

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 2014-11-14



Leanpub

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 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 and I can't wait to build something with it!

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	19
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	31
Debugger	31
Summary	32
Chapter Four: Modifying the User Model	34
Role and Status	35
The User Model	37
Properties of the Model	44
Constants	45
Identity Interface	46
Behaviors	49
Rules	50
Identity Methods	51
Boilerplate Methods	53
Other Models Accessing User	55
SignupForm Model	55
Summary	59
Chapter Five: Creating New Models with Gii	60
Creating Tables	60
Role Table	61
Status Table	62
User Type Table	62
Gender Table	62
Profile Table	63
Synchronize	64
Configuring Gii	65
Making Models with Gii	66
Create Role Model	67
Add Records To Role Table	70
Add Relationship To Role	72

CONTENTS

Update User Model with Role	73
Create Status Model	75
Update User Model with getStatus	77
Add Records to Status Table	78
Create UserType Model	79
Update User Model with UserType	81
Add Records to user_type Table	83
Create Gender Model	84
Add Records to gender Table	86
Create Profile Model	86
Update User Model with Profile	95
Finish Up User Model	96
The Complete User Model	97
Summary	109
Chapter Six: Helpers	110
Value Helpers	110
Permission Helpers	114
Record Helpers	120
Summary	122
Chapter Seven: Site Controller	123
Behaviors	124
Actions	125
Index Action	126
Login Action	126
Login Form Model	127
Logout Action	131
Contact Action	131
Contact Form Model	132
Captcha	133
Contact View Form	135
About Action	138
Signup Action	139
Signup Form Model	139
ResetPasswordForm Model	145
Backend Site Controller	148
Beginning Access Control	150
loginAdmin Method	151
Summary	153
Chapter Eight: Profile Crud	154
CRUD	154

CONTENTS

Profile Controller	156
Profile Search	156
_search	156
_form	156
Index	156
View	157
Create	157
Update	158
Modifying Profile Controller & Views	158
Modifying the Profile Controller	159
Index Action	161
View Action	163
Create Action	164
Update Action	166
Delete Action	169
FindModel Action	169
Modifying the Profile Views	170
View.php	170
Gender	176
Form Partial	177
DatePicker	180
Create	182
Update	182
Site Layout	183
Profile Link	186
Summary	187
Chapter Nine: Upgrade and Access Control	188
Upgrade Controller	190
Upgrade View	190
Require Upgrade To	190
Access Control	191
Passing A Variable From the Controller	196
Summary	198
Chapter Ten: Homepage Social Widgets	199
Implementing Homepage Social Widgets	199
Index	199
Facebook Widget	200
Facebook App Setup	201
Facebook Configuration	206
Extensions	208
HTML Helper	209

CONTENTS

Collapse Widget	216
Modal Widget	217
Alert Widget	217
Font-Awesome	218
Asset Bundle	219
Add Font-Awesome to Layout	222
Summary	224
Chapter Eleven: Backend Creation	225
Main.php	229
Updating Backend Views	234
backend\views\profile_form.php	234
backend\views\profile\view.php	234
backend\views\user\view.php	239
backend\views\user_form	242
Deeper Changes to Backend	243
backend\views\user\index.php	243
backend\views\profile\index.php	245
backend\views\profile_search.php	247
backend\views\user_search.php	248
User Search	250
Admin UI	269
Controller Behaviors	276
Match Callback	277
Summary	280
About The Author	281

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

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. And while I hope they get a lot out of it, I'm not really writing this book for them.

Instead, I'm writing it for the beginners. People who have some experience with PHP but have not really jumped up to advanced object-oriented programming. I can tell you from personal experience it's not easy to learn OOP simply by online tutorials.

Typically you get examples of class Bear extending class Animals, which isn't necessarily bad for teaching the basic concept, but on the other hand, isn't really going to give you a feel for working with nested objects. Learning Yii 2 however, does give you hands on experience with object-oriented programming with practical results. You end up with a working website.

Not only do we get to leverage object-oriented programming to build a data-driven web application, but we also get to use the latest version of PHP, which allows us to use cleaner notation for arrays, namespaces, and some other efficiencies which are new to Yii in Yii 2. And the really good news is that Yii 2 is now so strong as a framework, it is more accessible to beginners than it ever has been before. And with it's robust community support, it's the perfect framework to learn.

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, all in one go. 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 Schools¹](http://www.w3schools.com/PHP/)

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

And then of course there is [Php.net²](http://www.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 Beginners³](http://code.tutsplus.com/tutorials/object-oriented-php-for-beginners--net-12762)

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 [Google⁴](http://www.google.com) or [stackoverflow⁵](http://www.stackoverflow.com) or [PHP.net⁶](http://www.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.

¹<http://www.w3schools.com/PHP/>

²<http://www.php.net>

³<http://code.tutsplus.com/tutorials/object-oriented-php-for-beginners--net-12762>

⁴<http://www.google.com>

⁵<http://www.stackoverflow.com>

⁶<http://www.php.net>

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](#)⁷

[Install Video for Xampp and PhpMyadmin](#)⁸

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 will provide 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](#)⁹

You should familiarize yourself with how to create a database, how 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 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](#)¹⁰ are a great reference.

For my IDE, I use PhpStorm. 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 PhpStorm is a paid IDE. In order to install Eclipse, you will have to install the Java sdk first.

[Download Eclipse](#)¹¹

[Download Netbeans](#)¹²

You will also need to install Composer, which we will 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](#)¹³

⁷<https://www.apachefriends.org/index.html>

⁸<https://www.youtube.com/watch?v=dV3JjLhi4Jk>

⁹<http://dev.mysql.com/downloads/workbench/>

¹⁰<http://www.w3resource.com/mysql/mysql-tutorials.php>

¹¹<https://www.eclipse.org/>

¹²<https://netbeans.org/>

¹³<https://getcomposer.org/download/>

Enable Curl¹⁴

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 Git¹⁵

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 2¹⁶

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 devel, that is your choice.

I will provide 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](#)¹⁷ 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.

¹⁴<http://www.tomjepson.co.uk/enabling-curl-in-php-php-ini-wamp-xamp-ubuntu/>

¹⁵<http://git-scm.com/downloads>

¹⁶<http://sourceforge.net/projects/console/>

¹⁷<http://www.google.com>



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](#)¹⁸

You can format your code with a formatter at [Php Formatter](#)¹⁹, if you want to make it more readable. Obviously be careful not to break the code.

Contact Bill Keck

Please note that LeanPub does not forward me any contact info, so even though you may write to me to give me feedback, I have no way to respond. In light of that, please report any bugs to me at my Yii 2 blog:

[Bill Keck's Yii 2 Blog](#)²⁰

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.

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 programmers 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

¹⁸<https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-2-coding-style-guide.md>

¹⁹<http://beta.phpformatter.com/>

²⁰<http://yii2framework.wordpress.com/>

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 Setup²¹](http://www.yiiframework.com/doc-2.0/guide-tutorial-advanced-app.html)

Ok, let's get started:

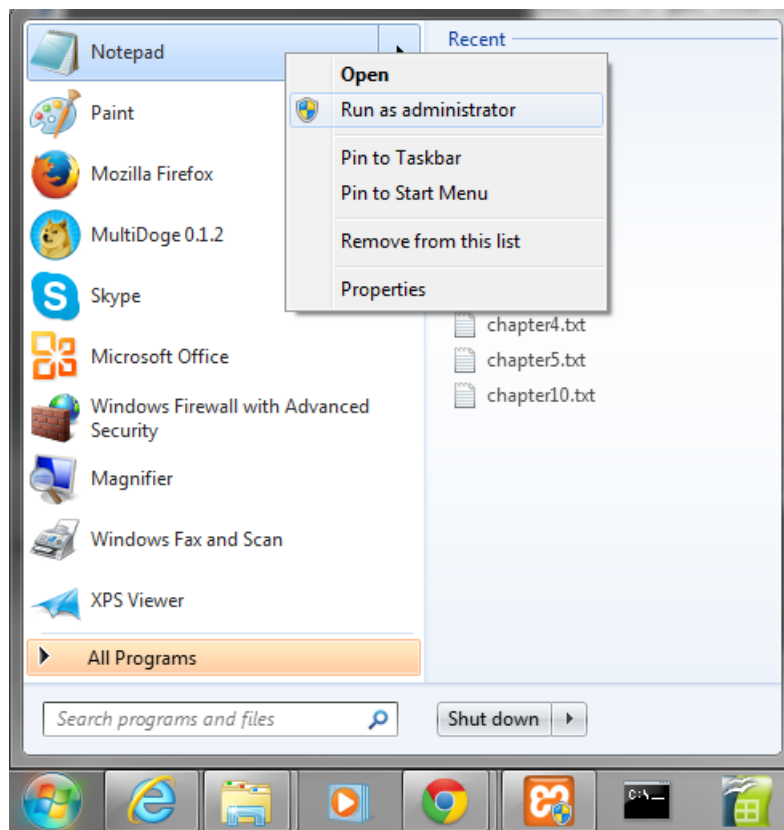
²¹<http://www.yiiframework.com/doc-2.0/guide-tutorial-advanced-app.html>

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.

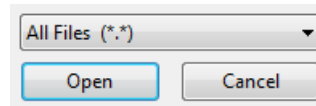


Notepad in Administrator Mode

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

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:

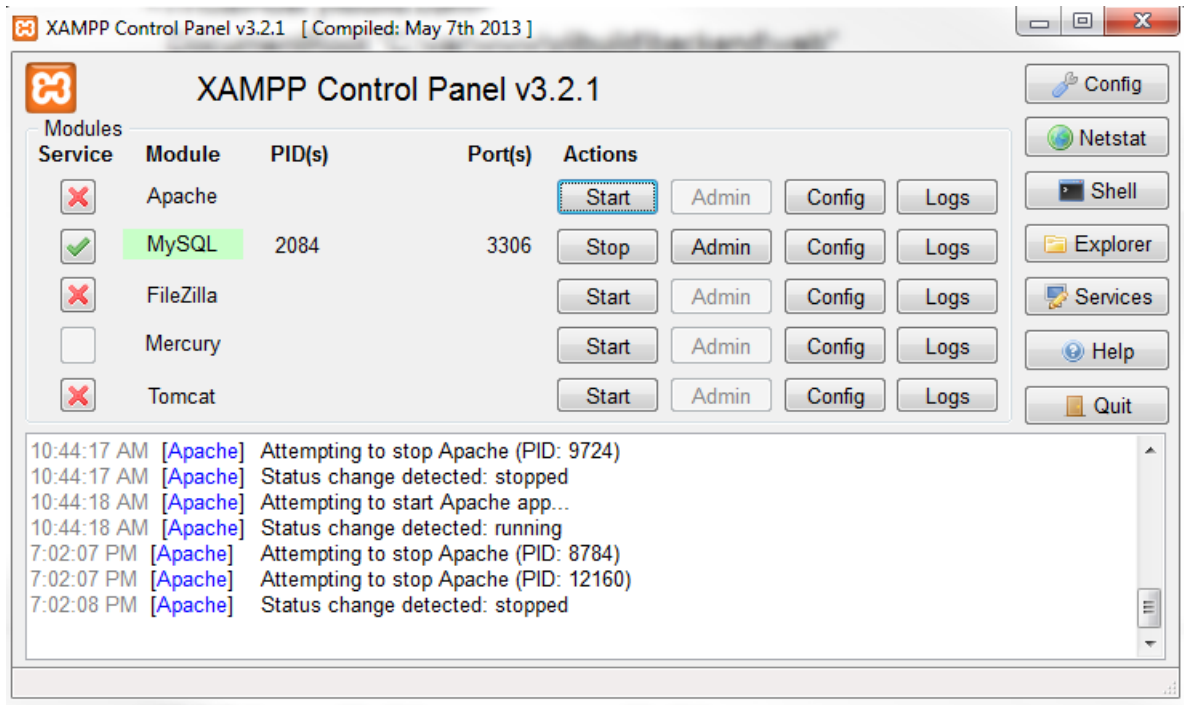
hosts

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.

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
```

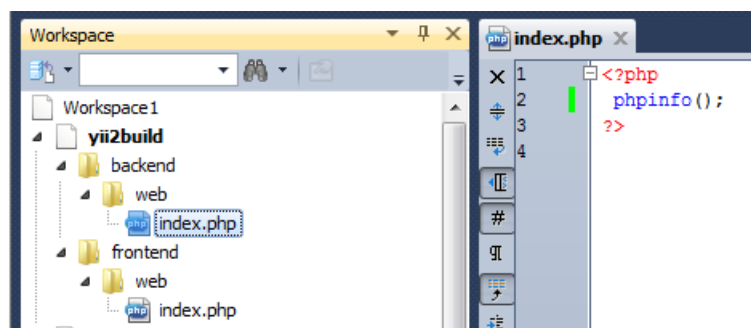
```
phpinfo();
```

```
?>
```

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

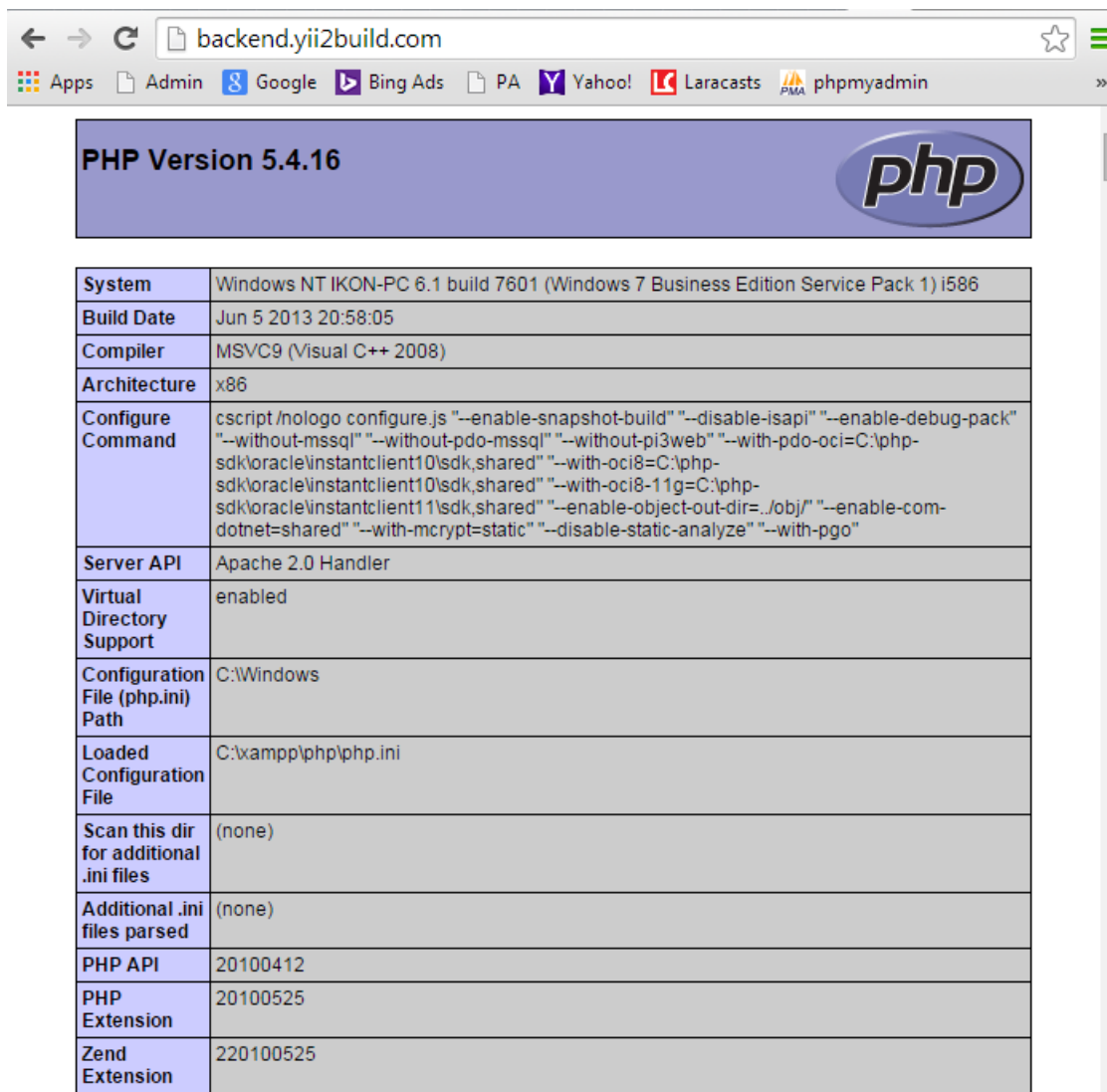
```
yii2build/frontend/web/index.php
```

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



Folder Structure for Test Host

If you type `yii2build.com` and `backend.yii2build` 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.



PHP Version 5.4.16	
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.

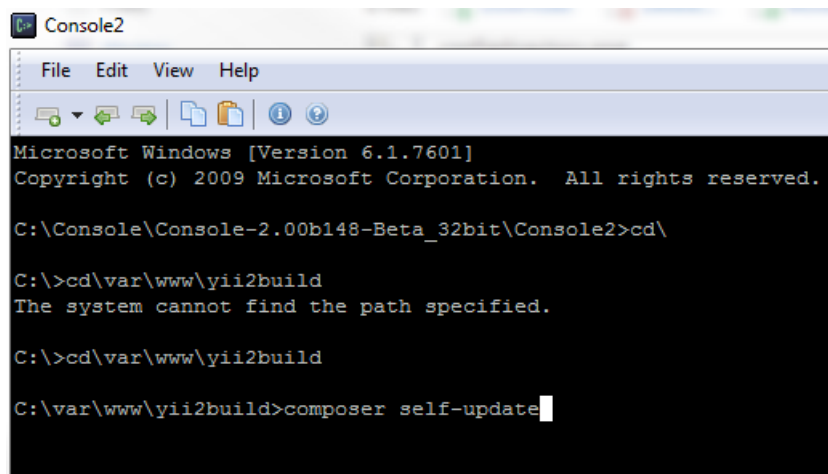
Step 6 - Find Command Line Path

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`.

A screenshot of a Windows Command Prompt window titled 'Console2'. The window shows the following text: 'Microsoft Windows [Version 6.1.7601] Copyright (c) 2009 Microsoft Corporation. All rights reserved. C:\Console2\Console2-2.00b148-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.*@dev"
```

```
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.

Tip

I checked the Yii 2 Guide and the latest recommended version of the plugin is “fxp/composer-asset-plugin:1.0.0-beta3” 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
```

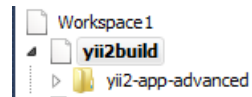
Composer Install Yii 2

Tip

The directions in the guide are slightly different in that you can set the root folder by the install. I don't like to do it this way 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.

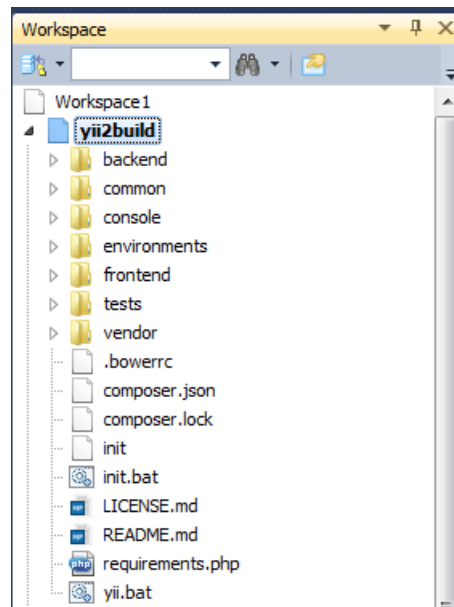
Step 9 - Check For Yii 2 Folder

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

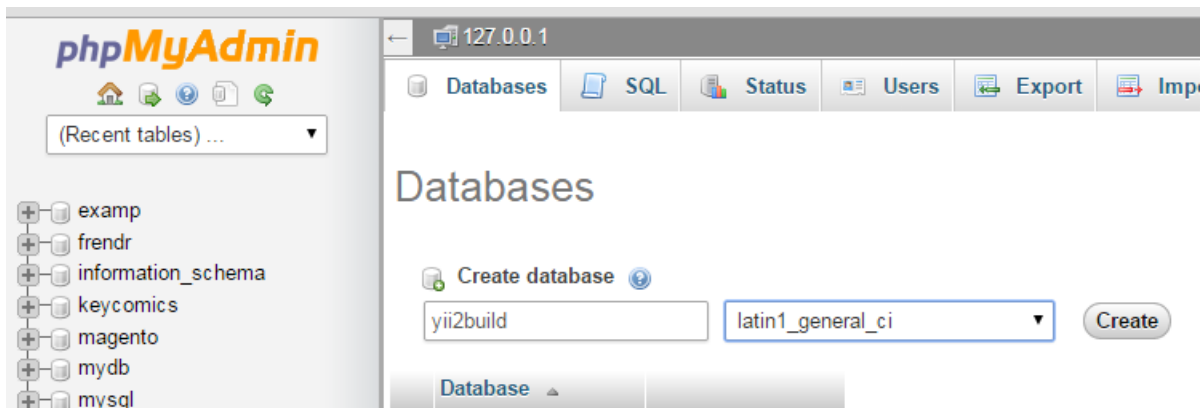
Your choice [0-1, or "q" to quit] 0

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 latin1_general_ci collation.



Create the Database

Step 12 - Set DB Connection

Adjust the components array in yii2build\common\config\main-local.php accordingly. Should look like this:

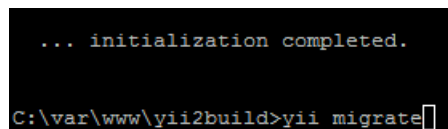
```
'db' => [  
    'class' => 'yii\db\Connection',  
    'dsn' => 'mysql:host=localhost;dbname=yii2build',  
    'username' => 'root',  
    'password' => 'yourpassword',  
    'charset' => 'utf8',  
],
```

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:

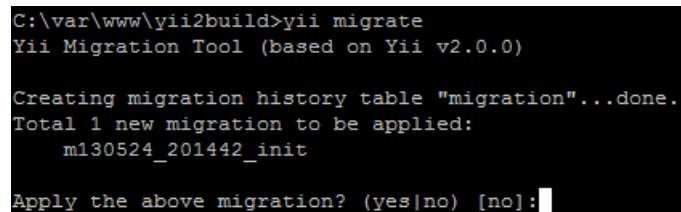
```
\var\www\yii2build>yii migrate
```



```
... initialization completed.  
C:\var\www\yii2build>yii migrate
```

Migrate

Confirm yes.

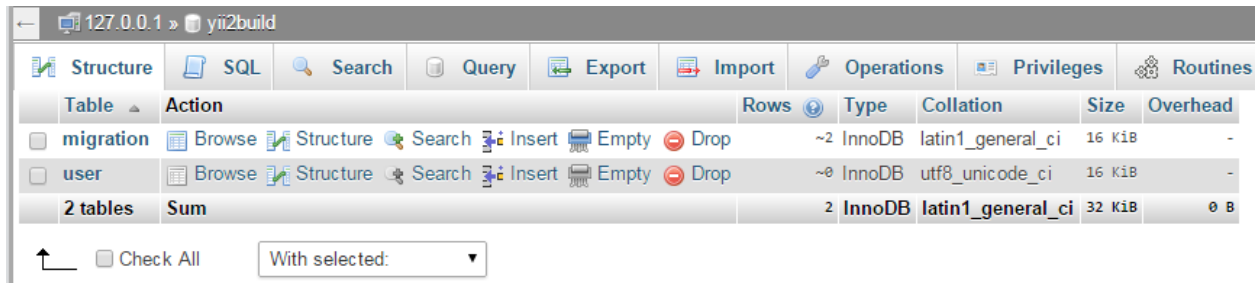


```
C:\var\www\yii2build>yii migrate  
Yii Migration Tool (based on Yii v2.0.0)  
  
Creating migration history table "migration"...done.  
Total 1 new migration to be applied:  
    m130524_201442_init  
Apply the above migration? (yes|no) [no]:
```

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:

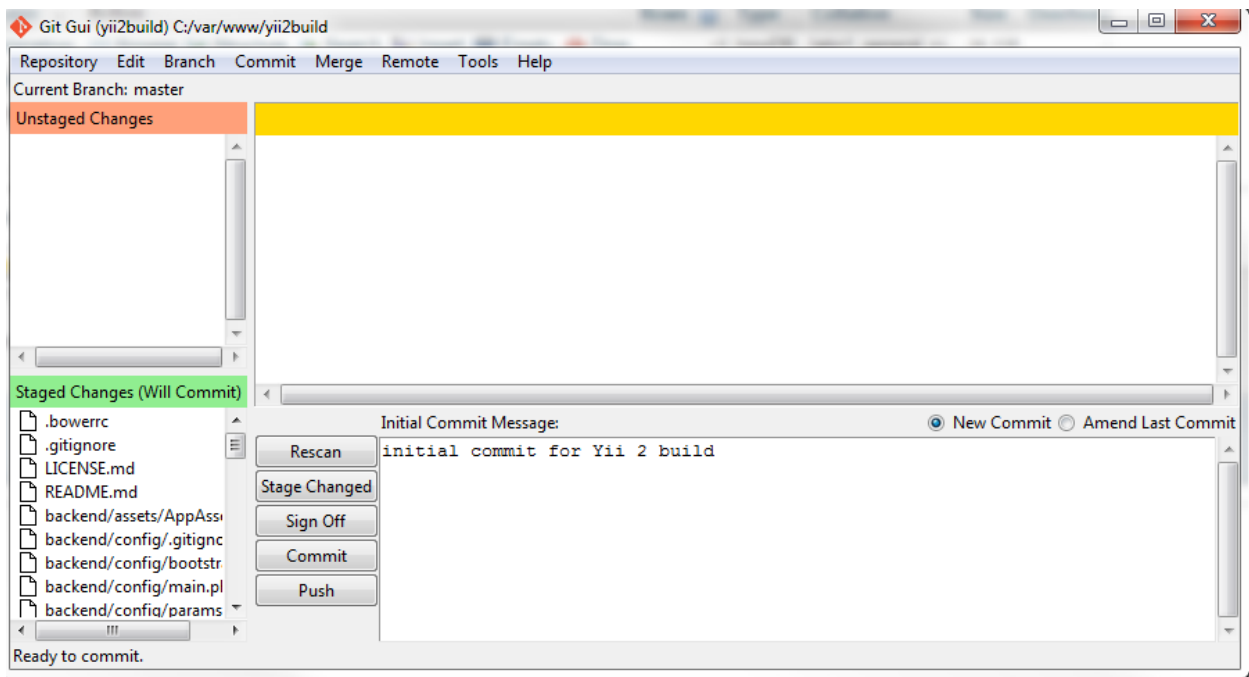
```
migrations  
user
```



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 Advanced App Install into the message area, then select commit.



Commit in Git

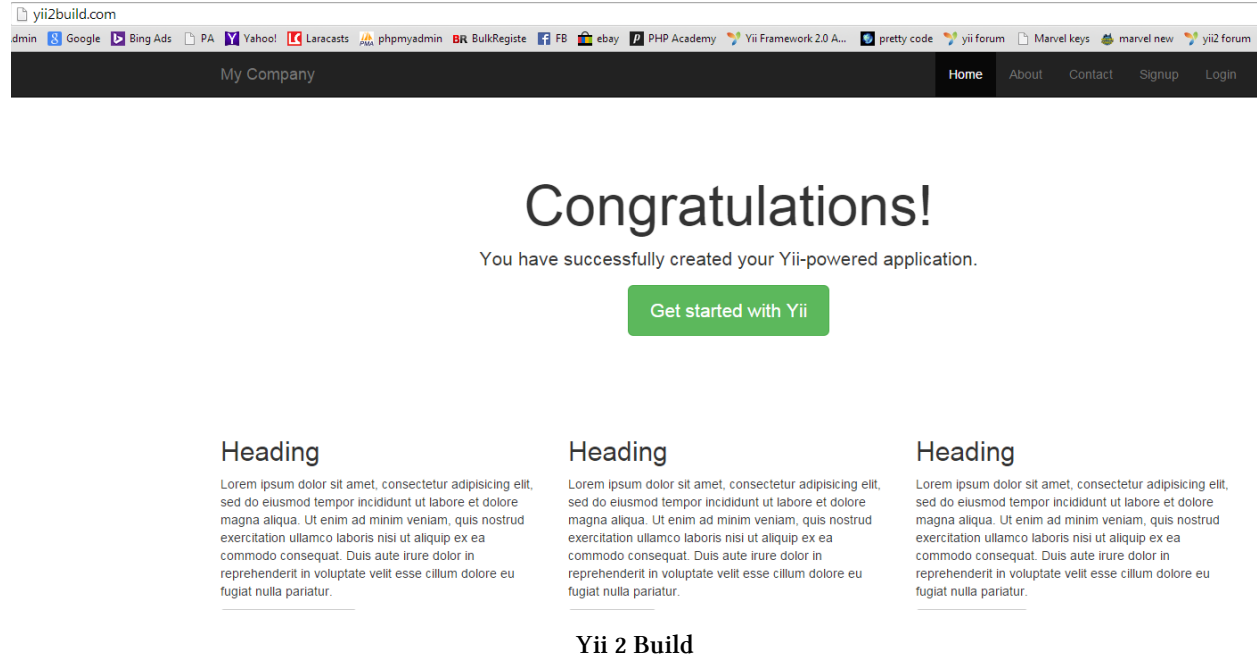
You have to stage changes before committing. You may have to unlock index or click continue from a pop up dialog box. You will also need to enter a comment before committing.

To view the repository from the repository menu, select visualize all branch history. This will show you the current master branch and it's 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.

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.



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

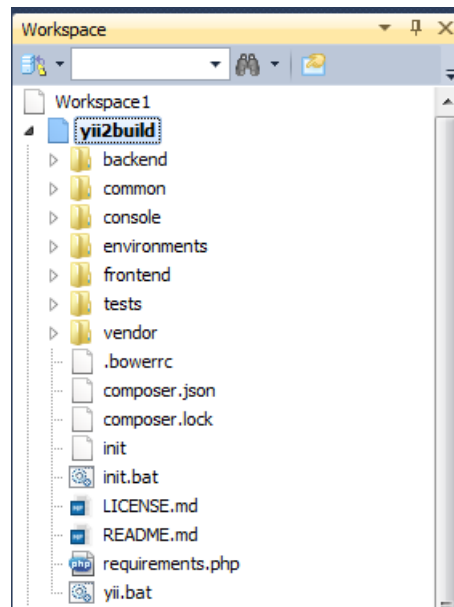
Congratulations, the hardest part of the book is over. Hopefully, this went smoothly for you. If you did have problems with setup, repeat the steps until you get it right. If you are sure everything is right, but it's still not working, consult with the individual docs of the components to see if something 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:

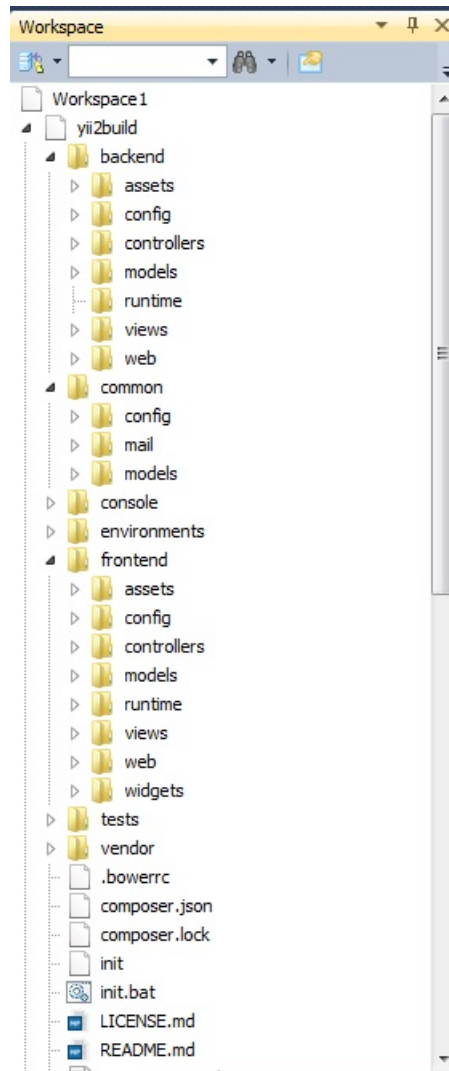


directory Structure

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.yiibuild.com goes to the backend folder version and yiibuild.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 and requires the files specified in the array:

```
$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, using the config for a dependency injection, which now becomes the container for 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.

Also, please note, I only mention dependency injection in passing because the subject seems to hold a lot of interest lately. 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:

```
yiibuild.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 doing this with our project. By waiting on that, we eliminate having to debug the url or apache, if a problem with the page should arise.

After we're done, you can easily consult with the Yii 2 guide to set pretty urls if you wish:

Pretty Urls²²

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 `yiibuild.com` for example, then, the route specified by `yii\webApplication::$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:

```
yiibuild.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 fuzzy 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.

²²<http://www.yiiframework.com/doc-2.0/guide-runtime-url-handling.html#customizing-urls>

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](#)²³

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 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

²³<http://getbootstrap.com/>

- 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?

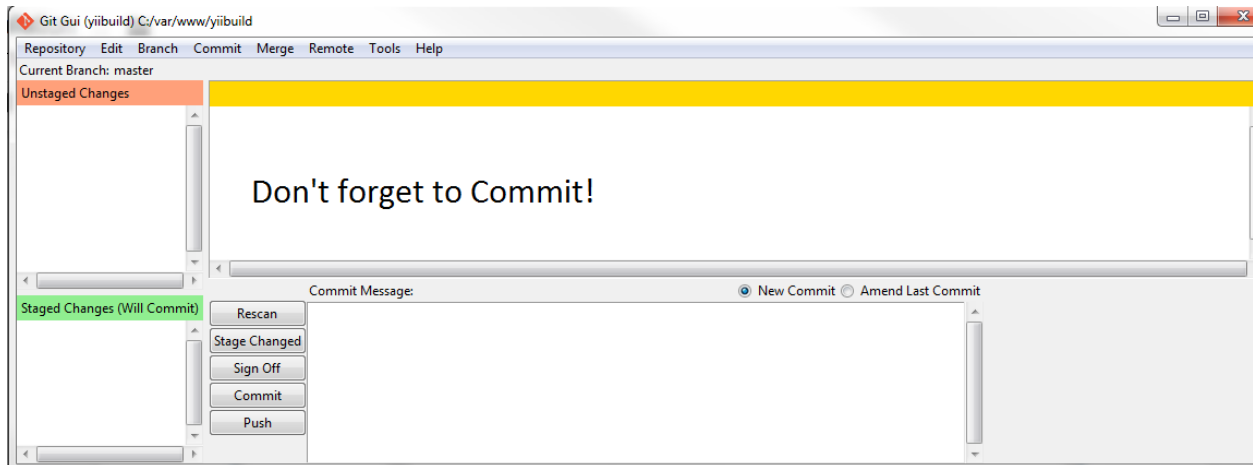
Asking myself those questions led me to conclude that this book should build an application template named Yii 2 Build. Now wait a minute, isn't the advanced application installation itself a template? Yes it is. But we are going to take things a little further.

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 personally I find a bit complicated and prefer a simpler 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 role and one for status in the user table. They 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 ambiguity problems if we left the existing fields on the user table as role and status. Ambiguity 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 field 'id', which most of our tables are going to have. If we had a field 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 role and status fields on the user table to role_id and status_id respectively. 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 change role to role_id and make sure it has a default value of '10'. Similarly, let's change status to status_id and give that a default value of '10' as well. Now let's add one more field to the user table, and we'll call that user_type_id and also give that a default value of '10'.

Here is a screenshot from MySQL Workbench:

The screenshot displays the MySQL Workbench interface. The top pane shows the 'user' table structure with the following columns:

- id INT(11)
- username VARCHAR(255)
- auth_key VARCHAR(32)
- password_hash VARCHAR(255)
- password_reset_token VARCHAR(255)
- email VARCHAR(255)
- role_id SMALLINT(6)
- user_type_id SMALLINT
- status_id SMALLINT(6)
- created_at DATETIME
- updated_at DATETIME

The right pane shows the 'gender' table structure with the following columns:

- last_name TEXT
- birthdate DATE
- gender_id INT
- created_at DATETIME
- updated_at DATETIME
- id INT
- gender_name VARCHAR

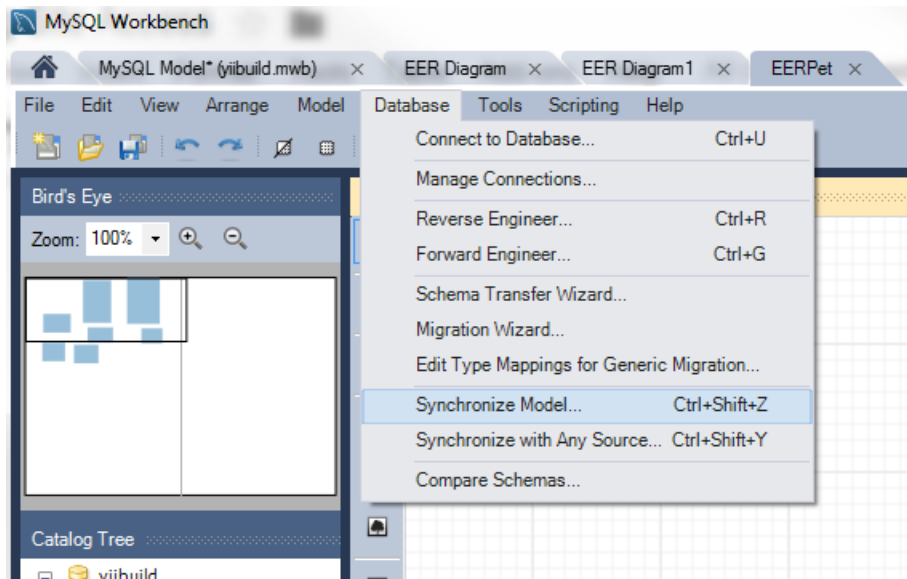
The bottom pane shows the 'user - Table' view with the following columns:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT(11)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input 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 checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
role_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"/>	'10'
user_type_id	SMALLINT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'10'
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"/>	'10'
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

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`.

Don't forget to sync to the database:



Synchronize to Database

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. Replace your existing User model with the following:

```
<?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 = 10;

    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' => 10],

        ['user_type_id', 'default', 'value' => 10],

        ['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]);
}

/**
 * @findIdentityByAccessToken
 */

public static function findIdentityByAccessToken($token, $type = null)
{
    return static::findOne(['auth_key' => $token]);
}
```

```
/**
 * 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)
{
    $expire = Yii::$app->params['user.passwordResetTokenExpire'];
    $parts = explode('_', $token);
    $timestamp = (int) end($parts);
    if ($timestamp + $expire < time()) {
        // token expired
        return null;
    }

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

/**
```

```
* @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

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. If you wish, you can use [phpformatter.com](http://beta.phpformatter.com)²⁴ to format your code, it's a handy tool.

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 field 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 dropped some constants and changed role and status to `role_id` and `status_id`. This matches the changes to our user table. Although we added `user_type_id` to the user table, we don't see any evidence of it here, and yet, as we described above, the model knows its attributes based on the table structure.

²⁴<http://beta.phpformatter.com/>

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](#)²⁵

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.

²⁵<http://www.yiiframework.com/doc-2.0/guide-db-migrations.html>

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 = 10;
```

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. 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, though that is not covered in this book. 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. The comments 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 secrete token.
 * @param string $token the secrete token
 * @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 attacks.
    *
    * 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. 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' => 10],

        ['user_type_id', 'default', 'value' => 10],

        ['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](#)²⁶

The first 3 rules deal with setting defaults and allowable range. The white space in between rows is simply cosmetic to make it easier to work on rules for a particular attribute, 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)
{
    return static::findOne(['auth_key' => $token]);
}
```

Also created for us by the advanced template:

²⁶<http://www.yiiframework.com/doc-2.0/guide-input-validation.html>

```

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

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

```

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](#)²⁷

Boilerplate Methods

The next few classes are all part of the boilerplate, which we did not change. The comments provide the 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
 */
```

²⁷<http://www.yiiframework.com/doc-2.0/guide-security-authentication.html>

```
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. Any old users were likely dropped when you made the changes to the schema. 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 SignupForm model, 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 `user_id` to 10, 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 we may as well take a quick peek, and see how this works by looking at the `actionSignup()` method on the `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 SignupForm 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.

Right now somebody's head is probably exploding because I explained something so basic. But just keep in mind this is a beginner's book and we want to refresh and grow our programming skills as we move along.

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 `signup` 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->goBack();
```

This simply takes you back to the page you started on, 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();
    }

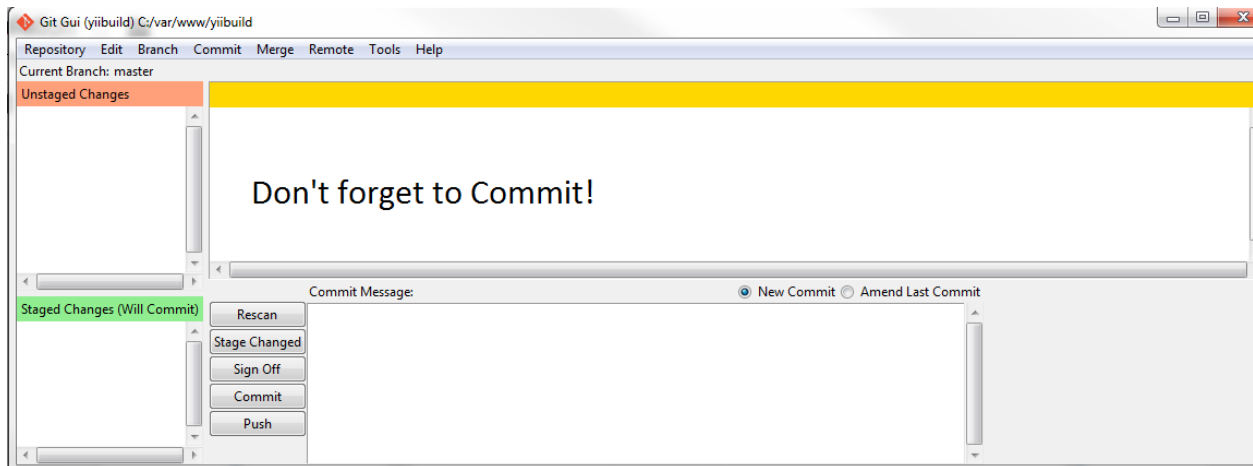
    $model = new LoginForm();
    if ($model->load(\Yii::$app->request->post()) && $model->login()) {
        return $this->goBack();
    } else {
        return $this->render('login', [
            'model' => $model,
        ]);
    }
}
```

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 :

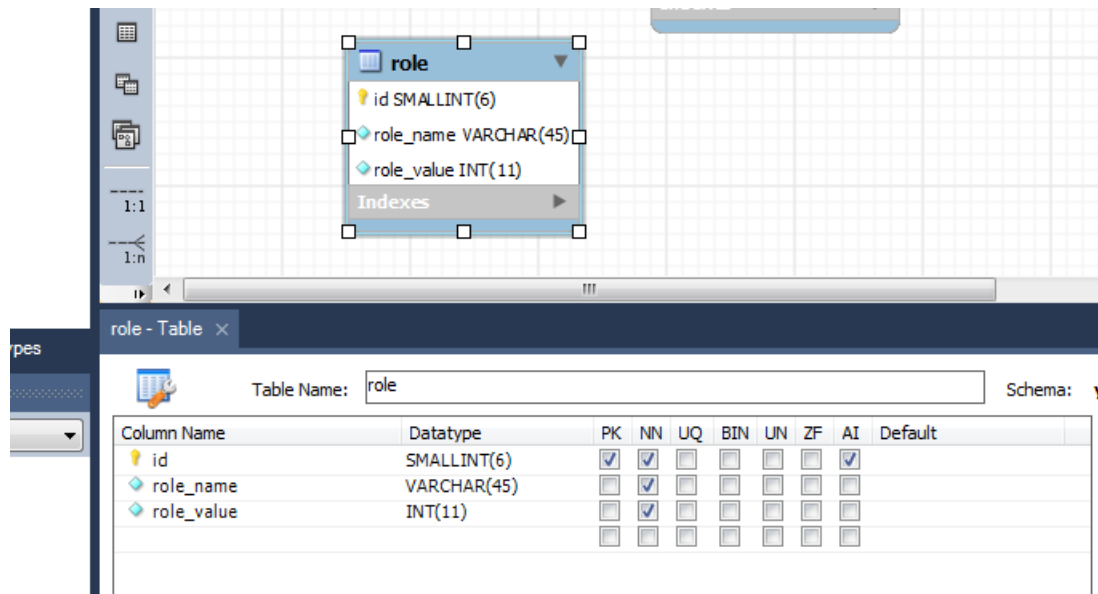
- NOT NULL

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

For a tutorial reference on Mysql, check [W3Resource](http://www.w3resource.com)²⁸.

Role Table

Here is the table for role:



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.

²⁸<http://www.w3resource.com/mysql/creating-table-advance/constraint.php>

Status Table

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

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 type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
status_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"/>	<input type="checkbox"/>
status_value	SMALLINT(6)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input 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:

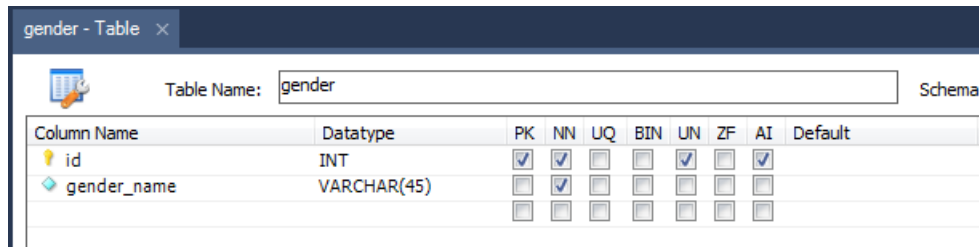
Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	SMALLINT(6)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
user_type_name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



The screenshot shows the Gii tool interface for creating a table named 'gender'. The table name is entered in the 'Table Name' field. Below it is a table with columns for 'Column Name', 'Datatype', and various constraints (PK, NN, UQ, BIN, UN, ZF, AI, Default). The 'id' column is an INT with PK, NN, and AI checked. The 'gender_name' column is a VARCHAR(45) with NN checked.

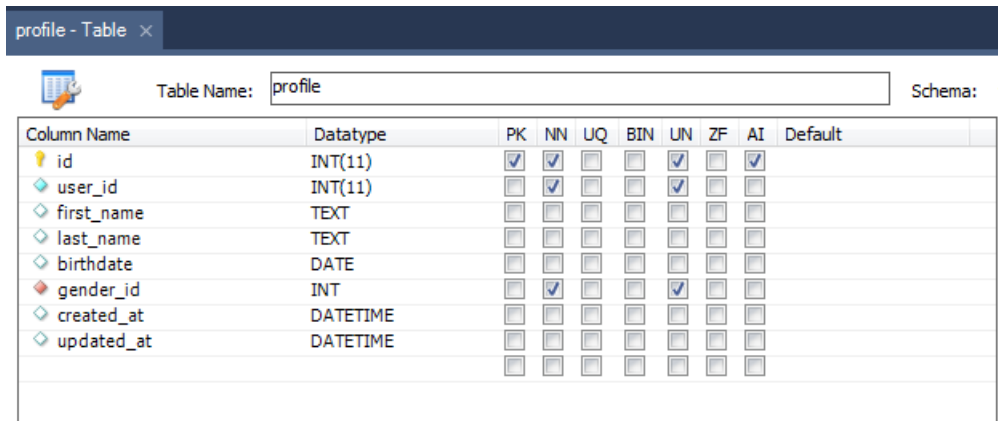
Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
gender_name	VARCHAR(45)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



The screenshot shows the Gii tool interface for creating a table named 'profile'. The table name is entered in the 'Table Name' field. Below it is a table with columns for 'Column Name', 'Datatype', and various constraints (PK, NN, UQ, BIN, UN, ZF, AI, Default). The 'id' column is an INT(11) with PK, NN, UN, and AI checked. The 'user_id' column is an INT(11) with NN and UN checked. The 'gender_id' column is an INT with NN and UN checked. The 'created_at' and 'updated_at' columns are DATETIME.

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	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
last_name	TEXT	<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	INT	<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. This is why on the User model we are going to add the following method:

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

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

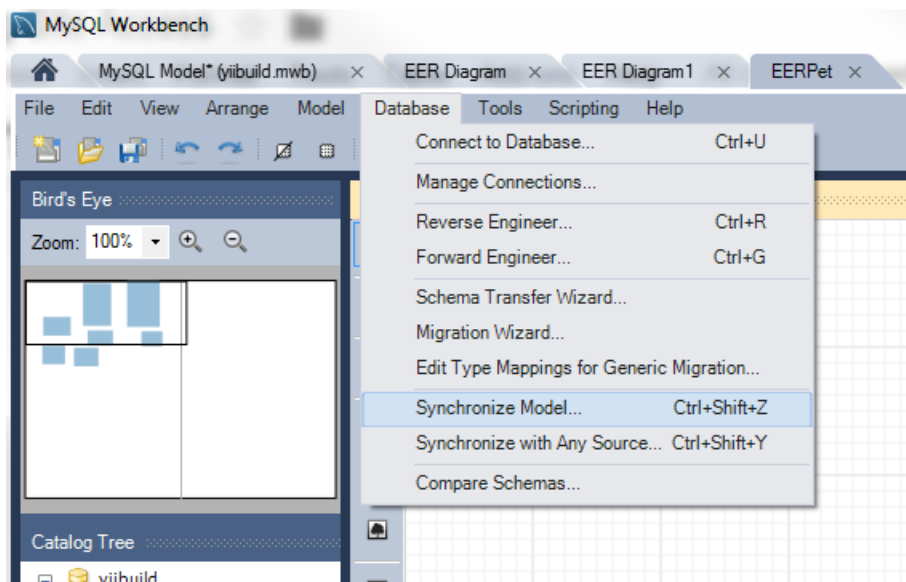
Note that on the profile int columns, I checked off UN, which stands for unsigned and does not allow negative numbers.

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.

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.

Synchronize

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



Synchronize

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

 A screenshot of the phpMyAdmin web interface. The 'Structure' tab is active, displaying a table structure overview for a database named 'yiibuild'. The interface includes a sidebar with a tree view of databases and a main table listing the tables in the database.

Table	Action	Rows	Type	Collation	Size	Overhead
gender	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	16 K1B	-
migration	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	16 K1B	-
profile	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	32 K1B	-
role	Browse Structure Search Insert Empty Drop	~3	InnoDB	latin1_general_ci	16 K1B	-
status	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	16 K1B	-
user	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	16 K1B	-
user_type	Browse Structure Search Insert Empty Drop	~2	InnoDB	latin1_general_ci	16 K1B	-
7 tables	Sum	15	InnoDB	latin1_general_ci	128 K1B	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:

`yii2build.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.

This is what my file looks like:

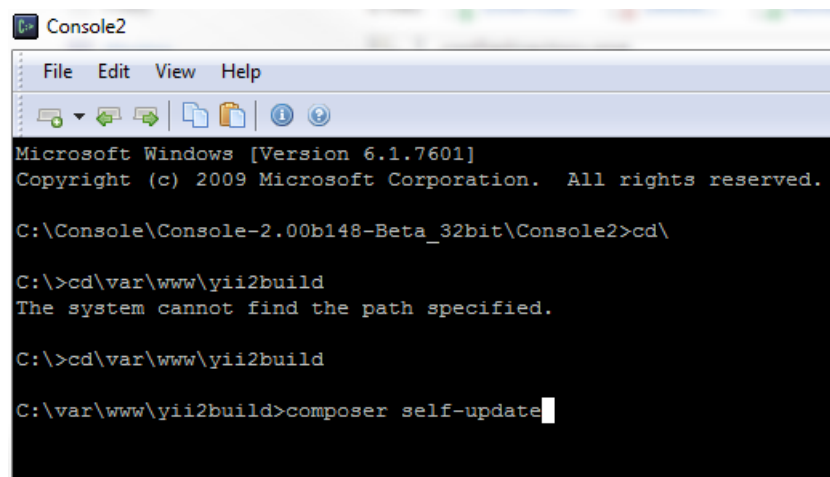
```
{
  "name": "yiisoft/yii2-app-advanced",
  "description": "Yii 2 Advanced Application Template",
  "keywords": ["yii2", "framework", "advanced", "application template"],
  "homepage": "http://www.yiiframework.com/",
  "type": "project",
  "license": "BSD-3-Clause",
  "support": {
    "issues": "https://github.com/yiisoft/yii2/issues?state=open",
    "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",
    "yiisoft/yii2-authclient": "*",
    "fortawesome/font-awesome": "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, authclient, 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:



```

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 Update

Now if you go back to your url:

yii2build.com/index.php?r=gii

You should get to Gii. If you had Gii in the first place, make sure you run the composer update, 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

Ok, point your browser to:

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

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'm not a big fan of this approach or at least I haven't seen it used in a way that makes sense to me. My feeling is that it's better to have one authoritative file for model, unless absolutely necessary to do otherwise.

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 >	<h2>Model Generator</h2> <p>This generator generates an ActiveRecord class for the specified database table.</p> <p>Table Name</p> <input type="text" value="role"/> <p>Model Class</p> <input type="text" value="Role"/> <p>Namespace</p> <input type="text" value="backend\models"/> <p>Base Class</p> <input type="text" value="yii\db\ActiveRecord"/> <p>Database Connection ID</p> <input type="text" value="db"/> <p><input type="checkbox"/> Use Table Prefix</p> <p><input checked="" type="checkbox"/> Generate Relations</p> <p><input type="checkbox"/> Generate Labels from DB Comments</p> <p><input type="checkbox"/> Enable I18N</p> <p>Code Template</p> <input type="text" value="default (C:\var\www\yii\build\vendor\yiisoft\yii2-gii\generators\model\default)"/> <p>Preview</p>
CRUD Generator >	
Controller Generator >	
Form Generator >	
Module Generator >	
Extension Generator >	

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
 * @property User[] $users
 */

class Role extends \yii\db\ActiveRecord
{

    /**
     * @inheritdoc
     */

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

    /**
     * @inheritdoc
     */

    public function rules()
    {
        return [

            [['role_name'], 'string', 'max' => 45],
            [['role_name', 'role_value'], 'required'],
        ];
    }
}
```

```
        [['role_value'], 'integer']
    ];
}

/**
 * @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.

Add Records To Role Table

Now that we have a role table and Role model, let's us 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 phpMyAdmin interface for inserting records into a table named 'role'. The interface includes a navigation menu on the left with various databases listed, and a top navigation bar with options like Browse, Structure, SQL, Search, Insert, Export, Import, and Operations. The main area displays two 'Insert' forms. The first form is for a record named 'User' with a role_value of 10. The second form is for a record named 'Admin' with a role_value of 20. Both forms have an 'id' field that is not required (Null) and a 'role_name' field that is required (Not Null). The 'role_value' field is also required (Not Null). The 'id' field is a smallint(6) type, 'role_name' is a varchar(45) type, and 'role_value' is an int(11) type. The 'id' field is not required for either record. The 'role_name' field is required for both records. The 'role_value' field is required for both records. The 'id' field is a smallint(6) type, 'role_name' is a varchar(45) type, and 'role_value' is an int(11) type. The 'id' field is not required for either record. The 'role_name' field is required for both records. The 'role_value' field is required for both records.

Column	Type	Function	Null	Value
id	smallint(6)			
role_name	varchar(45)		Not Null	User
role_value	int(11)		Not Null	10

Column	Type	Function	Null	Value
id	smallint(6)			
role_name	varchar(45)		Not Null	Admin
role_value	int(11)		Not Null	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:

Showing rows 0 - 1 (2 total, Query took 0.0000 sec)

```
SELECT *
FROM `role`
LIMIT 0, 30
```

Show : Start row: 0 Number of rows: 30 Headers every 100 rows

Sort by key: None

+ Options

		id	role_name	role_value
<input type="checkbox"/>				1 user 10
<input type="checkbox"/>				2 admin 20

Check All With selected: Change Delete Export

Show : Start row: 0 Number of rows: 30 Headers every 100 rows

Query results operations

PhpMyAdmin Role Records

If you recall, we set `role_id` on the `user` table to default to 10 when a user record is created. So it makes sense that `role_id` will map to `role_value`. 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 file add:

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

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

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 a `hasMany` 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' => 'role_value']
```

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' => 'role_value'])
```

`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:

```
/**
 * get role relationship
 *
 */

public function getRole()
{
    return $this->hasOne(Role::className(), ['role_value' => '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, 'role_value', '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 just simply says that `role_value` 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 values 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, 'role_value', 'role_name');
}

```

Here we create the local variable `$droptions` and assign an instance of `Role`, with all records returned as an array. 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](#)²⁹

²⁹<http://www.yiiframework.com/doc-2.0/guide-db-active-record.html>

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 `role_value` records. So this statement returns 10, 20 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 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 User[] $users
 */

class Status extends \yii\db\ActiveRecord
{
```

```
/**
 * @inheritdoc
 */

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

/**
 * @inheritdoc
 */

public function rules()
{
    return [

        [['status_name'], 'string', 'max' => 45],
        [['status_name', 'status_value'], 'required'],
        [['status_value'], 'integer']

    ];
}

/**
 * @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:

```
/**  
 * @return \yii\db\ActiveQuery  
 */  
  
public function getUsers()  
{  
    return $this->hasMany(User::className(), ['status_id' => 'status_value']);  
}
```

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:

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

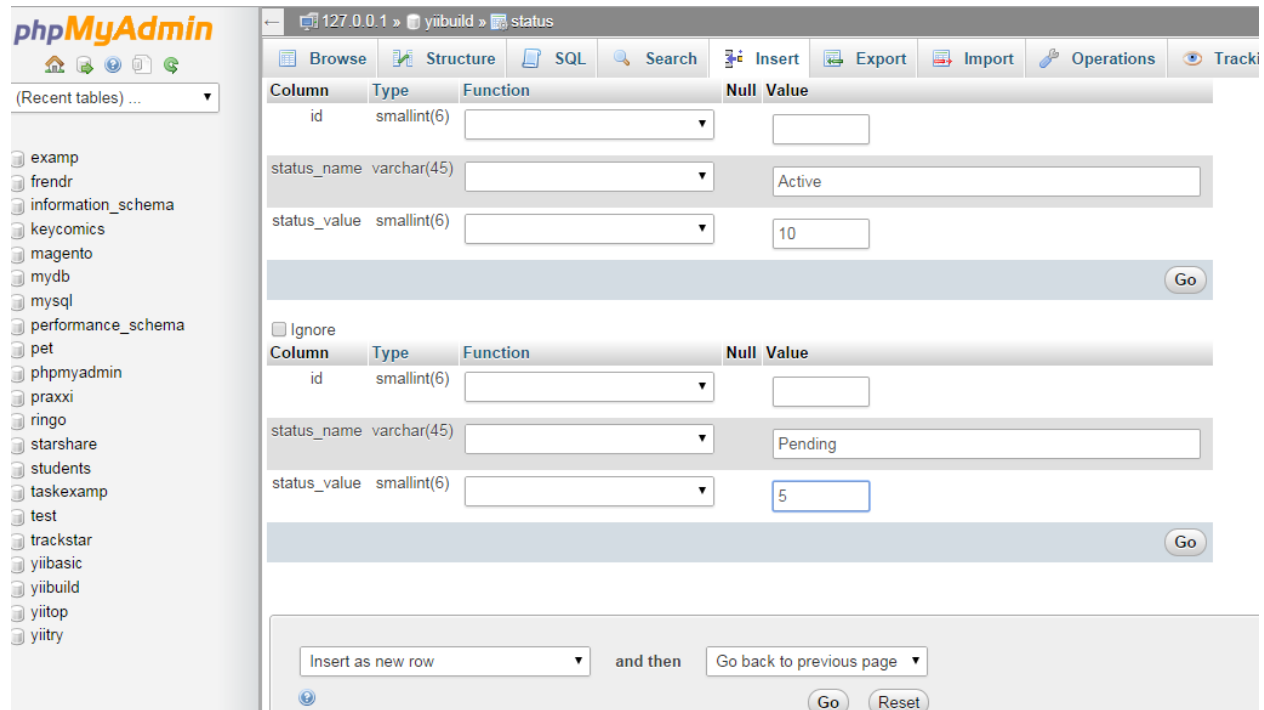
Add Records to Status Table

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:

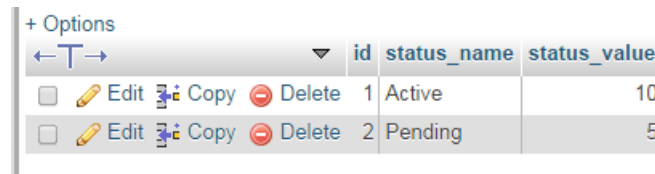
```
status_name: Active status_value: 10
```

```
status_name: Pending status_value: 5
```



Insert Into Status

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



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_value'], 'required'],
            [['user_type_value'], 'integer'],
        ];
    }
}
```

```

        [['user_type_name'], 'string', 'max' => 45]
    ];
}

/**
 * @inheritdoc
 */

public function attributeLabels()
{
    return [
        'id' => 'ID',
        'user_type_name' => 'Type Name',
        'user_type_value' => '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;
```

And then the getUsers method to setup the relation to users:

```

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

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

```

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:

```
/**
 *getUserType
 *line break to avoid word wrap in PDF
 * code as single line in your IDE
 */

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

```

/**
 * get user type id
 *
 */

public function getUserTypeId()
{
    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 upper case 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:

value = 10, name = Free value = 30, name = Paid

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	10
<input type="checkbox"/> Edit Copy Delete	2	Paid	30

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 string $id
 * @property string $gender_name
 *
 *
 */

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']);
}

}

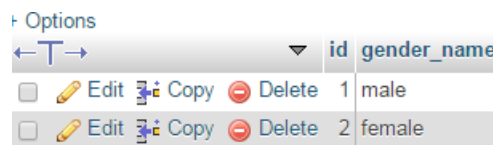
```

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:



+ Options				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 all be set by default when a user registers, and we have connected them to models that have a datastructure 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

Model Class

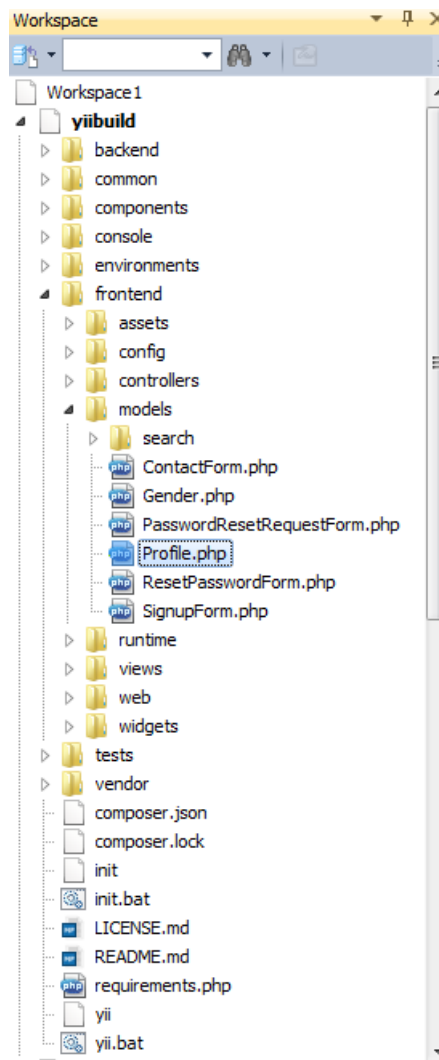
Namespace

Base Class

Database Connection ID

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\db\ActiveRecord;
```

```
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 [
        [['birthdate', 'created_at', 'updated_at'], 'safe'],
        [['gender_id', 'user_id', 'birthdate'], 'required'],
        [['gender_id', 'user_id'], 'integer'],
        [['first_name', 'last_name'], 'string', 'max' => 45],
    ];
}
```

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

/**
 * @inheritdoc
 */

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

/**
 * @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 the use statements:

```
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())]
```

Moving down the class, we need to add some attribute labels for the relationship methods under the `attributeLabels` method, so we can use them 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:

```
'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 calls out timestamp:

```
/**
 * 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.

```

/**
 * uses magic getGender on return statement
 *
 */

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 include 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 will need to include 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 it's 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.

Update User Model with Profile

So now we update the user model with methods to pull in the profile relation. Add the following methods to User.php:

```
/**
 * @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);
}
```


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.

```
/**
 * 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:

```

/* 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'),
        'userId' => 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:

```
<?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 = 10;

    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' => 10],
        [['role_id'], 'in', 'range' => array_keys($this->getRoleList())],

        ['user_type_id', 'default', 'value' => 10],
        [['user_type_id'], 'in', 'range' => array_keys($this->getUserTypeList())],
    ],
}
```

```
        ['username', 'filter', 'filter' => 'trim'],
        ['username', 'required'],
        ['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'),
        'userId' => 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)
{
    return static::findOne(['auth_key' => $token]);
}

/**
 * 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)
{
```

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

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

/**
 * @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(), ['role_value' => '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, 'role_value', 'role_name');
}

/**
 * @getStatus
 *
 */

public function getStatus()
{
    return $this->hasOne(Status::className(), ['status_value' => '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, 'status_value', '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
 * 2 lines to avoid wordwrap
 */

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

```
/**
 * @getUserTypeId
 *
 */
```

```
public function getUserTypeId()
{
    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()
{
```

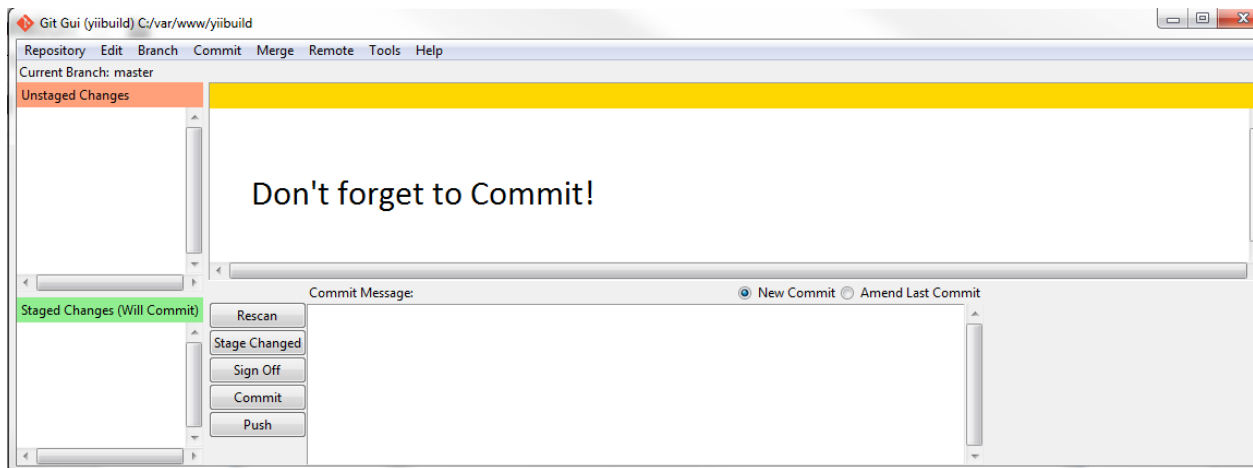
```

    $url = Url::to(['user/view', 'id'=>$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 stitched all these models together 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're 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 user_type_name? 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. Let's go ahead and create that 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:

```
<?php

namespace common\models;

Class ValueHelpers

{

    /**
     * return the value of a role name handed in as string
     * example: 'Admin'
     *
     * @param mixed $role_name
     */

    public static function getRoleValue($role_name)
    {

        $connection = \Yii::$app->db;
        $sql = "SELECT role_value FROM role WHERE role_name=:role_name";
        $command = $connection->createCommand($sql);
        $command->bindValue(":role_name", $role_name);
        $result = $command->queryOne();

        return $result['role_value'];
    }

    /**
     * return the value of a status name handed in as string
     * example: 'Active'
     * @param mixed $status_name
     */

    public static function getStatusValue($status_name)
    {

        $connection = \Yii::$app->db;
        $sql = "SELECT status_value FROM status WHERE status_name=:status_name";
        $command = $connection->createCommand($sql);
```



```

        $command->bindValue(":status_name", $status_name);
        $result = $command->queryOne();

        return $result['status_value'];
    }

    /**
     * returns value of user_type_name so that you can
     * used in PermissionHelpers methods
     * handed in as string, example: 'Paid'
     *
     * @param mixed $user_type_name
     */

    public static function getUserTypeValue($user_type_name)
    {

        $connection = \Yii::$app->db;
        $sql = "SELECT user_type_value FROM user_type
            WHERE user_type_name=:user_type_name";
        $command = $connection->createCommand($sql);
        $command->bindValue(":user_type_name", $user_type_name);
        $result = $command->queryOne();

        return $result['user_type_value'];
    }

}

```

You can see we have 3 methods are exactly the same, except they each reference a different model. So obviously, we could hand in a second argument for \$modelName and it would be less procedural and we would just have one method. But I chose not to do that because it's just a little more intuitive this way to understand the method, and these are methods we want quick access to, without having to think too much about how they work or what they do.

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:

The screenshot shows the phpMyAdmin interface. On the left is a tree view of databases. The main area displays a table named 'role' with the following data:

id	role_name	role_value
1	user	10
2	admin	20

The SQL query shown is: `SELECT * FROM `role` LIMIT 0, 30`. The interface also shows options for sorting and displaying rows.

Role Records

The answer is 20. So when I want to control access to the backend, for example, I could restrict access to users that have a minimum role value of 20. Or I could match 20 exactly if I wanted to just have one kind of role have access. We are not making that decision yet, we are just anticipating that we need the method for the future.

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. Again, that's why we have 3 methods, instead of one, you can see how intuitive the syntax is, at least that is what we were aiming for.

Even though these are simple methods, we should break at least one down line by line so we know how they work. Let's look at:

```
public static function getRoleValue($role_name)
```

The first thing we will do is set the connection:

```
$connection = \Yii::$app->db;
```

Next we define the sql statement:

```
$sql = "SELECT role_value FROM role WHERE role_name=:role_name";
```

This is followed by the syntax that tells it to use the \$sql.

```
$command = $connection->createCommand($sql);
```

Notice in the method, we are using a bound parameter for role_name, which we hand in from the function signature and then bind it to the parameter like so:

```
$command->bindValue(":role_name", $role_name);
```

So \$role_name, which we hand in as a string, in our example, we used 'Admin', gets bound to

```
:role_name
```

Next we execute the query:

```
$result = $command->queryOne();
```

And then we return the result and result is formatted as an array, so we have to name the key we want:

```
return $result['role_value'];
```

And that's it, we have the value of 'Admin'. The other 2 models are exactly the same except for using a different model.

I used raw sql for our methods for two reasons. 1. They return results faster and 2. We should know how to format this kind of query. It's easy to do.

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. Here is the entire file:

```
<?php
namespace common\models;

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

Class PermissionHelpers
{
```

```
/**
 * check if the user is the owner of the record
 * use Yii::$app->user->identity->id for $userid, 'string' for model name
 * for example 'profile' will check the profile model to see if the user
 * owns the record. Provide the model instance, typically as $model->id as
 * the last parameter. Returns true or false, so you can wrap in if statement
 * @param mixed $userid
 * @param mixed $model_name
 * @param mixed $model_id
 */
```

```
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;

    }
}
```

```
/**
 * method for requiring paid type user, if test fails, redirect to upgrade page
 * $user_type_name handed in as 'string', 'Paid' for example.
 *
 * used two lines for if statement to avoid word wrapping
 *
 * @param mixed $user_type_name
 * @return \yii\web\Response
```

```
*/

public static function requireUpgradeTo($user_type_name)
{

    if ( Yii::$app->user->identity->user_type_id !=
        ValueHelpers::getUserTypeValue($user_type_name)) {

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

    }

}

/**
 * @requireStatus
 * used two lines for if statement to avoid word wrapping
 * @param mixed $status_name
 */

public static function requireStatus($status_name)
{

    if ( Yii::$app->user->identity->status_id ==
        ValueHelpers::getStatusValue($status_name)) {

        return true;

    } else {

        return false;

    }

}

/**
 * @requireMinimumStatus
 * used two lines for if statement to avoid word wrapping
```

```
* @param mixed $status_name
*/

public static function requireMinimumStatus($status_name)
{
    if ( Yii::$app->user->identity->status_id >=
        ValueHelpers::getStatusValue($status_name)) {

        return true;

    } else {

        return false;

    }
}

/**
 * @requireRole
 * used two lines for if statement to avoid word wrapping
 * @param mixed $role_name
 */

public static function requireRole($role_name)
{
    if ( Yii::$app->user->identity->role_id ==
        ValueHelpers::getRoleValue($role_name)) {

        return true;

    } else {

        return false;

    }
}
```

```
/**
 * @requireMinimumRole
 * used two lines for if statement to avoid word wrapping
 * @param mixed $role_name
 */

public static function requireMinimumRole($role_name)
{
    if ( Yii::$app->user->identity->role_id >=
        ValueHelpers::getRoleValue($role_name)) {

        return true;

    } else {

        return false;

    }

}

}
```

We're going to move through these rather quickly, we get into more detail when we actually use the methods. The first 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. If they are, it returns true, if not returns false. When using it, the syntax would look 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.

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 next example. If you do build an application that has an area for users of a different type, 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::getUserTypeValue method. So already we are reusing code from our ValueHelpers class, you have to love that.

Ok, back to the test, 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, $redirect_destination)
{
    if ( Yii::$app->user->identity->user_type_id !=
        ValueHelpers::getUserTypeValue($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 'upgrade2/upgradenow' 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.

Next up, we have a requireStatus method. This is another super simple method to see if the current user has the status that you require. It also utilizes a ValueHelper method. Cha ching! Another code reuse.

This method will come in handy inside of controllers to check if the user has sufficient status to access a page. We will use this later when we are working with controllers to control access.

I did another version of this just for fun, `requireMinimumStatus`. The only difference in this method is that we use `>=` operator that allows us to set a minimum required status. Other than that, it's exactly the same as the previous method.

The `requireRole` method takes the role name as argument, so for example, if you want to require the user to have a role with the name of Admin, you can use this method. Once again, our `ValueHelpers` class comes in handy to help us make a very succinct method.

And finally, I did a `requireMinimumRole` method exactly like the `requireRole` method, but 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.

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. Just to be clear, it checks to see if the current user has a role with at least that `role_value`.

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.

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:

```
<?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.

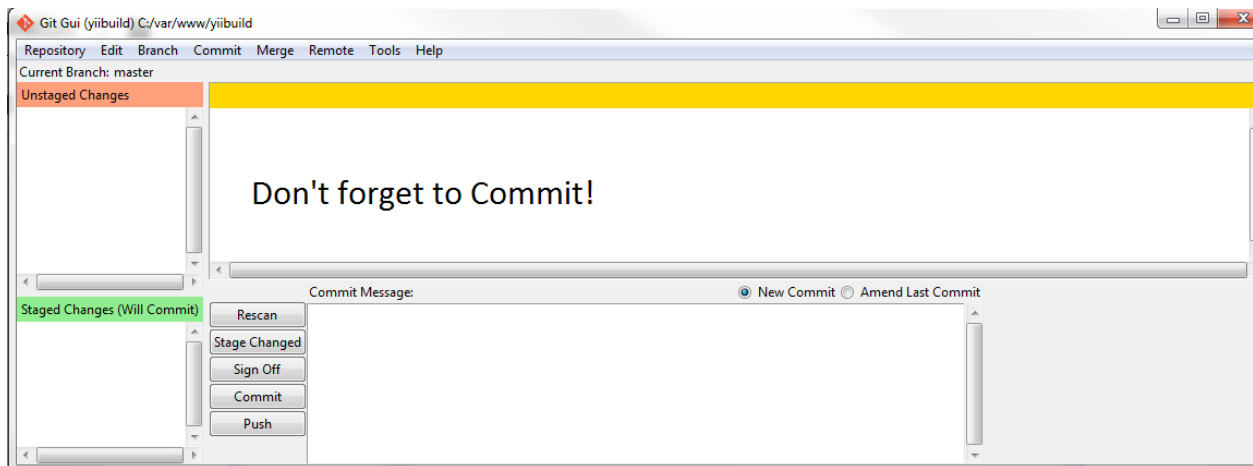
I also wrote it so I could use it with other models, I just need to hand in the model name as string. The method also uses:

```
Yii::$app->user->identity->id
```

We use this to set the id of the current user. The current user id is always available via this call throughout the application. So that is how we know we are dealing with the current user.

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



Commit!

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.

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 look at Site Controller and it's related views. This will introduce us to Yii 2's controllers, so we can gain knowledge of how they work. 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\InvalidParamException;
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 our 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 this configuration available to each action. So if you want to use Captcha for something, you need it configured in the controller, like it is above. Then it gets called out 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;
use yii\web\NotFoundHttpException;
```

Then we get our class declaration and attributes:

```
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 attributes. 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'],
    ];
}
```

We have 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:

```

/**
 * 2 lines in return statement to avoid wordwrap
 */

public function login()
{
    if ($this->validate() && $this->getUser()) {
        return Yii::$app->user->login($this->getUser(),
            $this->rememberMe ? 3600 * 24 * 30 : 0);

    } else {
        return false;
    }
}

```

Ok, slightly more complicated. `$this->validate()` is calling the `validate` method of `Model`. and `getUser` from the `LoginForm` model, which we will see in a moment. If we validate and get the user, we return the login method of user available to us from the `Yii::$app`.

This is actually a reference the `User` model in `vendor\yiisoft\yii\webUser`. 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, but that is the nature of the beast. 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 attribute `$_user` defaults to `false`, the condition in the `if` statement is going to be met if this method has not already been run. So if there is no username in `$_user`, then it uses a static method of the `User` model to return a model instance of the user and set it to `$_user`.

In order for this to work, the `LoginForm` model obviously has to get the values for its attributes 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 it's 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 attributes, and the first method, rules.

Captcha

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 on every action, and we see here that it is actually called as a validation rule on the form model. Cool stuff.

If you need to implement captcha on anything else on your future applications, you can use this as an example of how to implement it. These would be the steps for implementing captcha:

- Configure into controller via actions method
- Include as validator on form model rules
- Include widget on view

I don't know of any other PHP framework that makes it that easy. 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 how we setup our application on init in setup, emails will be sent to frontend/runtime/mail. So you can try your contact form to see if puts an email there. Other than that, I don't really cover that subject in this book. But you can check the Yii 2 guide for configuration instructions:

[Yii 2 Email Guide](#)³⁰

And that's it for our ContactForm model. Let's return now to SiteController and it's 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,
    ]);
}
```

³⁰<http://www.yiiframework.com/doc-2.0/guide-tutorial-mailing.html>

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\widgets\ActiveForm;
use yii\captcha\Captcha;

/* @var $this yii\web\View */
/* @var $form yii\widgets\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>

```

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 \$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\widgets\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>',
])
?>

```



Tip

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. This is the implementation of the captcha that we spoke about earlier. It's just a standard implementation, it will work in any form as long as you have done the first two steps that we outlined earlier.

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>
</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 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();  
        $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.

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) {
        $user->generatePasswordResetToken();
        if ($user->save()) {

```



```

        return \Yii::$app->mailer->compose('passwordResetToken',
                                          ['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'.

The sendemail method looks up the user by email address to see if they are active. If we're good there, we call the generatePasswordResetToken. 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\InvalidParamException;
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 attributes. 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\InvalidParamException 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 InvalidParamException(
            'Password reset token cannot be blank.');
```

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.

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 more ground to cover. That was the frontend.

Backend Site Controller

We have a backend Site Controller in `backend\controllersSiteController.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' => ['get', 'post'],
            ],
        ],
    ];
}
```

We had to add to verbs, both get and post for logout. That seems to be a symptom of some buggy behavior, since we didn't have to do it for frontend.

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, 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.

Change 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`:

```

public function loginAdmin()
{
    if ((\Yii::$app->user->login($this->getUser(),
        $this->rememberMe ? 3600 * 24 * 30 : 0);

    } else {
        throw new NotFoundHttpException('You Shall Not Pass.');
```

So what we have done here is added more to the `if` statement. Now we want to validate the user, get the user as long as they have a `roleValue` of `Admin` or greater and a `status_id` of `Active`. 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 class, ValueHelpers came in very handy and with those methods, we did not need verbose code to get the values we wanted. And we have already reused this helper class multiple times.

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.

Since we added active status as a requirement for logging into the backend, it makes sense we would want the same requirement on the frontend login. So on the same LoginForm.php file, let's change the login method to the following:

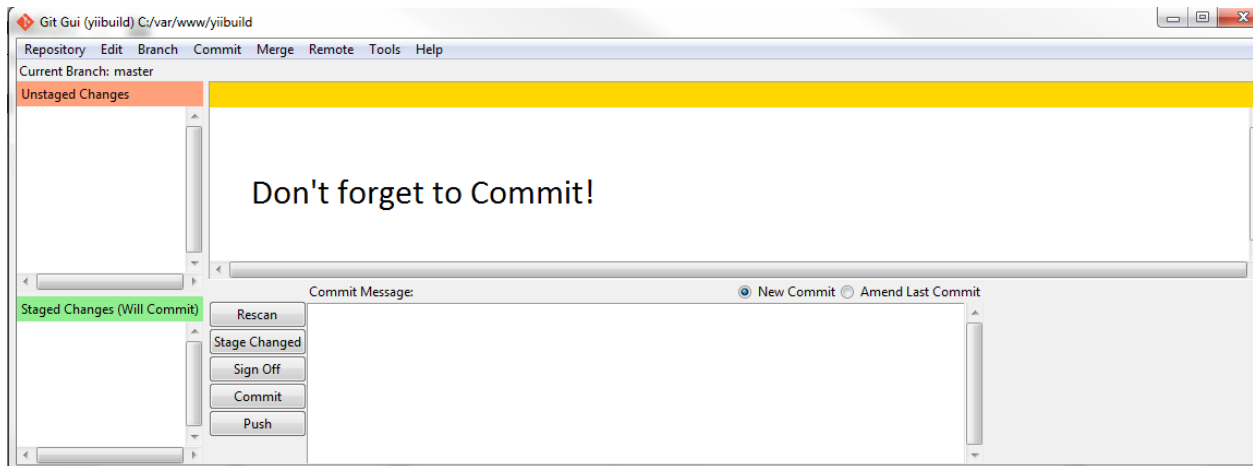
```
public function login()
{
    if ($this->validate() && $this->getUser()->status_id ==
        ValueHelpers::getStatusValue('Active')) {

        return Yii::$app->user->login($this->getUser(),
            $this->rememberMe ? 3600 * 24 * 30 : 0);

    } else {
        return false;
    }
}
```

Now users have to have an active status to login into either side of the site.

Summary



Commit!

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 try it. Let's navigate back to Gii:

yii2build.com/index.php?r=gii

The screenshot shows the 'CRUD Generator' configuration page in the Yii2 Code Generator. The sidebar on the left lists various generators, with 'CRUD Generator' selected. The main form contains the following fields and values:

- Model Class:** frontend\models\Profile
- Search Model Class:** frontend\models\search\ProfileSearch
- Controller Class:** frontend\controllers\ProfileController
- Base Controller Class:** yii\web\Controller
- Module ID:** [empty]
- Widget Used in Index Page:** GridView
- Enable I18N:**
- Code Template:** default (C:\var\www\yii2\build\vendor\yii2\soft\yii2-gii\generators\crud\default)

A 'Preview' button is located at the bottom of the form.

Gii CRUD

You can see that we need to namespace and follow the filename conventions.

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 namespaces are as follows:

Model Class is frontend\models\Profile

Search Class is frontend\models\search\ProfileSearch

Controller Class it is frontend\controllers\ProfileController

Gii should create the search folder under frontend\models, but I'm always in the habit of making sure it's there before I use Gii. So if you get an error, create that folder first.

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`

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 allow 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:

`yiiibuild.com/index.php?r=profile/index`

View

The view.php is a the details of an individual record and utilizes the DetailView widget. Once the file is made through Gii, you can reach this page at:

```
yiibuild.com/index.php?r=profile/view
```

Obviously since there are no profiles in the db, this doesn't do us much good yet.

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

Create Profile

You can feed in the user id manually if you wish to look up the user id via PhpMyadmin. If you want to test, using the existing form, leave `created_at` and `updated_at` blank, we know these will be filled in by behaviors. Also use, 2014-01-01 format for date, that is the only format that will validate. Pick a 1 or a 2 for gender, as long as you have created the gender records in a previous chapter. I wouldn't bother with it though, until we have made a few changes.

Update

Calling `update.php` renders the `_form.php` partial so you can input the needed data to update the record.

Once the file is made through Gii, you can reach this page at:

```
yiiibuild.com/index.php?r=profile/update
```

When you look at these view pages, you will notice that there are a lot of problems. Fields like `created_at` show up as text fields and gender returns a number.

Of course if you try these urls without having created a profile, you will get empty results. 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 boiler plate, 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 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:


```
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:

```
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' => ['@'],
    ],
],
],
];
}

```

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:

```
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'))
```

This may seem like a trivial 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.

Robert C. Martin wrote an excellent 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:

```
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:

```

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:

```

/**
 * 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:

```
public function actionDelete()
{
    $model = Profile::find()->where(['user_id' => Yii::$app->user->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 let's go through our views and fix this.

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 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;
```

`$this->params` comes from the view base class. When you render a view, in this case, `view.php`, you are calling an instance of `yii\web\View`, which extends the `yii\base\View` class, and through the magic of Yii 2's routing, making the object available as `$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:

```

<?php

use yii\helpers\Html;
use yii\widgets\DetailView;
use common\models\PermissionHelpers;

/**
 * @var yii\web\View $this
 * @var app\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 and User models in regards to attribute labels, so they display what we want correctly on the page. And since we’re taking a moment to update those models, we might as well add the relationships that we will need in the near future to the User model, while we are working on that file.

Gender

So let’s start with the simple one, `Gender.php` located in `frontend\models`.

We have one change only to the attribute labels:

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

should be changed to:

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

Once you save the change, hit refresh on `yiibuild.com/index.php?r=profile/view` and you will see that the label for gender has changed from Gender Name to Gender, which will make more sense to the user and is friendlier UI.

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 use a datepicker widget for the date. and create a dropdown list for Gender. Here is what the entire file should look like:

```

<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;
use yii\jui\DatePicker;
/**
 * @var yii\web\View $this
 * @var app\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/>

<?php echo $form->field($model, 'birthdate')->widget(DatePicker::className(),
    ['clientOptions' => ['dateFormat' => 'yy-mm-dd']]); ?>

<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. Also note the DatePicker widget configuration for the birthdate field.

DatePicker

Let's make sure we remembered to include the use statement:

```
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 yiibuild 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\yiibuild>composer update
```

This will bring in the yii2-jui dependencies. So now we are set to use it. If all went well, you should now see a difference on your update and create form from your Profile.

However, and I hate to throw in a wrinkle mid-development, but as of this writing, the DatePicker widget was not working properly. It's not that uncommon for you to face such a problem. The problem 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.

Now we could just have a regular text input field and stick in a note telling people what format we want, but that seems so subpar with the wonderful Bootstrap look and feel we have going, that I decided to work around the issue.

Basically, I need to figure out how to get the validation to pass. 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 Mysqlq is:

```
Y-m-d
```

This translates to 2014-10-14.

So obviously a conversion problem. I decided to solve it 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 my code:

```
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`. I actually wanted to use PHP's `DateTime` class, but Yii 2 would not instantiate it, so I used the older `strtotime` method.

Anyway, then 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.

This actually turned out to be a plus because now users can enter more valid date formats into the form and it will convert it before validating. The trade off is they can also enter gibberish, and it will get set to a date in 1970. But I can live with that trade off at this point since it's the most flexible option available to me, and I don't think everyone wants to be born in 1970. At some point I may write further validation rules to eliminate the gibberish, if possible.

Anyway, add the `beforeValidate` method above to your Profile model and it will solve the date problem and you can use the pretty jui datepicker.

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>
```

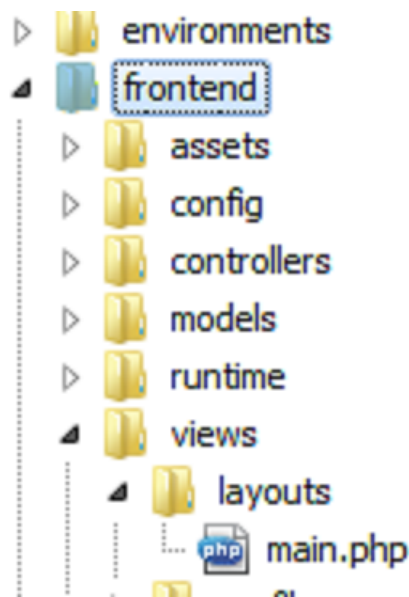
And just for consistency, let's take that 's' out of the word Profile on the view.php breadcrumbs as well:

```
$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`



`views\layouts\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']];
        }
    </?php
    </div>
</body>
</html>
```

```

    } 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. This way of referencing the app assets is not my favorite aspect of the framework, it seems overly complicated.

Yii 2 utilizes a publishing system to cache assets. I'm sure this serves a noble purpose, however it does make it harder to deal with structurally. 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:

```
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.

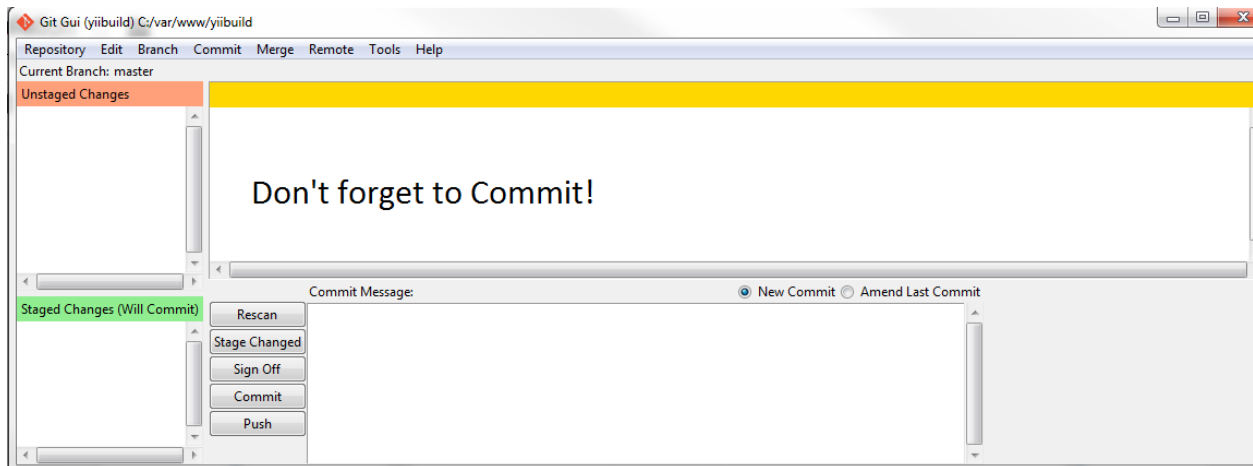
What we do want to test for however, is that the user is logged in. We do not want to show the profile link if the user is not logged in. Also note, we do not need to provide a get variable. We eliminated the need for that with the way we wrote the controller logic.

Insert the following block of code after the first `$menuItems` array:

```
if (!Yii::$app->user->isGuest) {
    $menuItems[] = ['label' => 'Profile', 'url' => ['/profile/view']];
}
```

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`.

Summary



Commit!

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 `PermissionHelpers::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.yiibuild.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. So in Controller ID, type in:

```
frontend/controllers/upgrade
```

Make sure the following fields are set if they do not auto-populate:

Action Ids: index

Controller Namespace: frontend\controllers

It should look like this:

Upgrade Controller in Gii

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](https://stripe.com/)³¹

Also, there is:

[PayPal](http://www.paypal.com)³²

You may want to offer both options.

³¹<https://stripe.com/>

³²<http://www.paypal.com>

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\controllersProfileController.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 we already added that requirement to the site controller, 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 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:

```
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 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 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:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => \yii\filters\AccessControl::className(),
            'only' => ['index'],
            'rules' => [
                [
                    'actions' => ['index'],
                    '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:

```

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. 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:

```
<?php
/* @var $this yii\web\View */
?>
<h1>Hey "<?php echo $name; ?>," This Requires Upgrade</h1>
```

```
<p>
    You can get the access you want by upgrading, but <?php echo $name; ?>,
    that's not all. You get to go everywhere, isn't that cool?
</p>
```

This is just goofy fun, but you get the point. So that's just a simple variable, let's try an object:

```
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 add a use statement to access Profile:

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

Then in our view, we can change it to:

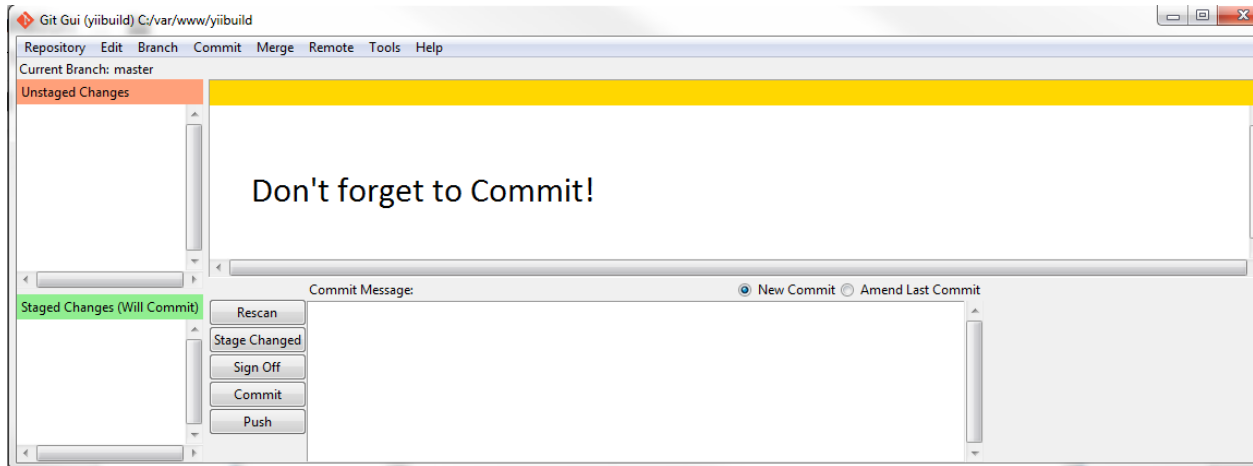
```
<?php
/* @var $this yii\web\View */
?>
<h1>Hey "<?php echo $name->first_name; ?>," This Requires Upgrade</h1>
```

```
<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:

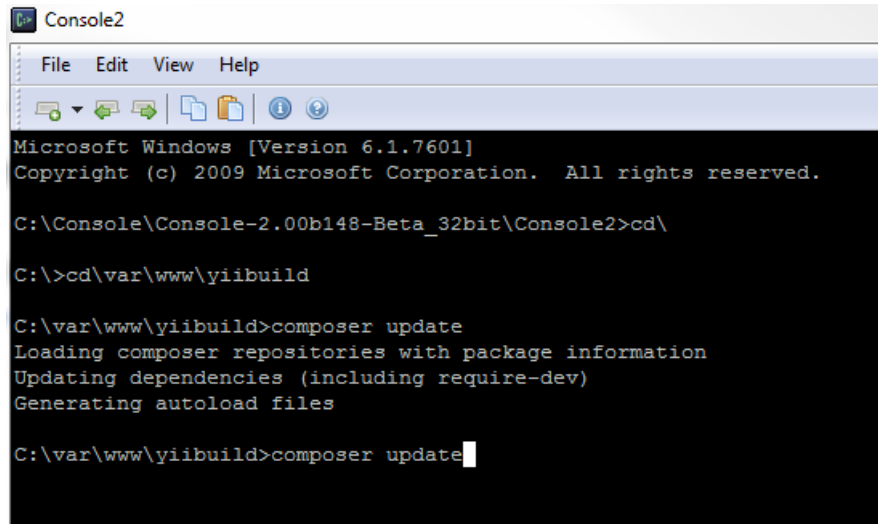
```
use \yii\bootstrap\Modal;  
use kartik\social\FacebookPlugin;  
use \yii\bootstrap\Collapse;  
use \yii\bootstrap\Alert;  
use yii\helpers\Html;
```

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"  
  },
```

You can see it on the last line. The run composer update from the command line, like we have done many times before.



```
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\yiibuild

C:\var\www\yiibuild>composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)
Generating autoload files

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

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](http://www.krajee.com)³³. 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 with:

³³<http://www.krajee.com>

```

<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 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.

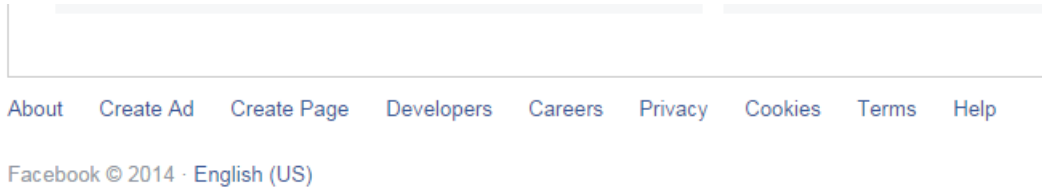
Then follow step by step for setting up an app. I will provide a lot of screenshots here for reference, but please keep in mind that things change over time, and it might not look the same. At any rate, it's fairly intuitive, so you should be able to figure it out.

Go to facebook and find your settings link from the down arrow all the way the right:



Facebook Nav

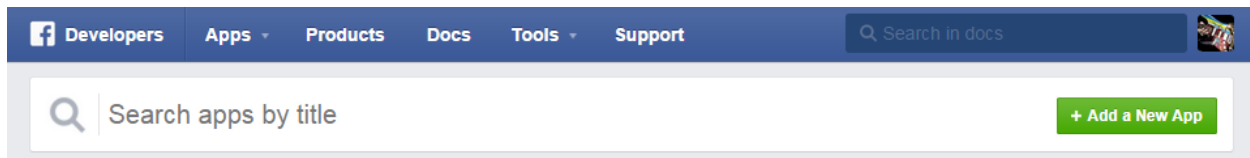
Once you are on your settings page, you can get to the footer and the developers link:



Facebook Footer

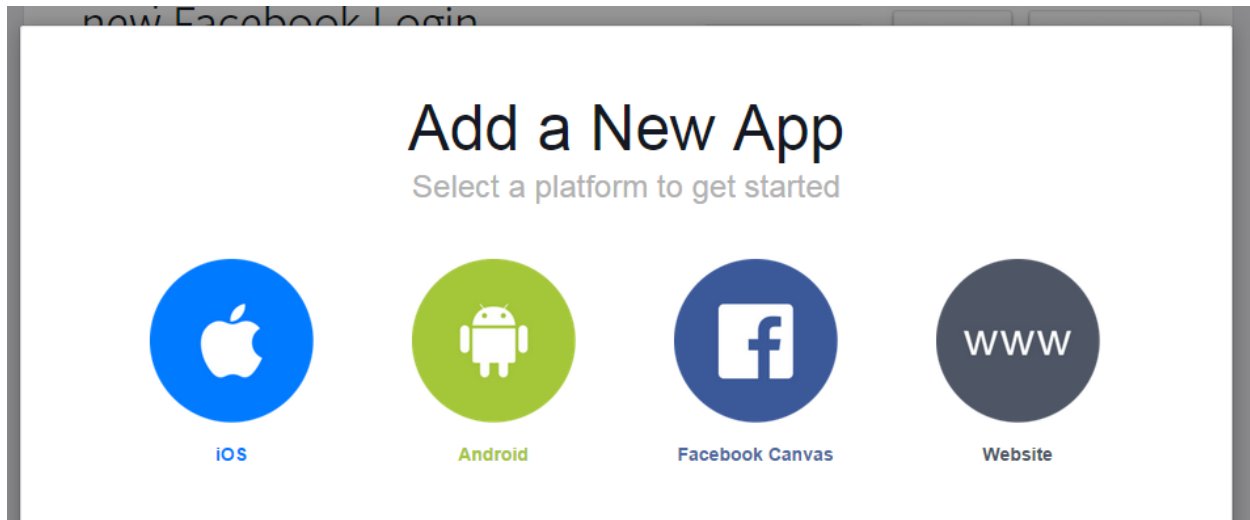
or you can just go to:

<https://developers.facebook.com/apps>³⁴



Facebook App Screen

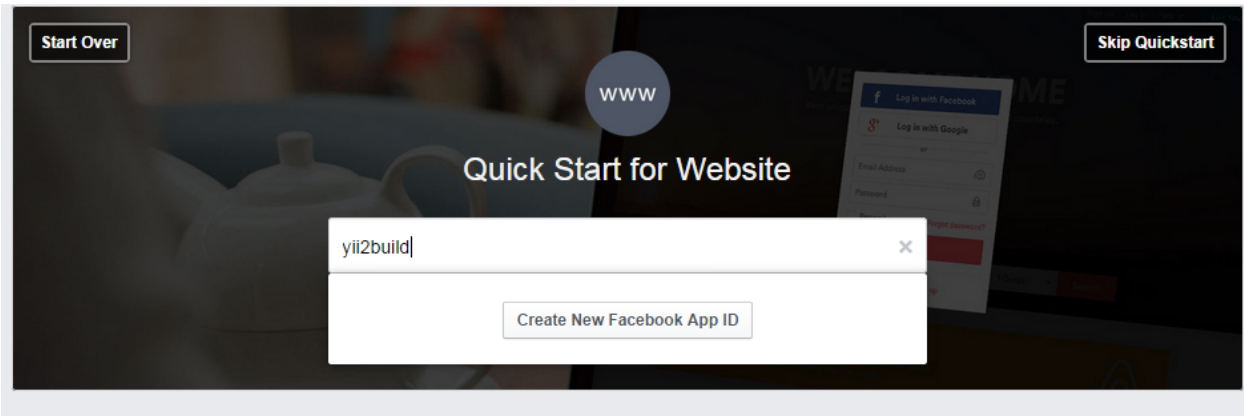
Select add new app:



Facebook Platform

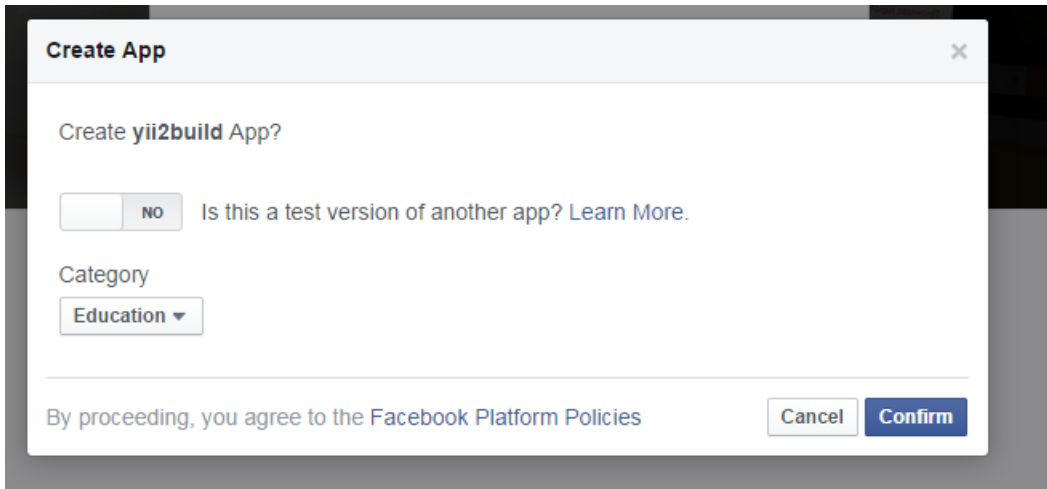
select website as platform and then choose name for app:

³⁴<https://developers.facebook.com/apps>



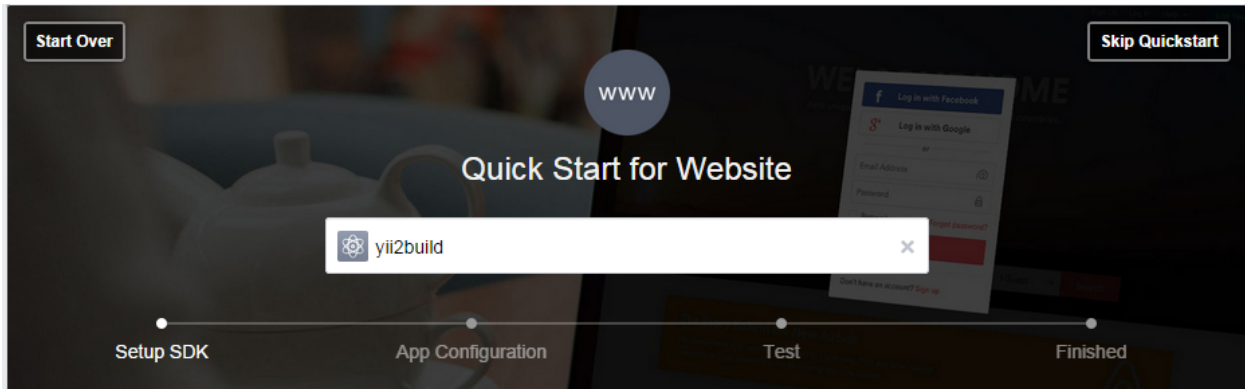
Facebook Select

Select category and confirm:



Facebook Confirm

You get the quick start screen:



Setup the Facebook SDK for JavaScript

The following snippet of code will give the basic version of the SDK where the options are set to their most common defaults. You should insert it directly after the opening `<body>` tag on each page you want to load it:

```
<script>
window.fbAsyncInit = function() {
  FB.init({

```

Facebook Quick Start

At the bottom of the quick start screen, fill in your site url:

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](#)

 Share Add the share dialog to let people share your content with their friends.	 Login Add Facebook Login to let people quickly and easily login to your app.	 Social Plugins See what your Facebook friends have liked, shared, or commented on across the Web.	 Ads Grow your app with Mobile App Ads.
--	---	--	---

Facebook Quick Start Form Finished

That will bring you to the Dashboard, where you click on settings:

The screenshot shows the Facebook App Dashboard for an application named 'yii 2 build'. The interface is divided into three main sections: Basic, Advanced, and Migrations. The Basic section contains fields for App ID (redacted), App Secret (masked with dots and a 'Show' button), Display Name (yii 2 build), Namespace, App Domains, and Contact Email. The Advanced section includes a 'Website' tab with fields for Site URL and Mobile Site URL, both containing 'http://www.yii2build.com/'. At the bottom of the dashboard, there are buttons for 'Delete App', 'Discard', and 'Save Changes'.

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 need to copy the app id and the app secret into the appropriate area in config as I describe below.

```
// 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:

```
<?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',
],

```

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 \socialModule, 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.

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 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:

```
<?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/">
    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 on 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.

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 to be perfectly honest, I don't understand why they made it that way. 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 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/font-awesome": "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 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:

```
<?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@jjmf.com>
 */

class FontAwesomeAsset extends AssetBundle
{
    # sourcePath points to the composer package.
```

```
public $sourcePath = '@vendor/fortawesome/font-awesome';

# CSS file to be loaded.
public $css = [
    'css/font-awesome.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',
    ];

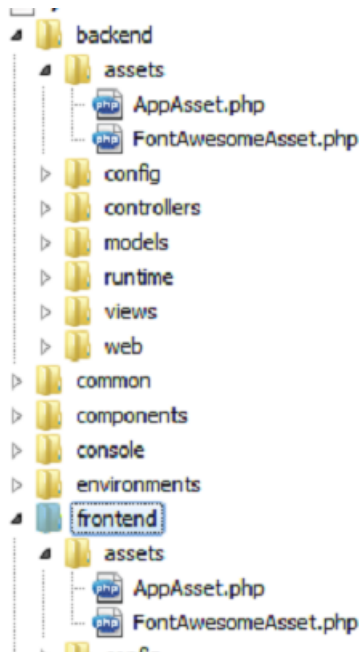
$pathItems = array_reverse(explode(DIRECTORY_SEPARATOR, $from));

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;
```



FontAwesomeAsset

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/>³⁵

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>³⁶

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:


³⁵<http://www.yiiframework.com/forum/index.php/topic/57902-using-fontawesome/>

³⁶<http://www.yiiframework.com/doc-2.0/guide-structure-assets.html>

Get Started Today

Yii 2 Build

Use this Yii 2 Template to start Projects.

 Like Be the first of your friends to like this.

Top Features

Top Resources


comments

Launch your project like a rocket... 

Free

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.


[Yii Documentation »](#)

 Like Be the first of your friends to like this.

Advantages

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!

[Yii Forum »](#)



Also post on Facebook

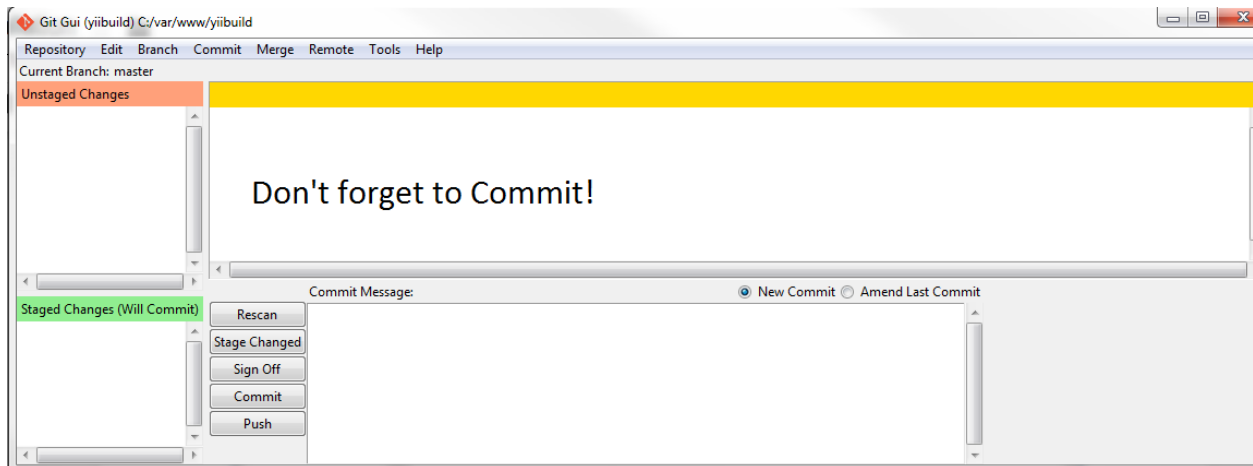
Code Quick, Code Right!

Leverage the power of the awesome Yii 2 framework with this enhanced template. Based Yii 2's advanced template, you get a full front end and backend implementation that features rich UI for backend management.

[Yii Extensions »](#)

Home Page

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>³⁷

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>³⁸

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>³⁹

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](http://fontawesome.io)⁴⁰

³⁷<http://www.krajee.com>

³⁸<http://www.yiiframework.com/doc-2.0/guide-widget-bootstrap.html>

³⁹<http://www.yiiframework.com/doc-2.0/guide-widget-jui.html>

⁴⁰<http://fontawesome.github.io/Font-Awesome/>

Chapter Eleven: Backend Creation

Ok, we're ready to move into creating our backend admin area. Let's start by using Gii to create controllers and CRUD views for all of our models. Now 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.yiibuild.com/?r=gii>

The user model namespace is as follows:

Model Generator >

CRUD Generator >

Controller Generator >

Form Generator >

Module Generator >

Extension Generator >

CRUD Generator

This generator generates a controller and views that implement CRUD (Create, Read, Update, Delete) operations for the specified data model.

Model Class

common\models\user

Search Model Class

backend\models\search\UserSearch

Controller Class

backend\controllers\UserController

Base Controller Class

yii\web\Controller

Module ID

[empty]

Widget Used in Index Page

GridView

Enable I18N

Code Template

default (C:\var\www\yii2\vendor\yii\soft\yii2-gii\generators\crud\default)

[Preview](#)

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 a namespace for a search model.

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.

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\ProfilerController

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

Ok, once you have generated the new files, I recommend saving your repository in Git. You should do this for each model, it will make it easier to step backward, should that become necessary. You can also overwrite easily with Gii, a mostly good thing, unless you do that by mistake, but it does tell you if you are going to overwrite and gives you the option not to.

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.

Ok, you can check the results individually by typing in a url, for example:

```
backend.yiibuild.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 backen.yii2build.com and the user logging in would have to have a role_id that matches role_value of 20, since we gave that value to role_name = Admin, and that is what we required when we created the seperate login for admin.

The user index page also has links to view and update 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 (don't repeat yourself). I used the frontend\models for Profile, but could have used common\models. I just chose the former because of work flow. 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. 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 value and the ability to change it, so we can grant additional privileges to users.

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:

```
<?php

use backend\assets\AppAsset;
use yii\helpers\Html;
use yii\bootstrap\Nav;
use yii\bootstrap\NavBar;
use yii\widgets\Breadcrumbs;
use common\models\ValueHelpers;
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

$is_admin = ValueHelpers::getRoleValue('Admin');

if (!Yii::$app->user->isGuest){

    NavBar::begin([

        'brandLabel' => 'Yii 2 Build <i class="fa fa-plug"></i> Admin',
        'brandUrl' => Yii::$app->homeUrl,
        'options' => [
            'class' => 'navbar-inverse navbar-fixed-top',
        ],
    ]);

} else {

    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::$app->user->isGuest
&& Yii::$app->user->identity->role_id >= $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']];

}

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\ValueHelpers;
```

We did that because we are going to use `ValueHelpers::getRoleValue('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::$app->user->isGuest
&& Yii::$app->user->identity->role_id >= $is_admin) {

    $menuItems[] = ['label' => 'Users', 'url' => ['user/index']];

}

```

The `$is_admin` variable holds the value of:

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

The reason why we take the extra step of setting up the `$is_admin` variable is that `getRoleValue()` performs a query, and even though it's a fast one, it wouldn't make sense to do a query each time we need this value. With a ValueHelper, it only returns the value we are looking for, so we need to also compare that value to the current user's `role_id`. We can access the current user's `role_id` by:

```
Yii::$app->user->identity->role_id
```

The logic behind this is if not a guest and the current user has `role_id` greater than or equal to `admin_value`, display the link. It's a very simple way to control access to the links.

```

if (!Yii::$app->user->isGuest
&& Yii::$app->user->identity->role_id >= $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 change the view.php for backend\views\profile\view.php. Here it is 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:text',
            'last_name:text',
            'birthdate',
            'gender_id',
            'created_at',
            'updated_at',
        ],
    ]) ?>
</div>

```

Let's change this to:

```

<?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>

```

Note that unlike when we used the ValueHelpers call for the value of admin, the PermissionHelpers method already compares the current user's role_id to the value of SuperUser, so we do't need to call it.

And finally, 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\modelsUser.php.

The method on the model:

```

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

```
<?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',
        // 'auth_key',
        // 'password_hash',
        // 'password_reset_token',
        'email:email',
        'roleName',
        'statusName',
        // lazyload 'status.status_name',

        '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',
        // lazyload 'status.status_name',

        '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:

```
<?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:

```
<?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 let's 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.

In the GridView widget, we left some attributes commented out for reference. You can see we added `userIdLink`, `userLink`, `ProfileLink`, `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.

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`:

```

<?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 '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:

```
use yii\helpers\Html;
use yii\widgets\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>

```

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:

```
<?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>
```

</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.

You'll also notice that we are getting Role name, instead of `role_id`. This is brought in through the relationships we created on the user model. But the problem is these too are lazy loaded, so we will have to modify our UserSearch model to account for these as well.

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 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->andWhere([
```

```

        'id' => $this->id,
        'role_id' => $this->role_id,
        'status_id' => $this->status_id,
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ]);

$query->andWhere(['like', 'username', $this->username])

->andWhere(['like', 'auth_key', $this->auth_key])
->andWhere(['like', 'password_hash', $this->password_hash])
->andWhere(['like', 'password_reset_token', $this->password_reset_token])
->andWhere(['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. Note: you might see a lower case u on user when it comes from Gii, probably a typo from Gii, but anyway, you'll see it uppercase in the code I've provided later.

Anyway, back to the search method on UserSearch. After setting the query to the model instance, we create an instance of Yii's CActiveDataProvider 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->andWhere([
    '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->andWhere(['like', 'username', $this->username])
->andWhere(['like', 'auth_key', $this->auth_key])
->andWhere(['like', 'password_hash', $this->password_hash])
->andWhere(['like', 'password_reset_token',
    $this->password_reset_token])
->andWhere(['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:

```
<?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
{

    /**
     * attributes
     *
     * @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, 'user.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->where('role.role_name LIKE "%' . $this->roleName . '%"');
}])

// filter by status

```

```

->joinWith(['status' => function ($q) {
    $q->where('status.status_name LIKE "%' . $this->statusName . '%"');
}])

    // filter by user type

->joinWith(['userType' => function ($q) {
    $q->where('user_type.user_type_name LIKE "%' . $this->userTypeName . '%"');
}]);

    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 inflection 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 its 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'
],

'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. 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, 'user.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, and on 'id', we specify the model to avoid ambiguity. 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 = 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]);
    }
}

```

The first part determines if there is a ‘.’ in the attribute. I added whitespace to make it easier to read:

```

if (($pos = strrpos($attribute, '.')) !== false) {

    $modelAttribute = substr($attribute, $pos + 1);

} else {

    $modelAttribute = $attribute;

}

```

If it 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.

In the case of the first attribute we hand in, there is a period, user.id. So this is telling the parameter that we want the id of the user model, which eliminates the ambiguity 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 `addSearchParameter` 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 ambiguity problems. Since this method runs one attribute at a time, we can safely assume that `$attribute` holds the string we intended.

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 `addSearchParameter` 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 `where` method to the model being joined:

```
// filter by role

$query->joinWith(['role' => function ($q) {
    $q->where('role.role_name LIKE "%' . $this->roleName . '%"');
}])

// filter by status

->joinWith(['status' => function ($q) {
    $q->where('status.status_name LIKE "%' . $this->statusName . '%"');
}])

// filter by user type

->joinWith(['userType' => function ($q) {
    $q->where('user_type.user_type_name LIKE "%' . $this->userTypeName . '%"');
}]);
```

Note that we stacked the `->joinWith` methods, but we have comments in between, be careful, the closing semicolon only comes at the very end. Also, note the quotation marks syntax is nested, be

careful to get it right. 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`:

```
'profileIdLink' => [
    'asc' => ['profile.id' => SORT_ASC],
    'desc' => ['Profile.id' => SORT_DESC],
    'label' => 'ID'
],
```

Let's take the entire `ProfileSearch.php` file and replace it with:

```
<?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, 'profile.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->where('gender.gender_name LIKE "%' . $this->genderName . '%"');
}])

    // filter by profile

    ->joinWith(['user' => function ($q) {
    $q->where('user.id LIKE "%' . $this->userId . '%"');
}]);

    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:

```
<?php

use yii\helpers\Html;
use common\models\ValueHelpers;

/**
 * @var yii\web\View $this
 */

$this->title = 'Admin Yii 2 Build';

$is_admin = ValueHelpers::getRoleValue('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::$app->user->isGuest
    &&
    Yii::$app->user->identity->role_id >=
    $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::$app->user->isGuest  
&& Yii::$app->user->identity->role_id >= $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 who is not. You can add a new role if you like, just click the link below to get started.

```
</p>
```

```
<p>
```

```
<?php
```

```
if (!Yii::$app->user->isGuest
    && Yii::$app->user->identity->role_id >= $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::$app->user->isGuest
    && Yii::$app->user->identity->role_id >= $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::$app->user->isGuest
    && Yii::$app->user->identity->role_id >= $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::$app->user->isGuest
    && Yii::$app->user->identity->role_id >= $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::$app->user->isGuest
    && Yii::$app->user->identity->role_id >= $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=ValueHelpers;
```

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::$app->user->isGuest
&& Yii::$app->user->identity->role_id >=$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 = ValueHelpers::getRoleValue('Admin');
```

As you recall, we are doing it this way because we only want one query to find the value of Admin, which is a more efficient use of resources.

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 with the following:

```

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' => ['get', 'post'],
    ],
],
];
}

```

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 someones 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:

```
public function behaviors()
{
    return [

        'access' => [
            'class' => \yii\filters\AccessControl::className(),
            'only' => ['index', 'view', 'create', 'update', 'delete'],
            'rules' => [
                [
                    'actions' => ['index', '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.

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 `PermissionHelpers` 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:

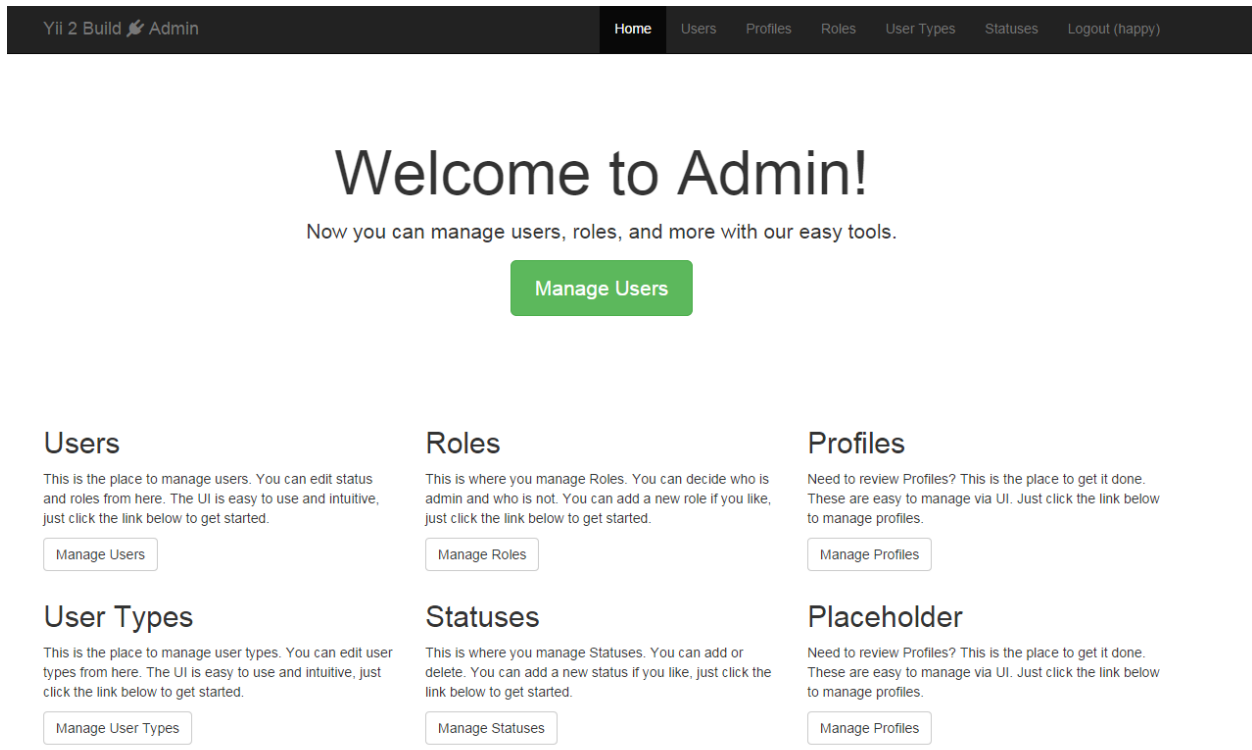
```
use common\models\PermissionHelpers;
```

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 value. Make sure you use the right operator, which is `==`, and not `=`, which would simply assign the value and not test for it, because it's a common programming mistake.

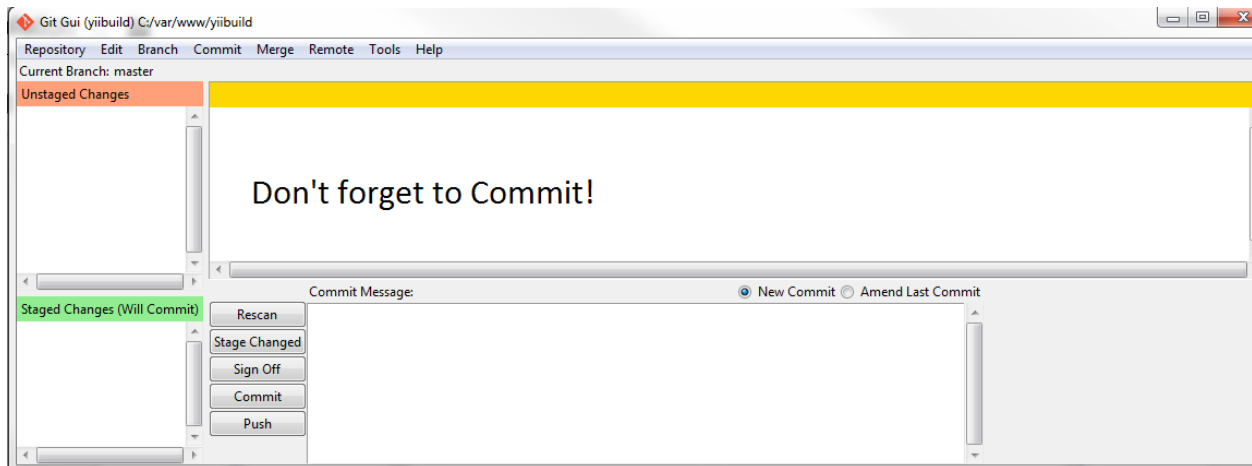
We were supposed to change the behaviors in the frontend in a previous chapter, so if you have not please go back and do so now.

Ok, finally we have our working back end. Now we can check out our new index page for admin when we are logged in:



Admin Homepage

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.

If you like this book, please recommend it to friends. You can visit my [blog⁴¹](#) and leave feedback and comments too, they 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.

⁴¹<http://yii2framework.wordpress.com>