox, which we implemented to confirm the user's agreement to terms of service. It was simple stuff, but we also added to our frontend AppAssets.php file to pull in the css, so we are getting more familiar with Yii 2's asset publishing.

Finally, we went back into the carousel to give ourselves full control over that marketing space, so we can nuance the image in a mobile browser by using a separate image for that purpose. Even though we expanded the functionality of the carousel, we still have 100% management of the carousel through our backend UI, creating a user-friendly admin environment that will be appreciated by those responsible for the marketing in the carousel. And that's it for this chapter.

Thanks once again to everyone for the positive comments and reviews, they keep me motivated to keep going with more bonus material. Please help spread the word about the book if you can. It will be greatly appreciated.

In the meantime, I will continue to work on bonus material and I will keep working on the template. Thanks again for supporting the book. See you soon.

Chapter 18: Bonus Material Returning Calculated Values in Gridview

Sorting A Calculated Value In Gridview

Welcome back to another bonus chapter. When we built our Faq ratings system, we noticed that the rating column in the gridview widget wasn't sortable.

The reason why it wasn't sortable in our first implementation is that we need to return the average rating of the faq, and that is a calculated value, not a strait DB column value. It was easy enough for us to build a method to return the average, but we didn't know how to sort on that method.

This might seem like a trivial detail, and maybe we could just forget about it, but of course clients and users love column sorts and with good reason. They allow us to organize our view and to see the most important data with the click of a button. So not having the sort capability is not really an option.

There's always more than one way to do things, so as I thought about the solution, I thought we might have to switch to SQL data provider because I knew the query was more complex than what we are used to dealing with.

For example, if we go straight to our SQL tab in PhpMyadmin and put in the following SQL:

```
select *
from faq as a
inner join
(SELECT faq_id, AVG(faq_rating) as 'average_rating'
FROM faq_rating
GROUP BY faq_id) as b
ON a.id=b.faq_id;
```

You will get something like:

The screenshot shows a MySQL query results grid. The SQL query is:

```

SELECT *
FROM faq AS a
INNER JOIN (
    SELECT faq_id, AVG(faq_rating) AS 'average_rating'
    FROM faq_rating
    GROUP BY faq_id
)

```

The results grid has columns: id, faq_question, slug, faq_answer, faq_category_id, faq_is_featured, faq_weight, created_by, updated_by, created_at, updated_at, faq_id, and average_rating. The data includes rows for various FAQ entries, with the last row being the calculated average rating.

<code>id</code>	<code>faq_question</code>	<code>slug</code>	<code>faq_answer</code>	<code>faq_category_id</code>	<code>faq_is_featured</code>	<code>faq_weight</code>	<code>created_by</code>	<code>updated_by</code>	<code>created_at</code>	<code>updated_at</code>	<code>faq_id</code>	<code>average_rating</code>
11	Should I use a framework?	should-i-use-a-framework	Probably, it's a good idea	1	0	3	1	1	2015-02-27 18:53:40	2015-03-01 11:43:38	11	4.5
12	Am I going to finish?	am-i-going-to-finish	Ysel	1	1	85	1	1	2015-02-27 18:54:21	2015-02-27 18:54:21	12	1
13	Am I still having fun?	am-i-still-having-fun	Probably	2	1	95	1	1	2015-02-27 18:55:03	2015-02-27 18:55:03	13	3
15	What time is it?	what-time-is-it	Now	1	0	100	1	1	2015-02-28 18:33:12	2015-02-28 18:33:12	15	3
16	is it similar to timestamp?	is-it-similar-to-timestamp	let's find out	1	1	100	1	1	2015-02-28 18:35:39	2015-02-28 18:35:39	16	4

Extended Table

It's hard to see everything because it's compressed down, but we get the extra row we need for average rating. I based this query on a tutorial:

Query Tutorial

You can see this isn't the friendliest format to work with. For one thing, the above query is not returning results when there is no rating, so the query is not even correct.

The simple solution to that problem is to make it a LEFT join, which will return all the results. Ok, so we overcame that.

When I first started using Yii 2, I tended to rely more on plain SQL than ActiveRecord, for the reason illustrated above. If you need to figure out a query, you can simply search for an answer, and it can be faster to develop that way. But now that I have more experience with ActiveRecord, I really love it. It's so much more intuitive and easy to use. It's worth taking the extra time to figure out how to work with it.

There is something else to consider, however, and that is the overhead you will incur by using any framework's ORM. For large databases and complex queries, it's generally not considered practical or scalable. But with PHP 7 coming, the framework overhead might be reduced to a very workable level for complex queries. And that means we can use ActiveRecord without worrying that we are harming our ability to scale.

Since PHP 7 is due in October of this year, I'm going on the assumption that ActiveRecord will be a viable solution for complex queries.

Also, as we will see as we develop this, we do not have to sacrifice MySQL's built-in functions like AVG and COUNT when we use ActiveRecord, so we still get to benefit from MySQL's performance capabilities for those kinds of queries.

So now we have to figure out how to translate our SQL query to ActiveRecord, if we want to use it with ActiveDataProvider and continue to use the existing solution we have for our FaqSearch search method. This is what we want to do.

The problem is that we have raw SQL, but we don't know how to translate it to ActiveRecord, or even if it can be translated to ActiveRecord.

To figure this out, we should reference two sections of the Yii 2 docs:

[Query Builder](#)

And

[ActiveRecord](#)

There's quite a lot of information there on all the methods available to us. That said, I was still a little unsure. So I went to the forum for help:

[Help with SQL to ActiveRecord](#)

Very quickly I received a couple of responses, and one of those was from Kartik. It turns out, he has written a web tip on his site for this scenario:

[Kartik's Web Tip](#)

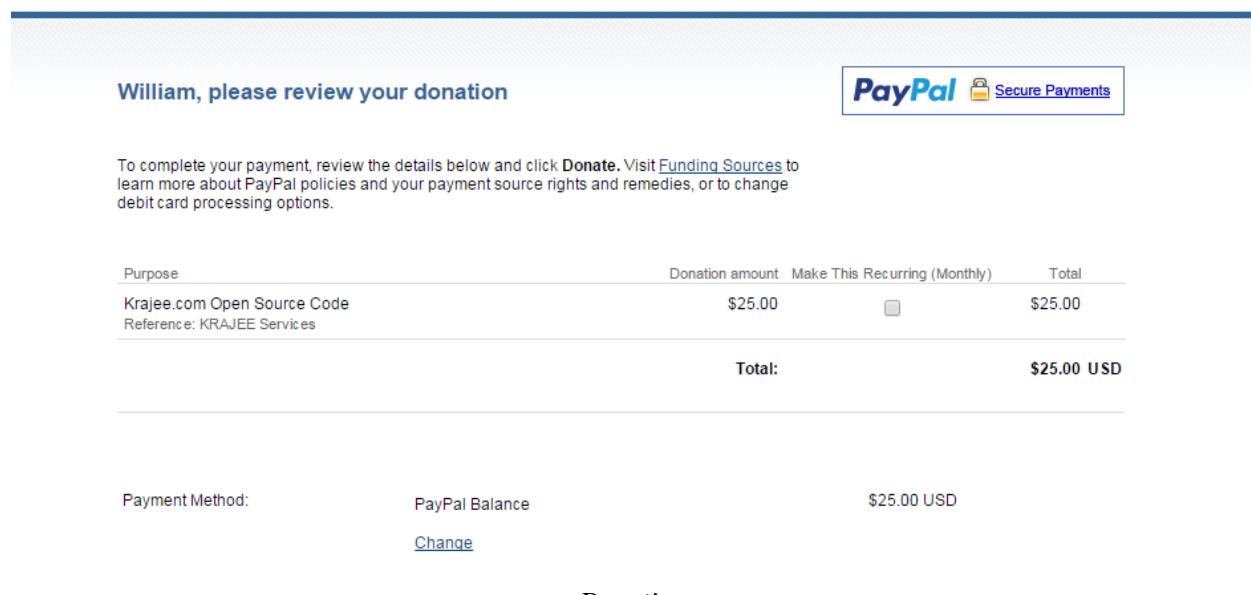
So, in addition to all the amazing widgets he has made for us, he also has anticipated that we would need some help with sorting a calculated field in Gridview. It's just incredibly helpful.

Donate To Kartik

Please donate to Kartik if you can, he is contributing a lot to the Yii 2 community. His donation button is at:

[Krajee.com](#)

Here is a copy of my donation:



The screenshot shows a PayPal donation page. At the top, it says "William, please review your donation". On the right, there is a "PayPal" logo with "Secure Payments" next to it. Below this, a message reads: "To complete your payment, review the details below and click **Donate**. Visit [Funding Sources](#) to learn more about PayPal policies and your payment source rights and remedies, or to change debit card processing options." A table then displays the donation details:

Purpose	Donation amount	Make This Recurring (Monthly)	Total
Krajee.com Open Source Code Reference: KRAJEE Services	\$25.00	<input type="checkbox"/>	\$25.00
	Total:		\$25.00 USD

At the bottom, it says "Payment Method: PayPal Balance" and "Change".

Donation

It's a great way to say thanks for all the hard work he puts into his extensions and tutorials, which he actively maintains.

I've described my research into this chapter fully, so you can understand how to approach this if you find yourself needing help with ActiveRecord or anything else.

You should always research thoroughly before going to the forum because there's no sense in asking a question that's already been answered. I made the mistake of focusing on the query, not thinking the full solution was out there, so I never saw Kartik's web tip. Next time I will try searching on the full solution before going to the forum.

Anyway, let's jump into this solution.

Average Rating For Gridview

The first step is to create a method on the base model, in this case, the Faq model:

Gist:

[getAverageRating](#)

From book:

```
public function getAverageRating()
{
    return $this
        ->hasMany(FaqRating::className(), ['faq_id'=>'id'])
        ->average('faq_rating');
}
```

So this is just a relationship, with the average method added on. Simple, but powerful.

I found the right syntax for average by referencing the query builder section of the docs. We'll see later that the average method inside of MySQL is AVG, so we have to be sure to use the right syntax in the right place.

During my research for this chapter, I also explored the possibility of creating a scope instead of the above method. However, scopes are not directly supported by Yii 2. Instead, you can create a custom query class that overrides your find method.

After looking at the two solutions, I felt the one offered by Kartik's tutorial was simpler. It is also listed in the Yii 2 guide as an alternative to creating a scope. So, all other things being equal, I chose the simplest implementation.

If I find a good use case for the custom query class on the template, I will include it in a future chapter.

Ok, back to the base model. We also add an attributeLabel:

```
'averageRating' => Yii::t('app', 'Rating'),
```

This isn't anything we haven't seen before. In case you need a reminder, we use the magic syntax, so there is no get and the average is lowercase since it's the first word.

Now we'll move on to the search model FaqSearch. Let's start with a use statement:

```
use backend\models\FaqRating;
```

and add a public property:

```
public $averageRating;
```

And we add the safe attribute in the rules:

```
public function rules()
{
    return [
        [['id', 'faq_category_id', 'faq_weight',
            'faq_is_featured', 'created_by', 'updated_by'],
            'integer'],
        [['faq_question', 'faq_answer', 'created_at',
            'updated_at', 'faqCategoryName', 'faqCategoryList',
            'faqIsFeaturedName', 'createdByUsername', 'updatedByUsername',
            'faq_category', 'faq_weight', 'averageRating'], 'safe'],
    ];
}
```

All we did was add the single attribute 'averageRating' to the safe rule.

Ok, moving on to our search method, we start by adding to our query definition. Before we had this:

```
$query = Faq::find();
```

Now we need this:

Gist:

```
query
```

From book:

```
$query = Faq::find();

$subQuery = FaqRating::find()
->select('faq_id, AVG(`faq_rating`) as average_rating')
->groupBy('faq_id');

$query->leftJoin([
    'faqAverage'=>$subQuery
], 'faqAverage.faq_id = faq.id');
```

You can see we created a subquery and did a left join. In the subquery, we are selecting AVG(faq_rating), which is the MySql function to calculate an average.

You can also see that we give the \$subQuery variable a name in the leftJoin array, in this case it's 'faqAverage.' And right away you can see that this alias works in the on part of the join. So instead of faq_rating.faq_id, we have faqAverage.faq_id to join on faq.id.

Doing it this way means that the AVG function will loop for each instance of FaqRating where faq_rating.faq_id is equal to faq.id. And this is what we're looking for.

The only thing I don't like about it is that we have already defined the getAverageRating method on the Faq model, so this seems a little redundant. I tested it to see if we needed both and it turns out that we do.

As far as I can tell, the sorting capability we want relies on this subQuery, and the actual line by line results rely on the model method. If we drop anything, it will not work.

So moving on to our setSort method, we add the following block.

Gist:

[setSort Block](#)

From book:

```
'averageRating'=>[
    'asc'=>[ 'faqAverage.average_rating'=>SORT_ASC],
    'desc'=>[ 'faqAverage.average_rating'=>SORT_DESC],
    'label'=>'Rating'
],
```

Remember that averageRating is our getAverageRating method on the base model and faqAverage is the label we gave our subQuery.

The last piece to pop in the search method is the line to filter by average rating:

```
// filter by average rating

$query->andWhere(['faqAverage.average_rating'=>$this->averageRating]);
```

The way we did this in previous examples was a little more verbose. It would look like this:

```
$query->joinWith(['faqRating' => function ($q) {
    $q->andFilterWhere(['faqAverage.average_rating'=>$this->averageRating]);
}]);
```

I checked the debug toolbar for DB queries and there was no difference in performance between the two, so I went with the shorter line.

Ok, so the last little bit is the addition of one line in Gridview in backend/views/faq/index.php:

```
'averageRating',
```

I put that on the line immediately following faq_weight. And that should do it. You now have a fully sortable column for a calculated value, in this the case, the average rating of the faq.

Times Rated

I was curious about how we might add additional calculated values, so I decided to return a calculated value for the number of times an faq is voted on.

I'm going to step through this quickly because most of it is exactly the same as what we just did. So we'll start with our Faq model method.

Gist:

[getRatingsCount](#)

From book:

```
public function getRatingsCount()
{
    return $this
        ->hasMany(FaqRating::className(), ['faq_id'=>'id'])
        ->count('faq_rating');
}
```

Next we make a label:

```
'ratingsCount' => Yii::t('app', 'Times Rated'),
```

Now we move on to FaqSearch model. We need to add a public property:

```
public $ratingsCount;
```

Then comes the safe rule, we just add the one attribute:

```
public function rules()
{
    return [
        [['id', 'faq_category_id', 'faq_weight', 'faq_is_featured',
            'created_by', 'updated_by'], 'integer'],
        [['faq_question', 'faq_answer', 'created_at', 'updated_at',
            'faqCategoryName', 'faqCategoryList', 'faqIsFeaturedName',
            'createdByUsername', 'updatedByUsername', 'faq_category',
            'faq_weight', 'averageRating', 'ratingsCount'], 'safe'],
    ];
}
```

Next we work on the query in the search method:

```
$query = Faq::find();

$subQuery = FaqRating::find()
    ->select('faq_id, AVG(`faq_rating`) as average_rating,
              count(faq_rating) as times_rated')
    ->groupBy('faq_id');

$query->leftJoin([
    'faqAverage'=>$subQuery
], 'faqAverage.faq_id = faq.id');
```

Ok, so you can see that we simply added to the subQuery instead of creating a new one.

In the subQuery, we are returning count(faq_rating) as times_rated, so we can use times_rated in our setSort.

That also means we use faqAverage, which identifies the subquery, for this block in the set sort.

```
'ratingsCount'=> [
    'asc'=> [ 'faqAverage.times_rated'=>SORT_ASC] ,
    'desc'=> [ 'faqAverage.times_rated'=>SORT_DESC] ,
    'label'=>'Times Rated'
],
```

And then we'll simply add the filter line in the appropriate place:

```
// filter by rating count
$query->andWhere(['faqAverage.times_rated'=>$this->ratingsCount]);
```

And finally, add the line to the index view in Gridview:

```
'ratingsCount',
```

When everything is saved, here is a screenshot of what you should see:

#	ID	Question	Answer	Category	Weight	Rating	Times Rated	Featured
1	11	Should I use a framework?	Probably, it's a good idea	general	3	4.5	2	no
2	16	is it similar to timesamp?	let's find out	general	100	4	1	yes
3	13	Am I still having fun?	Probably	Specific	95	3	1	yes

Gridview Screenshot

I'm going to give you the Faq model and the FaqSearch model gists for reference in case you need to troubleshoot:

[Faq Model](#)

[FaqSearch Model](#)

Summary

Sortable, calculated values are very important to the businesses that manage data through web applications. Important questions can be answered quickly such as what is the average rating, which one has the highest rating and which one was rated the most times.

The column sorts facilitate these answers, so admins can get to the key data quickly. This allows companies to follow trend data, which they use to make important decisions.

This was a short focused chapter on returning calculated values in Gridview, a nice way to end the book. I have had a lot of fun writing this stuff and I hope you found it helpful.

Thanks again to everyone who supports this book by sending in typo notices, leaving positive comments and positive reviews, I really appreciate it. See you soon.