# Testing Django Applications

by Jamie Counsell

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Related

# Course Overview

## Course Overview

Hi everyone, my name is Jamie Counsell, and welcome to my course, Testing Django applications. I'm a senior software developer at Conrad group, and I've been working with Django for many years. A good test plan can be the difference between a good Django site and a great one. It's easy to get started and can speed up development, save time, and reduce risk. This course is an introduction to testing your Django application. It'll cover all the tools you need to get started, as well as some tips and tricks to make testing easier and more effective. Some of the major topics that we will cover include, unit and integration testing, Python and Django testing features, third-party testing tools, and code coverage measurement. By the end of this course, you'll be able to begin writing tests for you Django-powered site. You'll have all the tools you need to build applications that are reliable and resilient. Before beginning the course, you should have some experience with Python and Django development. Come follow along, and we'll learn how to test your Django-powered site with the Testing Django Applications course at Pluralsight.

# Getting Started

## Setting Up

Hello, and welcome to my course on Testing Django Applications. I'm Jamie Counsell. I'm an avid Django developer, and I get excited about all things Django and Python. In this course, we're going to learn all about how to properly test our Django applications. In this first module, we're going to get started with some fundamentals. We're going to talk about virtualenv and how to get one set up. We'll talk a bit about some tools and resources as we go, and cover some dependencies and version information. Then, we'll start the installation of our app, which will include our database migrations and all of that. First, let's review the tools we'll be using in the course. We're going to be using Python 2. 7. 13, but any version of 2. 7 and above 2. 7. 10 is probably fine. Earlier versions of 2. 7 are likely fine as well, but I haven't tested it. The reality is that most of the course can likely be done in Python 3 as well, but there might be some small syntactic differences. We'll also be working with Django 1. 10, but the concepts and examples in these tests will work for any version of Django above 1. 8. At this time, Django 2 is still in alpha, so there might be some small changes when it's released. If you're new to Python and Django, you can just stick with Python 2. 7. 13 and Django 1. 10 to follow along. If you have some experience, these versions shouldn't make a big difference so long as all the ones you choose are recent. Most of our testing will be done with a few built-in tools including the Python unit test framework and Django's built-in test classes, which are largely extended from the until test module itself. We'll cover a few third-part tools like Ghost Inspector and Selenium, Which will help us enhance our test with powerful browser testing. We'll also be using virtualenv, which one of the most common and important tools in Python development. Let's take a moment to talk about virtualenv. Most people have used it before, but I think it helps to understand it on a bit more of a fundamental level. Virtualenv allows us to create multiple Python environments and switch between them. We can manage many conflicting versions of single Python packages, which allows us to seamlessly work on many projects that have different versions of dependencies. It also allows us to keep dependencies separate, so projects that don't require a specific dependency can run in an environment that doesn't have it. This is different than version management because it allows us to keep our environments clean, avoiding things like name conflicts that can arise with complex projects with many unnecessary dependencies. Most importantly, everyone needs to understand that virtualenv is not magic. In fact, it's not even doing anything that impressive. All it does is create a folder somewhere on your computer for each environment. Each folder holds the entire Python framework including Python itself, all the standard libraries, and any packages that you install into that environment. Then, virtualenv provides a little command line interface that allows us to easily switch between those environments, and that's it. Understanding how it works helps us use it in a more meaningful way, and can also help when we run into a problem.

## Creating Your Environment

Let's get our project set up. Make sure you have the source code downloaded from the course files. We'll install some dependencies, then we're going to migrate our database to make sure it's up to date, create our very first superuser, and then start up the server and use that user to log in and take a look around. Alright, I've got the project here that you can get from the course files. I'll be using the Atom text editor with the PlatformIO IDE terminal plugin. The text editor you choose doesn't matter for this course, so feel free to use whatever you're comfortable with. I'm also using virtualenvwrapper, which is just an extension of virtualenv that lets us avoid dealing with the folders that we generated for the environments by hiding them in a hidden folder inside my home directory. You can read more about virtualenvwrapper here at the Read the Docs, or you can just use virtualenv without the wrapper. I'm going to create an environment here called todo. I normally follow the convention of giving the environment name exactly the same name as my project name. It makes it easier to remember what the environment for our project is called. Once it's created, I can deactivate to leave, or use the workon command to switch back to it. When using virtualenvwrapper, this works from anywhere on the file system, which is another advantage over just using plain virtualenv. Now that the environment's created, I'll install the requirements. All the requirements for this project are contained in requirements. txt inside the course files. This name is purely conventional as we specify the file to install using the -r flag when running pip install, like this. We'll run that command and install our dependencies. Take some time to look through the requirements files to see what versions are being included. Next, we'll migrate our database. Currently, the project is set up to use SQLight. If you'd like to test any advanced database features, you'll need to configure a database that supports those features. With the database up to date, we'll create our first user. I'll keep Pluralsight as a Username, and leave the email blank for now. Now, let's start up the server and open our app in Chrome. The app is running on local host at port 8000. Here's our homepage, and we can use it to log in and check that everything's working. Looks good. We can also open the admin panel to get a better look at the data inside our project. We don't have any yet, but we can see our task form here, which can be used to create test data for fixtures later on. The Django admin panel is perhaps one of the best features for developers because it makes getting data into our development environment much easier than frameworks without a built-in backend.

## Summary

Now that we're set up, we're ready to start testing. We've covered the tools we're using and got our project dependencies and environment installed. With the app up and running, we can log in and use it. Not it's time to start testing. Let's go.

# Testing Concepts

## What You Should Test

Before we actually write any tests, we'll need to consider a few things. We need to understand why and when we should write certain tests, and exactly what needs to be tested. We'll be covering two main types of tests in this course, unit tests and integration tests. When writing tests, it helps a lot to have some good goals set out. In our case, we'll be writing tests to improve the consistency of our code, the robustness of our application and its ability to handle behavior of end users, and of course the quality of code over time as it's developed in a agile way. Though we're going to be focusing on unit and integration tests, it's important to note that there are many different types of tests. There's UI tests, functional tests, regression tests, load tests, security tests, compatibility tests, and more. These tests can be very valuable, but they're usually run at a much higher level than Django application itself, so we'll stick to unit and integration tests in this course. We need to understand what parts of our code we need to test. For Django, we're looking to cover a few things. First, we need to cover custom views and extended view logic. This includes custom function and class-based views, as well as any methods that we override from default generic Django views. We also need to test our models and their methods. This can be done using both unit and integration tests, each which serve a different purpose, but we'll cover that later. We'll cover custom helpers and middleware. This includes small helper methods, or Django implementations of things like password validators and context processors, and lastly, we'll test the results of template rendering with a special type of integration test. It is equally important to understand what we should avoid testing. Since Django is so highly generic and pragmatic, there's a lot of files, functions, and classes that simply won't benefit from our tests. We should generally avoid testing things like built-in Django views, at least those which are not customized, files such as urls. py or settings. py, the built-in Django admin panel, and any classes that are based mostly off highly generic built-ins. Let's take a look at some examples. We've got a few pieces of code here. The first is an example of a built-in generic Django view. A Legal page inheriting from Django's TemplateView. Though we have a single line of custom code here, we shouldn't test this. This is because the parameter we're setting is part of the Django framework, and we should trust that it behaves as the documentation claims, at least in terms of unit and integration tests. Next we have a custom helper method. This is a great candidate for our unit test because it has no dependencies inside the function that would need to be mocked out, and it has a clear set of inputs and outputs. Lastly, we have a small function-base view. If you have experience testing with other frameworks, this might look like another candidate for a unit test. We would mock out the post-model manager and then use that to write our unit test. But in Django, cases like this benefit a bit more from an integration test. Since Django makes it so easy to set up a test database and gives us things like setup and teardown helpers, it's often better to run tests like this against an actual database. It writes and behaves just like a unit test, but technically falls under the integration test category.

## Reviewing Your Application

It's time for a demo. Before we dive into writing our tests, we'll need a better understanding of the application we're working with. The application provided is a simple to do application with some basic features. We're going to walk through this application. We'll create a user, preview the features of the app, and identify any features that should be tested, then we'll take a quick walk through the code to identify some targets for our test and get a sense for which parts of the application are handled by Django, and which are our responsibility. Let's take a tour of our application as a user. First, we'll start up our server by running manage. py and using the command runserver. This will start a server on localhost at port 8000. We'll visit that in our browser. The first page we're presented with is the login page, but since we don't have an account yet, we'll need to sign up first. Let's click Sign Up. The form for sign up is pretty simple. Just enter a Username and a Password, then confirm that Password. Once we're registered, we'll automatically be logged in and taken to the task list. Right now, we have no tasks. Let's create one by clicking on the button with the plus symbol. A to do task in our app has a required Title, as well as an optional Description and Due date. We'll ender a Description and Due date so that we can use all of the features. (working) Once created, we'll return back to our list where we can see the task. We can click the checkmark to check the tasks as done. This is indicated by the checkbox turning red. Once it's marked as complete, we can undo this action by clicking it again. Doing so will put it back in the incomplete state, and the checkmark will go back to blue. We can edit our tasks as well. Just click on the task and we'll see a similar form to the one we used to create it. We can modify any of its values, and then click save to update the task. Again, we'll return to the task list after. Lastly, we can delete a task. Again, open the task by clicking the title, then click the small red X at the top right of the task card. The task will be deleted, and we'll see it's been removed from our list. Let's go back to our registration form. We'll Logout, then click Sign Up again. Let's go to create a new account. This time, the password I've chosen is plural. todo. We'll see an error that states that this word, as well as some others, is not allowed in our password. This is a custom password validator. Password validators were added in recent versions of Django and allows us to build simple, pluggable password validators to increase the security of user accounts. Removing this word and selecting a new password will allow us to continue. Let's take a look at our code. First, let's look at our model. Our primary model is our Task model. We've created a small custom model manager here to make sure that our tasks always come back ordered properly, sorted first by time completed, and then by due date. If you haven't worked with custom model managers before, don't worry, we only use this simple one here. That said, they are perhaps one of the most useful features of Django models. Our task object has a few properties. The owner, or the user who created it is the only person that can edit or delete it. We have our title, description, due_date, as well as a complete_time. Instead of storing a Boolean to check if our task is complete, we store a time. To check if it's complete, we just check that the time is not null. This gives us a bit more context. We've got a few custom properties too. The is_complete property will check if the task has been completed. Our due_soon property checks that the task is due within two days. Mark_complete will mark the task as complete by setting the current timestamp as its complete time, and mark_incomplete will remove that timestamp, causing the task to report as incomplete. These short descriptive methods are great candidate for unit tests; however, they are interestingly also candidates for integration tests since we'll need to understand how these properties behave with an object loaded from the database. Next, we've got our custom password validator. This is a simple object with only a few methods. We've got a few views in this application, most are inside the tasks app, but some are not related to tasks, so they're been moved into our main package. Some are testable, while others are largely based off Django views. Our Home view will redirect the user to either log in, or view their tasks if they're already logged in. This has a bit of customization. Our Registration view is fairly standard, but includes some extremely important logic that can impact security, so we'll need to test this completely. Note that our login and logout views are both built in, so we don't need to test those. The views in our tasks app really leverage Django's built-in views. Our TaskListView defines only a query set and some extra context, but perfectly handles get requests and loads a template. We only need to test the methods that we've developed ourselves. The new task view is similar. It's a Django CreateView with a customized form_valid method. We should only test this method. The DeleteTaskView is a great example of a view that we should not test. We are simply setting two attributes, the rest is handled by Django and a custom mixin. Our last two views are TaskView, which allows us to view and edit a task, and our view to toggle task completeness. Both have some testable pieces. Note the difference between class-based views and function-based views. In general, function-based views are much more testable because they consist of largely custom code. Class-based views are generally less testable because a lot of their logic is built into the framework. If we don't touch it, we don't test it. We have a few other things as well. There are two mixins, one to retrieve the success URL in a reusable way, and another to check that the task is owned by the current logged-in user before performing an action. Mixins are a great feature to the Python language, and they can really help us separate our concerns to create more easily testable code. Our TaskForm is a simple model form. It should not be tested for the same reasons as our delete view. Now, our templates are important, but they're very difficult to test. When we talk about integration tests, we'll talk about how we can test them. For now, let's just walk through what each one does. Our task template is our create an update form. Our task_list template shows us a list of tasks represented by another template, our task_item template. We've got some other templates including our base template, which loads our dependencies and creates an empty content block, it is extended by all other views. We also have a login view and a register view. these are quite similar and could probably be just one view, but we're going to leave them separate to give us a little bit more template testing options for later. Lastly, we have our urls. py and settings. py files. As I mentioned before, these files should probably not be tested, as they're simply declaring static values. That said, if there's something such as an important URL pattern, or an important setting, you could always create a unit test to validate its presence to stop other developers from changing it.

## Testing and Django

As I mentioned before, we're focusing on two types of tests for this course. Unit tests are small tests that are intended to isolate pieces of our code and test them at their most primitive level. Integration tests are meant to test the orchestration of many layers of our application, such as a database or web server. Let's talk about unit tests. Some great candidates for unit tests are smaller methods with easily defined roles and functions, such as any handlers, helpers, or model methods that we write. Some overridden view methods benefit from unit tests as well. Common pitfalls with unit tests involve over-testing, which is usually a result of bad planning or bad code, failure to consider and test real-world applications, that is, those outside an isolated and mocked-out environment, and tests which are written for the sole purpose of increasing our line coverage. Integration tests are different. Good candidates for those are full views, admin actions, or model behavior both in our code and with our database, as well as larger methods and methods with database transactions that can be difficult to unit test. Some common pitfalls with integration tests include a failure to test edge cases, writing tests that are too large, diluted their role in our test suite, or writing tests that rely on unreliable infrastructure, such as HTTP, APIs, or other third-party services that we don't control. The last thing we need to do is set out some reasonable metrics for ourselves. We need some that are easily measurable, but the main goal is for us to be able to check whether or not our tests are serving their purpose. Every project is different, but let's focus on a few things as we move along. Let's make efficient tests. These should be short, concise, and to the point. We want to avoid writing dramatically large amounts of code for small things, to make it easier to update tests and save time. We also want tests that respond change. Tests that never fail don't help us all that much. This is why we should avoid testing Django's features and focus on our core custom business logic. Lastly, for a more measurable metric, we'll be looking at code coverage. All too often, teams writing tests are too concerned with this metric, so we'll use it in combination with efficiency and responsiveness to keep us on the right track. Django is a little different than other frameworks when it comes to testing, so let's address some of these differences up front. Django's core principles make the framework great for building robust applications quickly, but they encourage different testing approaches. For example, Django's pragmatic design favors integration tests over unit tests, and code coverage is often not seen as the most important metric, in contrast to many other web frameworks. Each framework has its own requirement, so if you've worked with other frameworks before, use your knowledge of them as a tool, but remain open to Django's difference. Embrace Django's core principles, and always keep our goals and metrics in mind. We're about to move into writing our first test in Django. Along the way, we'll keep in mind that testing is most useful to us when we understand the requirements, use the right tests, focus on the application's critical points first, constantly improve, and always keep our goals and metrics in mind.

# Unit Testing in Django

## Unit Test Principles

Let's talk about unit testing in Django. We'll start by reviewing some unit test best practices, and then get to work writing our first unit test. Django comes with a few different options for unit testing. The built-in unit test library comes with a TestCase class, which Django has extended into its own SimpleTestCase class. We'll put both of these to good use. As we write our test, remember a few things. Our test should be concise, always with a clear purpose to keep them short and readable. It's also specifically important for unit tests to be complete as they're going to be the primary force behind testing our most important business logic. The way that Django discovers tests is important. Originally, the tests came from a single test. py file. It wasn't incredibly inconvenient, as test classes can be imported into it, but it was a little tougher to organize our tests in a meaningful way. Now Django will look inside each app in our installed apps for any files that match a specific pattern. That is, files that begin with the word test and end in the. py extension. This allows us to split our test into many files. I find it easiest to organize tests by creating a new module in each project called tests. Don't forget the __init__. py file here, otherwise Django will not recognize this directory as a valid module. As we go, we'll start to end up with a lot of tests, and as we do, it's important to focus on keeping things organized. This will vary from project to project, and the methods used in this course are by no means more correct than other ways, so we just need some general rules to guide us along. First, we need to create a common structure for our files. It doesn't matter what system you use so long as you stick with it. For this course, we'll be grouping tests the same was that Django groups files. Django has files like models, views, and forms for each app. Likewise, we will have test models, test views, and test forms files. On a similar note, we just need to make sure that we're grouping similar tests. This does not mean that we need to group tests by type, in fact, it's quite the opposite. Having unit and integration tests in one file can be very legible, so long as they test the same code. If our files grow too large, we can continue to break it into more files. We just need to make sure we follow the same process across our project. Let's look at a few primitive examples of test cases. In the first example, we'll see Python's built-in TestCase class. Our method is quite simple. It's an add_x function that adds X to some base number. We'll name our test AdditionTest. Also common is the naming convention AdditionTestCase, which we'll see later. Our methods, like our files, begin with the word test for Django to pick it up as a test. Our test method should describe exactly what we're testing. These names will be longer than normal methods, but if they're too long, it could mean that our tests cover too much at once. Here we start with setting up our test. We pick our base and X parameters, and then predetermine the expected value. We then get a result by calling the method and assert that that result is equal to the expected. We could also use the assert equal method here. This is just an example of one of a few cases we would need test. and it tests a largely general case. We would need to continue testing things like negative values, zero values, non-integer values, or even non-number values. Next, we have an example of Django's SimpleTestCase. There are very few differences between SimpleTestCase and unit tests TestCase, but they can be useful when needed. Django's documentation for SimpleTestCase has details on all the differences. We'll see here that our test works the same way as the built-in test case class, but we're using the assertRaises method instead. AssertRaises allows us to assert that a certain callable raises an exception of a desired type. We pass the exception class, the callable, and then its parameters. Both TestCase and SimpleTestCase have access to this method, but SimpleTestCase gives us some extra features that we'll see later. So what exactly is the difference between TestCase and SimpleTestCase? Well as I mentioned, the documentation goes into greater detail, but the differences are somewhat minute. Here's a short breakdown of those differences. The built-in TestCase has no dependencies. It's great for testing Python-only code, and it's quite fast, but it lacks some of the additional features of SimpleTestCase. SimpleTestCase extends TestCase and provides the ability to test template rendering, test that assertions were raised, and that the assertion contains a specific message, and a few other things like HTTP responses, redirects, as well as JSON, XML, or HTML output. That said, we should notice that some of these start to fall into the integration test category, especially those that deal with HTTP requests and their responses. So we'll see SimpleTestCase come up both in unit and integration testing. To run tests, we have a lot of options, but it's actually quite simple. The manage command test takes a few parameters, but the first is an optional name of the module we'd like to test. Following normal Python conventions, we can test an entire app, or just the tests in the test module. We can also test a specific file, class, or even just one test from a specific class. This gives us control over which test we run, and which ones we don't.

## Unit Testing Models

Let's do a demo. In this demo, we're going to identify some good candidates for unit tests, create our test file structure, and write a few unit tests using unittest. TestCase. We'll also see one example that uses Django's SimpleTestCase, but as we discussed, they're quite similar. First, we'll create a new folder in our tasks app called tests. Inside, we'll create a new file called init. py, which will tell Python that this is to be interpreted as a module. Then we'll create our first test file, test_models. py. In test_models. py, we'll start by importing the tools we need. TestCase, from the unit test module, our Task model itself, and lastly, Django's timezone utility. The timezone utility is essentially a wrapper around the Python date time module, but it adds support for dealing with timezones, something that most developers have a familiar fear of. We'll set up our test class. We'll call it TaskModelTestCase. Our first method will test if a model that should be complete is actually complete. Throughout our test writing, I'll include the code we're testing in the top right of the screen. A model is complete when it's complete time property is set to sometime in the past. We'll then set its complete time to be one day before this moment. And assert that the task is in fact complete. Note that we did not save the task model, as this is strictly unit test. It's testing how the task class performs as a Python class, not necessarily a Django model. We can write integration tests later that cover the behavior of the target tasks if they were first loaded from the database. Next, as is common with unit tests, we will test the opposite. We'll test the model that should be incomplete is in fact incomplete. A model is incomplete when its complete time property is none. Note that we use the assertFalse method here instead of something like assertTrue followed by a not expression. This keeps our code readable. Now let's test some edge cases. What happens if the complete time is set to sometime in the future? We ought to assert that it is not complete, since a task complete tomorrow is presently not complete. Let's test that. Again, we'll set up our Task, set its complete_time property to something interesting, and then assert the state we expect it to be in. Before we continue, let's run these tests. We'll open a terminal, activate our virtual environment, and then run manage. py test. Uh oh, an error. This particular error is quite common and has stumped me a few times. It happens when we have a module called tests, but have also left in Django's default tests. py file. So let's delete tests. py and run the test again. Great, 3 passing tests. We're well on our way to a completely tested application. So let's write a few more. The next few tests we're going to write are going to be testing the due soon method of our task model. This returns true if the task is due within two days, and false otherwise. Currently, this method isn't actually used in the application, but it's important that we test it before we implement it later. Perhaps we'll add a feature that highlights tasks in our list when the task is due soon. The first test here tests that a model that should be due soon, is. We'll test this by setting its due_date to just one day from today, and then asserting that it is in fact due_soon. Next, we'll repeat this process, but instead, we'll set the due_date to 3 days in the future, and then assert that the model is not due soon because it's not due within two days. Now what happens if a model does not have a due date? It should clearly be designed that a model with no due date is not due soon. So let's test that particular case. Note that there are a few interesting cases that remain untested. The first is what happens if an object with a due date in the past is checked. It really should be due soon. If we've said our task was due yesterday, we'd better get moving. Due to the similarity of this test to the previous test, there's not a lot of added value here. Though for us it seems that one day in the future and yesterday are vastly different, the objects that represent that time are not. It's just a date, and that date is in fact before the due date. The business logic we've defined is that a task is due soon when the due date is less than two days in the future, so that's all we need to test. The other interesting case that we've left out is what happens at the exact moment that a task is due, this was left out for two reasons. First, we haven't really defined the correct state for this, and it likely doesn't matter. Second, we would need to understand the implementation of the timezone module to understand what the outcome would be, since the results could differ, or even become unreliable based on the accuracy of the timezone module. Next up, we'll test some methods that mutate our task. The first is mark_complete. There's not a lot of ways to test this, In true black box unit testing, we might try a whole host of strange things, but the reality is that we likely will not derive much value from that. Let's just test that a model that is incomplete is complete after the mark_complete method is called. We'll do the same thing for mark_incomplete now. Let's run our test. We'll open the terminal, activate the environment, and run our test command. We can also run just the test in the tasks app by specifying the application name after the test command. We can even specify that we just want to run the tests module inside our tasks app. Now, let's do it again, but run it only in the test_models file. The same result, but that's to be expected. What about just our TaskModelTestCase class? Again, this is our only class, so the result is the same. To really see much of a difference, we'll need to write some more tests. So let's test our mixins. We'll create our test_mixins file, we'll then start again by importing the TestCase class, our target class, and the Django reverse function, which we'll need. The TestCase class provides us a setup method, which can be overridden to add some additional logic to run before each test. Testing mixins is an interesting process in Python. Since they're never meant to be used on their own, we should probably implement it. To do this, we'll create a new class called MyClass that extends the mixin. We don't need to add anything to this class, but it is now a concrete class we can use, and we'll set up a target as an instance of that class. This mixin as it stands is somewhat primitive and doesn't benefit all that much from our tests since our current implementation is doing almost exactly what the method is. However, getting the URL like this is definitely something we might change as we move on. So we should probably make sure it's covered to prevent changes without consideration. We'll set up our test by determining which URL we expect. We'll then assert that both are equal. Again, we'll run our test, and we'll repeat this process often. Since unit tests are somewhat repetitive, it's especially important to run them frequently while we're writing. Often, we'll make a mistake in our implementation, and then repeat it through a few more tests before even realizing. Running often prevents this. Obviously, our mixins test passes. We can run it on its own by specifying exactly which file to run.

## Other Unit Tests

We've completed unit testing for our models and our mixins, but there's a few things left to test. We're going to cover a simple test that we can use to protect various features of our application. Since Django is highly conventional, many third-party applications are installed simply by adding a few definitions to the settings. py file. Sometimes, you'll want to make sure that these settings are never modified without consideration by other developers. When we say without consideration, we're primarily talking about changes made without thought about what other services could be affected. If a change is made, the test will fail, prompting some consideration from the developer. We can add some comments to our test to justify the protection for a particular setting. Let's create a new folder in our main module called tests. Note that this will require us to add todo to the list of installed apps, something that is not done by default, but generally has not side effects. Inside, we'll create an __init__. py file to make sure it's recognized as a valid module. We'll now create a test_settings file that will hold the test for our settings. We'll import our unit test class, as well as our Django settings module. Always remember that you cannot import settings directly. It must be imported from the Django conf namespace. We'll create a new class called SettingsTestCase. We'll write a simple test to make sure that the default timezone is never changed. In our application, users may experience strange behavior if someone updated the default timezone depending on our current data, and the service configuration. We'll justify in our intended setting, which is UTC, and then assert that the current setting is equal to that value. Let's run that test. Looks great. Now no one will accidentally change the timezone. Next, let's test our password validator. We'll start by creating a file to test our validators. Then, we'll import the required dependencies. Note that we're using Django's SimpleTestCase this time. The validator we're using is simple, but it is a great example of one of many little handlers we can provide Django with. Others include things like context processors, middleware, or template renderers. Django makes it easy to build generic classes to handle such things, and they're generally good candidates for unit tests. This validator raises a validation error when a certain criteria fails. The first thing we need is to make sure that an otherwise valid password does not raise a validation error. We'll write a simple test to confirm that. We'll define a valid password, and then fail if a validation error is thrown. There are a few things to note here. This password must only pass our validator, and could potentially fail another validator's check. Let's keep it this way. Also, note how we're handling expected exceptions. When an exception is expected, some people may just leave the method as is, allowing the exception to bubble up and cause the test to fail. There are two problems with this. There's a technical difference between tests that fail and tests that throw an exception. Also, this allows us to only fail officially on the exception we expect, all other exceptions will bubble up normally. The fail method on the SimpleTestCase class allows us to define our own failing message. For our next test, we'll ensure that a password that should fail the check, does raise a validation error. We'll configure this by grabbing one of the disallowed passwords right off the class itself. It's reasonable to assume that the list of disallowed words will always contain at least one element. We'll set up our test and assert that a validation error is raised when this password is checked. Lastly, we'll cover an example of how to check that a certain message was supplied, as this message will be returned directly to the end user. This makes the message specifically important compared to other exception messages. Let's define the message we expect to see if a password is deemed invalid by this validator. The assertRaisesMessage method is one of the methods provided especially by the SimpleTestCase class, and allows us to check that this specific message was raised during an exception. Note that unlike other assertion methods, this method returns a context handler, so we'll use it along side Python's with keyword. Though this message is a little unreliable, since disallowed words may change often, this example serves to demonstrate the usefulness of this method, which is most useful when testing messages that are eventually returned to the frontend. Again, let's run our tests. we'll just run the test we're interested in for now. Looks good. Now let's run all of our tests together. Awesome, 13 passing tests. Note that Django prints a dot for each passing test. Tests that fail will display an F instead.

## Django's Test Classes

We've seen a few of the possible test cases that we can use when testing Django. We'll see a few more as we go, but it's important to get an idea of what's available. Everything is based in some way on unit tests TestCase class. Along the way, the Django team has added several features to this class by extending it several times. The simplest is, appropriately, the SimpleTestCase class. Above that is the TransactionTestCase, which provides support for database transactions and fixtures. We also have the Django TestCase, which is the most common test class for integration tests and wraps the entire test in two atomic blocks, one for the entire class, and one for each test. It also provides the ability to check deferrable database constraints after tests run, and provides a setup test data method to help us set up our data. The last class is the LiveServerTestCast, which is basically the same as TransactionTestCase, but actually runs a live, running server that binds to a valid TCP port, just like the manage runserver command does. We've now learned the basics for unit testing in Django. We've talked about how to organize tests and how their discovered, we've written tests using the standard Python unit test library, as well as with the Django TestCase. We've covered how to run our tests and control which tests run at which time, as we've learned a bit about what options are out there when selecting the correct class to test our application.

# Integration Testing in Django

## Integration Testing in Django

Let's take a look at integration testing in Django. Integration tests are much more common in Django than unit tests, and we'll see why shortly. We'll cover a few best practices for integrations tests, and then we'll write some tests for our to do application. We'll use three classes from the Django test library, the TransactionTestCase, the TestCase, and then the LiveServerTestCase. We need to keep a few things in mind as we design our tests, the test should be value driven. It's very easy to create tests that cover too much ground and don't really test any specific concept themselves. We also want to make sure that our tests are reproducible, perhaps the most common area of failure for integration tests, as well as one of the strongest cases for unit tests, is the idea of reproducibility. Any test should be idempotent. They should have the exact same output regardless of how many times they've been run. This can become very tricky in complex integration tests as we add things like web servers and databases into the mix. This can increase the entropy of the test, causing them to fail infrequently. Our tests should be hardened against this. As much as there is lots to do when we write tests, there's just as much that we should avoid. And in the case of integration tests, they're mostly related. We want to avoid unpredictable tests. These are tests that are either not reproducible, or which are, but are only so because we're handling multiple different situations. Assertion should always be an and operation, never an or operation. We also want to avoid tests that go into too much detail. Integration tests can cover much more ground than unit tests, but should have a clear purpose. If you can't come up with a good name for the test, it's probably too much. We're about to write our first integration test in the next module, to do so, we'll be following a similar process to name our files on our tests. Remember that it's fine to store integration tests and unit tests in the same file so long as they're related. Some of our files today will get up between 100 or 200 lines of code but they're quite manageable with a bit of code folding. Again, you can break these tests up further if you desire, so long as it's done consistently.

## TransactionTestCase

The first test type we're going to cover is the transaction test case. It supports database transactions, assuming they're supported by your chosen database, and it does a few special things. It removes the need for ordered tests by resetting the data before each test, and it also allows us to run tests in parallel by literally creating multiple databases, one for each thread, and then running the tests together in parallel. As a result, these tests are a little bit slower than unit tests, but not really by much. Let's write some integration tests using the transaction test case. We'll talk about some good candidates for this type of test, and apply it to cover our models even further. Note that code coverage won't really benefit a lot from these tests. To start, we're going to edit our existing test_models files to add our new integration test. To work in a smaller area, we'll just collapse our unit tests, and you can see here our large files can be quite easy to work with when they're written this way. Normally, I'm only concerned about size when there's a lot of imports, especially if a lot of those imports have the same name. One thing the TransactionTestCase supports is fixtures. Fixtures are just records that we can pre-populate a database with before running our tests. To use them, we just define the fixtures property of any TransactionTestCase. Note that fixtures are supported on other integration test classes as well. To create the fixtures initially, you just have to pre-populate your own local database with information, and then dump it to a JSON file using the Django command dumpdata, like this. I've included the fixtures I use for these tests in the course files. The path for fixtures is relative to the project root, so we'll specify the full path. Now, these fixtures are loaded each time a class is constructed, and since a class is constructed for each test, the data is never actually changed. Sometimes I like to write a short test to make sure the fixtures have loaded. This makes it easier to track down problems with the fixtures. Our first test is going to check that our mark_incomplete method actually persists the state of the task after being called. We'll start by updating our entire set of tasks to have a valid complete time, which will cause them to report as incomplete. We'll then grab the first task, mark it as incomplete, and then check its value again after loading it from the database. It's important that we load the model from the database during our assertion because it ensures that the state wasn't just stored in our Task. objects defined on line 75. It truly ensures that a fresh database lookup will get us the correct result. Of course, we'll do the same thing for mark_complete. This time, we're going to update all the records to have no complete time, that is, they're not complete, and then we're going to check our model again and load it straight from the database. The next test we'll write is a little clever. It's a bit of a sanity check that will ensure that real time actually affects the due soon property of our models. To do this, we're going to create a task that will be due in 0. 01 seconds, then we'll wait much more than that and check that it is due soon. This could be a unit test as well, and some people may disagree with the pattern, but I think it does a fine job of ensuring that the state updates properly without loading a fresh record from the database. There are a lot of features in Django, like the cached-property decorator that can cause this test to fail. So let's run the tests. We'll do the same thing as last time, and pass our file explicitly to avoid running the entire test suite. This will run our unit test too, but that's fine. Looks good. We've now added some practical tests to our models, and next we'll take a look at some heavier tests that we can use to test our views.

## Django TestCase

The most common integration test class Django is the TestCase class. It has the same name as the class from the unit test library, but it is inside the Django test module. It adds more functionality, but the most important part is that it wraps all of our tests in atomic transactions, which helps us keep our data consistent and speeds up the test, but stops us from being able to test some behavior of database transactions. It's normally used to test things like HTTP requests and middleware including most of our view testing. Let's write some tests using the Django TestCase. The best candidates for these tests are our views, so we'll achieve most of our view coverage here. I've created a new file under our todo test module that called test_views. We'll use this to test the views that are not inside our tasks app, that is, the ones that are under the folder todo. We'll import everything we need here, including the TestCase, the Django user model for authentication, and the reverse method to resolve our URLs by name. Notice that we're not actually importing the views. That's because we're going to call them through the normal URL resolver. This drastically simplifies the process of calling the views, including parameter generation, and it also tested the URLs resolve like we expected it to. In my opinion, names should always be required for all Django routes. This makes it easy to retrieve the URL again in a reusable way. In our setUp methods, we'll normally define some properties that are consistent throughout our test class. We'll define some user properties, like the username and password, and the client, which is the Django request client. It kind of behaves like a browser, and we'll use it to simulate requests and get response data for analysis. The URL we're going to be testing has the name home, so we'll store that. And then we'll create a user now to log in with. We'll just hardcode the email address because we won't need it again. And keep in mind as we go that the setup methods are called automatically by the TestCase class. We're just overriding them here to add our own logic. When we arrive at the homepage, one of two things happens. If we're logged in, we're redirected to our task list. If we're not, we're redirected to login. Let's test the first case. You might be surprised to learn how simple this really is. We'll use the client to send a get request to our URL and store it as a response, then we just assert that the response redirects to our login, and since we aren't already authenticated, it probably will. AssertRedirects is built into the Django TestCase class and handles testing whether or not we've been redirected by the response. That's it. Like I said, assert Redirects takes care of testing whether our response code is appropriate, and that the location had a required for our redirect is present in the response. So let's run that test. Who would have thought the testing of requests and response lifecycle would be so simple? Let's test the other case when we're logged in. In this case, we should be redirected to the task list. To log in, we'll use the client's login method. It takes a username and password, and performs authentication using the default authentication handler. It provides the client with a session cookie and so on so that we can test as if we're actually signed in. The rest of the test is the same. We send our request, and then we assert that the response redirects to the correct location. In this case, that's the task list, and that URL has the name tasks. And this is all the home view does as far as custom efforts go. The rest of the functionality of the view is built in, so we're not going to test it. We'll just move on. Next, we have our register view. This one's a little bit different than the other ones we'll be testing because it's not a class-based view, so there's not some sort of secret hidden logic or other built-in features. We'll essentially be testing the entire view on its own. We'll set up our class in a similar fashion to before, initializing our client and our username, we'll set the URL, and then define the data that we'll be providing to the view. We can send data in a dictionary, and in this case we're preparing the parameters that the view expects, two matching passwords, and email, and a username. We'll also create a set of bad data, which for this case is just going to be a copy of the other data set, except we're going to update it so that the second password no longer matches the first one. The two passwords have to match to be a valid login, so this should fail. We're also going to define some HTML that we're expect to see if the registration form is actually inside the HTML, and this is going to help us check whether or not it's present when we get our response. First, we're going to test the happy path. If we send a post request to the register on point and that contains valid data, we should see that a single user was created, and that that username is equal to the username we sent. Again, it's pretty simple for an integration test. Let's run it. Not bad. Note that although we said these tests were slower, they still run in a pretty small amount of time. They're slower than unit tests, but not something we'd really have to worry about. Not we're going to check that we get properly redirected after registration. Some people might put this in the same test, but that would be too much for a single integration test. Again, we're still trying to keep our tests a little bit more siloed. We'll send the same request, and then assert that the response redirects us properly to the task list view after we've been registered. We also have to check that the registration view returns the form when we send a get request. To do that, we'll send a request, and then use the assertTemplateUsed, and assert in HTML methods to make sure that both the right template was used, and that our submit button is present. Obviously if the submit button isn't present in the form, we're not going to be able to register. There are other convenience methods that are targeted at checking whether or not forms were used to render our view, but I find them pretty hard to work with, and I find that they work somewhat inconsistently. We can also check that the form is present in the request context. Let's check our progress. We're going to run our tests again. Looks good. Again, our tests are still running very fast considering that they're integration tests, and you know, from our perspective there's not much a difference between this and unit tests. Now we're going to test some failure cases of our registration view. We'll write another template test to check that the form was returned again if our request fails. That is, we'll see the form again if our registration didn't work for some reason. We'll do the same thing, but we'll just send the bad data that we created before with a post request instead of sending a get. We can also check that we receive a specific error. We'll send our bad data along, and then assert that a specific error message is present. I normally find it easiest to get the expected HTML from the DOM directly and then copy it into the test, which just watches for change instead of testing some dynamically generated HTML, which I find takes a lot more time and doesn't really give us a lot. We'll run this test as well, which will be the final test for our registration view for now. There are lots of additional tests that we could write, but this is a good starting point for now. Alright, now let's create some tests for our task views. We'll start by creating a new file in our tests module under the tasks app. I have prepared our import statements ahead of time here, and most of them are quite familiar by now, but I've included one new one, AutoFixture. AutoFixture is a third-party tool that allows us to create records in our database easily in code, without having to use JSON fixtures. The advantage here is that they're easier to create, as generating fixtures from a real database can be tedious, and they're much easier to update. They also survive most migrations whereas most fixtures will not. I've already included AutoFixture in the requirements, or we can just install it using pip. Once it's installed, we'll start off by testing our TaskListView. We'll override the setup method and add some data. Using AutoFixture is pretty straight forward for most cases, but it also has a lot of complex configuration options. The AutoFixture class is initialized with a model name, in this case user. AutoFixture will go ahead and initialize that object by filling in all the object's parameters with values. For example, it'll generate a random string for our character field, and a random number for any number field. Once created, the fixture has a create method. That takes a number that tells us how many of that object we'd like to create. We just need 1 user, so we'll just say 1, and then we'll grab the first and only value in the list of objects returned. I would prefer is AutoFixture returned a query set, but it returns a list, so we'll have to grab the object like this. In the next case, we're creating a fixture for a task. We can also provide values through the field_values parameter, which takes a dictionary mapping of property names on the model and their initial values. So here we'll create a fixture that can be used to create tasks, whose owner is set to the current user. Then we'll use that fixture to create 10 tasks and store them to our TestCase. Now we've got one user and 10 tasks belonging to that user. I find this a lot easier than generating fixtures because generating fixtures either forces me to deal with things like primary keys and password hashes and database constraints using JSON, or requires me to prepare a database ahead of time if I want to simplify that process a bit. The biggest thing I notice though is how AutoFixture is incredibly resilient to changes in our models, whereas almost all changes are going to break a set of JSON fixtures. Alright, so we've got our data and now we're going to write some tests. The list view is largely out of the box, but we've customized the query set in the context data, so we should probably test that. Let's start with testing that the task list view returns all other user tasks. Since we don't currently have pagination or anything, this should always be the case. Here, we're going to use the client's force_login method because we don't have a password for the user we created with AutoFixture. Force login just allows us to bypass any real security and log a user in by username. Once we're logged in, we'll just send our get request, and we're going to check the context of the data coming back. We want to check that the length of the tasks that we created is equal to the length of the list of tasks returned to us in the context. The next thing we're going to test is a pretty important feature of this view, that the list only displays tasks from our user. We want to make sure we don't accidentally show a user someone else's tasks. To do this, we're going to create another user, and then we'll give that user a task using AutoFixture. Then we're going to run basically the same tests as before. The assertions are the same. We could be a bit more picky by checking that each task in the result has the owner equal to the expected user, but it's not really necessary for now, and we can assume that if we receive all the tasks back and there's not an extra one, we're probably okay. Now let's run those tests and make sure they work. This is about all we need here for the TaskListView. Good so far. Next thing we're going to test is the new task view. This is the view we use to create new tasks. Our setup method will look pretty similar, but we're not going to create any tasks for the user this time. This makes it a lot easier to check if one was created. One thing we want to test is that our app doesn't let users create tasks for other users. This sort of pattern is something I see come up a lot when reviewing other applications. I see people taking user IDs from the frontend, which can easily be changed to simulate another user. To test that our application's not vulnerable to this, we'll create another user and send a request to create a task as that other user. Then we'll assert that once the task was created, it was created using the correct user as the owner. The follow parameter here in our post request just tells the response to follow redirects since we know we're going to be redirected to the task list afterward. Note that we don't have to test that redirect since it's covered reliably in our mixin unit tests. Next, we'll test the task view. This is the view that shows us details and lets us edit them. We don't really need to test the update feature since it's built in, but we could test some other stuff. Let's set up a similar TestCase to before where we create the user using AutoFixture and then create that user some tasks. In this case, we're just going to create one. We create the task ahead of time so that we can get its URL using the reverse method. Now let's test that the task view generates a form for us. We'll force the user to log in, and then get the view for the task that we created. This should return us the form where it shows us the details of the task and allows us to update them. We can now assert a few things. First, we'll assert that the task. html template was used to render this. Next, we'll test that we have a task form in our response context. See how this is a little bit easier than checking HTML? Lastly, we'll check that we have an update Boolean in our response context. The Boolean's what we use in the template to determine whether or not we're creating or updating a task. Running those tests, we should expect them to pass not problem. Awesome. Now moving one, we're going to test our ToggleCompleteView, which is a little bit more interesting because it's not built in, it's another function-based view, and it actually changes some state. To set up, we're going to create a user like we have before. We can also force a login in the setup method. This stops us from having to do it in each subsequent test method. I'm not sure how much I like this because it may bring a viewer to believe that a user's not logged in during that test, but I'll leave that up to you. Now, let's check that toggling a task to complete actually changes our complete value. We'll set up a task with no complete time and get its URL. Then, we'll first assert that the task is not complete when it's loaded from the database, then we'll send a request, and then after that request is sent, we'll assert the opposite, that the task is complete. Cool. To test the inverse, we can pretty much just copy our code from this test. We'll just update the title, switch the two assertions, and then change it so that the initial state of our task objects is going to be with a valid complete time in the past so that they start off as being complete instead. The last thing we need to test for this view is what happens when we can't find a task. This is pretty easy to test. We just send a request to a URL for a task that doesn't exist, and then assert that we receive a 404 like expected. This covers the majority of inputs for this view. Let's run all of our tests one more time. Awesome. We've added 7 great integration tests for our views. Let's run the whole test suite together now. Cool, 31 tests in less than a second is pretty good. That means we can start to get feedback about hour our application is changing a lot faster. Next up, we're going to talk about one of the most unique test cases in Django, the live server test case.

## LiveServerTestCase

We're now going to cover a really interesting and often overlooked feature of Django test ecosystem, and that's the live server test case. This test case creates a live, running Django server. It's kind of similar to running runserver during development. We can specify a different port to use, as well as all the normal parameters we'd set for a server. And there's a subclass of this called the static live server test case, which also handles static files serving like our server does locally. Normally in production, of course, that would be handled with a web server like NGIX. This test case is primarily used for things like remote testing, certain CI steps that require things like health checks from third-party services, or other third-party test tools. The scope of the functionality of this test is massive, but we're going to cover a very simple case. Let's do a demo. We're going to identify a case for, and write a live server test case, and then we're going to talk about a few common uses for this class. Here we are inside the To Do application, and there's a few things we might want to test about this UI, but as we saw before, doing things like checking the HTML output in a response is not a very graceful process and it's subject failure if we don't do it properly. When creating a task, our data input uses Chrome's built-in date picker. Maybe if this was a custom one we might want to check that this properly sets the date value in the backend. We could write a live server test case that simulates a browser, and then check the backend after. The great part about these cases is that they have the ability to check the backend state as well after we've interacted with the frontend. For now, we're going to test a simple case. We're going to test the functionality of the checkmark. We want to ensure that it's the correct color based on whether or not the task is complete. The color right now is set by a single CSS class. The documentation for the LiveServerTestCase offers some insight into how it can be used, as well as a few of the things to watch out for. One of the most common integrations used alongside this class is Selenium, which is a browser automation tool that allows us to simulate a user accessing the site with a real browser like Chrome or Firefox. Selenium is run by things called web drivers, and these are different drivers to support different types of browsers. Installation of web drivers varies depending on your operating system in the target browser, so I've already installed one for Chrome. To install Selenium for Python, we just need to pip install a package selenium. And note that I've already got this chromedriver installed and we can run it locally to check. I'm just going to create a new file for these tests called test¬_live_views, that is, live server tests of our application's views. I've started here with code taken from Django's documentation. This code just helps us officially set up an instance of Selenium's web driver that we can use in our test cases. These class methods are used in a really clever way. When we write these tests, we initialize the test once for every test method that we run. That means that if we do a lot of work in the set up method, we perform that work once for every single test. When we use the class methods, the method is run just once when the class itself is initialize, and not when we create an instance of that class. That means these two methods will only run once. What they do is pretty straight forward. They set up a web driver instance, and then they tear it down when we're done. We're going to start here by organizing our tests in a very similar way to the other tests. We'll create a user and give that user some tasks. We're going to use the normal user creation method here because we have to define our own passwords so that we can use it to simulate our user login through an actual browser. We'll just give the user one task for now, and that task will be incomplete so we can accurately predict the colors. Let's just run the test and make sure everything's working first. Selenium will complain at this point if we haven't installed everything correctly. Alright, let's start writing the test. To view a task, we've actually got to log in, and that means we have to actually follow through the login steps with Selenium because we can't use something like force login anymore. There are some clever ways to create a user session without actually filling out all the fields, but we're going to do it manually for now. To log a user in inside the UI, we'll use Selenium. We'll start by pointing Selenium at a URL. In the background, this runs a browser and navigates to that URL. In this case, we're using Chrome. Then, we're going to use the method find_element_by_name to find an input element with the name username. This is the actual DOM element inside our render view, inside Chrome. We'll do the same for password. Then we're going to use a method called send_keys to send actual keystrokes to each input field, like letters or numbers or spaces. We'll put the username in the username field, and then the password in the password field. To sign in, we've got to click the login button. To find it, we can use a CSS selector using this method. This is my favorite method for finding elements. They're a support for lots of options, like finding elements by IDs or XPath, but as a developer that's spent a lot of time writing frontend code like CSS or JavaScript, I'm quite comfortable writing CSS selectors. We'll write a selector to select a button with the type submit, and then click it, and this is actually going to sign us in. Now at this point, it's good to check that we're on the right track. These tests can take a long time to write because it can be tricky to predict exactly what the browser can and cannot see at every given time. Okay, so we're good so far. And so far we haven't really tested our test_toggle method. Like we haven't done anything to do with toggling, we've just logged in. So let's move this logic out and put it in another method called login, and then just call that method from our test_toggle method instead. This just separates our preparation logic from our actual test logic and allows us to reuse the login steps for another test later. Let's start by checking the initial color of the element. We'll track down the task complete button using a CSS selector, and I'll just tell you that the class that makes it blue is called glyphicon-blue. So we'll check that that class is actually applied. We can do this by checking the attribute class on the element that we found. Right now, we'll run our test again to just have Selenium check for us. And that looks good. We're going to need to find our button a few times, so let's wrap that again in another method. The second case we're going to check is that the task button has the class glyphicon-red, which is what's going to make it red. For this class to switch, the button should be clicked, and we can do that between our assertions by just taking the button that we found, and then calling the click method. Notice that we retrieve the button again after clicking it. The reason we have to do this is because clicking it actually caused the page to reload, and so our selector was referencing a now-gone element from the previous page. This is just one of many tricks you'll come across when writing true end-to-end tests like this in Selenium. Let's run the whole test together. Again, this starting a live server, hooking up a Selenium web driver in Chrome, and then actually interacting with our UI. Not bad for 6 seconds of work. Let's run all of our tests together. You may notice that larger tests like this one slow down the reactiveness of other tests. Django provides a few simple methods to control which tests run when so you can always control when longer tests run. In just 50 lines of code, we've got our very first end-to-end test.

## Summary

By now, it should be pretty clear why integration tests are preferred in Django. They're so similar to unit tests and they allow us to test our app even better without mocking out important services. Of course, there's a lot of valid arguments in favor of mocking and unit tests, and there's a lot of cases where they're superior, but I tend to find that in real scenarios, tests like we've covered are more effective at meeting the goals for testing, which is to ensure that applications work and adapt to change. We've covered a lot of best practices that are going to make integration tests do their job. We've covered the two main test case options, the transaction test case and the standard test case, and we've also covered the live server test case and used third-party tools to test our application in a full end-to-end test.

# Measuring Test Coverage

## Measuring Coverage in Django

Let's take a look at Measuring Test Coverage for the tests we've written so far. Measuring test coverage can provide us with a valuable insight into how well our app is tested, and that makes it a pretty valuable metric. Keep in mind that test coverage is still not our primary goal. Our primary goal is to ensure that our tests are useful and that they make our application better. Though high test coverage is ideal, we want to avoid testing just for the sake of testing. Measuring coverage in Django is no different than measuring coverage in other Python projects. We're going to use the coverage. py module to perform these measurements. Coverage is broken down into a few different metrics. The overall total coverage is the primary metric, but there are others that can provide some valuable insight into how our tests are performing. There's no true standard on how much test coverage we should be shooting for, but generally 90% to 95% is considered good coverage, 100% coverage is attainable, but it's not always useful. We'll be aiming for 95% coverage, but you should pick that value based on your project. To meet this goal, we'll need to configure our test coverage integrations properly. Failure to do so will result in either the measurement of unwanted files, or other undesired behavior that will either inflate or deflate the true coverage of our tests. To make sure that our tests continue to perform, we need to make sure that we have systems in place to maintain our coverage over time. This involves both starting with test coverage early in the project, and maintaining it as we make changes. Generally, new code should not be accepted into the project unless overall coverage has not been negatively impacted. Coverage. py is the standard for measuring test coverage in Python. It can already measure our code and generate both text outputs for our terminal, and rich interactive HTML output reports. There's a lot of third-party support, libraries, and plugins for this module. The coverage module also supports a file-based configuration called coveragerc, which we can use to check in the configuration that makes our measurements the most accurate. To generate coverage maps, we simply need to run a single command. The coverage run command acts like a normal Python interpreter. We just type coverage run before the normal Python commands that we use to run our tests, and it takes care of the rest. Here's an example of coverage's output. We can generate this with the command coverage report, which we'll cover later in the demo. There are a few important metrics here. Statements is the total number of reachable statements in the file. Miss is the number of lines that were note reached during the tests, and cover is the measure of our total average coverage for that file. We can also see a list of lines that were not covered in the missing column. Underneath, the Total row gives us the total for the entire test suite. This is where we're going to look when we're determining if we've reached our goals. If we haven't, then the per file information will will give us a hint as to what can be improved. We can also generate rich output in HTML by running the command coverage HTML. The output looks like this. This example also covers a few extra things. It tells us how many lines were excluded from our test by our configuration, and also indicates the total number of branches that exist in each file. Exactly what is measured can be configured using that configuration files we discussed earlier. Clicking any of these files will take us to another page that shows the impact of our test on a line-by-line basis. Green highlights on the left mean that the line was fully covered, We can see in this example in red that there were a few lines that weren't covered by the test. This makes it really easy to spot which areas of the code need to be tested more thoroughly.

## Measuring Your Own Coverage

Alright, let's measure our own test coverage. We've written unit and integration tests for our to do app, as well as one end-to-end test using the live server test case and Selenium. Let's see how well those tests are doing at actually covering our code. We're going to generate a coverage report, take a look at the HTML, and then configure our coverage reports to be a little bit more accurate by excluding certain files from our tests. To measure the coverage, we need to start with the coverage run command. After it comes our normal testing commands. In this case, manage. py test. Our test will run normally, but the code is actually being monitored by the coverage module while they run. Once the tests complete, we can run coverage report to generate a report inside our terminal. We can see right away that the coverage report is picking up files inside my virtual environment. We'll want to make sure that this is not the case, and we'll add some configuration later on to resolve that. Near the bottom, we'll see our project files and their coverage. The total here is inaccurate, but that's fine for now. We can take a look at what a rich report looks like from the same test results using the coverage html command. We can then open the index. html file that was generate in our browser and navigate those results. Here we can get a much better picture of what's going on. Let's take a look at one of our own files. This file is covered pretty well. This is the sort of result we're looking for. Now let's focus on getting rid of all those unrelated files. To change the way the coverage performs its measurement, we can create a. coveragerc file. It begins with a dot, and the file contains optional configuration parameters for generating our reports. Most, if not all, of these parameters can be set on the command line using various flags, but this will ensure two things. First, it ensures that the reports are being run the same each time regardless of who is running them. Next, and more importantly, we can actually check in this file to our version control, such as get, and it means that we can easily support different configurations on different versions and branches of the code. We'll configure the run section of our coveragerc file. We'll tell coverage to measure our branch coverage, which will measure the number of branch possibilities taken for each file. Next, we'll start to omit certain files from our tests. The first and most important step here is to omit the virtual environments from being tested. This syntax accepts a shell-like file pattern. The star is a wildcard, so this rule will match any files that has. virtualenvs in the path. Let's take a look now through our results and exclude some more things. We can see it's testing __init___. py files. These are empty in our case and don't need to be included. Some projects will leverage those files so don't omit them blindly. We can also see that our actual test files are showing up too, and we definitely don't want that, so we'll exclude those as well. We aren't testing migrations, so we can omit those, and we don't need to test the urls file because all it does is define a list that is used internally by Django. If we wanted to add some URL tests that validated that certain URLs were also defined or something like that, we could, but coverage for that is difficult to test since all the URL files do is define a single variable called URL patterns. We generally also want to exclude both our manage. py file, assuming we haven't done anything special to it, and the admin and apps files for each of our apps. Again, assuming we haven't written a lot of tests for them. If we want to test and measure any of these files, they shouldn't be omitted. These settings are good for this app, but will absolutely vary from project to project. Alright, with the new configurations in place, let's run our coverage report again. First, we need to run the tests again, then we'll take a look at the report. This report is a lot neater than before, and it gives a much better picture of the success of our tests. We can see that we've met our goal here, but haven't reached 100%. There's always room for improvement, so let's take a look at what we can improve using the HTML report. We'll generate the rich HTML report using the coverage html command, and then open it up. Let's start by taking a look at our views. We can see that our coverage is pretty good. The great thing about the HTML report is that is makes it really easy to determine what test we can add to improve coverage. Here's an example of our branch testing kicking in. The yellow highlighted if statement shows that only one of the two possible paths were taken. That is, the path where the statement is true has been tested, but the path where the statement is not has not been covered. We can test this by writing a test that causes this statement to be false. Let's see what else we can do to improve. Our mixins file appears to be the least covered, so let's open it up and see what's going on. The report shows that the task owned by user mixin was not covered. Now we have covered this feature implicitly in our tests by implementing the mixin in its own class. a mixin like this can be a lot of work to test on its own, but if we want to ensure that it's tested and covered, we get out a few unit tests with mocked-out requests to the suite. Now we've got some work for another day. Overall, our test coverage is pretty good, and our coverage is right around where we want it. Take a few minutes to browse the other files and see what else remains uncovered by our tests.

## Summary

We've now covered what we need to know to start measuring test coverage for our Django test suite. We learned how to analyze the output generated by our coverage tool, and configure our project to increase that accuracy of the coverage reports and remove unwanted files from our results. Remember that all of these things, when done early, will set us up for success. Increasing test coverage late in a project can be a lot of work, but incrementally working on test coverage throughout the lifecycle of the application will be much easier.

# Exploring Testing Options

## Overview

We've covered a lot of great testing options so far, but there's still more to be discovered. So let's take some time to go over a few of the other testing options. So far, our focus has been largely on Python's built-in test features and Django's extensions of them. These have been really great, and will cover most of our unit tests and early integration testing needs, but there are a few problems that they don't solve. We'll review some other options for unit testing including some third-party tools that can improve our workflow and offer some extra features. We'll cover some additional testing libraries that can fill some of the gaps left over when using the built-in methods, and we'll take a look at a great end-to-end testing option for Django. First up, let's talk about pytest. It's a fairly robust framework that helps us improve our testing workflow. It does a lot of things, but I'll talk about a few that are the most important to me. It provides a more detailed set of failure information when tests don't pass. It has improved autodiscovery for tests, and although this isn't as much of an issue when using Django, it's great to have other Python projects and Django projects that get a little bit more complex, and it also has great plugin support. There's a lot of features that aren't included here, but if you're interested, there's also a Django integration called pytest-django that gives us some more Django-specific features. Namely, it allows us to write function-based tests instead of class-based tests. These are great for short, sweet unit tests. It also allows us to reuse databases during certain tests that Django normally doesn't reuse. It supports file-based configuration, which we've seen the benefits of before when we were working with code coverage, and lastly, it provides a single-word command, pytest, to run our tests. Django's managed. pytest command is fairly simple, but as we start to add things like certain flags, it can become somewhat complicated. Pytest file-based configuration allows us to store these flags in a file, and keep using the short command name. So that means we can just run pytest instead of having to run the manage command with its optional flags. Tox is another awesome tool that has a very specific purpose. It's aimed at building reusable Django apps. Django's conventional app structure makes it really easy to reuse certain applications, but in reality, most people will never truly benefit from their power. When you do write a reusable app, it's often distributed to many projects, which makes things like version or operating system-based issues and regression issues a much bigger risks. Tox allows us to run our tests in multiple Python environments, and multiple Django versions. That means that you can test every supported version of Django and Python together in every different combination. It's also worth covering doctests. Python objects have a magic method called doc that returns the doc string, which is the multi-lined string directly under the class definition statement. Doctests allow us to write small, console-style tests right inside that string. Here's an example of a simple multiply function and some simple in-and-out test for it that we've written as a doctest. We're just asserting that multiply with the parameters 4 and 3 will return 12 for example. These tests are great for small, simple methods, and stop us from overcrowding our real test classes with small things like this. The last tool that we're going to talk about is Ghost Inspector. Ghost Inspector allows us to run UI tests in a real browser in a really simple way. There's a Chrome extension that allows us to record the tests, and then it allows us to run them again remotely with an emulated browser. Sort of like Selenium. When the tests complete, we can even verify that the resulting screen we're left with is identical to the last time we ran the tests. It's reasonably priced with a small free tier, I think it's like 100 tests per month, and is one of the easiest ways to get started with UI testing for your Django-powered site. That said, there are some limitations to Ghost Inspector. The tool doesn't work very well for AJAX-powered sites or single-page apps, which, you know, Django doesn't really support these two things out of the box easily unless you're using a tool like the Django REST framework. The tests can also be difficult to manage as the UI changes during development If you're in a process where your iteratively developing changes to the UI, tests can be a little difficult to keep up and will have to be either modified or re-recorded frequently. All in all, Ghost Inspector is a really powerful and super easy to use tool. So we'll take a look at how we would get that set up with our Django-powered site.

## Your First UI Test

Alright, let's take a look at how we can use Ghost Inspector to test our site. We'll create a Ghost Inspector test, run that test, and then take a look at the video recording and screenshot comparison tools that they provide. I've just registered for an account here and signed in. I don't yet have the Chrome extension so I'll download it now. I'll add that Chrome extension, and then just log in with my account credentials. I only have to do this the first time. Alright, now that we're all signed in, I've got the site running here on port 8000. However, for a Ghost Inspector to reach me, the site has to be able to reach this URL, which right now it doesn't. We need it to be publicly available. For now, I'm using ngrok, which is a tool that gives us a public URL that actually tunnels into our local machine. I've got mine running here, so we can just grab the URL that they give us and go to it. The site as we see it now is actually coming from my local Django server, but at least now it's publicly reachable, so services like Ghost Inspector can get to it. I'll pop open the Ghost inspector extension here and Start Recording. I'm going to log in using a test account to start. Note that this needs to be recorded so that the tool can log in later. For this test, I'm just going to add a few tasks. So I'll go ahead and add the first one here with a title, a description, and we'll pick a date, and then I'm going to add another one, which for this one I'm just going to do a title and a description, and I'm going to leave out the date. Okay, now I'm going to delete the tasks. It seems a little strange and this isn't maybe a great way to test, but currently we haven't built a way to reset the database after the test run, so we need to leave our app in the same state that we found it so that the tests run the same every time. For now, this is just for demonstration purposes, so this will work. We're done recording, so I'm just going to go ahead and stop the recording, and then give it a name, and save it to Ghost Inspector. By default, it's going to kick off an initial test run. So we'll see if that passes. Okay, our tests have failed. So let's take a look at why. On the bottom, we can scroll down and look through all the steps and then find the step that failed. It was looking for an anchor with a specific href, which as you can see is based in the ID of task. Since those IDs increment for every test, they're different and that anchor wouldn't be present for every single test. So let's fix that. We're just going to select instead the first anchor we see each time, which is kind of the same as selecting the first item in the list every time we run the test, and that's going to work just fine for our test. So let's save this, and run the test again. We'll wait for it to complete, and it passes. So we can now view a video of the test, which is going to show the entire process recorded through the browser, and it's also going to highlight things like certain interaction points, like inputs. And then we can also view the final screenshot that we ended up with. Note that this is just the last thing that the test saw. If there's a different between this result and that of the last test, it's actually going to be highlighted here. Hopefully you can already start to see the benefits of Ghost Inspector over other integration tests we wrote, like those in Selenium.

## Summary

We've now gone over some of the other test options out there for improving our tests. Pytest is a framework that extends the functionality we've worked with so far. Tox allows us to test reusable apps across multiple versions of Python and Django. Doctests are short little tests that we can write from right inside our code, and Ghost Inspector is an easy way to get started with full UI testing for our apps. Normally, UI testing can be somewhat complicated to get set up, but I think with Ghost Inspector it's a pretty simple process.

## Course Summary

Let's review. We're now ready to test our Django-powered apps with confidence. Armed with an arsenal of powerful tools, we can get to work making sure our applications are resistant to change and consistently perform well. We've got a whole host of tools at our disposal. Python's standard test classes can be used with zero dependencies and Django's simple extension of them add plenty of extra power and context in the web world. We've also covered some of the best practices for testing in Django, and some other things that might be different from other web frameworks or languages that you might have worked with before. We've learned to measure test coverage, which can help us in our endless goal to evaluate the effectiveness of our tests. We've talked about configuring our tests, and our coverage measurements for accuracy and most importantly, we talked about keeping that test coverage up as the project unfolds. Lastly, we covered some important end-to-end testing tools, like Selenium for code-powered tests, and then Ghost Inspector for tests that we can record right inside our browser. Thanks for joining me to learn about testing your Django applications, and happy testing.

## Course author

Jamie Counsell

Jamie Counsell is a Software Developer at Konrad Group. With a background in teaching, web, and API development, he has a passion for building modern web applications, leveraging cutting edge...

## Course info

| | |
|---|---|
| Level | Intermediate |
| Rating | ★★★★⯪ (11) |
| My rating | ★★★★★ |
| Duration | 1h 38m |
| Released | 17 Oct 2017 |

## Share course