

A Wedge of Django

Covers
Python 3.8

and
Django 3.x

The Django tutorial
by the authors of
*Two Scoops of
Django*

Daniel and Audrey
Roy Greenfeld

A Wedge of Django

Covers Python 3.8 and Django 3.x

Daniel Roy Greenfeld, Audrey Roy Greenfeld

2021-06-30

Contents

1 Copyright {{-}}	25
Introduction	26
Why Learn Django?	27
When the Going Gets Tough	29
This Book is intended for Django 3.x and Python 3.8	30
How This Book is Structured	30
Conventions Used in this Book	31

2 The Ultimate Django Setup	33
2.1 Goal: A Professional-Quality Development Environment	34
2.2 Core Concepts	35
2.3 Step 1: Install Visual Studio Code	36
2.4 Step 2: Installing Git	40
2.5 Step 3: Installing Conda	43
2.6 Step 4: Installing PostgreSQL	47
2.7 Summary	49
3 Hello, World in Django	50
3.1 The Hello, World of Python	51
3.2 Hello, World of Django: Simple on Purpose	52
3.3 Projects Goals	52
3.4 Prerequisites	53
4 Preparation: Install Dependencies	54
4.1 Set Up a New Conda Environment	54
4.2 Install Django	56
4.3 Confirming Django is Installed	57
4.4 Summary	57

5 Generate a Starter Django Project	59
5.1 Anatomy of a Bare-Bones Django Project	59
5.2 manage.py	60
5.3 settings.py	61
5.4 Other Settings	63
5.5 Exercise	64
6 Create a Database and Run the Project	65
6.1 Ensure We Are in the <code>helloworld</code> project	65
6.2 Creating the Database	66
6.3 Start Up Runserver	67
6.4 Runserver Has Started Up	68
6.5 Success!	68
7 Create a Django Superuser	70
7.1 Stop runserver if It Is Still Running	70
7.2 Run <code>createsuperuser</code>	70
7.3 Summary	71
8 Practice Using the Admin	72
8.1 Restart runserver	72
8.2 Visit the Admin	72

8.3	Log in as Our Superuser	73
8.4	Explore the Admin	74
8.5	Recreating the Hello, World in Django Database	74
8.6	Summary	75
9	Generate a Homepage App	77
9.1	Stop runserver if Needed	77
9.2	Generate a Starter Django App	77
9.3	Anatomy of a Simple 1-App Django Project	78
9.4	Exercise	79
9.5	Main Django Project vs Django Apps	79
10	Prepping the Project for Templates	80
10.1	Configure Templates	80
10.2	Save!	82
10.3	Lists? Dictionaries?	82
11	Add a Simple Homepage View	84
11.1	Get to Know URL Routing	84
11.2	Add a URL Pattern for the Homepage	86
11.3	Double-Check the Code	87
11.4	Terminology: Wire in the URL	88

11.5 Refresh the Page and Trigger an Error	88
12 Create Our First Django Template	90
12.1 Create a Templates Directory	90
12.2 Add Index Template	90
12.3 Start Up Runserver	91
12.4 View the First Page	91
12.5 Recap	91
12.6 Understanding Views and the Template-View Class	93
13 Working With Variables in Templates	94
13.1 Our First Templatized Value	94
13.2 What Is <code>my_statement?</code>	95
13.3 Extend HomepageView	95
13.4 Refresh the Homepage	96
13.5 Exercise: Play Around!	97
13.6 Common Questions	98
14 Calling View Methods From Templates	99
14.1 Define a View Method	99
14.2 Call the View Method From <code>index.html</code>	99
14.3 Reload the Homepage	100

15 Our First Project is Complete!	102
15.1 Summary	102
15.2 Next Steps	103
16 Enter the EveryCheese Project	105
16.1 Prerequisites	105
16.2 Prepare the EveryCheese Programming En- vironment	106
16.3 Create a New Environment	106
16.4 Reactivating Conda Environments	107
16.5 Summary	107
17 Cookiecutter and Template	109
17.1 Make Certain the (everycheese) virtual envi- ronment is activated	109
17.2 Install Cookiecutter	109
17.3 Cookiecutter Project Templates	112
17.4 Using django-crash-starter	112
17.5 What Happened?	116
17.6 Tips and Troubleshooting	117
17.7 Cookiecutter of Background	118

18 Exploring the Generated Boilerplate	120
18.1 Some Basic Components	120
18.2 Summary	122
19 Starting Up Our New Django Project	123
19.1 Go Into the EveryCheese Directory	123
19.2 Install the Dependencies	123
19.3 Defining the database	125
19.4 What to name the database?	128
19.5 Run the Migrations	130
19.6 Start Up Runserver	131
20 Cheese Boilers	132
21 Initializing the Git Repo Committing and Pushing	133
21.1 Why?	133
21.2 We'll Use GitHub	133
21.3 Create the GitHub Project	134
21.4 Examine the GitHub Instructions	134
21.5 Verify That the Files Got Committed	137
22 Git Is Not for Stinky Cheeses	138

23 What We Get From The Users App	140
23.1 Register a Normal User	140
23.2 Verify Our Email	142
23.3 Why Are Emails in the Console?	145
23.4 Explore the User Profile	146
23.5 Edit User Data	147
23.6 Introducing <code>UserUpdateView</code>	147
23.7 Introducing <code>UserDetailView</code>	148
23.8 Understand the Name Field	149
23.9 Understand <code>UserUpdateView</code>	150
23.10 Add Email Addresses	151
23.11 What About the Code?	152
24 Adding User Bios	154
24.1 Add a Bio Field	154
24.2 Create and Run Migrations	154
24.3 Update <code>UserUpdateView</code>	156
24.4 Try It Out	157
25 Adding Bio to User Detail Template	160
25.1 Update the User Detail Template	160
25.2 Add Line Breaks	161

25.3 Explaining <code>\nlinebreaks</code>	163
25.4 Commit Our Work	163
26 Introducing Tests	165
26.1 Why Test?	165
26.2 Running Tests	165
26.3 What Does a Test Look Like?	168
26.4 The Test Foundations	169
26.5 The Actual Test	169
26.6 Using the <code>assert</code> KeyWord	170
26.7 Introducing <code>coverage.py</code>	172
26.8 Using <code>coverage.py</code>	172
26.9 Generate HTML Coverage Reports	173
26.10 More Users App Tests	173
26.11 Summary	174
27 User Profile Cheese	175
27.1 Avatars Will Fit Into Cheese Holes	176
28 The Cheeses App and Model	177
28.1 Make Sure We're at the Project Root	177
28.2 Create an App Called Cheeses	177

28.3 Move It Into the Project	178
28.4 Set the Cheese Config	179
28.5 Add It to the List of Installed Apps	179
28.6 Add the Cheese Model	180
28.7 Create and Run the Initial Cheese Migration	184
29 Trying Out the Cheese Model	186
29.1 Let's Try It Out!	186
29.2 Django Shell vs. Shell Plus	186
29.3 Create a Cheese	187
29.4 Evaluate and Print the Cheese	187
29.5 View the Cheese's Name	188
29.6 Add a String Method	188
29.7 Exit shell_plus	189
29.8 Try It Again	189
29.9 Commit Our Work	189
30 Test Coverage	191
30.1 How to Check Test Coverage	191
30.2 Create a Module for Cheese Model Tests . .	193
30.3 Let's Test the <code>__str__()</code> Method.	194
30.4 Enter the Game of Test Coverage	196

30.5 Rules of the Game	196
30.6 What's the Point of All This Anyway?	197
30.7 Commit the Cheese Model Tests	198
30.8 Summary	198
31 The Cheeserator	199
32 Adding Cheeses to the Admin	200
32.1 We Need a Superuser	200
32.2 Go to the Admin	201
32.3 Explore the Admin So Far	201
32.4 Register Cheese Model in the Admin	201
32.5 Find Colby	202
32.6 Enter More Cheeses via the Admin	203
32.7 Some Notes About Using the Django Admin	204
32.8 Commit Changes	205
32.9 Summary	206
33 Behind the Curtain	207
33.1 We Are in Control	208

34 Class-Based View Fundamentals	209
34.1 The Simplest Class-Based View	209
34.2 Adding More HTTP Methods	210
34.3 Advantages of Class-Based Views	212
34.4 Composition	213
34.5 Composition Part II	214
34.6 Intelligent Defaults	214
34.7 Standardized HTTP Method Handling	215
34.8 Tips for Writing Class-Based Views	215
35 Writing the Cheese List View	217
36 Wiring in the List View URL	218
36.1 Define Cheese URL Patterns	218
36.2 Include Cheese URLs in Root URLConf	219
36.3 See the View in Action	219
37 The Cheese List Template	222
37.1 Create the List Template	222
37.2 Fill in the Template	222
37.3 See the List in Action	224
37.4 Add a Nav Link to the Cheese List	225

37.5 Explaining the <code>cheeses:list</code> Name	227
37.6 Commit Our Work	228
38 Add the CheeseDetailView	229
38.1 Add the CheeseDetailView URL Pattern	229
38.2 Link List Items to Detail Pages	230
39 The Cheese Detail Template	232
39.1 Add a Cheese Detail Template	232
39.2 Visit Cheese Detail	234
39.3 Commit Our Work	235
40 Where to Dip Class-Based Views	236
41 Writing Factories for Tests	237
41.1 Produce Cheese Test Data From Factories	237
41.2 Define a <code>CheeseFactory</code>	237
41.3 Try Out <code>CheeseFactory</code>	239
41.4 Delete the Bogus Cheese	240
41.5 Commit the Changes	241
42 Why Use Factories	243
42.1 Factories and Unit Testing	243

42.2 Approach 1: Calling <code>create()</code>	244
42.3 Approach 2: Calling <code>cheeseFactory()</code>	244
42.4 Other Uses	245
42.5 Learn More About Factory Boy	246
43 Using Factories in Tests	247
43.1 Passing in Explicit Field Values	247
43.2 Bulk Generation	247
43.3 Reviewing Our Cheeses	248
43.4 Cheese Cleanup	248
43.5 Replacing the Direct Cheese Model Call in Our Existing Test	249
43.6 Removing the Name Entirely	251
43.7 Commit Our Work	252
43.8 Summary	253
44 How Cheese Objects Feel About Tests	254
45 Finding and Adding a Third-Party Countries App	255
45.1 Country of Origin	255
45.2 Benefiting From Open Source	257
45.3 Check Django Packages	258
45.4 Review Django Countries on GitHub	259

45.5 Look for the Release Number on PyPI	260
45.6 Install It	260
45.7 Implement in cheeses/models.py	262
45.8 Migrations!	263
46 Display Country Data for Cheeses	265
46.1 Add Cheese Countries in the Admin	265
46.2 Display Country in Cheese Detail	266
46.3 Run the Tests	267
46.4 Update <code>CheeseFactory</code>	268
46.5 Verify That <code>CheeseFactory</code> Works	269
46.6 Commit Our Work	270
47 Implement Cheese Creation by Users	271
47.1 Add a Cheese Creation View	271
47.2 Add the Corresponding URL Pattern	272
47.3 Specify the Desired Form Fields	275
47.4 Define the Cheese Form Template	277
47.5 Submit the Form	279
47.6 Implement <code>get_absolute_url()</code>	281
47.7 Resubmit the Form	282
47.8 Link to the Add Cheese Form	284
47.9 Commit the Changes	285

48 Use Django Crispy Forms for Prettier Display	286
48.1 Isn't <code>as_p()</code> Enough?	287
48.2 A Quick History Lesson	287
48.3 Crispy Forms and Cookiecutter Django	288
48.4 Our Non-Crispy Form	288
48.5 Use the <code>crispy</code> Template Tag	290
48.6 Reload the Add Cheese Form	291
48.7 Commit the Changes	293
49 Understand View Mixins and <code>LoginRequiredMixin</code>	294
49.1 User-Generated Content and Accountability	294
49.2 Try Accessing the Add Cheese Form Anonymously	295
49.3 Require Login	296
49.4 Try Accessing It as Cheesehead	298
49.5 View Mixins	299
49.6 Commit the Changes	299
50 Add a Creator Field and Update Our Cheese Records	300
50.1 Add a Creator Field	300
50.2 Make and Apply the Migration	302

50.3 Commit the Changes	303
51 Track and Display the Cheese Creator	304
51.1 Set the Cheese Creator After Form Validation	304
51.2 Display the Creator on the Cheese Detail Pages	305
51.3 Try It Out	307
51.4 Commit the Changes	308
52 Update the Cheese Factory	309
52.1 Run the Tests	309
52.2 Modify Cheese Factory	309
52.3 Run the Tests, Again	310
52.4 Try <code>cheeseFactory</code> in the Shell	310
52.5 Delete the Random Cheese	311
52.6 Delete the Random User	311
52.7 Commit the Changes	312
53 Update the Cheese Model Tests	313
53.1 Test <code>get_absolute_url()</code>	313
53.2 Commit the Changes	315

54 Test All the Cheese Views	316
54.1 What to Test?	316
54.2 Start With Imports	316
54.3 The First Cheese View Tests	317
54.4 Write Our First Fixture	323
54.5 Write the Cheese Create Test View	323
54.6 Really Test the Cheese List View	324
54.7 Test the Cheese Detail View	325
54.8 Test the Cheese Create View	326
54.9 Commit the Changes	327
54.10 Conclusion	327
55 Test All the Cheese URL Patterns	329
55.1 Add the Imported Cheese	329
55.2 Write Our Second Fixture	329
55.3 Test the Cheese List URL Pattern	330
55.4 Test the Add Cheese URL Pattern	331
55.5 Test the Cheese Detail URL Pattern	332
55.6 Commit the Changes	333

56 Adding a CheeseUpdateView and Recycling a Form	334
56.1 Add the CheeseUpdateView	334
56.2 Wire in the URL Pattern	335
56.3 Try It in the Browser	336
56.4 What Happened?!?	337
56.5 Link to the Update Form	341
56.6 Try It in the Browser	342
56.7 Make Login Required	342
56.8 Commit the Changes	343
57 Test the Cheese Forms and Update View	344
57.1 Refactoring our Cheese Fixture	344
57.2 Test the Add Cheese Page Title	346
57.3 Test That <code>cheeseUpdateView</code> Is a Good View . . .	346
57.4 Test That Cheese Updates Correctly	348
57.5 Run the Tests And Confirm 100% Test Coverage	349
57.6 Commit the Changes	349
58 EveryCheese is the Foundation!	350
58.1 Take the Live, Online Version of this book!	350

58.2 Level Up With Two Scoops of Django	351
58.3 Other Django Books	351
58.4 Giving Us Feedback	351
I Aftermatter	352
59 Troubleshooting	352
59.1 Troubleshooting Conda Installations on Pre-Catalina OSX	352
59.2 Troubleshooting Conda Installations on Catalina or higher OSX	352
59.3 Troubleshooting PostgreSQL: Database EveryCheese Already Exists and/or Role myuser Already Exists	353
59.4 Troubleshooting PostgreSQL: Role Does Not Exist	355
59.5 Troubleshooting PostgreSQL: Cannot Create Database	356
59.6 Troubleshooting PsycoPG2	357
59.7 Troubleshooting GCC Errors on the Mac	357
59.8 Troubleshooting the MS Visual Studio Build Tools Error	358
59.9 Navbar Isn't Dark	358

60 Acknowledgements	359
60.1 Tech Reviewers	359
60.2 Contributors to 3.x Beta	360
60.3 Contributors to 3.x Alpha	361
60.4 Changelog	362
60.5 Typesetting	362

List of Figures

1 Two Scoops Travels the World	26
2 Scientific Cheese-Based Diagram Explaining Why To Learn Django	29
3 Let's start with the basics.	50
4 It worked! Hooray!	69
5 Admin login screen	73
6 Django Admin	74
7 Greetings, Hello World	92
8 Greetings	97
9 Greeting with view method	100
10 The Tasty Cookiecutter Logo	118

11	Illustration of cheese boiler, and the location of its boilerplate	132
12	Create repository on GitHub	135
13	Stinky cheeses getting kicked out by a foot.	138
14	Registration form	141
15	Verify email page	142
16	Verification email in console window	144
17	Page with Confirm button	145
18	Minimal user profile	146
19	UserUpdateView	147
20	UserDetailView	148
21	UserUpdateView With Bio	158
22	UserDetailView Without Bio	159
23	UserDetailView With Bio	161
24	No line breaks in UserDetailView	162
25	A block of user profile cheese	175
26	Slugs are awesome!	182
27	94% coverage	192
28	Cheese model coverage	193
29	The Cheeserator device for manipulating cheese.	199

30	Cheeses on the Django admin index page	202
31	Add cheese in the Django admin	203
32	The cheese behind the curtain	207
33	TemplateDoesNotExist	220
34	Cheese list	224
35	Cheese List With Nav Link	226
36	Missing cheese_detail.html TemplateDoes- NotExist exception	232
37	Cheese detail	234
38	Django admin list of cheeses, including bo- gus cheese	240
39	A good cheese that passed its test.	254
40	Django Packages' countries grid	258
41	Django Countries README on GitHub	259
42	Cheese country data in the admin	265
43	No cheese found!	273
44	Need Fields Attribute	275
45	Cheese Form Template Does Not Exist	277
46	Add Cheese Form	279
47	No URL to Redirect To	280
48	Resubmit Cheese	282

49	Cheese Detail for Havarti	283
50	Cheese List With Add Cheese Button	285
51	Cheese form with unstyled form fields	286
52	Add Cheese Form	289
53	Add Cheese Form, Rendered With Crispy Forms	292
54	Add Cheese Form, As Anonymous User	296
55	Incognito Redirect to Sign In Page	298
56	Cheese Detail With Creator	307
57	Pre-populated Add Cheese Form	336
58	Cheese Detail With Update Button	342

I Copyright {{-}}

A Wedge of Django

Covers Python 3.8 and Django 3.x

Copyright © 2020-2021 Daniel Roy Greenfeld, Audrey Roy Greenfeld,

and Two Scoops Press, PS, Feldroy, LLC.

All rights reserved.

Published by Two Scoops Press, an individual protected series of Feldroy, LLC, a Texas series limited liability company.

539 W. Commerce St.

Suite 942

Dallas, TX 75208-1953

United States

RC AWoD-2021-06-30

The only authorized sellers of A Wedge of Django are Feldroy Shop, PS (an individual protected series of Feldroy, LLC) and the vendors listed on <https://feld.to/authorized-vendors>.

Introduction

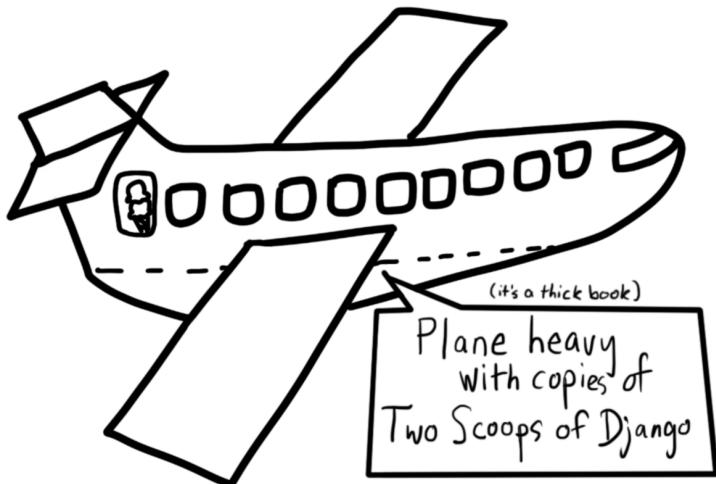


Figure 1: Two Scoops Travels the World

Once upon a time, we traveled the world doing Django training for companies. These private training sessions were custom-designed for companies needing to get engineers up and running fast with Django. Engineers who've been through this training describe them as compressing months of learning Django on your own into the span of one intense, exhausting week.

A Wedge of Django was the foundation portion of our in-person corporate training sessions. It was the first and part the second day of our week-long corporate training,

for which we charged **\$3000 an attendee**.

When people pay that much for a workshop, they make sure they're getting their money's worth. They take detailed notes, study every code sample as if it were gold, and appreciate every minute of it. They also complete every single part of the course, including the difficult or tedious parts.

If you follow that same pattern with this book you will learn. Trust us.



Take This Course Online!

Want to experience this course (and others) in a live, online, interactive setting? We offer it as a 2.5 day event once every three months. Find out more information at <http://feld.to/dcc-live>.

Why Learn Django?

In today's world, one might ask what's the point of learning a 17-year old application framework. Here are three reasons why:

Reason #1 - Django is Mature

Started in 2003, Django isn't some new framework that hasn't accounted yet for all common edge cases. Rather

Django's maintainers have handled those edge cases at least once. Instead, the maintainers are worried that Django is now old enough to be interested in dating and has a US driving learner's permit.

We like to think of Django like a delicious aged cheese like **Boerenkaas Gouda**, with a sweet, intense flavor that only comes with maturity.

Reason #2 - Django is Python

Python is an immensely popular programming language and is by far #1 in the field of data science. Python is easy to learn yet powerful in execution. It has a gigantic global community of users and hundreds of thousands of libraries to draw on, including Pandas, Numpy, Keras, and of course, Django.

We like to think of Python like **Mozzarella**, arguably one of the most popular cheeses in the world. Its use in pizza as the base cheese makes mozzarella such a universal cheese much in the same way that Django's use of Python makes it such a universal platform.

Reason #3 - Django is Proven

No tool remains popular for as long as Django unless it proves itself. Because it is so proven, Django is relied on by startups building dreams, science and fintech efforts

presenting their data, and of course, tech giants like Instagram.

We like to think of Django as the cheese on pie-sliced pizza, a dish proven around the world. No matter where one goes on the planet, a slice of cheese pizza is always available. That's because pizza is proven, much like Django is proven as an application framework.

Now that we know why to learn Django, let's get started!

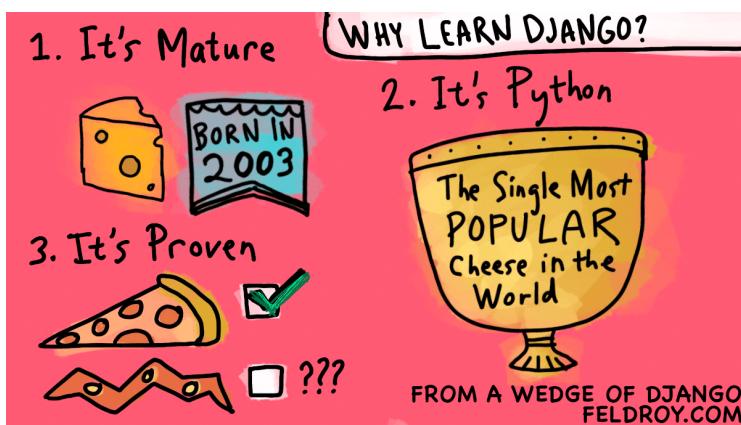


Figure 2: Scientific Cheese-Based Diagram Explaining Why To Learn Django

When the Going Gets Tough

We've tried to minimize the tedious parts, but sometimes it takes getting past that sort of material to break through to the fun stuff. We've also tried to keep the difficulty level manageable, but sometimes you'll find things you don't

understand, and you'll have to experiment and research concepts.

We've put so much love over the years into these materials, iterating and hand-crafting them with the utmost care. Yet you may find errors or omissions. This is not the final version of the book, so remember, you can help us make it even better.

We hope you have fun with this book. We've had fun putting it together for you.

This Book is intended for Django 3.x and Python 3.8

This book should work well with the Django 3.x series, less so with Django 2.x and so on. As for the Python version, this book is tested on Python 3.8. Most code examples should work on Python 3.7.x and 3.6.x, but some exceptions may have to be worked through.

How This Book is Structured

After this introduction chapter, A Wedge of Django¹ is structured in three parts:

The Ultimate Django Setup Starting and ending in chapter 1, our simple but comprehensive guide for setting up computers for building Django projects.

¹<https://www.fieldroy.com/products/django-crash-course>

Hello, World in Django Beginning in chapter 2, getting comfortable with the basics of Django development.

Enter the EveryCheese Project Starting at chapter 15, a gentle but deep introduction to techniques and patterns used by professional Django developers. It's also the foundation of the Extension Series.

Conventions Used in this Book

We use code examples like the following throughout the book:

```
class Cheese:
    def __init__(self, aged=False):
        self.aged = aged
```

To keep these snippets compact, we sometimes violate the PEP 8 conventions on comments, line spacing, and line length. Code samples are available at:

<https://github.com/feldroy/django-crash-course/tree/master/code>

We use the following typographical conventions throughout the book:

- Constant width for code fragments or commands.
- *Italic* for filenames.

- **Bold** when introducing a new term or important word.

Boxes containing notes, warnings, tips, and little anecdotes are also used in this book:



Something You Should Know

Tip boxes give handy advice



Some Dangerous Pitfall

Warning boxes help you avoid common mistakes and pitfalls.

We also use tables to summarize information in a handy, concise way:

Table 1: Three Delicious Kinds of Cheese

Name	Country of Origin	Hardness
Parmesan	Italy	hard
Cheddar	England	semi-soft
Raclette	Switzerland/France	semi-hard

2 The Ultimate Django Setup

This is how we like to set things up on our computers, as of 2020. Daniel uses a MacBook Air, and Audrey uses a MacBook Pro. We also share a Microsoft Surface Pro for Windows development and testing.

We've worked through Django setup countless times, both for ourselves and for students. Over the years our preferences have evolved. This is the latest iteration of how we like to set up our computers (and those of our students and readers) for Django development.



Got a problem with installation?

We've put a lot of work into this chapter, yet we know from experience that some people will run into challenges, or bugs, and errors. In that case, please open an issue at <https://github.com/feldroy/django-crash-course/issues> and we'll do our best to help you.

2.I Goal: A Professional-Quality Development Environment

The goal of this chapter is to get our computer fully set up for optimal Django development on Mac or Windows. This isn't just a toy set up for educational purposes; it's the real setup used by professional web developers and data scientists, and one which we can use to develop real-world Django projects.

Make sure we:

- Are comfortable using the command line. If not, work through one of the command line tutorials below:
 - Mac/Linux: <https://www.amazon.com/Linux-Command-Line-2nd-Introduction/dp/1593279523>
 - Windows: https://www3.ntu.edu.sg/home/ehchua/programming/howto/CMD_Survival.html
- Have a computer with at least a gigabyte (1 GB) of hard drive space.
- Have a fast internet connection. We're going to be here for a little while.

2.2 Core Concepts

2.2.1 Keep Tooling Updated

This book encourages us to use the latest versions of all available tools. If we have an outdated version of any particular tool, upgrade it whenever possible. Don't worry, this guide shows how to make those updates.

2.2.2 Perfection Is Impossible

When installing developer tools, it's extremely common for an install to fail partway, leaving crumbs on our system. While it can be frustrating, it's part of life as a developer. The more experience we gain as a developer, the more we learn to live with an imperfect system full of crumbs from half-installed tools all over our operating system's nooks and crannies. It's alright, though. If it's any consolation, as we gain more experience with Django and related tooling, we'll gain an intuition for when we can delete these crumbs.

We don't need a pristine, untouched operating system to set up Django and all its dependencies properly. We've had students purchase expensive new computers before our training, then panic when they accidentally install something the wrong way. Please don't buy a new laptop just to work with Django. Try installing these tools

on whatever computer that's available at home, school, or work.

Some may think their setup is too broken and unrecoverable to work with. There's a 99% chance that's not true. Just work with it and let's give it our best try.

2.2.3 Take Detailed Notes

We highly recommend keeping a journal of:

- All the commands typed in at the command line, and their results
- What is installed via GUI installers

This can be as simple as a text file in a `notes` directory named `2020-02-01.txt`.

Detailed notes are our best guard against things going wrong. Even if our notes are mostly just copy-pasted commands and output from our terminal that we don't understand, that's alright. The important thing is that we maintain a record of changes.

With that, let's get started!

2.3 Step I: Install Visual Studio Code

Visual Studio Code (VSCode) is a source code editor. Developed by Microsoft as an open-source project

(github.com/Microsoft/vscode²), according to Stack Overflow's 2019 Developer Survey it is the most popular developer environment tool in the world. (reference: insights.stackoverflow.com/survey/2019³).

We like VSCode because it is easy to install, features built-in source control management, and has a huge number of extensions.



If VSCode is already installed. Then skip to section on [Step 2: Installing Git](#).

2.3.1 Installing VSCode on Windows

1. Go to <https://code.visualstudio.com/download>.
2. Click the Windows download button.
3. On the computer double-click the .exe file.
4. Follow the instructions on the screen.
5. On the `Select Additional Tasks` step of the installer, mark the options:
 - Add “Open with Code” action to Windows Explorer file context menu.
 - Add “Open with Code” action to Windows Explorer directory context menu.
 - Register Code as an editor for supported file types.

²<https://github.com/Microsoft/vscode>

³<https://insights.stackoverflow.com/survey/2019>

6. Restart Windows after the installation is finished.
7. From the Start menu, open Visual Studio Code.



If we do not restart Windows.

If Windows is not restarted, the option to select Visual Studio Code as the default editor for Git may not be available when installing [Git for Windows](#)

2.3.2 Installing VSCode on Mac

1. Go to <https://code.visualstudio.com/download>.
2. Click the Mac download button.
3. On the computer double-click the .zip file.
4. Move the file inside to the Applications folder.
5. Launch Visual Studio Code from the /Applications directory.
If a dialog box appears that says, “*Visual Studio Code* is an app downloaded from the Internet. Are you sure you want to open it?, click on the **open** button.
6. On View menu, click into *Command Palette* (Shift + Command + P) and type ‘shell command’ to find the **Shell Command: Install 'code' command in PATH** Command.



Note about Safari Browser.

On step 3, after downloading the zip file with Visual Studio Code, it may not be possible to find the zip file after downloading it. If that is the case, look for a directory or an unzipped file. This is because by default Safari will automatically open some file types.

Users of Safari can change this behavior by unmarking the option `Open "safe" files after downloading` on *Safari -> Preferences* menu.

2.3.3 Installing VSCode on Linux

1. Open a terminal, and run

```
wget --quiet -O- https://feld.to/vscode-ubuntu-installer | bash
```

We will need to enter our Linux user's password, and after that, the installation will take place automatically.



To easily open a terminal on Ubuntu press the keyboard shortcut: **CTRL + ALT + T**

2.4 Step 2: Installing Git

Git⁴ is the predominant version control system used by the professional software development community.



If Git is already installed and configured.

Then skip to [Step 3: Installing Conda](#).

2.4.1 Installing Git on Windows

1. Go to <https://gitforwindows.org/>
2. Click on the download button.
3. On the computer double-click the .exe file.
4. Follow the instructions on the screen.
5. On the choosing the default editor used by Git step of the installer, select the option `Use Visual Studio Code as Git's default editor` and continue the installation.
6. To confirm that the git was installed successfully, Open a terminal (CMD) and type `git --version` and press ENTER, then we should get an output like:

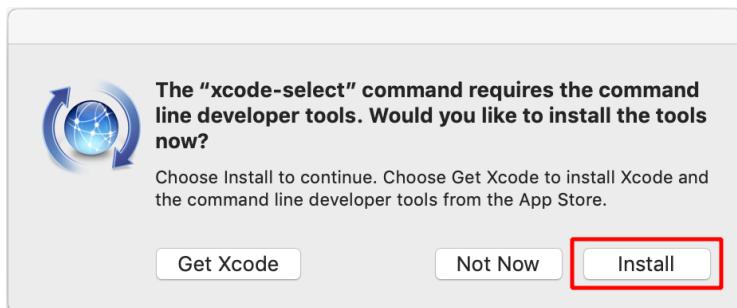
⁴<https://git-scm.com/>

```
git version 2.xx.x
```

2.4.2 Installing Git on Mac

To install Git on Mac we need to install the Apple Command-line tools for Xcode.

1. Open a terminal.
2. Type `xcode-select --install` and press ENTER.
3. Click on the `install` button.



4. Accept the license agreement.
5. Depending on our internet speed, installation may take some time. When the installation is finished, click on the `Done` button.
6. To confirm that the git was installed successfully, type `git --version` and press ENTER, then we should get an output like:

```
git version 2.xx.x
```

2.4.3 Installing Git on Linux

1. Open a terminal.
2. Type `sudo apt-get update` and press ENTER. Insert the password of your Linux user if it was asked.
The `sudo apt-get update` will update the local packages information database to the latest version.
3. Type `sudo apt-get install git --yes` and press ENTER to install `git`.
4. To confirm that the `git` was installed successfully, type `git --version` and press ENTER, then we should get an output like:

```
git version 2.xx.x
```

2.4.4 Configuring Git

To use `Git`, it is also necessary to inform `Git` which `name` and `email` address it will use to identify who is the author of a certain change.

To configure `Git`, first, we'll need access to the command line:

- Windows: Open the Start Menu and choose Git Bash.
- Mac/Linux: Open a terminal window.

```
git config --global user.name "replace with your full name"  
git config --global user.email "replace with email address"
```

To confirm that the git was configured successfully, type `git config --list` and press ENTER, then we should get an output that contains `user.name` and `user.email`, like

```
user.name=Your full name here  
user.email=your email address here
```



Don't Skip This Step!

This step is important. We use git extensively throughout this book, it is a necessary skill for every developer.

2.5 Step 3: Installing Conda

Conda is the tool we use for virtual environments.

Virtual environments allow us to do things like run one project with Django 3.2 and another with Django 2.2 on the same machine.



Is Conda already installed?

Here's how to check:

- Windows: Open the Start Menu and choose the Anaconda Prompt. If we find it, that means Conda is installed.
- Mac: Open a terminal and type `conda -v`.
 - * If the response says something like `conda 4.8.1`, Conda is installed and now we know its version.
 - * If that doesn't work, type `source ~/.bash_profile` and hit return. Once that's done, try `conda -v` again.

If any of these are true, skip forward to [Updating Existing Conda Installation](#).



Want to use pip, virtualenv, venv, or Poetry instead of Conda?

More advanced readers might already be proficient with tools like **virtualenv**, **poetry**, **docker**, or other environmental control tools. If that's the case, they can use those tools with small adjustments to the instructions we provide. Do note that we cannot provide any dependency-related support for people using the book with a tool that isn't Conda.

2.5.1 Installing Conda on Windows

1. Go to <https://docs.conda.io/en/latest/miniconda.html#windows-installers>
2. Find the Python 3.8 versions.
3. Click the link with 64-bit in the name and download the executable file.
4. On the computer double-click the .exe file. If we are unsure about any setting, accept the defaults. We can change them later.
5. Follow the instructions on the screen.
6. When installation is finished, from the Start menu, open the Anaconda Prompt.
7. In the Anaconda prompt, type `conda list` and press enter.

2.5.2 Installing Conda on Mac

1. Go to <https://docs.conda.io/en/latest/miniconda.html#macosx-installers>
2. Find the Python 3.8 versions.
3. Click the link with 64-bit pkg in the name and download.
4. On the computer double-click the .pkg file.
5. Follow the instructions on the screen. If unsure about any setting, accept the defaults. We can change them later.
6. When the installation is finished, open a terminal

window.

7. Type `conda list` and press enter.

If there is a problem, depending on the Mac go to:

- Troubleshooting Conda Installations on Pre-Catalina OSX
- Troubleshooting Conda Installations on Catalina or higher OSX

2.5.3 Installing Conda on Linux

1. Go to <https://docs.conda.io/en/latest/miniconda.html#linux-installers>
2. Find the Python 3.8 versions.
3. Click the link with 64-bit in the name and download.
4. On the computer, open a terminal. Go to the directory where you saved the file: `cd ~/Downloads` Make the file executable: `chmod a+x Miniconda3-latest-Linux-x86_64.sh` Execute the file: `./Miniconda3-latest-Linux-x86_64.sh`.
5. Follow the instructions on the screen. Type yes when asked for init. Otherwise, if unsure about any setting, accept the defaults. We can change them later.
6. When the installation is finished, close the current terminal window and open a new one.
7. Type `conda list` and press enter.

2.5.4 Updating Existing Conda Installation

To update a Conda installation, enter this command:

```
conda update conda
```

2.6 Step 4: Installing PostgreSQL

2.6.1 Installing PostgreSQL on Windows

For the time being, we can't support installations of PostgreSQL on Windows. Fortunately, the Sqlite engine is part of the Python installation of Windows. That is what Windows users should use in this course.

2.6.2 Installing PostgreSQL on Mac

The best way to install PostgreSQL on the Mac is by following these steps:

1. Go to <https://postgresapp.com>
2. Click on the “Downloads” tab
3. Choose the most recent release and download it.
This should be the first download option on the page
4. Once downloaded, open the `dmg` file locally

5. Move the **Postgres** file in the opened `dmg` to our `/Applications` folder like we would any other application
6. Launch Postgres from the `/Applications` directory.
If a dialog box appears that says, “*Postgres* is an app downloaded from the Internet. Are you sure you want to open it?, click on the `open` button.
7. Click on the `Initialize` button to the Postgres app creates the PostgreSQL cluster configuration directories and files.
8. Once that’s done, we need to configure our `$PATH` so that we can use PostgreSQL easily on the command-line. To do that, execute this in the terminal:



For the next commands work as expected, we should press the `ENTER` key after typing the `\` character.

```
sudo mkdir -p /etc/paths.d && \
PART1=/Applications/Postgres.app/Contents
PART2=/Versions/latest/bin
echo $PART1$PART2 | sudo tee /etc/paths.d/postgresapp
```

We need to close the terminal window and open a new one for the changes to take effect.

2.6.3 Installing PostgreSQL on Linux

1. Open a terminal, and run

```
wget --quiet -O- https://feld.to/pg13-ubuntu-installer | bash
```

We will need to enter our Linux user's password, and after that, the installation will take place automatically.

2.7 Summary

Getting a computer setup for development is a bit of work, but worth the effort. Now that we're done, let's move on and crash into coding some Django!

3 Hello, World in Django



Figure 3: Let's start with the basics.

When most people learn how to ride a bicycle, they are given the simplest model. Only one gear, simple brakes, and nothing more. Yet the skills we learn on that bicycle stay with us for the rest of our life.

Think of this project as our first bicycle. It's not fancy, it won't do much. Yet it's going to provide us with the skills needed to start truly understanding Django.

In programming terms, minimal projects like this are

called “Hello, World” programs⁵.

3.1 The Hello, World of Python

For example, this is a “Hello, World” program in Python.

```
print("Hello, world!")
```

We would save it as `hello.py` and then run it by typing `python hello.py` at the command line. Then it would print out:

```
Hello, world!
```

As we can see, this program is not a very useful program. It doesn’t do anything except show us how to print a string. But it’s useful if we’re a programmer new to Python, trying to understand what a Python program looks like. It teaches us:

- How a very simple Python program is run
- The basic Python syntax for printing the string “Hello, world!”
- Whether Python is available to run Python programs, or if we have a problem with our setup

Likewise, a “Hello, World” Django project would teach us how a very simple Django project works and is run, and it can help us make sure that we have Django set up correctly.

⁵https://en.wikipedia.org/wiki/%22Hello,_World!%22_program

3.2 Hello, World of Django: Simple on Purpose

The project will be simple, but it will expose the foundations needed to begin your Django journey. We'll want to keep it on hand as a reference project, one that we go back to on later projects to remind ourselves how the basics are done.

What we won't be doing is expanding this **Hello, World** project to become a real-world project that does anything meaningful. Just like we wouldn't use our first bicycle to enter a serious race, this project isn't meant for serious use. The foundations just aren't there in the boilerplate code.

Later in this book, we'll show us how to develop a realistic, fully-featured Django project. Be patient. Study the foundations, and we'll be equipped to get even more out of the later projects in this book. Onward and upward!

3.3 Projects Goals

- Get a taste of working with Conda, pip, and Django together
- Understand the minimum Django project
- Understand a minimal Django app
- Use a minimal Django template

- Understand the minimal template components: context data and method calls

3.4 Prerequisites

We will need to have our computer ready for software development. If we don't have it ready, go to [The Ultimate Django Setup](#) chapter.

3.4.1 Still Learning to Code?

This book is best once we already know at least a little programming. If we are still learning to code, we will benefit from going through at least one intro to programming tutorial before this one.

It's fine if we don't know everything about Python or HTML when we start this project. Just having a taste of writing simple Python and HTML is fine. We will learn a lot just by following along and doing the exercises.

4 Preparation: Install Dependencies

A **dependency** is a package that a project requires in order to run. In our case, the only dependency we will need to install is Django.

4.1 Set Up a New Conda Environment

Create and activate a **hellodjango** conda environment. First, we'll need access to the command line:

- Windows: Open the Start Menu and choose Anaconda Prompt.
- Mac and Linux: Open a terminal window.

In the terminal window, type the following:

```
conda create -n hellodjango python=3.8
```

The screen will then display a message that should look similar to this:

```
The following NEW packages will be INSTALLED:
```

```
<snip for brevity>
```

```
Proceed ([y]/n)?
```

Type y and hit return.

Depending on our computer, internet speed, and if we have done this before, the computer will process for 5 seconds to 5 minutes, then return us to the command-line. Once that's done, we type this at the command-line:

```
conda activate helldjango
```

After that, our command-line should look like something like:

```
(helldjango) $
```

If it's not exactly what we see above, that's okay. Just so long as it's prefixed with `(helldjango)`, then it's been done correctly.

Until we finish this project, we should always keep the `helldjango` conda environment active.



If We Lose the Hellodjango Terminal

Don't worry. We can reactivate the conda environment with these simple instructions:

- Windows: Open the Start Menu and choose Anaconda Prompt.
- Mac and Linux: Open a terminal window.

In the terminal window, type the following:

```
conda activate hellodjango
```

4.2 Install Django

Install the latest version of Django into our conda env:

```
pip install django
```

We should see something like this happen:

```
Collecting django
  Downloading Django-3.1-py3-none-any.whl (7.8 MB)
[██████████| 7.8 MB 8.9 MB/s]
Collecting asgiref~=3.2.10
  Downloading asgiref-3.2.10-py3-none-any.whl (19 kB)
Collecting pytz
  Downloading pytz-2020.1-py2.py3-none-any.whl (510 kB)
```

```
[...]  
| 510 kB 13.0 MB/s  
Collecting sqlparse>=0.2.2  
  Downloading sqlparse-0.3.1-py2.py3-none-any.whl (40 kB)  
[...]  
| 40 kB 24.2 MB/s  
Installing collected packages: asgiref, pytz, sqlparse, django  
Successfully installed asgiref-3.2.10 django-3.1 pytz-2020.1  
sqlparse-0.3.
```

4.3 Confirming Django is Installed

Do this at the command-line:

```
python -m django --version
```

If Django was installed successful, typing that will produce a value that looks similiar to one of the values listed below:

- 3.1.2
- 3.2.8

4.4 Summary

We just created the `hellodjango` conda environment. While it doesn't seem like much, this is a big deal. As we make more and more projects, isolating versions of libraries like Django from project to project becomes very important.

We also installed Django and its dependencies. This gave us a tool called `django-admin`, which lets us run management commands. We'll be using that in the chapters ahead.

5 Generate a Starter Django Project

First, let's make certain we are within the `hellodjango` conda environment. To do this, check that the command-line is prefixed with `hellodjango`. Once we've done that, type the following:

```
django-admin startproject hellodjango
```

This creates a minimal Django project called `hellodjango`. Type `ls` on Mac or `dir` on Windows to check that there's now a `hellodjango` directory in our current working directory.

5.1 Anatomy of a Bare-Bones Django Project

Open the newly-created project in Visual Studio Code. At the command-line, type:

```
code hellodjango
```

Take a look at the files that were created. We should see a file structure that looks like this:

```
helldjango
├── helldjango
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

This is the minimal Django project that `startproject` creates by default.

Let's look at a few modules.

5.2 manage.py

This `manage.py` module is something we call to run various Django commands that we'll be teaching during this project and other projects to follow. If we look inside, it will look something like this:

```
#!/usr/bin/env python
"""
Django's command-line utility for administrative tasks.
"""

import os
import sys

def main():
    os.environ.setdefault('DJANGO_SETTINGS_MODULE',
```

```
'helldjango.settings')

try:
    from django.core.management import execute_from_command
except ImportError as exc:
    raise ImportError(
        "Couldn't import Django."
        "Are you sure it's installed?"
        "Did you forget to use a virtual environment?"
    ) from exc
execute_from_command(sys.argv)

if __name__ == '__main__':
    main()
```

Don't worry if this code doesn't make any sense at this time. In fact, for most Django projects this code is never touched by developers.

5.3 settings.py

The `settings.py` module is much larger than the `manage.py` module. It's where we can globally change Django's behavior via settings. These settings are special variables identified by their `UPPERCASE` nature. In Django this is more than just a convention, all formally defined settings must be `UPPERCASE` and are considered to be `constants`.



How Django Settings Are Constants

While Python normally doesn't allow for constant-style variables, this is a special case. The way that the `django.conf.settings` module works, Django needs to restart for their value to be picked up by the rest of a project. In local development this can happen so fast we barely notice, which is convenient there.

Deployment into QA and production is a different story, but that topic is outside the scope of this book. We'll cover it in our forthcoming deployment extensions.

Here's an important setting, `SECRET_KEY`:

```
# SECURITY WARNING: Keep the Secret Key Used
#                     in Production Secret!
SECRET_KEY = '39)b-_1aga9eauyda((b^4+1hmao'
```

Every time a new Django project is created, a new `SECRET_KEY` is generated as well. This is part of the security system of Django, and this value is critical for keeping a Django project secure and unhacked. If that sounds like a heavy responsibility, that's because it is. In later projects we'll cover how to safely manage `SECRET_KEY` and other security related settings.

In fact, let's cover another important setting, `DEBUG`:

```
# SECURITY WARNING: Don't Run With Debug
#
#                         Turned on in Production!
DEBUG = True
```

When `DEBUG` is set to `True`, we get very sophisticated and very understandable debug messages while we are developing. In production, however, it must be set to `False`.



Any Django setting with a `SECURITY WARNING` must be taken seriously. Doing so ensures that our projects are much safer from evil people who want to do bad things.

5.4 Other Settings

One thing to note is that many sections inside `settings.py` have comments with links to the official documentation. As we get more familiar with Django, we'll find these links invaluable. Example:

```
# Internationalization
# https://docs.djangoproject.com/en/3.1/topics/i18n/

LANGUAGE_CODE = 'en-us'

TIME_ZONE = 'UTC'
```

```
USE_I18N = True
```

```
USE_L10N = True
```

```
USE_TZ = True
```

If a setting or group of settings doesn't have a reference to the Django documentation, it's still possible to look it up. In many cases we can enter the word "Django" and the name of the setting into Google or other search engines and get a result.

Try this with `ALLOWED_HOSTS`.

5.5 Exercise

Look up each setting from `settings.py` in the following docs:

- Settings topic overview⁶
- Settings reference docs⁷

For now, don't change anything. Instead, take a few minutes to jot down notes on the ones we think are interesting or important.

In the future we'll modify these settings to empower our projects to do great things.

⁶<https://docs.djangoproject.com/en/stable/topics/settings/>

⁷<https://docs.djangoproject.com/en/stable/ref/settings/>

6 Create a Database and Run the Project

All Django projects use databases to store persistent data supplied by users. Even this “hello, world” project follows this pattern.

6.1 Ensure We Are in the `hellodjango` project

If not done so yet, `cd` into the outer `helloworldjango` folder:

```
cd hellodjango
```

List the contents of the directory, using `ls` on Mac/Linux or `dir` on Windows, to ensure we are in the same directory as `manage.py`. We should see this output:

```
helloworldjango manage.py
```

The current directory should contain an inner `helloworldjango` directory and a `manage.py` file.

6.2 Creating the Database

When we run the `migrate` command for the first time, a database gets created, and some starter tables in the database are set up.

At the command line, type:

```
python manage.py migrate
```

We will see output that looks something like this:

```
Operations to perform:
  Apply all migrations: admin, auth,
                      contenttypes, sessions

Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_a... OK
  Applying admin.0003_logentry_add_acti... OK
  Applying contenttypes.0002_remove_con... OK
  Applying auth.0002_alter_permission_n... OK
  Applying auth.0003_alter_user_email_m... OK
  Applying auth.0004_alter_user_usernam... OK
  Applying auth.0005_alter_user_last_lo... OK
  Applying auth.0006_require_contenttyp... OK
  Applying auth.0007_alter_validators_a... OK
```

```
Applying auth.0008_alter_user_usernam... OK
Applying auth.0009_alter_user_last_na... OK
Applying auth.0010_alter_group_name_m... OK
Applying auth.0011_update_proxy_permi... OK
Applying sessions.0001_initial... OK
```

Now if we type `ls` on Mac/Linux or `dir` on Windows, we will see this output:

```
db.sqlite3 helldjango manage.py
```

What's that new file? `db.sqlite3` is the database that got created.

6.3 Start Up Runserver

At the command line, type this to start up Django's development server:

```
python manage.py runserver
```

We should see something like this:

```
Watching for file changes with StatReloader
Performing system checks...
```

```
System check identified no issues  
(0 silenced).  
January 25, 2020 - 19:34:39  
Django version 3.x  
using settings 'helldjango.settings'  
Starting development server at:  
http://127.0.0.1:8000/  
Quit the server with CONTROL-C.
```

This means that the development server is now running on our computer.

6.4 Runserver Has Started Up

To go to our Django site, visit <http://127.0.0.1:8000> in a browser.

6.5 Success!

If we see the image below (or on the next page), our Django project is now running on runserver (the local development server).

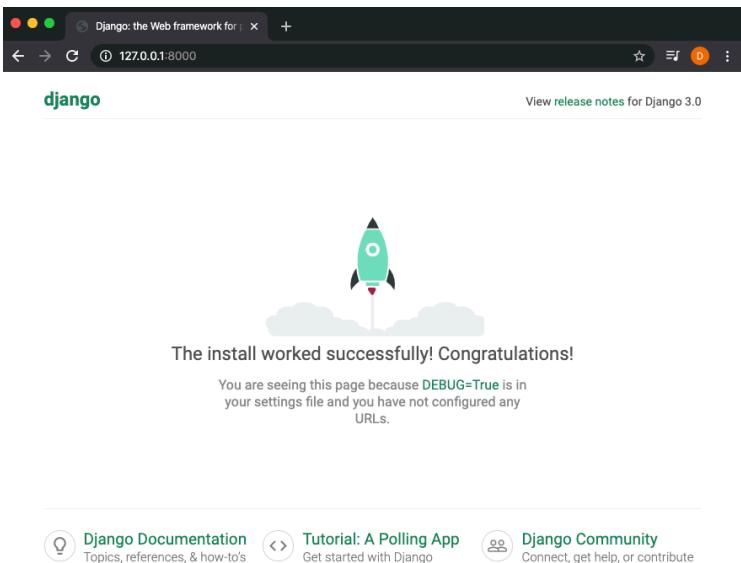


Figure 4: It worked! Hooray!

7 Create a Django Superuser

In order to log into the Django admin, we need to create a Django superuser. Ordinary users will not be able to access the admin.

7.1 Stop runserver if It Is Still Running

Go to your terminal window. If runserver is still running, use CTRL-c to stop it.

Note: Make sure to press *Control-c*, not *Command-c*.

7.2 Run createsuperuser

Enter a username, email, and password for our Django superuser. For this minimal project, use the values of `admin` for the username, `admin@example.com` for the email, and create an eight (8) or more character password.

```
python manage.py createsuperuser
Username (leave blank to use 'scoopy'): admin
Email address: admin@example.com
Password:
Password (again):
Superuser created successfully.
```



Copy this password into a password manager!

It's bad practice to use simplistic passwords, even for local development. We want to promote strong security habits and this is one of them. Also, don't stick passwords into plaintext or spreadsheet files. Instead, lean on secure password managers like our preferred **1Password** or open source options such as **Bitwarden** and **KeePass**.

7.3 Summary

We just created a superuser for a Django project. Out of the box this superuser has access to Django's admin tool. This allows them to make changes to database records, including the user list. As one might imagine, this is a very powerful role and not to be taken lightly.

8 Practice Using the Admin

8.1 Restart runserver

If `runserver` isn't on, restart it in the shell:

```
python manage.py runserver
```

8.2 Visit the Admin

In our browser, go to <http://127.0.0.1:8000/admin/> (the previous URL + `/admin/`).

We should see the admin login page below or on the next page:

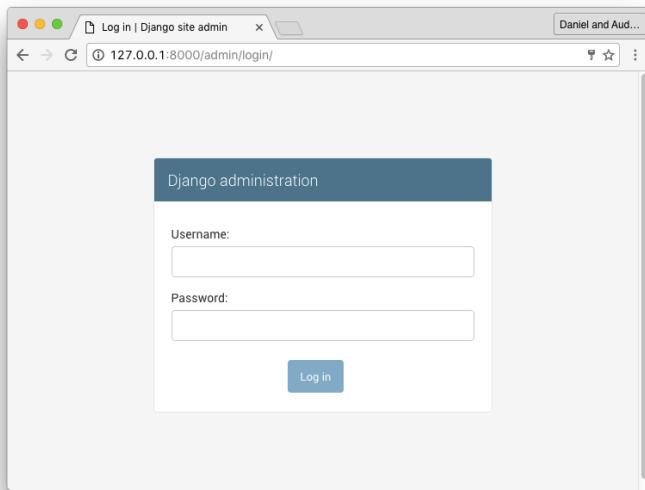


Figure 5: Admin login screen

Since we are not logged in, the Django admin redirects us to its login screen.

8.3 Log in as Our Superuser

Enter the Django superuser's username “`admin`” and password now. Then click “Log In”.

8.4 Explore the Admin

In a barebones Django project, the admin is empty except for the Authentication and Authorization tools:

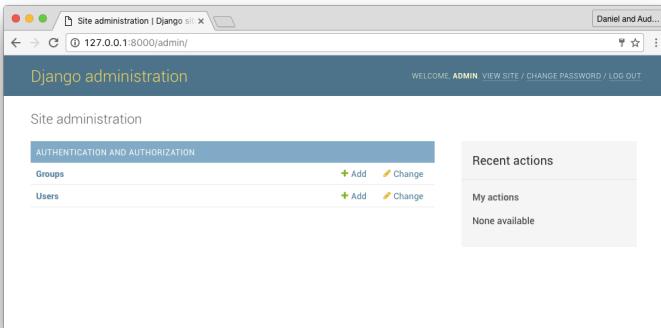


Figure 6: Django Admin

Explore everything in the Admin. Click on various things, try entering data, and get a feel for what the Django admin has provided so far. Don't worry if things break, we can and will quickly recreate the database. In fact, we cover that in the next section.

8.5 Recreating the Hello, World in Django Database

Let's recreate the database.

8.5.1 Step 1: Stop Runserver

At the command-line, type `ctrl-c`. This should stop the server.

8.5.2 Step 2: Delete the Database

In Visual Studio Code, find and right-click on the `db.sqlite3` database file. Choose the `Delete` option.

8.5.3 Step 3: Recreate the Database and Superuser

At the command-line, run `migrate` again:

```
python manage.py migrate
```

Once the database is created, we can recreate the superuser:

```
python manage.py createsuperuser
```

8.6 Summary

In this chapter we familiarized ourselves with not just the Django admin, but also creating the superuser. We even deleted the database and recreated the Django admin

user. This should give us confidence about exploring systems in the future. Part of being a good developer is knowing that if we can restore a system easily, that gives us the luxury of being able to explore how to break it.

9 Generate a Home-page App

9.1 Stop runserver if Needed

If we still have the Django development server running, stop it with CTRL+c.

9.2 Generate a Starter Django App

Use the `startapp` management command to create a new Django app called `homepage`:

```
python manage.py startapp homepage
```



If something goes wrong

If this doesn't work, double-check that we're still inside the outer `hellodjango` project directory, at the same level as `manage.py`.

9.3 Anatomy of a Simple 1-App Django Project

The `startapp` command created a new directory in our project called `homepage`.

Switch back to Visual Studio Code. Expand the `homepage` directory to see what files it contains.

We should see this directory/file structure:

```
helldjango
├── db.sqlite3
└── helldjango
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── homepage
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── manage.py
```

9.4 Exercise

What files are in the new Django app created by startapp? Study the files of our new `homepage` app. Don't touch them yet, just get familiar with the code inside each one.

9.5 Main Django Project vs Django Apps

- A Django project is a folder containing all the code (Python, templates, etc.) needed to run a website.
- A Django app is a folder within a Django project.
- Django apps are isolated components that do one specific thing in a Django project.
- Combining Django apps is one of the primary things we do when we build a Django project.

10 Prepping the Project for Templates

Before we can start writing views that use templates, we need to modify our project's settings.

10.1 Configure Templates

Search the `settings.py` module for the `TEMPLATES` setting. It's a list of dictionaries that looks like this:

```
TEMPLATES = [
{
    'BACKEND': 'django.template.backends.DjangoTemplates',
    'DIRS': [],
    'APP_DIRS': True,
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.msg',
        ],
    },
},
```

```
 ]
```

Notice how `DIRS` is an empty list. We can change that to specify where we want Django to look for templates.

Go ahead and modify that slightly, by adding `BASE_DIR / "templates"`, to the `DIRS` list on the fourth line. Here's how it should look:

```
TEMPLATES = [
{
    'BACKEND': 'django.template.backends.djangoproject.DjangoTemplates',
    'DIRS': [BASE_DIR / "templates", ],
    'APP_DIRS': True,
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.messages',
        ],
    },
},
]
```

This change tells Django to look inside a `templates` directory within the base directory of our project, which we'll create later in this book.

One might notice that `BASE_DIR / "templates"`, is not a typical Python string concatenation. Specifically, `pathlib.Path` overwrites the slash (/) to create child paths. To learn more, visit <https://docs.python.org/3/library/pathlib.html>.

10.2 Save!

As always, let's not forget to save our work! Save `settings.py` so that Django's development server picks up the change we made.

10.3 Lists? Dictionaries?

We mentioned some Python terms in this lesson. While we assume in this book that readers already know the basics of Python, here's a recap in case we forgot what these terms mean.

10.3.1 What's a Python List?

In Python, a list consists of comma-separated values wrapped in square brackets. For example:

```
['Ice cream!', 2, 'cones', 3, 4]
```

Lists are one of the most important and frequently-used data types in Python. If we're new to lists, we'll greatly

benefit from going through the Lists section of the official Python tutorial⁸. The rest of that tutorial is highly recommended.

10.3.2 What's a Python Dictionary?

A set of **key: value** pairs.

For example, here's a dict of ice cream flavors, where the keys are the flavors in English and the values are the corresponding flavors in Spanish:

```
{  
    'vanilla': 'vainilla',  
    'chocolate mint': 'choco menta',  
    'coffee': 'café',  
}
```

If not familiar with dicts in Python, we strongly recommend going through the Dictionaries section of the official Python tutorial⁹. Type the examples into the Python shell and get comfortable adding and retrieving values.

⁸<https://docs.python.org/3/tutorial/introduction.html#lists>

⁹<https://docs.python.org/3.8/tutorial/datastructures.html#dictionaries>

II Add a Simple Home-page View

Open the `views.py` file of the `homepage` app. Remove the code that is there and replace it with the following:

```
from django.views.generic import TemplateView

class HomepageView(TemplateView):
    template_name = 'index.html'
```

Congrats, we've written our first Django view! What we've done:

- We created the Python class `HomepageView`
- `HomepageView` subclassed `TemplateView`. Django comes with a number of built-in views, which we can inherit from. We use these built-in views throughout the rest of this book
- Associated the `index.html` template with the class

II.I Get to Know URL Routing

In the same inner `hellodjango` directory containing `settings.py`, there's also a file called `urls.py`. Let's open it up in

the text editor.

Notice that it defines a list of `urlpatterns`:

```
"""Hellodjango URL Configuration

The `urlpatterns` list routes URLs to views.
For more information please see:
    https://docs.djangoproject.com/en/3.1/topics/http/urls/
Examples:
Function views
1. Add an import: from my_app import views
2. Add a URL: path('', views.home, name='home')
Class-based views
1. Add an import: from other_app.views import Home
2. Add a URL to urlpatterns:
    path('', Home.as_view(), name='home')
Including another URLconf
1. Import the include() function:
    from django.urls import include, path
2. Add a URL: path('blog/', include('blog.urls'))
"""

from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Right now, there's only one item in the `urlpatterns` list: a URL pattern for `admin/`. That corresponds to the Django admin UI, which we saw earlier. When a browser requests a page starting with `admin/` (i.e. matching the regular expression '`admin/`'), it gets routed by the server to `admin.site.urls`.

In a typical Django project, this file will be full of URL routes. But it starts almost empty.

II.2 Add a URL Pattern for the Home-page

First, import `HomepageView` into `urls.py`. Right below the Django imports in `urls.py`, add this line:

```
from homepage.views import HomepageView
```

Then add this URL pattern to the `urlpatterns` list, right after the one for the Django admin:

```
path('' , HomepageView.as_view() , name='home') ,
```

What we've done thus far:

- When a browser requests a page matching the blank value of `'/'` (that is, the root at <http://127.0.0.1:8000>)

- Route it to `HomepageView.as_view()`.
- And name this route `home`

Remember how we defined a `HomepageView` class in the `views.py` of our `homepage` app? That's what we're importing here.



HomepageView.as_view() generates a callable object

The callable object is analogous to a function. Django calls this object by passing in an `HttpRequest` generated by the user, and then the object returns an `HttpResponse`. Django supplies the `HttpResponse` back to the user.

Don't worry if this doesn't make sense yet. Much later in this book we go over this explicitly in the testing chapters.

II.3 Double-Check the Code

Putting it all together, the code part of `urls.py` should now look like this:

```
from django.contrib import admin  
from django.urls import path  
  
from homepage.views import HomepageView
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', HomepageView.as_view(), name='home'),
]
```



II.4 Terminology: Wire in the URL

Sometimes we refer casually to defining a URL pattern as wiring in a URL for a view. When we say “Wire in the URL”, that means to associate the Django view we just wrote with a web location on our site.

II.4.1 Start Up `runserver` Again

If Django isn't running `runserver`, start it up:

```
python manage.py runserver
```

II.5 Refresh the Page and Trigger an Error

We're going to purposefully trigger an error. That's okay. We're doing it intentionally.

Go back to the browser, and refresh <http://127.0.0.1:8000>. If we did things right, we should see an error page with a header that says:

TemplateDoesNotExist at /

If we see that above error, we are doing it right! Specifically, our view is correctly wired into the `urls.py` module, but its template, `index.html`, doesn't exist yet. Don't worry, we'll demonstrate how to do that in the next chapter.

I2 Create Our First Django Template

Templates are what Django uses to generate the HTML that is shown in browsers.

I2.I Create a Templates Directory

At the same level as `manage.py`, create a templates directory:

```
mkdir templates
```

We can do this through the text editor GUI if we prefer.

Our project file structure should now look like this:

```
helldjango/
├── helldjango/
├── homepage/
└── templates/
└── manage.py
```

I2.2 Add Index Template

Inside of `templates/`, create a file called `index.html` containing the following:

```
<h1>Greetings</h1>
<p>Hello, world!</p>
```

As of now, this is just plain HTML.

I2.3 Start Up Runserver

If `runserver` isn't running, go ahead and start it now:

```
python manage.py runserver
```

I2.4 View the First Page

Open the browser and go to <http://127.0.0.1:8000>. What we see should look something like the image below.

The error went away because we created the template that `HomepageView` was looking for.

I2.5 Recap

We now have a homepage. Here's how it works:

- Our browser requests <http://127.0.0.1:8000>
- Django's development server looks up where to route the request
- Since the index was requested, it matches the path of ''
- Therefore the matching URL pattern is:

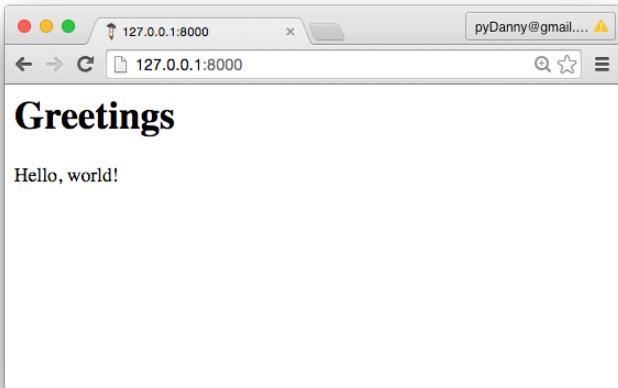


Figure 7: Greetings, Hello World

```
path('' , HomepageView.as_view() , name='home') ,
```

- `HomepageView` is the view corresponding to that URL:

```
class HomepageView(TemplateView):  
    template_name = 'index.html'
```

- It's a `TemplateView`, so it renders its template and returns the result
- We see the page in the browser!

12.6 Understanding Views and the TemplateView Class

- Views are how a Django project communicates with the user. They send out web pages, JSON, spreadsheets, PDFs, and so much more.
- Templates are often used by views to render HTML for web pages.
- A `TemplateView` renders a Django template and sends it to a browser.

I3 Working With Variables in Templates

Without variables we would not have a dynamic web application, there would be no way to interact with the data in the database, essentially we would have a standard static website. For example, we couldn't log in to a website to get results for just the things we care about. By using variables, web pages can display content specific to each user.

I3.I Our First Templated Value

Right now, `templates/index.html` is a Django template that only contains static HTML. But it can also contain variables that get populated dynamically from Python code.

Open up `templates/index.html` and add the following line at the end:

```
<p>{{ my_statement }}</p>
```

The curly braces around `my_statement` mean that it won't be displayed on the page as-is. Instead, it'll be evaluated on the Python side and then the result will be displayed.

I3.2 What Is `my_statement`?

At this point we haven't defined `my_statement` yet in our Python code. It can be any of the following:

- A Python expression, such as a string, number, or instantiated object
- A function or method call

I3.3 Extend HomepageView

Open up the homepage app's `views.py` again. So far, the `HomepageView` is this:

```
class HomepageView(TemplateView):  
    template_name = 'index.html'
```

We're now going to show you how to define a view method called `get_context_data()`. This is a special method that lets you pass variables to your template.

Right below `template_name = 'index.html'`, add a blank line. Then below it, add this method to your `HomepageView` class:

```
def get_context_data(self, **kwargs):  
    context = super().get_context_data(**kwargs)  
    context['my_statement'] = 'Nice to see you!'  
    return context
```

Look carefully at the line involving `my_statement`. There, you defined a **context variable** called `my_statement`. You assigned it the value of `Nice to see you!`.

Note that `context` is a Python dictionary. `my_statement` is a key, and `Nice to see you!` is its corresponding value.



Indent Code Per the Examples!

With Python, indentation is of absolute importance. It is how code blocks are defined. In this case, make certain that `get_context_data()` is indented from the class. Also, the last three lines of the `get_context_data()` method are indented.

This isn't just a code clarity issue in Python, if it is not done then Python will throw errors.

13.4 Refresh the Homepage

Go back to the browser, and refresh <http://127.0.0.1:8000>. (If runserver isn't still running from before, start it up again.)

The value of `{{ my_statement }}` should now be replaced with `Nice to see you!`, as shown in the image below or on the next page.

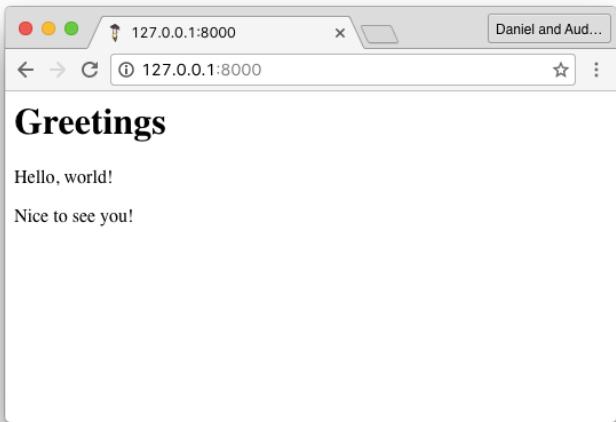


Figure 8: Greetings

I3.5 Exercise: Play Around!

Go ahead and modify the context data, either by changing the value of `my_statement` or adding new keys and values. This practice is important. Django developers working with templates get very experienced working with `get_context_data`.

13.6 Common Questions

13.6.1 What Is a Method?

A method is a Python function that's a member of a class. In our example, `get_context_data()` is a function that's a member of the `HomepageView` class.

Further reading:

- [stack overflow .com/questions/3786881/what-is-a-method-in-python](https://stackoverflow.com/questions/3786881/what-is-a-method-in-python)¹⁰

¹⁰<https://stackoverflow.com/questions/3786881/what-is-a-method-in-python>

I4 Calling View Methods From Templates

The standard Django way to add data to a template's context is via `get_context_data()`. Here's an alternative approach.

Here, we're going to add data by calling a method and displaying the result.

I4.1 Define a View Method

Add the following `say_bye` method to the `HomepageView` class:

```
class HomepageView(TemplateView): # Don't add this line!
    ... # Don't add this line!
    def say_bye(self): # add this line
        return 'Goodbye' # and this line too!
```

I4.2 Call the View Method From index.html

Add this to the bottom of our index template:

```
<p>{{ view.say_bye }}</p>
```

Notice something strange: in the template, we call the `say_bye()` method, but there are no parentheses after it. That's because Django templates intelligently check to see if something can be called as a method.

I4.3 Reload the Homepage

Go back to the browser, and refresh <http://127.0.0.1:8000>.

We should now see the return value of `say_bye()` displayed on our homepage.

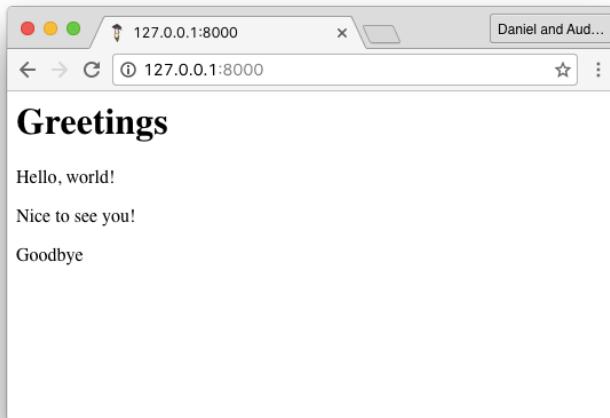


Figure 9: Greeting with view method



How Is a View Method Different From Context Data?

It's the difference between evaluating a variable and evaluating a function call.

Context data Variables used in the template.

View methods Functions that we can call from the template.

For now it's a small distinction. Typically projects rely on Context Data, but there are advanced use cases for using view methods instead.

15 Our First Project is Complete!

Congratulations! By now we understand what makes up a very minimal Django project.

We recommend keeping the `helldjango` project we created as a reference project, so we can refer back to it when we need to revisit how Django views, templates, and URL patterns work together.

15.1 Summary

We started off with a barebones Django project that we generated with `django-admin startproject`. We created a database and started up Django's development server.

Then we created a Django superuser so that we could access the Django admin. We explored the admin UI.

Next, we added a simple Django app to the project. We called it "homepage" and explored what files it gave us to start with.

After that, we learned about views and templates in Django. We created our first view by subclassing `TemplateView`. We connected it to a URL pattern and a simple static HTML template.

Finally, we turned the static HTML template into a dynamic one. We learned how to put variables into the template and populate them in our Python code. We also learned how to call view methods from our template.

15.2 Next Steps

Now that we've finished the **Hello, World** in Django project, let's move on to the EveryCheese on the next page. It's a much deeper dive into Django development and is based on real-world project development.

Blank page

I6 Enter the EveryCheese Project

We're your friendly hosts, Daniel and Audrey, and we love cheese. This tutorial is based on real-world project development, teaching the fundamentals needed to build professional projects. It includes:

- Type-along lesson plans to cover core concepts
- Code examples for each section
- Tutorial project code that prepares students for more advanced techniques

I6.I Prerequisites

We will need to have our computer ready for software development. If it isn't ready, follow the instructions in the chapter called [The Ultimate Django Setup](#).

We'll want to have basic Django experience before starting. If the reader hasn't ever touched Django before, we recommend the previous project in this book, Hello World in Django.

I6.2 Prepare the EveryCheese Programming Environment

As mentioned earlier in this book, **Conda** is a tool for creating isolated Python environments. We use it with pip in this lesson.

If this is starting to feel familiar, that's a good thing. Being practiced at isolating programming environments is a critical skill for any serious developer.

I6.3 Create a New Environment

Create a new Conda environment called `everycheese` and configure it to use `python3.8`:

```
conda create -n everycheese python=3.8
```

Activate the `everycheese` Conda environment:

```
conda activate everycheese
```

After that, our command-line should look like something like:

```
(everycheese) $
```



Don't reuse pre-existing Conda envs for new projects!

It's always good to start fresh with a new Conda env. This way old and new projects won't collide. We'll still be able to maintain successful but older projects and experiment with the latest and greatest pieces of technology.

I6.4 Reactivating Conda Environments

Sometimes we need to reactivate our conda environment. There's lots of reasons to do this, including:

- Our computer needed to restart for any reason
- We just bought a new computer
- Sometimes a second (or third) terminal window is very useful

To do this, in the terminal type:

```
conda activate everycheese
```

I6.5 Summary

If this feels like a repetition of what we did in the **Hello, World** in Django tutorial, that's intentional on our part.

We can't reinforce enough the importance of knowing how to easily move from one programming environment to another. It's a foundation every software developer should know, be they coding with Django, Python, JavaScript, or pretty much anything else.

I7 Cookiecutter and Template

The first tool we install into our projects is **Cookiecutter**, which we'll be using in future extensions and books.

I7.I Make Certain the (everycheese) virtual environment is activated

Your command line should be prefixed with:

(everycheese)

If not, go back to chapter 15 and restart it.

I7.2 Install Cookiecutter

Installing it the first time may take up to 30 seconds, as you might not have Cookiecutter's dependencies already cached on your system. So long as your shell is prefixed with (everycheese) as shown above, Conda will install Cookiecutter in the right place, where you are doesn't matter.

At the command line, type:

```
conda install -c conda-forge cookiecutter
```

There will be output along these lines:

```
Collecting package metadata (current_repodata.json): done
Solving environment: done
<snip for brevity>
Proceed ([y]/n)?
```

Type `y` and then press the return button. We should see something like:

Downloading and Extracting Packages

poyo-0.5.0	14 KB	#####	100%
cryptography-2.8	603 KB	#####	100%
six-1.14.0	13 KB	#####	100%
certifi-2020.4.5.1	151 KB	#####	100%
arrow-0.15.5	97 KB	#####	100%
python_abi-3.8	4 KB	#####	100%
pycparser-2.20	89 KB	#####	100%
chardet-3.0.4	170 KB	#####	100%
jinja2-2.11.2	93 KB	#####	100%
binaryornot-0.4.4	370 KB	#####	100%
urllib3-1.25.9	92 KB	#####	100%
jinja2-time-0.2.0	10 KB	#####	100%

click-7.1.1	64 KB	#####	100%
openssl-1.1.1g	1.9 MB	#####	100%
markupsafe-1.1.1	25 KB	#####	100%
pysocks-1.7.1	27 KB	#####	100%
brotlipy-0.7.0	353 KB	#####	100%
cookiecutter-1.7.0	83 KB	#####	100%
cffi-1.14.0	216 KB	#####	100%
future-0.18.2	717 KB	#####	100%
python-dateutil-2.8.	220 KB	#####	100%
requests-2.23.0	47 KB	#####	100%
ca-certificates-2020	146 KB	#####	100%
whichcraft-0.6.1	8 KB	#####	100%
idna-2.9	52 KB	#####	100%
pyopenssl-19.1.0	47 KB	#####	100%

At this time, when we install Cookiecutter, 26 other packages are also installed. These are Cookiecutter's dependencies and are necessary for Cookiecutter to do its magic.

These are the beginning of the tools we'll be using to create the Everycheese project.

Now that we have Cookiecutter installed, let's use it! This lesson is all about using Cookiecutter with **django-crash-starter** to create a project skeleton.

17.3 Cookiecutter Project Templates

A Cookiecutter project template is a version control repository containing a bunch of files. Those files are used to generate a starter project of some sort. In Cookiecutter parlance, these templates are called *Cookiecutters*.

The `django-crash-starter` project template is at: <https://github.com/feldroy/django-crash-starter>. It provides the foundation for this project and the various Crash Course Extensions.

17.4 Using django-crash-starter

Let's go!

17.4.1 Go to Our Projects Directory

Using the command line, navigate to the place on the computer our coding projects are kept.

For example, we keep ours in a directory called `projects` inside our home directory. We would go into `~/projects`:

On Linux and Mac:

```
mkdir ~/projects
```

On Windows:

```
mkdir %USERPROFILE%\projects
```

Change directories as needed now, so that we are in the right place to create a new Django project.

On Linux and Mac:

```
cd ~/projects/
```

On Windows:

```
cd %USERPROFILE%\projects\
```



User Home Directory

On Linux and Mac, the **~ (tilde character)** shorthand command refers to the current logged user home directory. The **~** is basically an alias to the **\$HOME** environment variable.

On Windows, the current logged user home directory is the **%USERPROFILE%** environment variable.

Based on the assumption that we have the default configuration generated after installing Windows, Linux and Mac, our user's home directory can be found at:

Windows: `c:\Users\<username>`

Mac: `/Users/<username>`

Linux: `/home/<username>`



Open the GUI file manager from the command-line.

One thing that can be very useful to know is how to open the graphical file manager from the terminal/command-line.

1 - Open the current directory using the `.` (dot character):

Windows: `explorer .`

Linux: `xdg-open .`

Mac: `open .`

2 - Open the another directory:

Windows: `explorer path\to\the\directory`

Linux: `xdg-open path/to/the/directory`

Mac: `open path/to/the/directory`

17.4.2 Run the Cookiecutter Command

Here, use the `django-crash-starter` template to power Cookiecutter. Type this at our command line:

```
cookiecutter gh:feldroy/django-crash-starter
```

This means:

- Run the command called `cookiecutter` now.
- Use <https://github.com/feldroy/django-crash-starter> as the Cookiecutter project template.

When we first run this command, the result should look like:

```
Cloning into 'django-crash-starter'...
remote: Counting objects: 3955, done.
remote: Compressing objects: 100% (142/142), done.
remote: Total 3955(delta 85), pack-reused 3811
Receiving objects: 100% (3955/3955) | 0 bytes/s, done.
Resolving deltas: 100% (2425/2425), done.
Checking connectivity... done.
project_name [EveryCheese]:
```

That last line is a prompt asking for the name of the project. Default values are in the brackets. For example, by default, our project is going to be called “EveryCheese”. Go ahead and hit enter for each value. If you are on Windows, choose ‘y’ when you come to `windows [n]:`

```
project_name [EveryCheese]:  
project_slug [everycheese]:  
description [The Ultimate Cheese Index!]:  
author_name [Your name here]:  
domain_name [everycheese.com]:  
email [your-name-here@example.com]:  
timezone [UTC]:  
windows [n]: # On Windows change to 'y'  
Select database:  
1 - PostgreSQL  
2 - SQLite  
Choose from 1, 2 [1]:
```

Let's not worry if we don't understand what every single value means. We'll learn more as we progress through this book.

I7.5 What Happened?

We've now used `django-crash-starter` to generate boilerplate code for our Django project!

A directory called `everycheese/` should have been generated in our current working directory. Check that it exists by typing `ls` on Linux and Mac or `dir` on Windows at the command-line. We should see the following:

```
everycheese/
```

If it's there, you're ready to move on to the next lesson.

17.6 Tips and Troubleshooting

17.6.1 Delete and Re-Clone!

If you have used `django-crash-starter` before, we may see this message when we run `cookiecutter`:

```
You've cloned /Users/audreyr/.cookiecutters/
django-crash-starter before. Is it okay to
delete and re-clone it? [yes]:
```

We can hit Return to accept the default of `[yes]`.

(Note: The only time we wouldn't want to pick `[yes]` is if we manually edited the files in that directory for some strange reason and didn't want them to be overwritten. It's best not to edit files in `~/.cookiecutters/`, though.)

17.6.2 Unfamiliar Options

If `django-crash-starter` offers us an option that we haven't listed above, go with the defaults for that option. (It may change after the printing of this book! We'll try not to change it too much.)

17.6.3 Fear of Making a Mistake

Don't worry about accidentally generating our project with the wrong `django-crash-starter` options.

That's because we can always look through the `django-crash-starter` source tree to see what code would have been generated if we had picked a different option. The next lesson will give us some insight into this.

17.7 Cookiecutter of Background

Cookiecutter is a popular cross-platform project template generator created by author Audrey Roy Greenfeld in the summer of 2013. The simplicity of the UI hides a sophisticated framework that can be extended easily for use in frameworks and other tools.

Because of this combination of simple UI and extendability, there are approximately 3500 project templates powered by Cookiecutter on GitHub, and an unknown number of private ones. In the past six years, we've yet to work for an employer or consulting client who wasn't using Cookiecutter.



Figure 10: The Tasty Cookiecutter Logo

I8 Exploring the Generated Boilerplate

Now we're going to explore the boilerplate that Cookiecutter generated from the following components:

- The `django-crash-starter` project template.
- The values you entered.

I8.I Some Basic Components

Open `everycheese/` in your text editor.

If you're using VS Code, you can do this by typing:

```
code everycheese/
```

I8.I.1 Readme

If we go into `README.md`, we'll see that the header, EveryCheese, comes from what we entered as `project_name`.

Every Django project that we work on has a `README` which provides an overview of the project. You'll want to add more to the starter file. Typically it will also include a list of features of the project.

18.1.2 Settings

Notice that a `config/` directory got generated. Among other things, inside is a `settings/` directory with a bunch of settings files for different purposes:

base.py Any settings that are used identically in all places

local.py Local development on our computers

production.py Live production server settings

If we go into `everycheese/config/settings/base.py`, we can see that a value was set for `TIME_ZONE`:

```
TIME_ZONE = 'UTC'
```

This value came from what we entered as `timezone` during the `django-crash-starter` prompts.

For everything that Cookiecutter asked questions about, it has filled out the values throughout the project.

18.1.3 Contributors

If we go into `config/settings/base.py`, we can see that this got placed in there:

```
ADMINS = [("Your name here", "your-name-here@example.com")]
```

Whoa, that's not quite right! Or is it?

When we created our project using Cookiecutter, we accepted the default values. Those are the ones now seen in the `base.py` settings file. Go ahead and change those values to match your name.

18.2 Summary

`django-crash-starter` is more than just README and configuration. It comes with a `users/` app for account management including `django-allauth` preconfigured, templates that work with Bootstrap right out of the box.

This kind of boilerplate normally takes forever to get right. Fortunately, `django-crash-starter` gets it out of the way.

There's a lot to cover. We'll get to explaining more of it in the next chapter and the rest of this book.

I9 Starting Up Our New Django Project

The code in `everycheese/` is a real Django project with tons of working features. We can run EveryCheese, just like any other Django project.

Let's make that happen!

I9.1 Go Into the EveryCheese Directory

Navigate into the main `everycheese/` directory:

```
cd everycheese
```

Generally, this is where we'll be working.

Now, if we type `ls` (or `dir` on Windows), we should see a bunch of files and directories. One of these should be `manage.py`.

I9.2 Install the Dependencies



We must make sure that we have the `everycheese` Conda environment activated. If we have something like `(everycheese)` at the beginning of the

terminal text, `everycheese` Conda environment is activated. If not, we must reactivate it as described in [Reactivating Conda Environments](#)

Inside of the `requirements/` directory are the environment-specific requirement files for our project:

- `base.txt` for any packages that are used in all places.
- `local.txt` for packages required for local development on our computer. Also used in testing.
- `production.txt` for our live production server packages.

The file we want to use is `local.txt`, which represents our local development requirements.

To install `local.txt` on Linux and Mac:

```
pip install -r requirements/local.txt
```

To install `local.txt` on Windows:

```
pip install -r requirements\local.txt
```

If there are problems installing the dependencies, we may cover the solutions in our [Troubleshooting](#) appendix:

- Troubleshooting PsycoPG2
- Troubleshooting GCC Errors on the Mac
- Troubleshooting the MS Visual Studio Build Tools Error

19.3 Defining the database

In our first django project, we did not change the default database configuration that `django startproject` provides. This meant that users of all operating systems used Sqlite for the Hello Django project.

In the `everycheese` project, we will use an `.env` file to make Django configuration more flexible.

19.3.1 Creating the .env file

On the root directory of the `everycheese` project, there are two files:

- `.env.sample.mac_or_linux`
- `.env.sample.windows`

Choose the file that matches your operating system. Copy that to a new file in the same directory named `.env`.



Make certain that the command-line is at the right level of the project. We need to be at the same level as `manage.py`.

To create the .env on Linux and Mac:

```
cp env.sample.mac_or_linux .env
```

To create the .env on Windows:

```
copy env.sample.windows .env
```



If `env.sample.mac_or_linux` OR `env.sample.windows` does not exist in our project

Get the correct file by going to the link appropriate for your operating system.

Windows: http://feld.to/env_sample_windows

Mac or Linux: <http://feld.to/env-sample-mac-or-linux>

19.3.2 Configuring git to ignoring the .env file

The .env file contains credentials and data that should be kept secret and never saved with git. Therefore, we need to configure git to never track the .env file.

If our project was created after September 8th, 2020, ignore this subsection and move on to the next. Otherwise, in Visual Studio Code, open the .gitignore file and add these two lines to the end of the file:

```
# Block secrets from being saved to version control  
.env
```

19.3.3 Start PostgreSQL if Needed

This only applies to Mac and Linux. If on Windows, please ignore this subsection.

Different PostgreSQL installers make different decisions about whether PostgreSQL automatically runs as a service upon system startup.

To check if PostgreSQL is running, try typing this:

```
pg_isready
```

If PostgreSQL is running, we'll see an `accepting connections` message, like:

```
/var/run/postgresql:5432 - accepting connections
```

If PostgreSQL isn't running, we'll see a `no response` error message, like:

```
/var/run/postgresql:5432 - no response
```

And if PostgreSQL is not already running, we'll need to start PostgreSQL:

- On Mac: Open up Applications > Postgres.app¹¹
- On Linux: Start the service if it's not already running with `sudo service postgresql start`

On Mac and Linux we'll need PostgreSQL running later in this chapter to install certain Python libraries and create a PostgreSQL database.

I9.4 What to name the database?

Let's look at the common settings file, inside of `config/settings/`. If we open up `base.py` and look for the `DATABASES` setting, we can see that the database is called `everycheese`:

```
DATABASES = {  
    # Raises ImproperlyConfigured Exception  
    # if DATABASE_URL Not in os.environ  
    "default": env.db(  
        "DATABASE_URL", default="postgres://everycheese",  
    )  
}
```

Windows users may see something like:

¹¹<http://postgresapp.com/>

```
sqlite_path = str(BASE_DIR / 'everycheese.db')

DATABASES = {
    "default": env.db(
        "DATABASE_URL",
        default=f"sqlite:///{sqlite_path}",
    )
}
```

These are the defaults, which are used by `local.py` for local development. If we open up `local.py`, we'll see that there is no `DATABASES` setting in there, and so it defaults to the above.

In production, we do not provide a default, as we can see in `production.py`. We have to handle that ourselves. We'll explain more about that when we get around to deployment.

One might notice that `BASE_DIR / 'everycheese.db'`, is not a typical Python string concatenation. Specifically, `pathlib.Path` overwrites the slash (/) to create child paths. To learn more, visit <https://docs.python.org/3/library/pathlib.html>.

19.4.1 How to Create It

This only applies to Mac and Linux. If on Windows, please ignore this section.

To create a database called `everycheese`, we can type this at the command line:

On Mac:

```
python manage.py sqlcreate | psql
```

On Linux:

```
python manage.py sqlcreate | sudo -u postgres psql -U postgres
```

On Windows:

Since we are using SQLite 3 on Windows, no extra command is needed to create the database before use it.

If there are any problems, one of the following troubleshooting segments might prove useful:

- [Troubleshooting PostgreSQL: Database EveryCheese Already Exists and/or Role myuser Already Exists](#)
- [Troubleshooting PostgreSQL: Role Does Not Exist](#)
- [Troubleshooting PostgreSQL: Cannot Create Database](#)

19.5 Run the Migrations

Whenever we set up a Django project, we want to run migrate in case any migrations need to be run. Type this at the command line:

```
python manage.py migrate
```

19.6 Start Up Runserver

Finally, start Django's local development server so we can see the project in action:

```
python manage.py runserver
```

Open a web browser to <http://127.0.0.1:8000>. We should see a barebones EveryCheese website running.

20 Cheese Boilers

When cheese is melted in industrial processing, it goes into cheese boilers.

Cheese boilers have boilerplates. Do not remove the boilerplate while a cheese boiler is in operation, or else the cheese will flow out of the boiler.

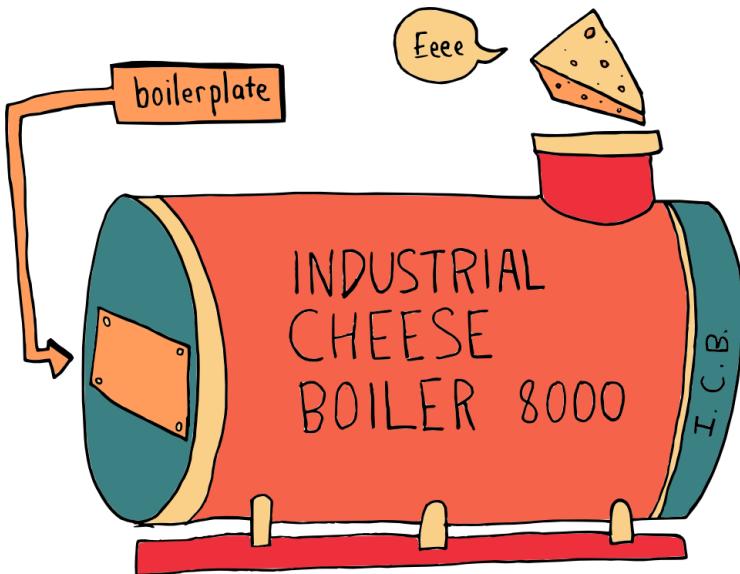


Figure 11: Illustration of cheese boiler, and the location of its boilerplate

2I Initializing the Git Repo Committing and Pushing

We've generated a bunch of starting Django project boilerplate and seen the barebones site running on the development server.

Before we move on, we'll walk us through creating a repo so we can start saving our changes periodically.

2I.I Why?

We're going to be building upon this code and changing a lot of files.

The problem is that it's easy to break things and lose our work.

That's where version control comes in.

2I.2 We'll Use GitHub

We're going to create a private repo for the project on GitHub.

- Sign up for a GitHub¹² account if we don't have one yet.
- Add an SSH key¹³ to our GitHub profile, so that we can push code changes to our repo. If you don't know how to do that yet, follow the instructions at <https://docs.github.com/en/github/authenticating-to-github/connecting-to-github-with-ssh>

21.3 Create the GitHub Project

Then go to <https://github.com/new> and fill it out with:

- Project name: **everycheese**
- Description: **The ultimate cheese index.**
- Visibility Level: **Private**

Click the *Create repository* button.

21.4 Examine the GitHub Instructions

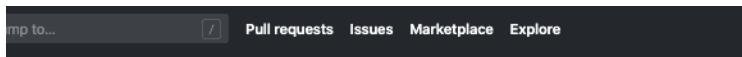
GitHub provides instructions for adding files from an “Existing folder or Git repository.” We have an existing folder of code.

We’re going to skip changing directory into our folder, since we’re already in there.

We’re going to walk us through what these instructions actually mean.

¹²<https://github.com>

¹³<https://github.com/settings/keys>



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner



pydanny ▾

Repository name *

/ everycheese



Great repository names are short and memorable. Need inspiration? How about [bug-free-pancake](#)?

Description (optional)

The Ultimate Cheese Index

Public

Anyone can see this repository. You choose who can commit.

Private

You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

Initialize this repository with a README

This will let you immediately clone the repository to your computer.

Add .gitignore: None ▾

Add a license: None ▾



Create repository

Figure 12: Create repository on GitHub

21.4.1 Initialize the Local Repo

If we still have `runserver` running, we can stop it and type these commands in the same terminal. Or we can open a new terminal if preferred. It doesn't matter.

Initialize the current folder as a Git repo:

```
git init
```

Add the GitHub remote to the repo as the origin. Replace `ourusername` with our actual GitHub username:

```
git remote add origin git@github.com:ourusername/everycheese.git
```

21.4.2 Add Our Files

Add all our files to the first commit to the project:

```
git add .
```



Before the next step

The next step hopes that we've already configured `git` (by following the [Configuring Git](#) instructions). We should be in the root of the `everycheese/` directory structure.

Commit the first commit with a brief, descriptive message:

```
git commit \  
-m "Generated from the django-crash-starter project template."
```

Push our changes:

```
git push -u origin master
```

21.5 Verify That the Files Got Committed

In the browser, refresh the GitHub page for the repo created earlier in this chapter.

Go back to the terminal and check the git log as well:

```
git log
```

Now we are free to code without worrying about losing our work if we accidentally break something and can't fix it.

22 Git Is Not for Stinky Cheeses

You know what really, absolutely doesn't belong in a git repo?

Stinky cheese!

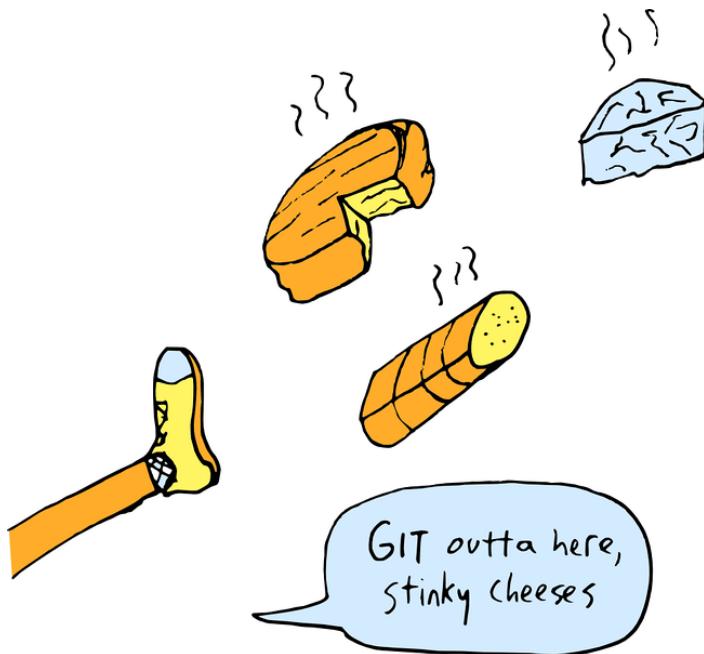


Figure 13: Stinky cheeses getting kicked out by a foot.

Don't even try to put it in there! That's because other files may leave a repo due to the pungent odor. No file wants to be in a stinky cheese-filled repo.

23 What We Get From The Users App

`django-crash-starter` provided us with a simple `users` app, preconfigured right out of the box. Let's explore what it gives us.

23.1 Register a Normal User

Start up `runserver` again:

```
python manage.py runserver
```

Go to <http://127.0.0.1:8000> to see the starter EveryCheese website.

Click on the “Sign Up” link in the navbar. This will bring us to a registration form. This is where the visitors to our website will create their own user accounts.

We’re going to try out the website’s registration system now. Enter these values:

- email: `cheesehead@example.com`
- username: `cheesehead`
- password: `*QjkXCPUm8iuaeP`

E-mail*

cheesehead@example.com

Username*

cheesehead

Password*

Password (again)*

Sign Up >

Figure 14: Registration form



Copy this password into a password manager!

It's bad practice to use simplistic passwords, even for local development. We want to promote strong security habits and this is one of them. Also, don't stick passwords into plaintext or spreadsheet files. Instead, lean on secure password managers like our preferred **1Password** or open source options such as **Bitwarden** and **KeePass**.

23.2 Verify Our Email

When we submit the form, we'll be asked to verify our email address before we can continue on the website:

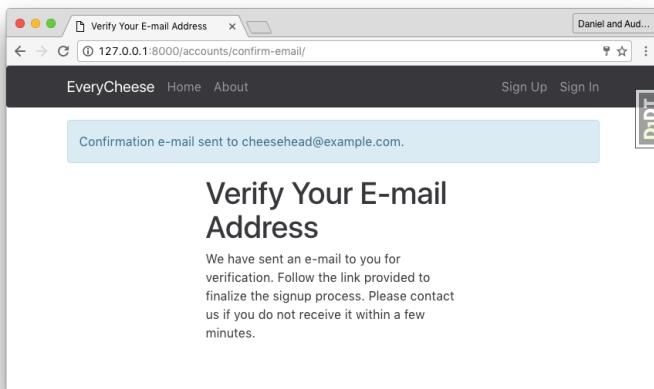


Figure 15: Verify email page



What to do if the navbar is light colored

The correction for this is found at [Navbar Isn't Dark](#).

23.2.1 Grab the Email Confirmation Link

We're not going to get an email confirmation by regular email. `django-crash-starter` isn't set up to send out real emails during local development, and for good reason. We wouldn't want to accidentally send out emails to real people during local development.

However, `django-crash-starter` has configured our project to display all outgoing emails in the console, where `runserver` is running.

Go to the console window, the one where we started up `runserver`. We should see the confirmation email there, as shown below or on the next page.

```
(everycheese) everycheese a&w ./manage.py runserver — 95x50
Performing system checks...

System check identified no issues (0 silenced).
August 18, 2016 - 13:41:33
Django version 1.9.9, using settings 'config.settings.local'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

[18/Aug/2016 13:42:14] "GET / HTTP/1.1" 200 13053
[18/Aug/2016 13:42:14] "GET /static/css/project.css HTTP/1.1" 304 0
[18/Aug/2016 13:42:14] "GET /static/js/project.js HTTP/1.1" 304 0
[18/Aug/2016 13:42:14] "GET /static/debug_toolbar/css/toolbar.css HTTP/1.1" 304 0
[18/Aug/2016 13:42:14] "GET /static/debug_toolbar/js/jquery.print.js HTTP/1.1" 304 0
[18/Aug/2016 13:42:14] "GET /static/debug_toolbar/js/jquery_post.js HTTP/1.1" 304 0
[18/Aug/2016 13:42:14] "GET /static/debug_toolbar/img/ajax-loader.gif HTTP/1.1" 304 0
[18/Aug/2016 13:42:14] "GET /static/debug_toolbar/js/toolbar.js HTTP/1.1" 304 0
[18/Aug/2016 13:42:14] "GET /static/debug_toolbar/css/print.css HTTP/1.1" 304 0
[18/Aug/2016 13:44:27] "GET /accounts/signup/ HTTP/1.1" 200 14838
MIME-Version: 1.0
Content-Type: text/plain; charset=utf-8"
Content-Transfer-Encoding: 7bit
Subject: [EveryCheese] Please Confirm Your E-mail Address
From: webmaster@localhost
To: cheesehead@example.com
Date: Fri, 19 Aug 2016 00:37:31 -0000
Message-ID: <20160819003731.46250.35439@audreyr.local>

Hello from EveryCheese!

You're receiving this e-mail because user cheesehead at everycheese.com has given yours as an e-mail address to connect their account.

To confirm this is correct, go to http://127.0.0.1:8000/accounts/confirm-email/MQ:1baXoR:6b7WEncZz06H10cgGdWgnUGDR4

Thank you from EveryCheese!
everycheese.com

[18/Aug/2016 17:37:32] "POST /accounts/signup/ HTTP/1.1" 302 0
[18/Aug/2016 17:37:32] "GET /accounts/confirm-email/ HTTP/1.1" 200 13441
[18/Aug/2016 17:37:33] "GET /static/css/project.css HTTP/1.1" 304 0
[18/Aug/2016 17:37:33] "GET /static/debug_toolbar/js/toolbar.js HTTP/1.1" 304 0
[18/Aug/2016 17:37:33] "GET /static/debug_toolbar/css/toolbar.css HTTP/1.1" 304 0
[18/Aug/2016 17:37:33] "GET /static/debug_toolbar/js/jquery.print.js HTTP/1.1" 304 0
[18/Aug/2016 17:37:33] "GET /static/js/project.js HTTP/1.1" 304 0
[18/Aug/2016 17:37:33] "GET /static/debug_toolbar/img/ajax-loader.gif HTTP/1.1" 304 0
[18/Aug/2016 17:37:33] "GET /static/debug_toolbar/js/jquery_post.js HTTP/1.1" 304 0
[18/Aug/2016 17:37:33] "GET /static/debug_toolbar/js/toolbar.js HTTP/1.1" 304 0
[18/Aug/2016 17:40:50] "GET /accounts/signup/ HTTP/1.1" 200 14834
```

Figure 16: Verification email in console window

Copy/paste the email confirmation link into the browser. We will be taken to a page with a *Confirm* button, as shown below or on the next page.

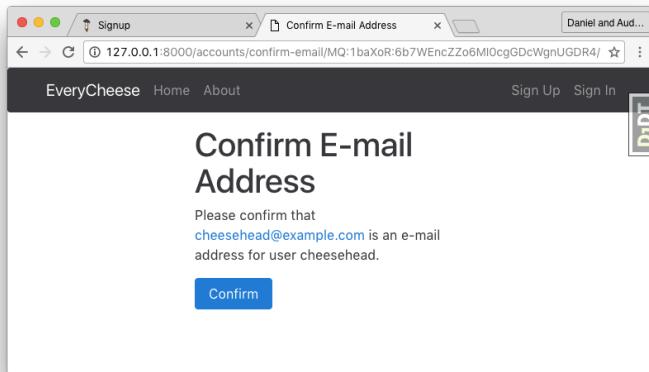


Figure 17: Page with Confirm button

Then click *Confirm* to confirm the user account.

23.3 Why Are Emails in the Console?

Open `settings/local.py` in a text editor. Find the `EMAIL_BACKEND` setting.

```
EMAIL_BACKEND = env(
    "DJANGO_EMAIL_BACKEND",
    default="django.core.mail.backends.console.EmailBackend",
)
```

Now open `settings/production.py` and find the `EMAIL_BACKEND` setting. Compare it.

```
EMAIL_BACKEND = "anymail.backends.mailgun.EmailBackend"
```

For local development, we use Django's console email backend, which prints all emails to the console. But in production we switch to the Mailgun backend, which sends out emails to users.

23.4 Explore the User Profile

If needed, sign in with the new account.

Click through to the user profile. We might not recognize it as a user profile because of how minimal it is to start. It looks like this:

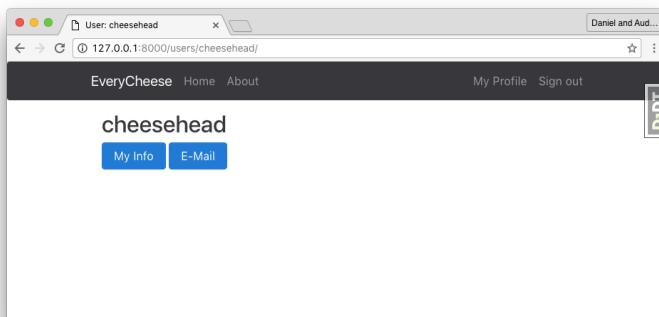


Figure 18: Minimal user profile

23.5 Edit User Data

Let's see what user info is editable:

1. Click on *My Info*.
2. Try giving the user a full name.
3. Click *Update* and see where the data is displayed in the user profile.

23.6 Introducing `UserUpdateView`

Remember where we entered a value for *Name of User*?

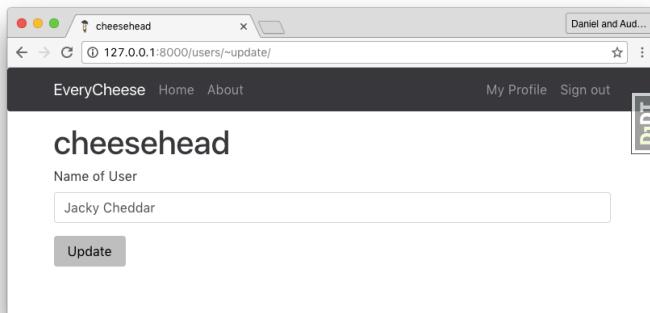


Figure 19: UserUpdateView

That page corresponds to `UserUpdateView`. An `UpdateView` is a view for editing an existing object. In this case, the object edited is the `user` object.

We'll show us the code for this shortly. But first, let's look at a simpler view together.

23.7 Introducing `UserDetailView`

After entering `Name of User`, we were redirected back to the user profile. This view is the `UserDetailView`.

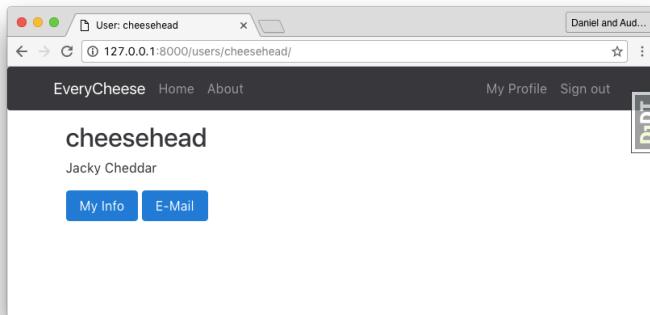


Figure 20: `UserDetailView`

A `DetailView` displays detailed data for a particular object. Here, it displays info about our `user` object.

Open up `views.py` of the `users` app and find `UserDetailView`:

```
class UserDetailView(LoginRequiredMixin, DetailView):
    model = User
    # These Next Two Lines Tell the View to Index
    #   Lookups by Username
```

```
slug_field = 'username'  
slug_url_kwarg = 'username'
```

Some things to notice:

- `UserDetailView` is a subclass of Django's generic `DetailView`.
- The view can only be accessed by logged-in users because of `LoginRequiredMixin`.
- We have to explicitly specify the model with `model = User`.
- Note the URL `/users/cheesehead/` and how `username` is used as a slug in the URL.

23.8 Understand the Name Field

Where is this *Name of User* field coming from? Let's find out.

In the text editor, open the `models.py` of the **users** app. Note these lines:

```
class User(AbstractUser):  
  
    # First Name and Last Name Do Not Cover Name Patterns  
    # Around the Globe.  
    name = models.CharField(  
        max_length=150,  
        blank=True,  
        null=True,  
        help_text='Name of User')
```

```
_("Name of User"), blank=True, max_length=255  
)
```

The `name` field looks like it corresponds to *Name of User*. But we have to look at the views to see what's really going on.

23.9 Understand UserUpdateView

Open the `views.py` of the users app now and look for the `UserUpdateView`. It looks like this:

```
class UserUpdateView(LoginRequiredMixin, UpdateView):  
    fields = [  
        "name",  
    ]  
  
    # We already imported user in the View code above,  
    # remember?  
    model = User  
  
    # Send the User Back to Their Own Page after a  
    # successful Update  
    def get_success_url(self):  
        return reverse(  
            "users:detail",  
            kwargs={"username": self.request.user.username},
```

```
)  
  
def get_object(self):  
    # Only Get the User Record for the  
    # User Making the Request  
    return User.objects.get(  
        username=self.request.user.username  
    )
```

This view corresponds to the page where we entered in the name of our user. Note the following:

- The `fields` list contains the fields that are part of the form. The one field that exists, `name`, is in this list.
- Even though it's obvious from `UserUpdateView` that this view affects the `User` model, we still have to set the view's model explicitly with `model = User`.

Study the rest of the views in the users app. Figure out which view corresponds to each page.

23.10 Add Email Addresses

What else can we do?

1. Click on *E-Mail*.
2. Try adding an email address. Remember, it doesn't have to be a real email, since the confirmation email will be printed to the console anyway.

3. Check for a confirmation email in the terminal window where `runserver` is running.
4. Copy and paste the email verification link into our browser.

Feel free to play around with the other email address management buttons as well. It's good to see what they do.

23.II What About the Code?

Where is the code for this email functionality? We may have looked through the `users` app to try and find it, but it's not there. That's because the code is in **django-allauth**.

Let's look at the template that is used with `UserDetailView`. The templates for the `users` app are in `templates/users/`. Open `user_detail.html` in our text editor.

Find the HTML for the two buttons:

```
<a class="btn btn-primary" href="{% url 'users:update' %}">  
    My Info</a>  
  
<a class="btn btn-primary" href="{% url 'account_email' %}">  
    E-Mail</a>
```

Both buttons are actually just HTML links that are styled to look like buttons. The `url` tag is a built-in template tag that converts a URL name to an absolute path URL.

The *My Info* button points to a URL named `update` within the `users` namespace, which is wired up with `UserUpdateView` in the `users/urls.py` module. We saw how that wiring is done at a fundamental level back in [Get to Know URL Routing](#).

The *E-Mail* button link points to a URL named `account_email`. That URL is defined in allauth's `urls.py` file, which we can find at <https://github.com/pennersr/django-allauth/blob/master/allauth/account/urls.py>. Search for the `path()` declaration with a `name` argument of `account_email`.

`django-crash-starter` came with `django-allauth` pre-configured with its `users` app. Since it's a dependency of the `everycheese` project that we generated, links in `users` app templates can go to URLs defined by allauth.

24 Adding User Bios

Out of the box, user profiles are pretty minimal. We're going to give the `User` model a `bio` field to make it a little more interesting.

24.1 Add a Bio Field

Add this to the `User` model in the `users` app `models.py`:

```
bio = models.TextField("Bio", blank=True)
```

We can put that line right below the `name` field.

Save the file again before proceeding.



Save your files constantly!

Anytime you leave a code file to visit another file or run something on the command, save your file. In fact, with VS Code, we recommend going to the file menu at the top of the screen and choosing “Auto Save”. This will autosave files within VS Code any time you leave them.

24.2 Create and Run Migrations

To do this, we can either:

- Stop `runserver` and use the same terminal,

or

- Open up a new command line tab or window where we run `makemigrations` and `migrate`

Then make and apply our migrations, this time append it with the word, “users”. This ensures that the migration only affects the `users` app:

```
python manage.py makemigrations users  
python manage.py migrate users
```

Now the `User` model has a `bio` field, and the field exists in the database.

24.2.1 What Are Migrations?

Whenever we do something that affects the structure of the database (mostly when we modify `models.py`, but also when we create the database), we need to apply these changes. That is called a migration. We do it in two steps (Don’t run these next two commands, just understand what they are):

- first we create the commands that will change the database

```
python manage.py makemigrations
```

- then we apply these changes

```
python manage.py migrate
```

When migrations have already been created, for example by third-party libraries, we only need to apply them. That's not the case here, we are making our own (`makemigrations`) and running them (`migrate`).

24.3 Update UserUpdateView

In the `views.py` of the users app, update `fields` by adding `bio` to the list:

```
class UserUpdateView(LoginRequiredMixin, UpdateView):  
    fields = [  
        "name",  
        "bio",  
    ]
```

Now the form should allow us to modify our user's bio.

24.4 Try It Out

Start up `runserver` so that we can try out our new bio field on our site.

```
python manage.py runserver
```

Go back to the EveryCheese website in our browser and refresh our user profile page:

- <http://127.0.0.1:8000/users/cheesehead/>

Click on the *My Info* button again. We should see the new bio field. Fill in some text, then click *Update* to save our bio as shown in the image below or on the next page:

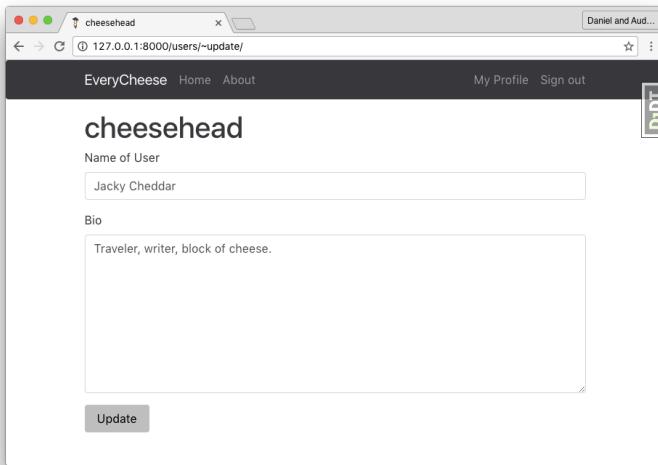


Figure 21: UserUpdateView With Bio

As we can see, now users can add and edit their bios. But the bios **field** still **doesn't** show up on user profiles. Notice that `UserDetailView` still looks like the image below or on the next page:

But if we try clicking on *My Info* again, we'll see that the bio we wrote earlier is there. It has in fact been saved to the `user` object; it's just not being shown in `UserDetailView` yet.

We'll fix that in the next chapter.

One more thing, you don't always need to make or run migrations. Typically migrations are only done when

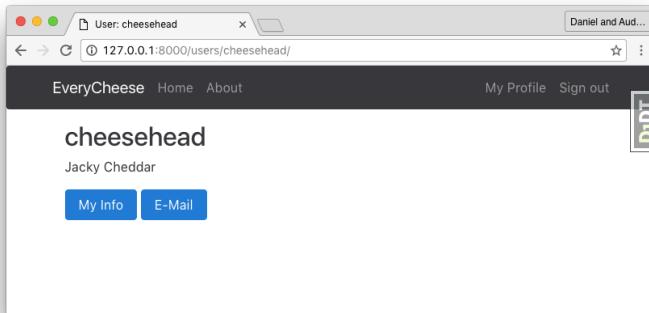


Figure 22: UserDetailView Without Bio

classes within `models.py` are changed. For example, the next chapter is about templates, so we won't be doing any migrations there.

25 Adding Bio to User Detail Template

Let's allow users to provide some general information about themselves.

25.1 Update the User Detail Template

Go back to our text editor. Open up `user_detail.html` again (find it with the templates for the users app).

Find the place where the user's name is displayed if it exists. It looks like this:

```
{% if object.name %}  
  <p>{{ object.name }}</p>  
{% endif %}
```

Below it, add a similar block to display the user's bio:

```
{% if object.bio %}  
  <p>{{ object.bio }}</p>  
{% endif %}
```

Make sure that the bio is in the same place, before the closing divs.

This code says to only display that line if the bio exists. We do this because we don't want an empty paragraph element if the user has not filled in a bio.

Go back to our profile page in the browser. Reload it if necessary. We should now see our bio:

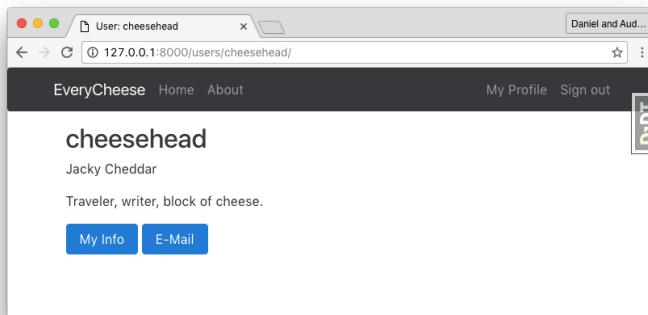


Figure 23: UserDetailView With Bio

Looks good so far. But what happens when a user bio has more than one paragraph?

25.2 Add Line Breaks

Now try adding line breaks and a second paragraph to our bio:

```
Traveler, writer, block of cheese.
```

```
What is life without cheese?  
Nothing.
```

Save it and look at the user detail page. Notice how the line breaks aren't preserved. Our poetic multiline bio is all on one line. That's a problem because a lot of people will probably enter line breaks, expecting them to be preserved.

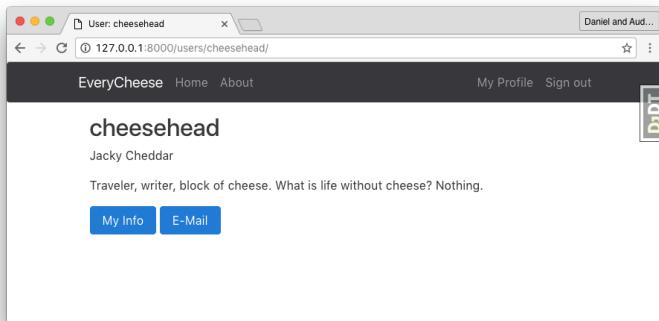


Figure 24: No line breaks in UserDetailView

To fix that, change the line displaying the bio to:

```
<p>{{ object.bio|linebreaksbr }}</p>
```

Now when we reload our profile page in the browser, we should see that the line breaks are displayed properly. If you see line breaks then you got it working!

25.3 Explaining `|linebreaksbr`

We just used `|linebreaksbr` to modify the `object.bio` variable so line breaks in our bio would show up on a rendered HTML page. Let's break down how that works:

1. The “|” or “pipe” symbol in a template signifies to Django that the next word is a “filter”. Filters are used to modify variables in Django templates.
2. The “`linebreaksbr`” filter modifies text to replace every carriage return with the HTML `
` tag in the page being rendered. The `
` tag tells browsers to break the line.

All of this is a long-winded way of explaining that `|linebreaksbr` allows our `bio` to show proper line breaks.

Django provides over 50 built-in template filters, which are very useful in presenting information to users. It's possible to write your own, and there are third-party libraries written around them. We plan to cover these techniques and more in a future extension to this book.

25.4 Commit Our Work

We've done a lot. The `-A` in `git add -A` below tells git to add all changes, including new files and deletions:

```
git status  
git add -A  
git commit -m "Add bio field to User model, views, templates"  
git push origin master
```

26 Introducing Tests

In this lesson we're going to learn the basics of writing tests against changes we've made in our code. Then we're going to show a few tricks to help ensure we have healthy test writing habits.

26.1 Why Test?

Why are we spending our time writing tests? What's the point? Well, tests help ensure that our project is more stable in the long run.

Specifically:

Fixing bugs on existing projects. Minimize the chances of our bug fix breaking other things.

Upgrading our software and operating system.

Formal tests help identify where things are going to break

There are many more reasons why testing is important, but those are two of our favorites.

26.2 Running Tests

To run the test suite in Django is just a management command away.

At the command line, type:

```
coverage run -m pytest
```

As the tests are run, we'll see text printed out that looks something like:

```
Test session starts
platform: darwin, Python 3.8.1, pytest 5.3.4, pytest-sugar
django: settings: config.settings.test (from option)
rootdir: /Users/drg/projects/everycheese, ini file: pytest.ini
plugins: sugar-0.9.2, django-3.8.0
collecting ...
everycheese/users/tests/test_forms.py ✓    12% [■]
everycheese/users/tests/test_models.py ✓    25% [■■]
everycheese/users/tests/test_urls.py ✓✓✓    62% [■■■■■■]
everycheese/users/tests/test_views.py ✓✓✓✓ 100% [■■■■■■■■■■]

Results (1.94s):
    9 passed
```

Let's go over what happened step-by-step:

1. Django created a test database, so there's no chance of it hurting our real data.
2. Django ran 9 tests in 1.94 seconds, then reported 'OK'. That means all the tests pass.
3. Django finally destroyed the test database, making it possible to run the tests again with a clean slate.

If any of those 9 tests had failed, Django would have reported the failures to us via error messages. These failure messages indicate where the problem has occurred.

26.2.1 If We Get Any “Already Included” Warnings

Should the tests pass but there are lots of warnings that start with this:

```
Coverage.py warning: Already imported a file  
that will be measured
```

That means that **coverage.py** is trying to look in third-party packages for files that need more test coverage. To correct the issue, go to your project’s `.coveragerc` file and add this to the bottom:

```
disable_warnings = already-imported
```

26.2.2 If We Get a LOT of Deprecation Warnings

Should the tests pass but there are dozens of warnings about `django.utils.translation.ugettext_lazy()`, that’s okay.

One of the dependencies we’re relying on may not be updated yet to run on Django 3.x. If these warnings appear, go into the root of the project and find the `pytest.ini` file.

Add this value to the second line, with a space between it and the text that comes before it:

```
-p no:warnings
```

26.3 What Does a Test Look Like?

In our text editor or IDE, go to `everycheese > users > tests > test_models.py`. Inside there is a single function, `test_user_get_absolute_url`:

```
import pytest

from everycheese.users.models import User

pytestmark = pytest.mark.django_db


def test_user_get_absolute_url(user):
    url = f"/users/{user.username}/"
    assert user.get_absolute_url() == url
```

Next, let's look at each part of this code in detail.

26.4 The Test Foundations

```
import pytest

from everycheese.users.models import User

pytestmark = pytest.mark.django_db
```

- We import pytest¹⁴, our favorite testing framework.
Think of it as the Django of testing.
- We import the `User` model from our `users` models.
- We define `pytestmark` in our module, which drives the test database system.

26.5 The Actual Test

The function in the module is a test. We know it is tests because it is prefixed with “`test_`”.

```
def test_user_get_absolute_url(user: User):
    assert user.get_absolute_url() == f"/users/{user.username}/"
```

`test_user_get_absolute_url` is a validation against the `everycheese.users.models.User` model’s `get_absolute_url()` method.

¹⁴<https://docs.pytest.org/>



Every model method should have a test

If we add another model method, for example, `__str__`, we would write a test for that as well. In fact, any time we add any method or function to the project, we write tests for them. Doing it upfront is a good way to ensure improved code stability.

26.6 Using the assert KeyWord

Assert is a Python built-in that lets us run little tests. Let's go into the Python shell and try it out. Start it by typing `python` at the command-line:

```
python
Python 3.8.1 (default, Jan  8 2020, 16:15:59)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda
Type "help", "copyright", "credits" or "license"
>>>
```

Once inside the Python shell, we can try out the `assert` keyword:

```
>>> assert 1 == 1
>>> assert 1 == 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
AssertionError
```

```
>>>
```

By writing functions that use the `assert` key, we are able to write simple yet effective tests for anything we need in our projects.

This tests ensure that if we make any breaking changes to the `get_absolute_url()` method of our `user` model, these changes are caught quickly. This is a simple, clear example, which is a *good thing*. In our experience clarity in testing goes a long way towards preserving the stability of a website. If we have to write tests for our tests because our tests are too complex, we're probably making a mistake.

Now that we've seen how tests are constructed, let's write some more.

But how do we know where to begin writing tests?



Python shell or Python REPL?

Technically speaking, the Python shell is a REPL. Indeed, a REPL can be a programming shell, but not necessarily.

If that sounds confusing (or seems pedantic), just stick with the term "shell". It's what most people are familiar with anyway.

26.7 Introducing coverage.py

The `coverage.py` library is a tool that examines our project for code not covered by tests. This allows us to quickly determine places where we can add meaningful tests.

Do keep in mind that `coverage.py` isn't perfect and test coverage doesn't equate to bug free. Nevertheless, test-covered code is always more stable in the long run.

26.8 Using coverage.py

Normally there is a bit of setup to use `coverage.py`, but fortunately for us `django-crash-starter` has taken care of all the work in advance. All we have to do is run the following at the command line:

```
coverage report
```

It should display the percentage of test coverage per module and the overall total as shown below:

Name	Stmts	Miss	Cover
<hr/>			
<code>everycheese/__init__.py</code>	2	0	100%
<code>everycheese/conftest.py</code>	13	0	100%
<code>everycheese/contrib/__init__.py</code>	0	0	100%
<code>everycheese/contrib/sites/__init__.py</code>	0	0	100%

everycheese/users/__init__.py	0	0	100%
everycheese/users/admin.py	12	0	100%
everycheese/users/apps.py	10	0	100%
everycheese/users/forms.py	18	0	100%
everycheese/users/models.py	9	0	100%
everycheese/users/urls.py	4	0	100%
everycheese/users/views.py	23	0	100%
<hr/>			
TOTAL	91	0	100%

26.9 Generate HTML Coverage Reports

In addition to the broad coverage report that we just saw, we can generate a more detailed interactive HTML coverage report. Type this:

```
coverage html
```

Now open `htmlcov/index.html` in a browser. Notice how we can click on any particular module to see exactly which lines are missing test coverage. Not that there are any missing lines, but as we add code in future chapters this will display code we've added that doesn't have tests yet.

26.10 More Users App Tests

There are 3 other modules of tests generated for our starter users app:

- `test_forms.py`
- `test_urls.py`
- `test_views.py`

Study the comments in those tests to learn how those tests work.

26.II Summary

In this lesson, we covered:

1. Reasons to write tests for our project
2. How to run tests
3. The starter tests that come with the users app
4. How to determine how much of a project is covered by tests

27 User Profile Cheese

What if, instead of using a computer or tablet, we could browse the internet with a block of cheese?

Our device would technically be a cheese computer, or a “cheeseputer” for short. It would be the ideal device for browsing EveryCheese.

On our cheeseputer, here is what the user profile (user detail) page would look like.

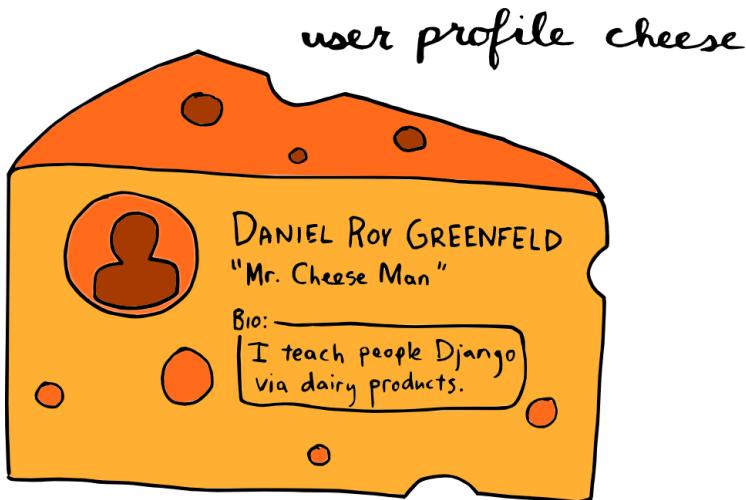


Figure 25: A block of user profile cheese

27.I Avatars Will Fit Into Cheese Holes

We haven't implemented avatars yet, but when we do in an upcoming A Wedge of Django extension, they'll fit into the holes of our cheeseputer, as shown.

28 The Cheeses App and Model

EveryCheese is still missing one thing: cheese! We're going to start building out the cheese-related functionality by creating a **cheeses** app. We'll then create a **cheese** model, and we'll make and apply migrations.

Remember, a Django app is a directory within our Django project containing code focused on doing one thing well. A typical Django project contains many Django apps.

28.1 Make Sure We're at the Project Root

At the command line, let's check that we are still in the root of the project. If we type `ls`, we should see a file called `manage.py` among many others.

28.2 Create an App Called Cheeses

Use `startapp` to create a new app called **cheeses**:

```
python manage.py startapp cheeses
```

The `startapp` command creates starting boilerplate for a Django app. After it runs, we'll see that an app directory called `cheeses` has been created. Inside, all the familiar Django app files are there, such as `models.py`.

28.3 Move It Into the Project

The `cheeses` app was created in the current directory, but the other Django apps aren't in the current directory. They're all in an inner `everycheese/` directory.

It's a common practice in the Django world to keep all our Django apps inside one directory. In the `django-crash-starter` project layout, the `apps` directory has the same name as the outer project directory.

Move `cheeses/` to the inner `everycheese/` directory:

On Linux and Mac:

```
mv cheeses everycheese/
```

On Windows:

```
move cheeses everycheese\
```

We should now have it at the same level as our `users` app, with all our other Django apps.

28.4 Set the Cheese Config

In the `everycheese/cheeses/apps.py` module, change the value of `name` to be “`everycheese.cheeses`”. It should look like this:

```
name = "everycheese.cheeses"
```

28.5 Add It to the List of Installed Apps

Open up the common settings file `base.py`. We can see here that `django-crash-starter` breaks up the `INSTALLED_APPS` tuple into `DJANGO_APPS`, `THIRD_PARTY_APPS`, and `LOCAL_APPS`:

```
INSTALLED_APPS = DJANGO_APPS + THIRD_PARTY_APPS + LOCAL_APPS
```

This is to help keep our apps organized. The naming convention here is:

- `DJANGO_APPS`: Apps that are part of the core Django codebase.
- `THIRD_PARTY_APPS`: Reusable Django apps available from the Python Package Index. *Third-party* means not part of Django core.
- `LOCAL_APPS`: Django apps that are internal to our Django project, such as the `users` app.

Add `everycheese.cheeses` to `LOCAL_APPS`:

```
LOCAL_APPS = [
    "everycheese.users.apps.UsersConfig", # custom users app
    "everycheese.cheeses.apps.CheesesConfig", # cheese info app
]
```

Now the *cheeses* app is installed into the project.

28.6 Add the Cheese Model

Open up `cheeses/models.py`.

Delete the placeholder comment saying `# Create your models here.`. It's fine to delete comments like this – there's no reason to keep it around. In the Django world, a `models.py` is obviously where we create our models.

28.6.1 Start the Cheese Model Like Any Other Model

Define a `Cheese` model which subclasses the `TimeStampedModel` from `django-model-utils`, a third-party package. Give it a standard `name` field.

```
from django.db import models

from model_utils.models import TimeStampedModel
```

```
class Cheese(TimeStampedModel):
    name = models.CharField("Name of Cheese", max_length=255)
```

`TimeStampedModel` automatically gives the model `created` and `modified` fields, which automatically track when the object is created or modified. We like to define all our models as subclasses of `TimeStampedModel`.

28.6.2 Add an Autopopulating Cheese Slug

Add a slug field. The `slug` will populate automatically from the `name` field, rather than requiring the user to enter it.

```
from django.db import models

from autoslug import AutoSlugField
from model_utils.models import TimeStampedModel


class Cheese(TimeStampedModel):
    name = models.CharField("Name of Cheese", max_length=255)
    slug = AutoSlugField("Cheese Address",
        unique=True, always_update=False, populate_from="name")
```



Illustration © Audrey Roy Greenfeld

Figure 26: Slugs are awesome!

28.6.3 Add Description Field

Continuing from above, we'll give cheeses a description. Add this to the `Cheese` model:

```
description = models.TextField("Description", blank=True)
```

This is a `TextField` because there's a good chance that users might enter long descriptive text. A good rule of thumb is to use `TextField` rather than `CharField` whenever there might be a need for more than 255 characters.

28.6.4 Add Firmness Field

Finally, add a `firmness` field to the same `cheese` model. Give it enum-powered choices ranging from soft to hard.

```
class Cheese(TimeStampedModel):  
  
    class Firmness(models.TextChoices):  
        UNSPECIFIED = "unspecified", "Unspecified"  
        SOFT = "soft", "Soft"  
        SEMI_SOFT = "semi-soft", "Semi-Soft"  
        SEMI_HARD = "semi-hard", "Semi-Hard"  
        HARD = "hard", "Hard"  
  
        # Other Fields Here...  
  
    firmness = models.CharField("Firmness", max_length=20,  
                               choices=Firmness.choices, default=Firmness.UNSPECIFIED)
```

Note that we defined the firmness constants as variables within the scope of the `cheese` model. This allows us to do things like this comparison:

```
if cheese.firmness == Cheese.Firmness.SOFT:  
    # Do Something
```

No need to type the above comparison. It's here just to show what we can do.



About Django 3+ choice Classes

For those with pre-3.0 Django experience, `models.TextChoices` and `models.IntegerChoices` are new features of the framework. Built as an extension of Python 3.4+ Enum, it makes choice field documentation easier on the coder, hence projects more maintainable. We just scratch the surface here, and more information can be found at:

- Django docs on Choice classes: <http://feld.to/django-3-choice-classes>
- <https://docs.python.org/3/library/enum.html>

28.7 Create and Run the Initial Cheese Migration

Run `makemigrations` and `migrate`:

```
python manage.py makemigrations cheeses
python manage.py migrate cheeses
```

This migration should have created a table for the `cheese` model.

29 Trying Out the Cheese Model

We're going to experiment with our `cheese` model via `shell_plus`, an enhanced version of the interactive Django shell.

29.1 Let's Try It Out!

At the command line, start up `shell_plus`:

```
python manage.py shell_plus
```

29.2 Django Shell vs. Shell Plus

The regular Django shell (`python manage.py shell`) is the same as starting the Python shell, but when we run it, it preloads the appropriate Django settings file as specified in `manage.py`.

`shell_plus`, provided by the `django-extensions` third-party package, takes this a step further by autoloading all our Django model classes. In the plain Django shell, we would have to type this in order to import the `cheese` model class:

```
>>> from everycheese.cheeses.models import Cheese
```

However, in `shell_plus`, the above import is already done for us as a convenience.

29.3 Create a Cheese

Let's instantiate a `Cheese` object. Type this now:

```
in [1]: cheese = Cheese.objects.create(  
...     name='Colby',  
...     description='Similar to Cheddar but without undergoing'  
...             'the cheddaring process, Colby is a mild,'  
...             'creamy cheese that was first created in 1885',  
...             'in Colby, Wisconsin.'),  
...     firmness=Cheese.Firmness.SEMI_HARD)
```

Behind the scenes, the above code created a new record in the `cheeses_cheese` table in the database.

29.4 Evaluate and Print the Cheese

If we evaluate the object that we just instantiated, we get:

```
in [2]: cheese  
<Cheese: Cheese object>
```

Printing the string representation of our object gives us:

```
in [3]: print(cheese)
Cheese object
```

Cheese object? That's not very descriptive. How do we know which cheese object this refers to? We don't. It's also rather insulting to Colby cheese.

29.5 View the Cheese's Name

To see the actual name of the cheese, type:

```
in [4]: cheese.name
'Colby'
```

We can make cheese objects print out better. Exit the command line.

29.6 Add a String Method

When Python classes have a `__str__()` method, that method is called and the result is printed whenever we try to print an instance object. We're about to see this in action.

Open up `models.py` of our *cheeses* app again.

Add a `__str__()` method to our `cheese` model. This goes at the bottom, after all the fields:

```
def __str__(self):
    return self.name
```

29.7 Exit shell_plus

For this to take effect, we need to exit `shell_plus`. Do that by pressing `ctrl+D`.

29.8 Try It Again

Let's get that cheese object and try to print it again:

```
python manage.py shell_plus
in [1]: cheese = Cheese.objects.last()
in [2]: print(cheese)
Colby
```

Great! Now when we printed `cheese`, we got `Colby` as the string representation of the `cheese` object.

Exit `shell_plus` again. We're done using it for now.

29.9 Commit Our Work

We've made some major changes to `EveryCheese`, so it's time to back up our work in version control.

Check which files have changed, then add them to a commit.

```
git status  
git add everycheese/cheeses/  
git add config/settings/base.py
```

Commit the changes and push:

```
git commit -m "Add cheeses app"  
git push origin master
```

30 Test Coverage

30.1 How to Check Test Coverage

Now's a good time to recheck test coverage. At the command line:

```
coverage run -m pytest
coverage report
coverage html
```

Open `htmlcov/index.html` in a browser:

Why isn't our test coverage at 100%? Well, `django-crash-starter` provided us with 100% test coverage, but adding the `cheeses` app dropped us to its current score.

Click around the coverage report to find exactly what dropped our score. In this case, look at the coverage for `everycheese.cheeses.models`.

The line that dropped our test coverage was this line inside of the `__str__()` function beginning with `return`:

```
def __str__(self):
    return self.name
```

Coverage report: 99%

Module	statements	missing	excluded	coverage
everycheese/__init__.py	2	0	0	100%
everycheese/cheeseses/__init__.py	0	0	0	100%
everycheese/cheeseses/admin.py	1	0	0	100%
everycheese/cheeseses/apps.py	3	0	0	100%
everycheese/cheeseses/models.py	16	1	0	94%
everycheese/conftest.py	13	0	0	100%
everycheese/contrib/__init__.py	0	0	0	100%
everycheese/contrib/sites/__init__.py	0	0	0	100%
everycheese/users/__init__.py	0	0	0	100%
everycheese/users/admin.py	12	0	0	100%
everycheese/users/apps.py	10	0	0	100%
everycheese/users/forms.py	18	0	0	100%
everycheese/users/models.py	9	0	0	100%
everycheese/users/urls.py	4	0	0	100%
everycheese/users/views.py	23	0	0	100%
Total	111	1	0	99%

coverage.py v5.1, created at 2020-06-17 14:14

Figure 27: 94% coverage

You see, coverage assumes that imported code like `models.CharField` just works. There's no need to test the `Cheese` model's fields, because the usage of those fields has already been tested in Django's codebase.

However, the `__str__()` function that we defined is specific to our code. `Coverage.py` didn't find any test coverage for it, so it marked it as uncovered.

Coverage for `everycheese/cheeses/models.py` : 94%

16 statements 15 run 1 missing 0 excluded

```
1 from django.db import models
2
3 from autoslug import AutoSlugField
4 from model_utils.models import TimeStampedModel
5
6
7 class Cheese(TimeStampedModel):
8     FIRMNESS_UNSPECIFIED = "unspecified"
9     FIRMNESS_SOFT = "soft"
10    FIRMNESS_SEMI_SOFT = "semi-soft"
11    FIRMNESS_SEMI_HARD = "semi-hard"
12    FIRMNESS_HARD = "hard"
13    FIRMNESS_CHOICES = (
14        (FIRMNESS_UNSPECIFIED, "Unspecified"),
15        (FIRMNESS_SOFT, "Soft"),
16        (FIRMNESS_SEMI_SOFT, "Semi-Soft"),
17        (FIRMNESS_SEMI_HARD, "Semi-Hard"),
18        (FIRMNESS_HARD, "Hard"),
19    )
20
21    name = models.CharField("Name of Cheese", max_length=255)
22    slug = AutoSlugField("Cheese Address",
23        unique=True, always_update=False, populate_from="name")
24    description = models.TextField("Description", blank=True)
25
26    firmness = models.CharField("Firmness", max_length=20,
27        choices=FIRMNESS_CHOICES, default=FIRMNESS_UNSPECIFIED)
28
29    def __str__(self):
30        return self.name
```

Figure 28: Cheese model coverage

30.2 Create a Module for Cheese Model Tests

Django's `startapp` command gave our `cheeses` app a `tests.py` file by default. However, a single `tests` module

is going to get crowded with code. Let's turn our `tests` module into a `tests` package.

In `everycheese/cheeses/`, let's now:

- Delete `tests.py`
- Create a directory in its place called `tests/`
- Inside of `tests`, create `__init__.py` and `test_models.py`.

Common practice is to delete `tests.py` and create individual test modules for each file of the app. (Later, the `cheeses` app may have a `test_views.py`, `test_admin.py`, etc.)

Note: even though `__init__.py` is technically optional on Python 3, create it anyway. By including the `__init__.py`, it empowers us to use relative imports in our tests.

30.3 Let's Test the `__str__()` Method.

One might say, this is silly! But it's actually important because we never know, there could one day be code in `EveryCheese` that relies on the value of `__str__()` behaving a certain way. If that behavior changes and we don't catch it in time, it will cause us to be embarrassed in front of our users.

Before we can write a test for this, we need to import the `pytest` library and link it to our database. Start off `cheeses/tests/test_models.py` with this:

```
import pytest

# Connects our tests with our database
pytestmark = pytest.mark.djangoproject_db
```

Import the `cheese` model class, since it's what's being tested in this module:

```
from ..models import Cheese
```

Here we use a relative import to get `cheese` from `models.py` of the `cheeses` app. This pattern is the same as that of the `users` app's `test_models.py`.

Then add our test function:

```
def test__str__():
    cheese = Cheese.objects.create(
        name="Stracchino",
        description="Semi-sweet cheese eaten with starches.",
        firmness=Cheese.Firmness.SOFT,
    )
    assert cheese.__str__() == "Stracchino"
    assert str(cheese) == "Stracchino"
```

This is a good starter example of how to write tests on model methods. (In a later lesson we'll improve upon this by using a factory, but for now this is a good place to start.)

Run coverage.py again:

```
coverage run -m pytest  
coverage report  
coverage html
```

Open the `htmlcov/index.html` file again. The test coverage of `everycheese.cheeses.models` should have increased to 100%. If we click through to see the file in detail, we should see that no lines are highlighted for lacking test coverage.

30.4 Enter the Game of Test Coverage

Since test coverage is a measure of how much of our project is covered by tests, we can use this value as part of a game. The goal of the game is simple:

Get our test coverage score to 100% and keep it there.

30.5 Rules of the Game

Any time we add new code, we'll be running a coverage report.

- If the number drops, then we are losing.
- If the number is maintained, we draw.
- And if the number goes up (even after a drop), then we are on our way to victory.

The way to maintain our score is by adding more tests to cover things that aren't covered. That's the challenge. Every time we add or modify existing code we risk dropping our score, so we're playing constantly.

As for what's a good coverage percentage, there's no one correct answer. Read more: What is a reasonable code coverage % for unit tests (and why)?¹⁵

30.6 What's the Point of All This Any-way?

Primarily, it's a chance to justify eating more cheese.

Other, less important things to think about are:

1. Do we want to be able to easily upgrade our site whenever Django or any of the dependencies of our site put out new releases?
2. Do we want our deployments to go smoothly?
3. Are people's health or finances dependent on our project?

If we answered yes to any of the above, then maximizing our test coverage is in our best interest.

¹⁵<https://stackoverflow.com/questions/90002/what-is-a-reasonable-code-coverage-for-unit-tests-and-why>

30.7 Commit the Cheese Model Tests

```
git add -A  
git commit -m "Add Cheese model tests"  
git push origin master
```

30.8 Summary

We've covered:

1. Testing the `cheese` model
2. The game of test coverage
3. Committing our new cheeses app tests

3I The Cheeserator

With Django, we can build absolutely anything imaginable.

Now imagine that we wanted to build a contraption called The Cheeserator, a machine for doing whatever we want to do to cheese.



Figure 29: The Cheeserator device for manipulating cheese.

Yes, we can build this with Django. We can do anything.

32 Adding Cheeses to the Admin

32.1 We Need a Superuser

Create a Django superuser to get full admin access.

```
python manage.py createsuperuser
```

Enter these values:

- username: admin
- email: admin@example.com
- password: Peau18mUPCXkjQ*



Copy this password into a password manager!

It's bad practice to use simplistic passwords, even for local development. We want to promote strong security habits and this is one of them. Also, don't stick passwords into plaintext or spreadsheet files. Instead, lean on secure password managers like our preferred **1Password** or open source options such as **Bitwarden** and **KeePass**.

32.2 Go to the Admin

Start up `runserver` if it's not already running.

Go to <http://127.0.0.1:8000/admin>. Log in with our superuser account.

32.3 Explore the Admin So Far

Look around to see what's currently available in the Django admin. We'll notice some functionality that comes with Django by default, and some that comes with `django-crash-starter`.

32.4 Register Cheese Model in the Admin

Open `admin.py` of our `cheeses` app in our text editor. Replace the code in the file with the following:

```
from django.contrib import admin
from .models import Cheese

admin.site.register(Cheese)
```

Save the file and go back to the Django admin in our browser. Refresh the page. We should now have an admin UI for entering cheeses, accessible from the *Cheeses* section:

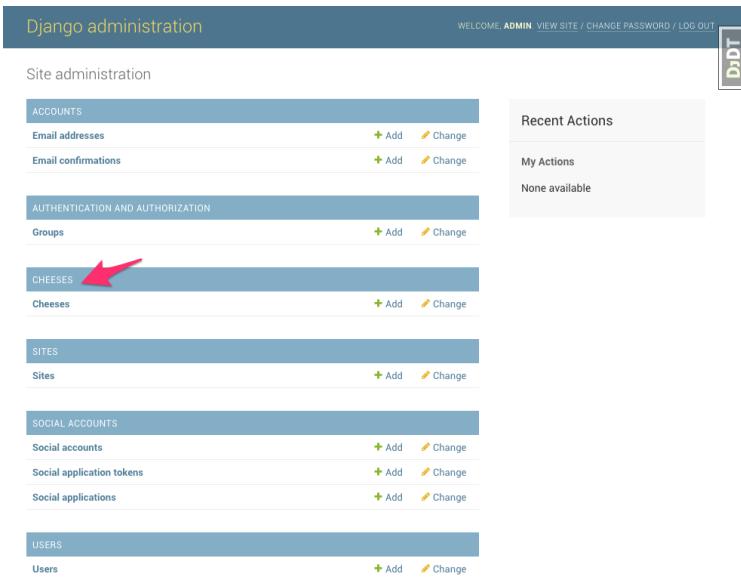


Figure 30: Cheeses on the Django admin index page

32.5 Find Colby

Earlier, we created a cheese by calling `cheese.objects.create()` in `shell_plus`. The end result was that a new cheese record was created.

In the admin, find the list of cheeses. **Colby**, the cheese we added earlier, should be the only cheese in the list.

As of now, there are 2 ways to create new cheeses in `EveryCheese`:

1. By writing code that creates cheeses, as we did earlier

lier.

2. By creating cheeses in the Django admin.

We've already created a cheese programmatically; next, we'll create 3 more via the Django admin.

32.6 Enter More Cheeses via the Admin

Under *Cheeses*, click on *Add*. We will be taken to the **Add cheese** form:

The screenshot shows the Django Admin interface for adding a cheese. The title bar says "Add cheese | Django site admin". The URL in the address bar is "127.0.0.1:8000/admin/cheeses/cheese/add/". The main title is "Django administration". The breadcrumb navigation shows "Home > Cheeses > Cheeses > Add cheese". The form has fields for "Name of Cheese" (Camembert), "Description" (A French cheese with a white, powdery rind and a soft, delicately salty interior.), and "Firmness" (set to Soft). At the bottom are buttons for "Save and add another", "Save and continue editing", and a large blue "SAVE" button.

Figure 31: Add cheese in the Django admin

Add the following cheeses via the Django admin UI:

1. Camembert

- (a) Description: A French cheese with a white, powdery rind and a soft, delicately salty interior.
- (b) Firmness: soft

2. Gouda

- (a) Description: A Dutch yellow cheese that develops a slight crunchiness and a complex salty toffee-like flavor as it ages.
- (b) Firmness: hard

3. Cheddar

- (a) Description: A relatively hard, pale yellow to off-white, and sometimes sharp-tasting cheese.
- (b) Firmness: hard

Don't skip this step. We'll need these cheeses later in this course.

32.7 Some Notes About Using the Django Admin

32.7.1 Keep People Who Use It to a Small Number.

- 1. Must be power-users, understanding how models work
- 2. Not optimized for scaling

3. The Django admin can do a lot of amazing things for a little amount of work. But then we run out of options and are stuck



Don't Use `list_editable`. Ever!

As records are tracked not by primary keys but by their position in a displayed list, this is a serious risk. On multi-user projects:

1. Cheeses are created in descending order.
2. Daniel brings up the list field for cheeses and begins making changes
3. Audrey decides to add the cheese *Brie*. Since it's the last item added, it's the first Cheese returned on a query.
4. Daniel finally saves his cheese changes, but the records he saw all receive the data from the next record, corrupting at least 50 records (Django's default admin display number).

32.8 Commit Changes

```
git commit -am "Register Cheese model with the admin"  
git push origin master
```

32.9 Summary

In this lesson we:

1. Created a superuser
2. Learned how to use the Django admin to **rapidly** create a barebones cheese management system.
3. Admonished us to keep the volume of users of the admin system to a small number.
4. Warned against use of `list_editable`.

33 Behind the Curtain

Having access to the Django admin is like having access to the Wizard of Oz's control panels. It makes us more powerful than we actually are.

Imagine if a wedge of cheese had access to the Django admin controls of the EveryCheese site. Who knows what might happen. It could be rather dangerous.

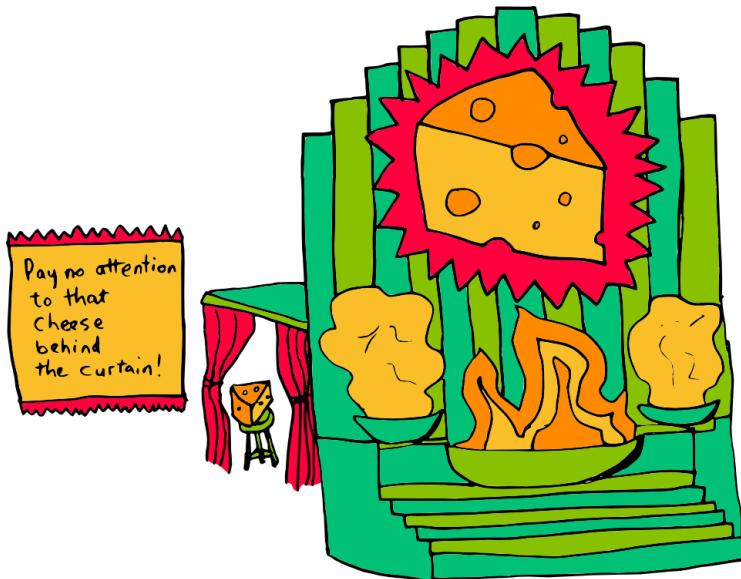


Figure 32: The cheese behind the curtain

33.I We Are in Control

Fortunately for us, we don't have to create superuser accounts for cheeses if we don't feel like it. Just say no. If they ask, tell them to follow the yellow brick road back to the refrigerator.

34 Class-Based View Fundamentals

This lesson will cover:

- The Simplest Class-Based View
- Advantages of Class-Based Views
- Tips for Writing Class-Based Views

There's no need to type out the examples in this lesson.
Just sit back and learn the concepts.

34.1 The Simplest Class-Based View

Here, study the code and observe that:

- We create a very simple view by subclassing the base `View` class.
- It returns an HTTP response consisting of a simple text string.

```
# No need to type this out, just study it!
from django.http import HttpResponseRedirect
from django.views.generic import View
```

```
class MyView(View):

    def get(self, request, *args, **kwargs):
        return HttpResponse('Response to GET request')
```

The above is analogous to this function-based view (or FBV for short):

```
# No need to type this out, just study it!
def my_view(request, *args, **kwargs):
    return HttpResponse('Response GET request!')
```

34.2 Adding More HTTP Methods

Just as we can define a `get()` method to handle GET requests, we can also do the same with other HTTP methods such as POST and DELETE.

```
# No need to type this out, just study it!
class MyView(View):

    def get(self, request, *args, **kwargs):
        return HttpResponse('Response to GET request')

    def post(self, request, *args, **kwargs):
```

```
    return HttpResponse('Response to POST request')

    def delete(self, request, *args, **kwargs):
        return HttpResponse('Response to DELETE request')
```

The above CBV code is analogous to this FBV approach:

```
# No need to type this out, just study it!
def my_view(request, *args, **kwargs):
    if request.method == 'POST':
        return HttpResponse('Response POST request!')
    elif request.method == 'DELETE':
        return HttpResponse('Response DELETE request!')
    return HttpResponse('Response GET request!')
```

Or this FBV approach:

```
# No need to type this out, just study it!
def my_view(request, *args, **kwargs):
    METHOD_DISPATCH = {
        'POST': HttpResponse('Response POST request!'),
        'DELETE': HttpResponse('Response DELETE request!'),
    }
    DEFAULT = HttpResponse('Response GET request!')
    return METHOD_DISPATCH.get(request.method, DEFAULT)
```

Or any number of other FBV approaches. Which makes us wonder, what is the correct FBV approach?

Hint: There isn't a 'correct' or 'standard' FBV approach. It's every coder for themselves.



What is the difference between GET, POST, and DELETE?

These are different HTTP methods used by clients such as web browsers to access application servers like Django. Here's a list of samples of those three methods in use in the context of interacting with a web site:

GET: Used to read web pages

POST: Used to submit forms

DELETE: Used with an API to delete a resource such as a web page

34.3 Advantages of Class-Based Views

- Composition
- Intelligent Defaults
- Standardized HTTP method handling
- Every action generally has a place to be
- Easier than Writing Decorators

34.4 Composition

Here's an example mixin:

```
# No need to type this out, just study it!
class BaseProjectMixin:
    def complex_data(self, request):
        # Sophisticated Logic Here
        return 'sophisticated result'
```

Every view inheriting from `BaseProjectMixin` gets a `complex_data()` method as a result:

```
# No need to type this out, just study it!
from django.shortcuts import render
from django.views.generic import View
from .mixins import BaseProjectMixin

class ItemDetailView(BaseProjectMixin, View):
    def get(self, request, *args, **kwargs):
        value = self.complex_data(request)
        return render(request, 'details.html',
                      {'value': value})

class ThingDetailView(BaseProjectMixin, View):
    def get(self, request, *args, **kwargs):
        value = self.complex_data(request)
        return render(request, 'things.html', {'value': value})
```

<http://ccbv.co.uk/projects/Django/3.0/django.views.generic.base/View/>

34.5 Composition Part II

Various open-source packages expect you to extend the default views they provide. Django REST Framework is the most notable project to take this approach.

34.6 Intelligent Defaults

A CBV-based update view

```
# No need to type this out, just study it!
from django.views.generic import UpdateView
from .models import Item

class ItemUpdateView(UpdateView):
    model = Item
    fields = ['name', 'description', 'price']
```

- Auto-creates form based off `Item` and `fields`, called `form` in template.
- Default template name of `items/item_form.html`.
- Item record can be accessed in template as `item` or `object`.

<https://ccbv.co.uk/projects/Django/3.0/django.views.generic.edit/UpdateView/>

34.7 Standardized HTTP Method Handling

```
# No need to type this out, just study it!
from django.http import HttpResponseRedirect
from django.views.generic import View

class SpecialView(View):

    def delete(self, request, *args, **kwargs):
        return HttpResponseRedirect('HTTP deletes!')

    def post(self, request, *args, **kwargs):
        return HttpResponseRedirect('HTTP posts!')
```

- Returns HTTP 405 for these HTTP methods:
 - GET
 - PUT
 - OPTIONS

34.8 Tips for Writing Class-Based Views

- Stick with the defaults
- Don't go crazy with multiple inheritance

- `request.GET` and `request.POST` are dictionary-like objects that contain data sent by the user
- For extra CBV behaviors not provided by core Django, django-braces can provide to be useful:
<https://django-braces.readthedocs.io>
- For quick reference documentation: <https://ccbv.co.uk>
- If doing something crazy-sophisticated, drop down to a basic `django.views.generic.View`.
- For more advanced views:
 - GET/POST arguments are inside of `self.kwargs`
 - You can access the request at `self.request`

35 Writing the Cheese List View

In our text editor, open `cheeses/views.py`. There will be some stub code in there. Replace it with the following:

```
from django.views.generic import ListView, DetailView

from .models import Cheese

class CheeseListView(ListView):
    model = Cheese
```

This is a simple view that lists all the cheeses on a single page.

When we import the `Cheese` model, we use a relative import because we're importing it from within the same app.

36 Wiring in the List View URL

36.1 Define Cheese URL Patterns

In our cheeses app, create a new file called `urls.py`. Open it in our text editor and add the following code:

```
# everycheese/cheeses/urls.py
from django.urls import path
from . import views

app_name = "cheeses"
urlpatterns = [
    path(
        route='',
        view=views.CheeseListView.as_view(),
        name='list'
    ),
]
```

36.2 Include Cheese URLs in Root URLConf

In order to make the cheeses app's URLConf work, we still have to include it in our root URLConf. Open `config/urls.py` and add this line right after the comment “`# Your stuff: custom urls includes go here`”:

```
# Your stuff: custom urls includes go here
path(
    'cheeses/',
    include('everycheese.cheeses.urls',
            namespace='cheeses'),
),
```

36.3 See the View in Action

We're about to trigger an error, but that's okay!

In our web browser let's navigate to `http://127.0.0.1:8000/cheeses/`

We should see a `TemplateDoesNotExist` error:

The screenshot shows a browser window with the URL `127.0.0.1:8000/cheeses/`. The error message is: **TemplateDoesNotExist at /cheeses/cheeses/cheese_list.html**. The stack trace includes:

```
Request Method: GET
Request URL: http://127.0.0.1:8000/cheeses/
Django Version: 3.0.2
Exception Type: TemplateDoesNotExist
Exception Value: cheeses/cheese_list.html
Exception Location: /opt/anaconda3/envs/everycheese/lib/python3.8/site-packages/django/template/loader.py in select_template, line 47
Python Executable: /opt/anaconda3/envs/everycheese/bin/python
Python Version: 3.8.1
Python Path: ['/Users/drg/projects/everycheese',
 '/opt/anaconda3/envs/everycheese/lib/python38.zip',
 '/opt/anaconda3/envs/everycheese/lib/python3.8',
 '/opt/anaconda3/envs/everycheese/lib/python3.8/lib-dynload',
 '/opt/anaconda3/envs/everycheese/lib/python3.8/site-packages',
 '/Users/drg/projects/everycheese/everycheese',
 '/Users/drg/projects/everycheese/everycheese']
Server time: Thu, 30 Jan 2020 01:01:11 +0000
```

Figure 33: TemplateDoesNotExist

This is good news! It means:

1. Django's URL dispatcher found match for '`cheeses/`' and sent the rest of the URL to `everycheese.cheeses.urls`.
2. The remainder of the URL is a match for '`''`', so `CheeseListView.as_view()` was called.
3. The `CheeseListView` looked for a template to display, found nothing, so returned a `TemplateDoesNotExist` error.

We'll add that template in the next chapter.

37 The Cheese List Template

We love looking at lists of cheeses. So much taste and flavor!

37.1 Create the List Template

Let's create a blank template for `CheeseListView`:

1. In `templates/`, create a `cheeses/` directory. This will hold all the templates for our *cheeses* app.
2. In `templates/cheeses/`, create a file called `cheese_list.html`

Make sure we have `runserver` still running.

Go to <http://127.0.0.1:8000/cheeses/> again. When we load the page, it shows up blank as expected.

37.2 Fill in the Template

Fill `cheese_list.html` with the following code:

```
{% extends "base.html" %}

{% block title %}Cheese List{% endblock title %}
```

```
{% block content %}

<h2>Cheese List</h2>

<ul>
    {% for cheese in cheese_list %}
        <li><a href="#TODO">{{ cheese.name }}</a></li>
    {% endfor %}
</ul>

{% endblock content %}
```

In this template, we display all the cheeses in a bulleted list. We iterate over `cheese_list`, which `CheeseListView` automatically provides to the template. This is a built-in copy of the default `object_list` value that `ListView` provides by default. If our model was `Butter` instead of `Cheese`, then we would access `butter_list` instead.

The `#TODO` links will go to the detail pages for each cheese. We'll implement them very soon.

37.2.1 How Django Templates Use “extends” and “block”

The Django Template Language is designed to maximize code reuse. This isn't just to make things easier for developers, it also encourages projects to have a unified look and feel. To facilitate that code reuse, Django primarily

relies on two template tags, `{% extends %}` and `{% block %}`. Here's how they are used in this chapter:

`{% extends "base.html" %}` This tag calls the projects base HTML template. That becomes the foundation of the `cheese_list.html` template we are writing.

`{% block content %}` This tag tells Django that it wants to put content from `cheese_list.html` into the `base.html` file's content block.

37.3 See the List in Action

Refresh <http://127.0.0.1:8000/cheeses/> again. We should see the cheese list in action:

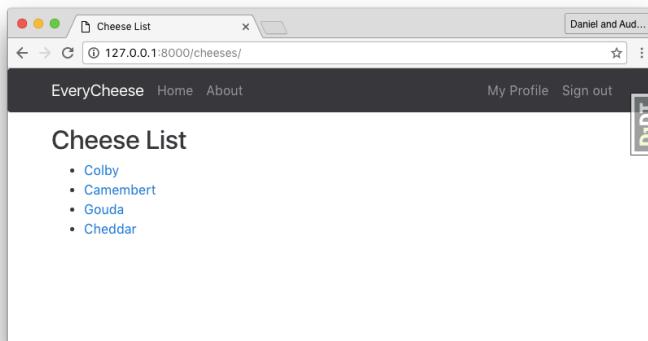


Figure 34: Cheese list

- All the cheeses we entered earlier via the Django admin or shell should be in the list.
- The links should be `#TODOs`. We haven't implemented them yet.

37.4 Add a Nav Link to the Cheese List

Open up `templates/base.html` in our text editor. This is the project's main sitewide base template.

Find the code for the navbar:

```
<ul class="navbar-nav mr-auto">
  <li class="nav-item active">
    <a class="nav-link" href="{% url 'home' %}">Home</a>
    <span class="sr-only">(current)</span>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="{% url 'about' %}">About</a>
  </li>
</ul>
```

Insert a link to the Cheese List page:

```
<li class="nav-item">
  <a class="nav-link" href="{% url 'cheeses:list' %}">Cheeses</a>
</li>
```

Here we use the `{% url %}` template tag to generate the absolute URL. This is better than linking to the absolute URL directly. The reason is that no matter what we change the link to, the `base.html` template will pick it up. For example, if we decided to switch the URL from **cheeses** to **les-fromages** (French for “cheeses”) in `urls.py`, we wouldn’t have to go into templates to accomodate that change. Instead, `{% url 'cheeses:list' %}` would automagically link to `127.0.0.1:8000/les-fromages/`.

In any case, now when we go back to the browser and refresh the Cheese List page, we should see the *Cheeses* link in the navbar:

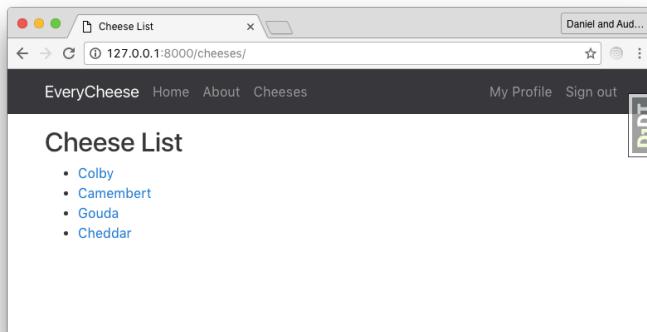


Figure 35: Cheese List With Nav Link

Click around the site and then click on *cheeses* to make sure the link works as expected.

37.5 Explaining the cheeses:list Name

In the previous chapter, inside the `cheeses/urls.py` module we set two important values, the `app_name` and the `path name` for the `CheeseListView`. The specific values we set are seen below:

```
# Code inside cheeses/urls.py
app_name = "cheeses"
urlpatterns = [
    path(
        route='',
        view=views.CheeseListView.as_view(),
        name='list'
    ),
]
```

app_name This is how we set the `cheeses` app name in an explicit manner. Amongst other things, setting this here makes it easy for us as developers to identify which file we are working in.

path name The `CheeseListView` is given a URL name of `list`. We do this because it allows us greater flexibility in naming views across a project. Specifically, a too-simple naming approach starts to be uncomfortable on even medium-sized projects.

37.6 Commit Our Work

As always, commit the work!

```
git add -A  
git commit -m "Implement cheese list page, add navbar link"  
git push origin master
```

38 Add the CheeseDetailView

So far we can see the list of cheeses at <http://127.0.0.1:8000/cheeses/>. Next, let's make it possible for users to click through to see the detail page for any cheese.

Add this to the bottom of `cheeses/views.py`:

```
class CheeseDetailView(DetailView):  
    model = Cheese
```

Now we have the view that will power each of the cheese detail pages.

38.1 Add the CheeseDetailView URL Pattern

As usual, after defining a view, the next step is to wire in its corresponding URL pattern.

Add this to the `urlpatterns` list in `cheeses/urls.py`:

```
# URL Pattern for the CheeseDetailView  
path(
```

```
        route='<slug:slug>/',
        view=views.CheeseDetailView.as_view(),
        name='detail'
    ),
```

Our `urlpatterns` in `cheeses/urls.py` should look like this now:

```
urlpatterns = [
    path(
        route='',
        view=views.CheeseListView.as_view(),
        name='list'
    ),
    path(
        route='<slug:slug>',
        view=views.CheeseDetailView.as_view(),
        name='detail'
    ),
]
```

Now when we visit <http://127.0.0.1:8000/cheeses/colby/>, the requested URL will match '`<slug:slug>`' where `slug` is `colby`.

38.2 Link List Items to Detail Pages

In `cheeses/cheese_list.html` change:

```
<a href="#TODO">
```

To:

```
<a href="{% url 'cheeses:detail' cheese.slug %}">
```

Here we are using the `url` template tag to generate the absolute URL to each cheese's detail page.

If we refresh `http://127.0.0.1:8000/cheeses/` and click on **Colby** we'll go to the CheeseDetailView results. However, as we haven't added a template yet, what we get is the following:

If we see the image above, then we are doing it right. Go to the next chapter where we add the `cheese_detail.html` html template.



The screenshot shows a browser window with the URL `127.0.0.1:8000/cheeses/colby/cheeses/cheese_detail.html`. The page title is "TemplateDoesNotExist at /cheeses/colby/". The error message is "TemplateDoesNotExist at /cheeses/colby/ cheeses/cheese_detail.html". Below the error message, there is a detailed stack trace:

```
Request Method: GET
Request URL: http://127.0.0.1:8000/cheeses/colby/
Django Version: 3.0.2
Exception Type: TemplateDoesNotExist
Exception Value: cheeses/cheese_detail.html
Exception Location: /opt/anaconda3/envs/everycheese/lib/python3.8/site-packages/django/te
Python Executable: /opt/anaconda3/envs/everycheese/bin/python
Python Version: 3.8.1
Python Path: ['/Users/drg/projects/everycheese',
 '/opt/anaconda3/envs/everycheese/lib/python38.zip',
 '/opt/anaconda3/envs/everycheese/lib/python3.8',
 '/opt/anaconda3/envs/everycheese/lib/python3.8/lib-dynload',
 '/opt/anaconda3/envs/everycheese/lib/python3.8/site-packages',
 '/Users/drg/projects/everycheese/everycheese',
 '/Users/drg/projects/everycheese/everycheese']
Server time: Thu, 6 Feb 2020 06:32:28 +0000
```

Figure 36: Missing cheese_detail.html TemplateDoesNotExist exception

39 The Cheese Detail Template

39.1 Add a Cheese Detail Template

In `templates/cheeses/`, create a file called `cheese_detail.html`.

Fill in the detail template with this code:

```
{% extends "base.html" %}

{% block title %}Cheeses: {{ cheese.name }}{% endblock title %}

{% block content %}

<h2>{{ cheese.name }}</h2>

{% if cheese.firmness %}
    <p>Firmness: {{ cheese.get_firmness_display }}</p>
{% endif %}

{% if cheese.description %}
    <p>{{ cheese.description }}</p>
{% endif %}

{% endblock content %}
```

Some things to note:

- In a `DetailView`'s template, the object is accessible as the lowercased model name. In this case, the object is `cheese`.
- Before displaying any optional field, we first check if a value exists.
- `get_firmness_display` shows the cheese's firmness attribute in an attractive format. `get_FOO_display()` is a utility method created automatically for any

Django model field. So where we had a `firmness` choice field defined on our Cheese model, Django adds a `get_firmness_display()` method which we can call in templates or any place the instantiated model is available.

39.2 Visit Cheese Detail

Go back to the cheese list page in our browser. Refresh the page.

Click on a cheese in the list. This will take us to the cheese detail page:

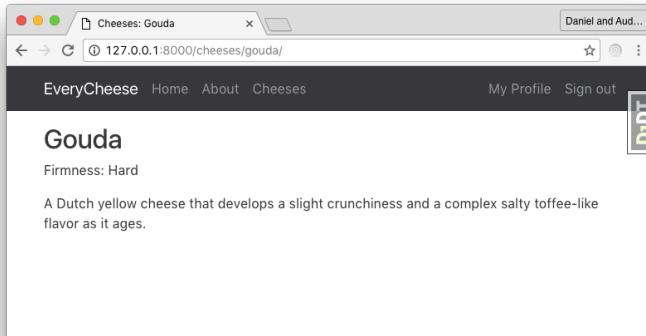


Figure 37: Cheese detail

Compare it to the template that we just filled in.

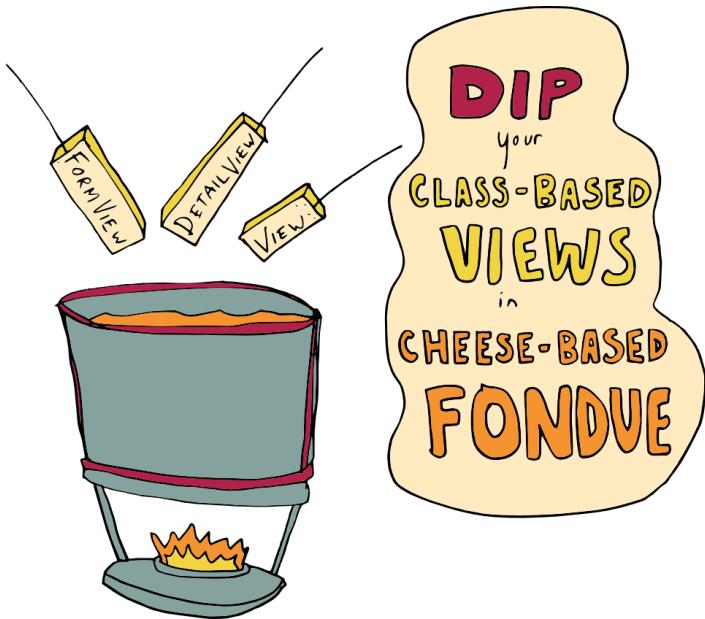
39.3 Commit Our Work

```
git add -A  
git commit -m "Implement cheese detail page"  
git push origin master
```

40 Where to Dip Class-Based Views

Class-based views are great. But there's something about them that is very difficult to understand.

That is, most people don't know where to dip them. Fortunately, we're about to explain where CBVs should be dipped. Grab a large fondue pot and make some cheese-based fondue. Any cheese will do.



4I Writing Factories for Tests

It's fun to describe lots of cheeses. We've done it in tests, in the admin, and we'll be doing it through the course.

4I.1 Produce Cheese Test Data From Factories

However, since we're building a real site, we're going to need real amounts of data. Hand coding cheeses and other things into our site works to some degree, but it doesn't let us test things in quantities of tens, hundred, or thousands.

What we're going to do instead is use a factory. One powered by `Factory Boy`, a third-party library included in the `django-crash-starter` library.

4I.2 Define a `CheeseFactory`

Create a new module in our cheeses app's test directory for factories: `cheeses/tests/factories.py`

Place the following code inside:

```

from django.template.defaultfilters import slugify

import factory
import factory.fuzzy

from ..models import Cheese

class CheeseFactory(factory.django.DjangoModelFactory):

    name = factory.fuzzy.FuzzyText()
    slug = factory.LazyAttribute(
        lambda obj: slugify(obj.name))
    description = factory.Faker(
        'paragraph', nb_sentences=3,
        variable_nb_sentences=True
    )
    firmness = factory.fuzzy.FuzzyChoice(
        [x[0] for x in Cheese.Firmness.choices]
    )

    class Meta:
        model = Cheese

```

CheeseFactory generates cheese objects (model instances). Think of it like cheese.objects.create(), but where the cheese data is autogenerated.

Observe that:

- The `name` of the cheese is autogenerated using `FuzzyText()`.
- The `slug` of the cheese is the slugified version of the name.
- The cheese's `description` is a randomly-generated paragraph.
- The cheese's `firmness` is randomly selected from all possible firmness choices.

4I.3 Try Out CheeseFactory

Let's go to `shell_plus` and try it out:

```
python manage.py shell_plus
```

In the shell, type this:

```
>>> from everycheese.cheeses.tests.factories import CheeseFactory
>>> cheese = CheeseFactory()
>>> cheese
<Cheese: sGwDovpLpggc>
```

What's going on here:

1. We import the `CheeseFactory` class that we just defined.
2. We instantiate a `cheese` object named `cheese` using `CheeseFactory`.

3. We evaluate the `cheese` object.

When we evaluate that cheese, we get a funny result. That's no cheese title!

That's because what `cheeseFactory` did was create a cheese with random data. While that's no good for looking at, it's great for testing. We'll see shortly how to use factories in unit tests.

41.4 Delete the Bogus Cheese

Because we just created a random cheese with `cheeseFactory`, it now exists in the database. We can see that it's there by looking at the list of cheeses in the Django admin:

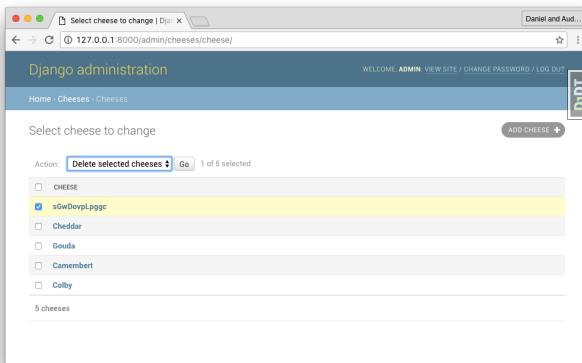


Figure 38: Django admin list of cheeses, including bogus cheese

Go to the admin and delete the one bogus cheese from the list:

1. Select the checkbox to the left of the cheese.
2. Choose Action > **Delete selected cheeses**.
3. Click **Go**.

Now our database should only contain good cheeses.

41.5 Commit the Changes



All cheese is made in cheese factories

It is our understanding that in English, the term “Cheese Factory” means any place where cheese is made for non-personal use. In other words, cheese factories can mean giant mechanized facilities and small family-run kitchens. It’s important to remember that the word “factory” for making things predates the industrial revolution and has its roots in the Latin *facio*, and before that the *fakiō* of Proto-Italic and before that *θakjō* from Proto-Indo-European.

Let's save our changes:

```
git status  
git add -A  
git commit -m "Add a cheese factory"  
git push origin master
```

42 Why Use Factories

Why are factories so great for unit testing?

42.1 Factories and Unit Testing

Well, suppose that for test purposes we wanted to create a new cheese record containing data for the following fields:

- `name`
- `slug`
- `description`
- `firmness`

For unit-testing purposes, we don't actually care if `name` is the name of a real cheese. Likewise, `description` can just be any random paragraph, and `firmness` can be any of the firmness choices that we defined in our `cheese` model.

Here are 2 ways of creating test cheese:

1. Call `cheese.objects.create()` and pass in values for `name`, `description`, and `firmness` (`slug` will be autogenerated).
2. Call `CheeseFactory()` without passing in values (since they will all be autogenerated).

Let's compare both approaches.

42.2 Approach I: Calling `Create()`

First we'll create a Cheese via `cheese.objects.create()`:

```
>>> cheese = Cheese.objects.create(  
... name="Stracchino",  
... description="Semi-sweet cheese eaten with starches.",  
... firmness=Cheese.Firmness.SOFT,)  
>>> cheese  
<Cheese: Stracchino>
```

Here, we had to type out values for `name`, `description`, and `firmness` and pass them into `create()`. It took a few lines of code to define a cheese.

42.3 Approach 2: Calling `CheeseFactory()`

Now let's generate a cheese from the `CheeseFactory`. One... two... three...

```
>>> from everycheese.cheeses.tests import factories  
>>> cheese = factories.CheeseFactory()  
>>> cheese  
<Cheese: dEiwndWjefsq>
```

Here, we didn't have to explicitly pass values for `name`, `description`, OR `firmness` into `CheeseFactory()`. That means:

1. We didn't have to look up the `cheese` model definition to figure out how the fields were defined. That's a lot of mental overhead that we saved. We also reduced the risk of making mistakes during cheese creation.
2. Creating a cheese took 1 line of code instead of 4.

Factories like `CheeseFactory` are much more efficient and less error-prone to create test data with.

`CheeseFactory` doesn't create real cheeses unless you explicitly give it real values. For unit testing purposes, that's fine. For example, if you're testing that `cheese.__str__()` returns `cheese.name`, the test checks for equality only. It doesn't care whether `cheese.name` is a real cheese.

42.4 Other Uses

Besides unit testing, factories can be used in place of fixtures. For example, suppose you need to generate 5000 cheeses for local development purposes, to see how the site's UI behaves when it's well-populated with cheese data. You can call `CheeseFactory()` in a loop.

42.5 Learn More About Factory Boy

Our preference for testing factories, `Factory Boy` is a mature library used around the world in tens of thousands of projects. Here's more information about it:

- <https://factoryboy.readthedocs.io/>
- https://github.com/FactoryBoy/factory_boy

43 Using Factories in Tests

Let's apply what we covered in the previous chapter.

43.1 Passing in Explicit Field Values

When using a factory, we can optionally specify the value of one of the fields. For example, try typing this in `shell_plus`:

```
from everycheese.cheeses.tests.factories import CheeseFactory
>>> cheese = CheeseFactory(name="Sample Cheese From Factory")
>>> cheese
<Sample Cheese from Factory>
```

Here, we specify that we want the name of our new cheese to be *Sample Cheese From Factory*. But it could be any name we want. We pass the `name` argument into `CheeseFactory()` the same way we would normally pass it into `create()`.

43.2 Bulk Generation

Let's use this feature to generate a whole bunch of sample cheeses at once. Here we only create 10 cheeses, but

it could be hundreds or more if we needed that many.

In **shell_plus**, type this:

```
>>> for x in range(10):
...     CheeseFactory(name=f"Sample Cheese {x}")
```

The names of the cheeses we created will be *Sample Cheese 1*, *Sample Cheese 2*, *Sample Cheese 3*, etc.

43.3 Reviewing Our Cheeses

In **shell_plus**, type this:

```
>>> Cheese.objects.all()
```

If we do this query on cheeses we get some interesting results:

- See all those new test cheeses? Ten more of them, all created by our `CheeseFactory`.
- Since we specified the `name` in the `CheeseFactory()` call, the cheeses have our desired names instead of being random characters.

43.4 Cheese Cleanup

Okay, it's time to clean up our database so we can move on to the next lesson. Let's delete the fake sample cheeses we just made.

```
>>> Cheese.objects.filter(name__startswith='Sample').delete()
```

If we prefer, we could delete the fake cheeses via the Django admin, as we did before. But if we had hundreds of cheeses, we'd see why deleting via the shell can be nice. It would save us from clicking on pages and pages of cheese.

In this case, we created the cheeses manually by calling `CheeseFactory()` from the shell. However, if we had called `CheeseFactory()` from a unit test, we wouldn't need to do this cleanup, as tests clean up the database for themselves. We'll now see how factories are used in unit tests.

43.5 Replacing the Direct Cheese Model Call in Our Existing Test

One of the tenets of good programming is that we're not supposed to repeat ourselves. If we have to describe building cheeses again and again, that's going to be a lot of repetition. And that's where factories come into play. They build test data for us.

Let's change up our test data set of cheese.

In `cheeses/tests/test_models.py` add this import right after where we imported the `Cheese` model (`from ..models import Cheese`):

```
from .factories import CheeseFactory
```

Then change this part where we created a cheese via `create()`:

```
cheese = Cheese.objects.create(  
    name="Stracchino",  
    description="Semi-sweet cheese eaten with starches.",  
    firmness=Cheese.Firmness.SOFT,  
)
```

to this, where we now create a cheese via `CheeseFactory()`:

```
cheese = CheeseFactory(name="Stracchino")
```

Putting it all together, the test should look like this:

```
def test__str__():  
    cheese = CheeseFactory(name="Stracchino")  
    assert cheese.__str__() == "Stracchino"  
    assert str(cheese) == "Stracchino"
```

Run the test and make sure it passes:

```
coverage run -m pytest
```

We should see something similar to this output:

```
coverage run -m pytest
Test session starts
(platform: darwin, Python 3.8.2, pytest 5.3.5, pytest-sugar)
django: settings: config.settings.test (from option)
rootdir: /Users/drg/projects/everycheese, ini file: pytest.ini
plugins: sugar-0.9.2, django-3.8.0
collecting ...
everycheese/cheeses/tests/test_models.py ✓
everycheese/users/tests/test_forms.py ✓
everycheese/users/tests/test_models.py ✓
everycheese/users/tests/test_urls.py √√√
everycheese/users/tests/test_views.py √√√

Results (0.66s):
    10 passed
```

43.6 Removing the Name Entirely

There's no need to specify the cheese's name. Replace the test with this:

```
def test__str__():
    cheese = CheeseFactory()
    assert cheese.__str__() == cheese.name
    assert str(cheese) == cheese.name
```

This is even better than what we had before because:

1. The code is even shorter.
2. Now we are validating that the value of `cheese.__str__()` and `str(cheese)` is equal to `cheese.name` rather than a hardcoded string.
3. The value of `cheese.name` is randomly generated every time we run the test. While in this example it doesn't matter too much, if we were testing a more complex method, we might discover occasional test failures due to improper Unicode or special character handling.

Run the test again and make sure it passes:

```
coverage run -m pytest
```

43.7 Commit Our Work

```
git add -A  
git commit -m "Use CheeseFactory in a test"  
git push origin master
```

43.8 Summary

A running theme of our course is finding safe ways to accelerate how fast we do things. Generating test data is one of the more time consuming parts of writing tests, and factories simply accelerate the process.

If we are uncomfortable with values the `CheeseFactory` stores in particular fields, we can change that by specifying field values during instantiation.

44 How Cheese Objects Feel About Tests

It's easy to think of our tests' cheese objects as merely testing data. But remember, even cheese objects have feelings. When a cheese object passes its tests, it beams proudly with joy and satisfaction.

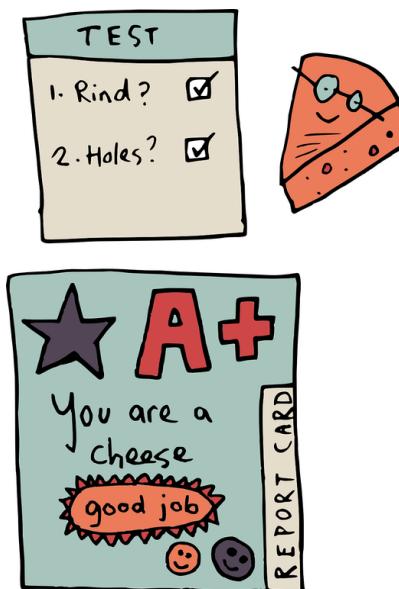


Figure 39: A good cheese that passed its test.

45 Finding and Adding a Third-Party Countries App

We're going to make changes to our Cheese model.

It would be interesting to track where a cheese is from. There are different approaches we could take: tracking the country, region, city, etc. What makes this even more complicated is the fact that a cheese's country can be ambiguous: is it the country where the cheese originally was created, where the cheese is now produced, or where the cheese is available for purchase?

It's always good to limit scope and aim for simplicity, so we'll do the following:

- Add some sort of country field to the `cheese` model.
- Limit it to the country where the cheese originally came from. That is, the cheese's country of origin.

45.1 Country of Origin

To be more explicit, let's call this field `country_of_origin`.

Let's now talk about the implementation.

To represent a cheese's `country_of_origin` value, here are some possible ways we could implement the field:

45.1.1 Option 1: Use a Plain `CharField`

The easiest way to implement the `country_of_origin` field would be to define it as a `CharField` like this:

```
country_of_origin = models.CharField(  
    "Country of Origin", max_length=255  
)
```

There's a big problem with this implementation, though. With a plain `CharField`, there are no constraints on what text we enter. We might accidentally enter `us` for one cheese and `usa` for another cheese. `us` is different from `usa` or `United States` in the database, making it challenging if we later decide to filter cheeses by country.

45.1.2 Option 2: Use a `CharField` With Choices

Another option would be to specify choices for the country like this:

```
country_of_origin = models.CharField(  
    "Country of Origin",
```

```
    max_length=20,  
    choices=COUNTRY_CHOICES,  
    default=COUNTRY_UNSPECIFIED,  
)
```

Here, `COUNTRY_CHOICES` would be a list of all possible countries.

While this resolves the problem of having different strings representing the same country, there's still a big problem. It takes substantial work to create and maintain a list of countries. Every now and then, new countries form, countries that were split unite, and more.

We could reduce our burden by sticking to the ISO 3166-1¹⁶ standard list of country names, but then we would still have to keep the EveryCheese country list in sync with the ISO 3166-1 country list. We'd have to check it frequently and keep updating our code.

45.2 Benefiting From Open Source

Option 2 seems the best, but it's a substantial amount of work.

Providing a list of country choices seems like such a common problem. For common problems like this, it's good to search for an open source solution before attempting to implement anything. Open source to the rescue!

¹⁶https://en.wikipedia.org/wiki/ISO_3166-1

45.3 Check Django Packages

First step, go to <https://djangopackages.org> and enter *countries* in the search bar. One of the things that comes up is a *grid*, which allows us to easily compare packages. Let's go there:

The screenshot shows a web browser window for 'Django Packages : Countries'. The top navigation bar includes links for Home, FAQ, audrey, Admin, and Log out. Below the navigation is a grid of package descriptions. The columns are labeled: PACKAGE, DJANGO-CITIES, DJANGO-COUNTRIES, DJANGO-CITIES-LIGHT, DJANGO-AIRPORTS, DJANGO-GEO, NANO APPS, DJANGO-COUNTRIES, and DJANGO WORLD REGIONS. The first row, 'Description', contains a single entry for 'Django Countries'. The 'DJANGO-COUNTRIES' column for this entry is highlighted in green.

PACKAGE	DJANGO-CITIES	DJANGO-COUNTRIES	DJANGO-CITIES-LIGHT	DJANGO-AIRPORTS	DJANGO-GEO	NANO APPS	DJANGO-COUNTRIES	DJANGO WORLD REGIONS
Description	Countries and cities of the world for Django projects		A simple app providing three models: Country, Region and City model. Also provided: a command to insert or update data ...	It's like django-cities, but django-airports	django application to manage administrative geographical info (country, city, region ...)	Does less! Loosely coupled mini-apps for django.		Simple Django app to group countries in world's regions.
Category	App	App	App	App	App	Framework	App	App

Figure 40: Django Packages' countries grid

Scanning the list, we can see Django Countries, a popular package that might fit what we need. Let's do some analysis off the data Django Packages provides:

- Says it's **Production/Stable**.
- Has a good number of stars. This is a tricky value because popular does not always equate to quality.

- It's maintained and has a history of such.
- SmileyChris, aka Chris Beaven, a Django core team member, maintains it.

Yeah! Let's dive in!

45.4 Review Django Countries on GitHub

The instructions look nice and clear:

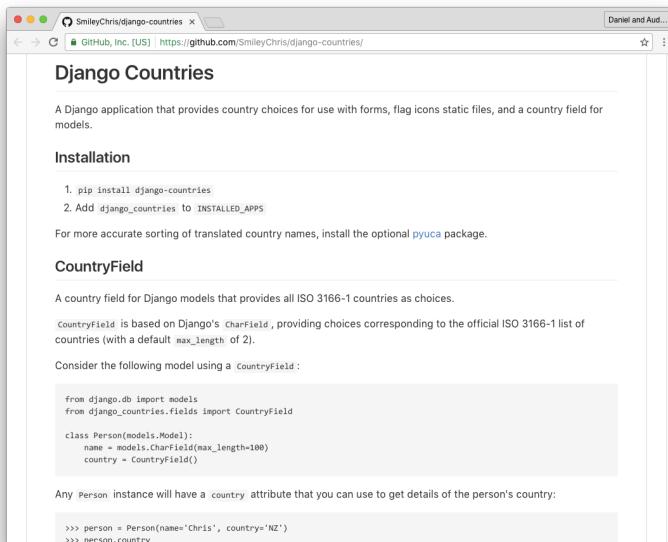


Figure 41: Django Countries README on GitHub

45.5 Look for the Release Number on PyPI

Let's grab the version number so we can pin the dependency.

We prefix the name of the project with <https://pypi.org/project/> and check it in our browser thus: <https://pypi.org/project/django-countries>.

45.6 Install It

As usual, add the package and its version number to `requirements/base.txt`, which probably won't match the "6.1.2" we list here:

```
django-countries==6.1.2
```

The reason we choose to add to the common `base.txt` is that we want the package installed in all environments: local, test, and production.

Then pip install the local requirements into our local virtualenv:

```
pip install -r requirements/local.txt
```

This does the same as if we had typed:

```
pip install django-countries==6.1.2
```

45.6.1 Don't Do This: Copy/Paste Installation

This is bad. Really bad. Don't do it!

1. git clone git@github.com:SmileyChris/django-countries.git
2. cd django-countries
3. mv django_countries > everycheese/everycheese

Don't do the above! Why this is bad:

- We've just lost ALL the advantages of open source.
This is what's called a HARD fork.
- Getting code updates is MUCH harder
- Quick testing/reversion is MUCH harder
- Changes we make to our own Django Countries can/will cause breaking differences between implementations
- Makes handling upgrades or installing security patches MUCH harder

In summary, by cloning instead of extending we are hurting ourselves and whoever follows us in maintaining our projects. Be a good coder and stick to using package installers like `pip` instead of hard forks.

And now back to our normally scheduled EveryCheese lesson...

45.7 Implement in cheeses/models.py

Confident that we pip installed our django-countries requirement, we follow the instructions in <https://github.com/SmileyChris/django-countries>.

45.7.1 Installation

Add django_countries to settings/base.py's THIRD_PARTY_APPS setting:

```
THIRD_PARTY_APPS = [
    "crispy_forms",
    "allauth",
    "allauth.account",
    "allauth.socialaccount",
    "django_countries", # CountryField
]
```

45.7.2 Adding the Country Field to the Cheese Model

In cheeses/models.py, import CountryField and define country_of_origin like this:

```
from django_countries.fields import CountryField
```

```
class Cheese(TimeStampedModel):  
    # ... already existing fields here  
    country_of_origin = CountryField(  
        "Country of Origin", blank=True  
    )
```

Since `CountryField` is an extension of `CharField`, we treat it similarly. We want `country_of_origin` to be optional, since it might be unknown for some cheeses, and so we set `blank=True`. But we don't set `null=True` because Django's convention is to store empty values as the empty string, and to retrieve NULL/empty values as the empty string. (See [Two Scoops of Django](#) for a more detailed explanation.)

45.8 Migrations!

The next stage is creating a database migration. From the command line:

```
python manage.py makemigrations cheeses
```

Apply the migration:

```
python manage.py migrate
```


46 Display Country Data for Cheeses

46.1 Add Cheese Countries in the Admin

Startup `runserver` if it's not already running.

Then go the Django admin. For each cheese, choose a **Country of Origin** and save our changes.

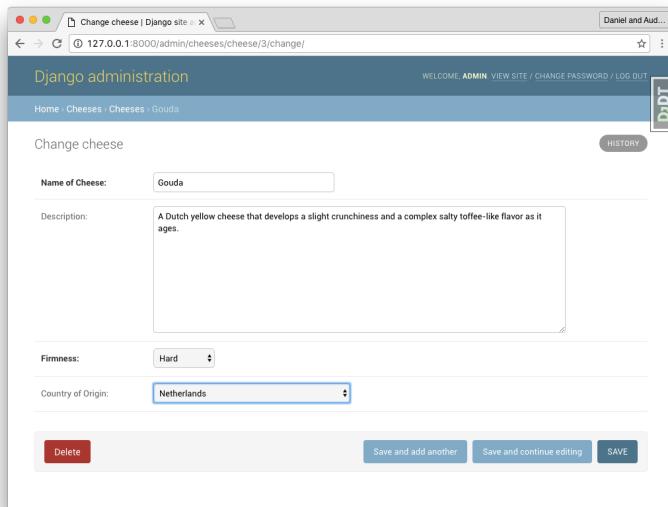


Figure 42: Cheese country data in the admin

The countries of origin are:

Cheese	Country of Origin
Cheddar	United Kingdom
Gouda	Netherlands
Camembert	France
Colby	United States of America
Stracchino	Italy

Now each cheese should have country data.

46.2 Display Country in Cheese Detail

At this point, the only place on the site where we can see Country of Origin data is in the Django admin. Let's change that by showing country data on the cheese detail pages.

Put this code into `templates/cheeses/cheese_detail.html` after the part where the cheese description is shown:

```
{% if cheese.country_of_origin %}  
    <p>Country of Origin: {{ cheese.country_of_origin.name }}  
      
    </p>  
{% endif %}
```

How this works:

1. As we did before with other fields, we check if the cheese has any value set for `country_of_origin` before even trying to display it.
2. Then we display the name of the country with `cheese.country_of_origin.name`. How did we know this would work? We found this usage example in the Django Countries documentation¹⁷:

```
>>> person.country.name  
'New Zealand'
```

Let's start up the server and see what we get. Voilà, we have cheese Country of Origin displayed.

46.3 Run the Tests

Let's run the tests in case anything broke:

```
coverage run -m pytest
```

The tests still pass:

```
Test session starts  
(platform: darwin, Python 3.8.1, pytest 5.3.4, pytest-sugar 0.9.2)  
django: settings: config.settings.test (from option)
```

¹⁷<https://github.com/SmileyChris/django-countries#countryfield>

```
rootdir: /Users/drg/projects/everycheese, ini file: pytest.ini
plugins: sugar-0.9.2, django-3.8.0
everycheese/cheeses/tests/test_models.py ✓
everycheese/users/tests/test_forms.py ✓
everycheese/users/tests/test_models.py ✓
everycheese/users/tests/test_urls.py ✓✓✓
everycheese/users/tests/test_views.py ✓✓✓

Results (0.55s):
    10 passed
```

46.4 Update CheeseFactory

Open the cheeses app's `factories.py`. Add this field to `CheeseFactory`:

```
country_of_origin = factory.Faker('country_code')
```

The Faker documentation for `faker.providers.address`¹⁸ says that we can generate a fake country code like this:

```
>>> from faker import Faker
>>> fake = Faker({'en-US': 1})
>>> fake.country_code()
# 'NI'
```

¹⁸<https://faker.readthedocs.io/en/master/providers/faker.providers.address.html>

Translating that to actual usage on our project, we pass the function name string of `country_code` into `factory.Faker()` to get a randomly-selected country code. In future A Wedge of Django extensions, we'll go over even more testing tricks that can be used with factories and the faker library.

46.5 Verify That CheeseFactory Works

Make sure that `CheeseFactory` still can generate cheeses, and that it sets `country_of_origin` to random countries correctly. Open `shell_plus` and type:

```
from everycheese.cheeses.tests.factories import CheeseFactory
for i in range(5):
    cheese = CheeseFactory()
```

Then go to the cheese list in the Django admin. Click on each random new cheese and make sure that it has a random country of origin.

When we're done, delete the new cheeses via the Django admin.

46.6 Commit Our Work

```
git status  
git add -A  
git commit -m "Implement cheese country of origin"  
git push origin master
```

47 Implement Cheese Creation by Users

Right now only admins can add, edit, and delete cheese data.

But wouldn't it be nice if users who aren't admins could enter cheese data too? That would mean EveryCheese could gather content much faster.

That's the whole point of a user-generated content site: the admins don't have to write all the content themselves. User-generated content means trusting users to enter in real cheeses.

47.1 Add a Cheese Creation View

Open `cheeses/views.py` in our text editor. Remember, this is where we'll put all our cheese-related views.

We'll be using a `CreateView` to display a cheese creation form on the EveryCheese website. First, import the `CreateView` class at the top of the `views.py` module:

```
from django.views.generic import CreateView
```

Define a `CheeseCreateView` by subclassing `CreateView`:

```
class CheeseCreateView(CreateView):
    model = Cheese
```

Here we specify its model as `cheese` because it will be used for creating `cheese` objects.

47.2 Add the Corresponding URL Pattern

We want the URL of the cheese form page to be `/cheeses/add/`.

Therefore, add this URL pattern to `cheeses/urls.py` in the `urlpatterns` list:

```
path(
    route='add/',
    view=views.CheeseCreateView.as_view(),
    name='add'
),
```

Where should we add it? Well, let's first try adding it to the bottom, after `CheeseDetailView`. Put it there, start up `runserver` if needed, and then open our browser `http://127.0.1:8000/cheeses/add/`. We will see this:

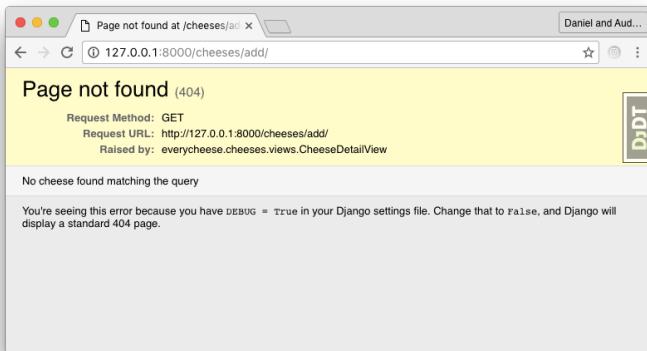


Figure 43: No cheese found!

Look closely at that page and note that it says:

Raised by: `everycheese.cheeses.views.CheeseDetailView`

Oddly enough, it looks like the URL is matching `CheeseDetailView` when it should be matching `CheeseCreateView`. Django thinks we're looking for the Cheese Detail page for a cheese with the slug of **add**.

Django tries each URL pattern in order until it finds a match. When it finds the first match, it calls the corresponding view.

Here the first match is `CheeseDetailView` because **add/** matches '`<slug:slug>/`'. Django never gets to the `CheeseCreateView` which comes next.

Look closely at `cheeses/urls.py` until we understand how this all works.

Then move the URL pattern for `CheeseCreateView` up before `CheeseDetailView`. Our code should now be:

```
from django.urls import path

from . import views

app_name = "cheeses"
urlpatterns = [
    path(
        route='',
        view=views.CheeseListView.as_view(),
        name='list'
    ),
    path(
        route='add/',
        view=views.CheeseCreateView.as_view(),
        name='add'
    ),
    path(
        route='<slug:slug>/',
        view=views.CheeseDetailView.as_view(),
        name='detail'
    ),
]
```

Return to our browser and refresh the page. We will see a new error:

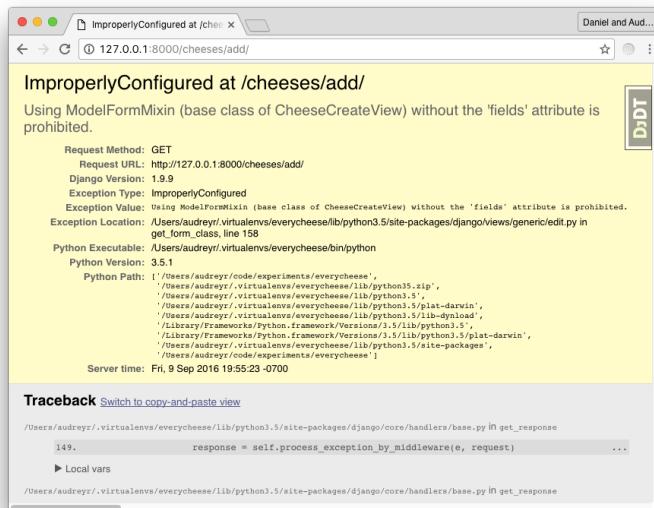


Figure 44: Need Fields Attribute

That's because when we use a `CreateView`, we need to specify a `fields` attribute.

47.3 Specify the Desired Form Fields

The `fields` attribute tells the `CreateView` which of the model's fields should appear on the form page.

Open `cheeses/models.py`. Take a look at our `Cheese` model. There are a number of fields. What do we want in the

cheese creation form? The values a user should fill out for a cheese are:

- `name`
- `description`
- `firmness`
- `country_of_origin`

We don't want a user to enter a `slug`, because that should be automatically generated from the value of `name`. We also don't want a user to enter anything for the `created` and `modified` fields inherited from `TimeStampedModel` because those values are automatically taken care of for us.

Open `cheeses/views.py` again. Add this attribute to `CheeseCreateView`:

```
fields = [  
    'name',  
    'description',  
    'firmness',  
    'country_of_origin',  
]
```

Refresh the page in our browser. We should get a new error:

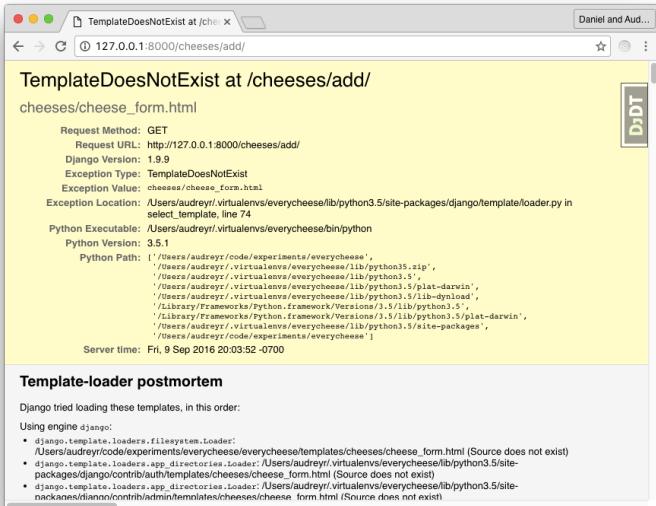


Figure 45: Cheese Form Template Does Not Exist

47.4 Define the Cheese Form Template

In `templates/cheeses/`, create a new file called `cheese_form.html`. Put this code inside:

```
{% extends "base.html" %}

{% block title %}Add Cheese{% endblock title %}

{% block content %}
<h1>Add Cheese</h1>
```

```
<form method="post" action="{% url 'cheeses:add' %}">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit" class="btn btn-primary">Save</button>
</form>
{% endblock content %}
```

Here's what's going on:

- Each item in the `fields` list is turned into the appropriate HTML label/input, as part of the form.
- Each label/input pair is displayed in a `<p>` element.
- When the user clicks Save, the form sends a POST request to the same URL with the form data.



For those reading the PDF or print version of this book

This empty space is intentional. Please go on to the next page.

47.5 Submit the Form

Save the file and go back to our browser. Refresh the page. We should now see the Add Cheese form as shown in the image below:

The screenshot shows a web browser window with the title 'Add Cheese'. The URL in the address bar is '127.0.0.1:8000/cheeses/add/'. The page has a dark header with the logo 'EveryCheese' and navigation links for 'Home', 'About', 'Cheeses', 'My Profile', and 'Sign out'. On the right side of the header, there is a small 'DDT' icon. The main content area is titled 'Add Cheese'. It contains several input fields: 'Name of Cheese:' with an empty text input, 'Description:' with an empty text area, 'Firmness:' with a dropdown menu set to 'Unspecified', 'Country of Origin:' with a dropdown menu showing a list of countries, and a 'Save' button at the bottom.

Figure 46: Add Cheese Form

Try entering a cheese:

- Name: Havarti
- Description: A mild, buttery cheese that comes in loaves or blocks.

- Firmness: Semi-Soft
- Country of Origin: Denmark

Click Save. Be forewarned as we're going to generate an error page:

ImproperlyConfigured at /cheeses/add/

No URL to redirect to. Either provide a url or define a get_absolute_url method on the Model.

Request Method: POST
 Request URL: http://127.0.0.1:8000/cheeses/add/
 Django Version: 1.9.9
 Exception Type: ImproperlyConfigured
 Exception Value: No URL to redirect to. Either provide a url or define a get_absolute_url method on the Model.
 Exception Location: /Users/audreyt/virtualenvs/everycheese/lib/python3.5/site-packages/django/views/generic/edit.py in get_success_url, line 193
 Python Executable: /Users/audreyt/virtualenvs/everycheese/bin/python
 Python Version: 3.5.2
 Python Path: ['/Users/audreyt/code/experiments/everycheese', '/Users/audreyt/virtualenvs/everycheese/lib/python35.zip', '/Users/audreyt/virtualenvs/everycheese/lib/python3.5', '/Users/audreyt/virtualenvs/everycheese/lib/python3.5/plat-darwin', '/Users/audreyt/virtualenvs/everycheese/lib/python3.5/lib-dynload', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/plat-darwin', '/Users/audreyt/virtualenvs/everycheese/lib/python3.5/site-packages', '/Users/audreyt/code/experiments/everycheese']
 Server time: Mon, 12 Sep 2016 12:16:43 -0700

Traceback [Switch to copy-and-paste view](#)

```
/Users/audreyt/virtualenvs/everycheese/lib/python3.5/site-packages/django/views/generic/edit.py in get_success_url
190.         url = self.object.get_absolute_url() ...
▶ Local vars
```

Figure 47: No URL to Redirect To

47.6 Implement `Get_absolute_url()`

Whenever we define a `createView`, we also need to define a `get_absolute_url()` method on the corresponding model if one doesn't exist yet.

Open `cheeses/models.py` in our text editor and add this import to the top of the module:

```
from django.urls import reverse
```

Then add the `get_absolute_url()` method to the `Cheese` class:

```
def get_absolute_url(self):
    """Return absolute URL to the Cheese Detail page."""
    return reverse(
        'cheeses:detail', kwargs={"slug": self.slug}
    )
```

Now our `CheeseCreateView` will use this method. When a user fills out the Add Cheese form and submits the data, Django will create the cheese and redirect them to the Cheese Detail page for the cheese that they just created.

47.7 Resubmit the Form

Let's go back to our browser and resubmit the form:

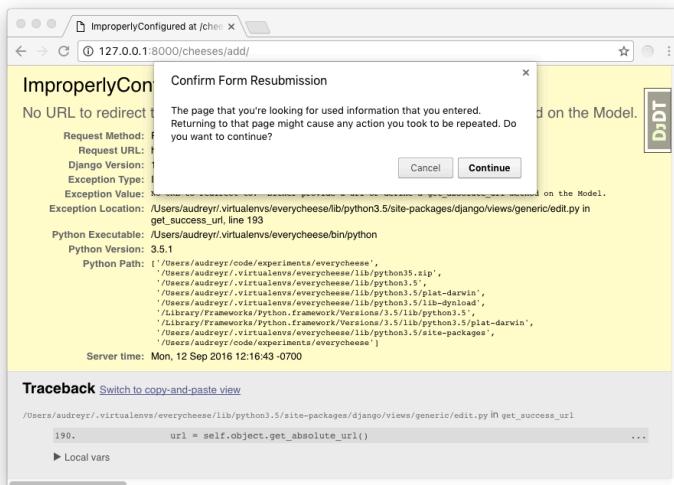


Figure 48: Resubmit Cheese

The submission will go through. Then we'll be redirected to the Cheese Detail page for the cheese that we just created:

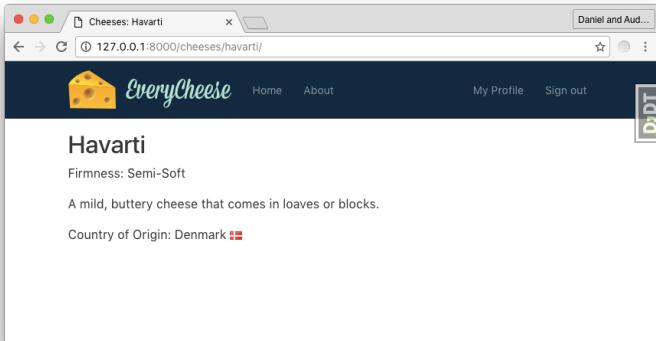


Figure 49: Cheese Detail for Havarti

Congrats! Our `cheesecreateView` now lets users create new cheeses.

47.7.1 Troubleshooting Non-ASCII Names

This can occur when we enter a cheese with non-ascii characters like Κεφαλοτύρι and that generates this error: `django slug is defined before trying to ensure uniqueness.` This may occur if we created the **EveryCheese** project before February 20th, 2020.

If this happens, the solution is to install the `unidecode` library thus:

```
pip install unidecode==1.1.1
```

Don't forget to add the library to our project's `requirements/base.txt` file. Here's how it looks in the `django-crash-starter`: https://github.com/feldroy/django-crash-starter/blob/master/%7B%7Bcookiecutter.project_slug%7D%7D/requirements/base.txt

47.8 Link to the Add Cheese Form

The form works, but users still need a way to get to it. Open `templates/cheeses/cheese_list.html` in our text editor and add this button link to the bottom of the page:

```
<hr/>
<p>Don't see a cheese listed here?</p>
<p>
  <a class="btn btn-primary"
     href="{% url 'cheeses:add' %}" role="button">
    Add Cheese
  </a>
</p>
```

Make sure we put the button inside of the `content` block, before `{% endblock content %}`.

Now when we click on **Cheeses** in the navbar, we should see the button at the bottom of the page (the image showing this may be on the next page):

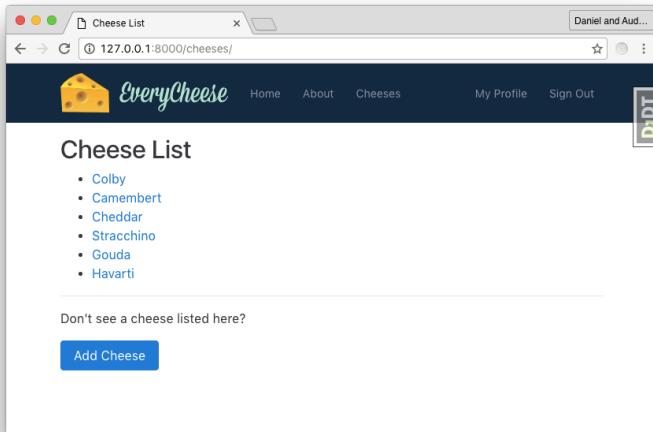


Figure 50: Cheese List With Add Cheese Button

Click on the button to make sure that it brings up the Add Cheese form.

47.9 Commit the Changes

We've done a lot this lesson. It's time to commit:

```
git status  
git add -A  
git commit -m "First pass at working Add Cheese form"  
git push origin master
```

48 Use Django Crispy Forms for Prettier Display

If we're observant, we'll have noticed that the Add Cheese form controls aren't lined up properly.

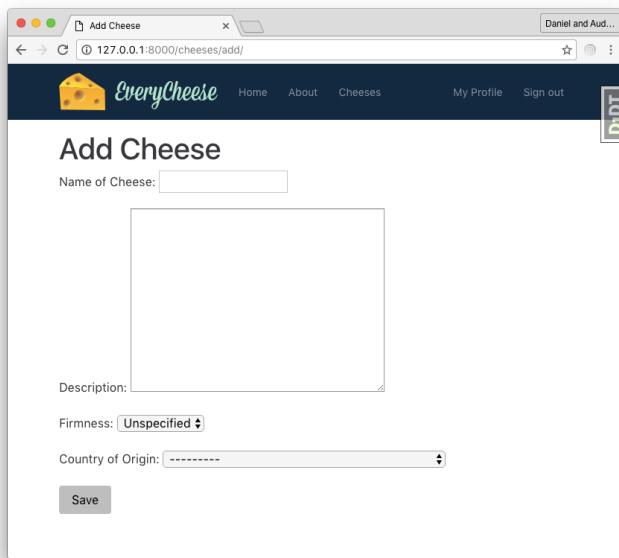


Figure 51: Cheese form with unstyled form fields

48.1 Isn't `as_p()` Enough?

Even if we use Django's built-in form display methods `as_p()`, `as_table()`, or `as_ul()`, there's no way to format the form so that it uses Bootstrap's form styles properly.

48.2 A Quick History Lesson

Django started as a project in 2003, back when the internet was a lot simpler. Then it was enough for a form to work. It didn't have to be pixel-perfect. This was the time before HTML form UI toolkits like Bootstrap.

Flash forward to January of 2009. Daniel Roy Greenfeld, employed at NASA¹⁹ at the time, was given a task. He had to convert around 70 Django forms from using `as_table()` to something that would match the Section 508 Web Content Accessibility Guidelines²⁰. Rather than convert them manually, which would be both tedious and error-prone, Daniel elected to write a custom template filter to do the work.

And thus was `django-uni-form`²¹ born!

The project rapidly grew in features²². Over time support for formsets, custom layouts, and other features was added. The project increased in popularity²³.

¹⁹<https://www.nasa.gov/>

²⁰<https://www.section508.gov/create/web-content>

²¹<https://pydanny.blogspot.com/2009/01/django-uni-form-lives.html>

²²<https://pydanny.blogspot.com/2009/02/should-django-uni-form-handle.html>

²³<https://github.com/pydanny-archive/django-uni-form>

The code having grown organically had a few problems. Daniel and Miguel Araujo, the other major contributor at the time, began a rewrite. One issue was that the library only supported an obscure JavaScript/CSS library, and Bootstrap was becoming much more popular. Eventually Daniel decided to hand the project off to Miguel, who in accepting it took Audrey Roy Greenfeld's suggestion for a new form library name: django-crispy-forms²⁴.

And that is the superhero origin story for django-crispy-forms.

48.3 Crispy Forms and Cookiecutter Django

The django-crispy-forms package should have already been installed earlier when we installed all of django-crash-starter's local requirements. Therefore, we don't need to install it again.

django-crash-starter uses Crispy Forms to control the way forms are rendered in the *users* app. We'll do something similar in the *cheeses* app.

48.4 Our Non-Crispy Form

This is the form that we're about to change:

²⁴<https://github.com/maraujop/django-crispy-forms>

The screenshot shows a web browser window with the title bar "Add Cheese". The address bar displays the URL "127.0.0.1:8000/cheeses/add/". The page header includes the "EveryCheese" logo, navigation links for "Home", "About", "Cheeses", "My Profile", and "Sign out", and a "Django" watermark in the top right corner. The main content area is titled "Add Cheese". It contains the following fields:

- Name of Cheese:
- Description:
- Firmness:
- Country of Origin:

A "Save" button is located at the bottom left of the form.

Figure 52: Add Cheese Form

Right now form elements are rendered using `<p>` tags. That's why they look so poorly aligned.

Once we modify it to use Crispy Forms, the fields will align nicely. That's because by default, Crispy Forms renders form elements with Bootstrap-compatible HTML.

48.5 Use the Crispy Template Tag

In our text editor, let's open the file `templates/cheeses/cheese_form.html`.

Load the Crispy Forms template tags at the top of the template. Add this line right below `{% extends "base.html" %}`:

```
{% load crispy_forms_tags %}
```

Now the `crispy` tag is available for us to use in the template. Find this line:

```
{{ form.as_p }}
```

And replace it with this line:

```
{{ form|crispy }}
```

Now we are using the `crispy` filter.

Putting it all together, our code should now be:

```
{% extends "base.html" %}  
{% load crispy_forms_tags %}  
  
{% block title %}Add Cheese{% endblock title %}  
  
{% block content %}  
<h1>Add Cheese</h1>  
<form method="post" action="{% url 'cheeses:add' %}">  
    {% csrf_token %}  
    {{ form|crispy }}  
    <button type="submit" class="btn btn-primary">Save</button>  
</form>  
{% endblock content %}
```

48.6 Reload the Add Cheese Form

Let's go back to our browser and reload the **Add Cheese** form at <http://127.0.0.1:8000/cheeses/add/>. It should now look like this page below:

The screenshot shows a web browser window titled "Add Cheese" with the URL "127.0.0.1:8000/cheeses/add/". The page is part of the "EveryCheese" website, as indicated by the logo and header. The header also includes links for "Home", "About", "Cheeses", "My Profile", and "Sign out". A "DRAFT" watermark is visible in the top right corner of the form area. The main content is a "Add Cheese" form with the following fields:

- Name of Cheese*: An input field.
- Description: A large text area for entering a description.
- Firmness*: A dropdown menu currently set to "Unspecified".
- Country of Origin: A dropdown menu currently set to "-----".

A "Save" button is located at the bottom left of the form.

Figure 53: Add Cheese Form, Rendered With Crispy Forms

It looks much better.

Now the form is rendered with Bootstrap controls instead of `<p>` tags.

That's just the start of the power of Crispy Forms. There's so much more that we can do to make our forms render exactly how we want. Crispy Forms gives us fine-grained form rendering control.

48.7 Commit the Changes

Let's save our changes:

```
git status  
git add -A  
git commit -m "Use django-crispy-forms to better Add Cheese display"  
git push origin master
```

49 Understand View Mixins and Login- RequiredMixin

In the previous chapter we made it possible for any user on EveryCheese to create new cheeses. But what happens when people use the **Add Cheese** form to enter fake data or spam the site?

49.1 User-Generated Content and Accountability

Inevitably, some users will try to mess with the site and enter in garbage data:

- People who are just trying out the site and don't know what to enter.
- Spammers adding links for SEO.

To control the creation of bogus cheeses, some level of accountability would help. We'll start by requiring users to be logged in when they create cheeses.

49.2 Try Accessing the Add Cheese Form Anonymously

Open a new private browser window:

- If we're using Chrome: **New Incognito Window**
- If Firefox: **New Private Window**
- If Edge: **New InPrivate Window**

In the incognito/private browser window, let's go to our Add Cheese form: <http://127.0.0.1:8000/cheeses/add/>

A screenshot of a web browser window titled "private window (incognito)". The URL in the address bar is "127.0.0.1:8000/cheeses/add/". The page itself is titled "Add Cheese" and features a logo for "EveryCheese". The form has fields for "Name of Cheese*", "Description", "Firmness*", and "Country of Origin". A "Save" button is at the bottom. A red arrow points to the "private window (incognito)" text in the title bar.

Figure 54: Add Cheese Form, As Anonymous User

Here are two important observations:

- In that window, we are not logged in.
- Yet we can access the Add Cheese form. That's bad.

49.3 Require Login

We're going to make the *Add Cheese* form accessible only to users who are logged into the site.

Open `cheeses/views.py` in our text editor.

At the top, add this import:

```
from django.contrib.auth.mixins import LoginRequiredMixin
```

Then modify `CheeseCreateView` to use `LoginRequiredMixin` like this:

```
class CheeseCreateView(LoginRequiredMixin, CreateView):
```

```
    ...
```

Now `CheeseCreateView` cannot be accessed unless we are logged into EveryCheese.

49.3.1 Try Accessing the Add Cheese Form Incognito

In our private (incognito) browser window, let's reload the Add Cheese form (<http://127.0.0.1:8000/cheeses/add/>).

We should now see this:

What happened?

- Because we were an anonymous user in that window, we were redirected to the login page.
- Note the URL in the address bar: <http://127.0.0.1:8000/accounts/login/?next=/cheeses/add/>

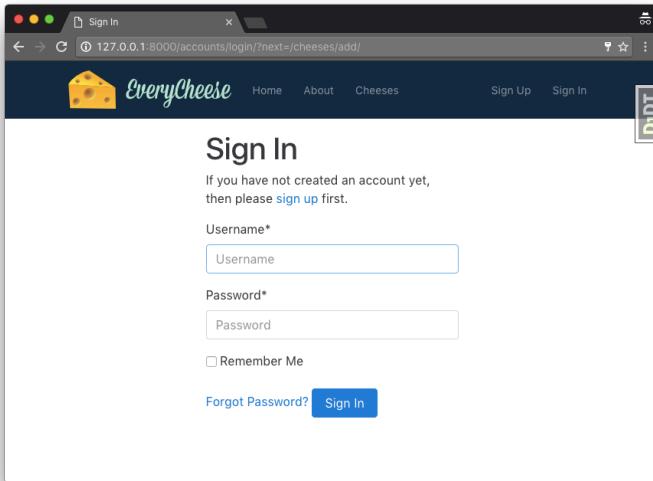


Figure 55: Incognito Redirect to Sign In Page

- **/accounts/login/** is the standard login page.
- The URL contains the parameter **next=/cheeses/add/**. That is the page that we will be redirected to after we log in.

49.4 Try Accessing It as Cheesehead

Switch to the browser window where we are logged in as **cheesehead**.

Refresh the Add Cheese form page.

We should be able to access the form. That's because we

are already logged in, so we can access that page normally.

49.5 View Mixins

Here we saw a common example of a **view mixin**. View mixins are used with class-based views. We use them by putting them to the left of the subclassed view name.

View mixins are frequently used to restrict access to a view.

Mixins seem very fancy, but they can be quite simple. A simple view mixin looks like this:

```
class MyMixin:  
    def some_method(self):  
        return self.something
```

If we were to use `MyMixin` in our `CheeseCreateView`, it would give `CheeseCreateView` a method called `some_method`.

49.6 Commit the Changes

Save our changes:

```
git status  
git add -A  
git commit -m "Constrain Add Cheese form to logged-in users only"  
git push origin master
```

50 Add a Creator Field and Update Our Cheese Records

To increase accountability, we'll start tracking the creator of each cheese.

50.1 Add a Creator Field

First, add this import to the top of the `cheeses/models.py` module. Remember to put it in the Django imports section, keeping that section organized alphabetically:

```
from django.conf import settings
```

Add the `creator` field to the `Cheese` model. Put this under the other field definitions but before the method declarations of `__str__` and `get_absolute_url`:

```
creator = models.ForeignKey(  
    settings.AUTH_USER_MODEL,  
    null=True,
```

```
    on_delete=models.SET_NULL  
)
```

This field is a `Foreign Key` to a `User` object. In this case since we are defining a field and just need a string representation of the `User` model name, it's fine to import that string from `settings` rather than calling `get_user_model()`.

50.1.1 Explaining Foreign Keys

In relational databases, a foreign key is used to link two tables together. The first table has a field dedicated to tracking the key, which can only select amongst the rows of the second table. Selections cannot typically be made for rows not in the second table. This protects the integrity of data.

To use our own example, a cheese creator must be a user within the EveryCheese system. Our example also includes an `on_delete` action of `models.SET_NULL`, which means if the `user` selected as `creator` is deleted, the cheese is not deleted. Instead, the cheese record has a `null` value assigned to the creator. Other options used to protect the integrity of data might be to delete the cheese or assign it to a default “EveryCheese Master” record. It really depends on the business requirements for a project and how the data needs to be protected.

In summary, the idea that `foreign keys` are a critical part of the `relational in relational databases` is accurate and worth investigating:

- Django Docs on Foreign Keys: <http://feld.to/django-db-models-ForeignKey>
- https://en.wikipedia.org/wiki/Foreign_key

50.2 Make and Apply the Migration

At the command-line run `migrate`, which will apply the migration that we just created and edited:

```
python manage.py makemigrations cheeses
python manage.py migrate
```

We don't want the new `creator` to ever be empty. It's a matter of data (and cheese) integrity. Once we run that, go into the Django shell with `python manage.py shell_plus`.

```
>>> cheesehead = User.objects.get(username='cheesehead')
>>> for cheese in Cheese.objects.all():
...     cheese.creator = cheesehead
...     cheese.save()
```

Now each `cheese` record will have a `creator` field with the value set to the `cheesehead user` object. We can confirm

this by checking cheese creators in either the Django admin or inside `shell_plus`. For example:

```
for cheese in Cheese.objects.all():
...     print(cheese, cheese.creator)
```

50.3 Commit the Changes

Let's save our changes:

```
git status
git add -A
git commit -m "Add creator to Cheese model and custom migration"
git push origin master
```

5I Track and Display the Cheese Creator

When a user submits a new cheese via the Add Cheese form, the `cheeseCreateView` will tie their `user` record to the new cheese that they created. That way if someone enters in bad data, we can go into the admin and warn or delete their account.

5I.I Set the Cheese Creator After Form Validation

The `creator` field shouldn't be shown on the Add Cheese form page. That would allow users to set it to whatever they want, which would be bad.

Instead, we want to automatically set the value of `creator`. A good place to do this is after the form data has been validated. We can override a `createView`'s `form_valid()` method to insert form data after validation occurs.

Open `cheeses/views.py` in our text editor. Add this `form_valid()` method to `cheeseCreateView`:

```
def form_valid(self, form):
    form.instance.creator = self.request.user
```

```
    return super().form_valid(form)
```

Here, the value of `creator` is set to the `user` object associated with the current request.

We can set `creator` programmatically *after* form validation because there's no need to validate the value of `creator`: it comes from our code, not from user input.

The finished view class should now look like this:

```
class CheeseCreateView(LoginRequiredMixin, CreateView):
    model = Cheese
    fields = ['name', 'description', 'firmness',
              'country_of_origin']

    def form_valid(self, form):
        form.instance.creator = self.request.user
        return super().form_valid(form)
```

51.2 Display the Creator on the Cheese Detail Pages

On the Cheese Detail page, we're going to display a bunch of info about the creator of the cheese:

- Their username
- Their name, if one exists in the database

- Their bio, if one exists in the database

Let's open `templates/cheeses/cheese_detail.html` in our text editor. At the bottom of the template, right before `{% endblock content %}`, add a section containing info about who submitted the cheese:

```
<hr/>

<h3>Submitted by

<a href="{% url 'users:detail' cheese.creator.username %}">
    {{ cheese.creator.username }}
</a>

</h3>

{% if cheese.creator.name %}
    <p>{{ cheese.creator.name }}</p>
{% endif %}
```

Powered by `creator` foreign key to the `UserModel`, what we're providing:

- The creator's username
- A link to the creator's bio
- If they've filled it out, the creator's name

Inside of Django when we ask for information about the cheese creator it queries the user table for us. This is really useful, taking away the minutia of writing small queries.

51.3 Try It Out

Start `runserver` if it's not already running.

Go to any Cheese Detail page. Look at the bottom for info about who created the cheese.

EveryCheese Home About Cheeses My Profile Sign Out

Cheddar

Firmness: Hard

A relatively hard, pale yellow to off-white, and sometimes sharp-tasting cheese.

Country of Origin: United Kingdom 

Submitted by [cheesehead](#)

Cheesy Head

Figure 56: Cheese Detail With Creator

While logged in as `cheesehead`, try adding a cheese:

- Name: Kesong Putî
- Description: A fresh, salty white cheese made from Philippine water buffalo milk.
- Firmness: Soft
- Country of Origin: Philippines

It's good to do this in order to ensure that our overridden `CheeseCreateView.form_valid()` is saving the current user as the cheese's creator.

Save the cheese. We should be redirected to the Cheese Detail page, which should have our *cheesehead* user as the creator.

51.4 Commit the Changes

Save our changes:

```
git status  
git add -A  
git commit -m "Track and display the creator of each cheese"  
git push origin master
```

52 Update the Cheese Factory

Alright, we've made a number of code changes. We haven't run the tests in awhile, and there are probably tests that we need to update or write.

52.1 Run the Tests

Let's see what happens when we run the test suite:

```
coverage run -m pytest
```

The tests still pass. That doesn't mean that we're off the hook. We still have some changes to make.

52.2 Modify Cheese Factory

In VS Code, open `cheeses/tests/factories.py`. Study what we have so far in our `CheeseFactory` class. Notice that it doesn't set the value of `creator` anywhere.

To set `creator`, we'll need a `User` object. The easiest way to get one for our `CheeseFactory` is to create a `User` with `UserFactory`.

First, import `UserFactory` near the top of the `cheeses/tests/factories.py` module:

```
from everycheese.users.tests.factories import UserFactory
```

Then add this line to our `cheeseFactory` definition right underneath the `country_of_origin` definition:

```
creator = factory.SubFactory(UserFactory)
```

Now when we use `cheeseFactory` to create a `Cheese` instance, the cheese's creator will be a user created by `UserFactory`.

52.3 Run the Tests, Again

```
coverage run -m pytest
```

The tests should still pass.

52.4 Try CheeseFactory in the Shell

Okay, the tests passed. But how do we know that `CheeseFactory` is creating cheeses with creators properly? Well, we can see for ourselves in the shell.

Start up Shell Plus:

```
python manage.py shell_plus
```

Instantiate a `cheese` object using `CheeseFactory`, and see what the value of `cheese.creator` is set to:

```
In [1]: from everycheese.cheeses.tests.factories import CheeseFactory  
  
In [2]: cheese = CheeseFactory()  
  
In [3]: cheese.creator  
Out[3]: <User: millsdustin>
```

As we can see, a `User` object was created with a username of `millsdustin`. Of course, the name of the user in this book might be different from what we get in our shell.

52.5 Delete the Random Cheese

Clean up by deleting the random cheese that we created:

```
In [4]: cheese.delete()  
Out[4]: (1, {'cheeses.Cheese': 1})
```

52.6 Delete the Random User

Clean up the random user as well:

```
In [5]: u = User.objects.last()
```

```
In [6]: u
```

```
Out[6]: <User: user-0>
```

```
In [7]: u.delete()
```

```
Out[7]:
```

```
(1,
 {'admin.LogEntry': 0,
 'users.User_groups': 0,
 'users.User_user_permissions': 0,
 'users.User': 1})
```

We're done experimenting and cleaning up. Exit the shell with Ctrl+D.

52.7 Commit the Changes

Save our changes:

```
git status
git add -A
git commit -m "Modify factory to create cheese creator user"
git push origin master
```

53 Update the Cheese Model Tests

Open `cheeses/tests/test_models.py` in our text editor. We should have one test in there:

```
def test__str__():
    cheese = CheeseFactory()
    assert cheese.__str__() == cheese.name
    assert str(cheese) == cheese.name
```

Now look at `cheeses/models.py` to determine if any new model tests need to be written:

- `__str__()` has a test already.
- `get_absolute_url()` still needs a test.

53.1 Test `Get_absolute_url()`

Now we have a cheese ready to use in our next test. When we test `get_absolute_url()`, we want to call it and check that the result matches the expected absolute URL.

Suppose we have a cheese named *My Happy Cheese*. Its absolute URL should be `/cheeses/my-happy-cheese/`, using `cheese.slug` in the URL.

Add this test to the bottom of the `cheeses/tests/test_models` module:

```
def test_get_absolute_url():
    cheese = CheeseFactory()
    url = cheese.get_absolute_url()
    assert url == f'/cheeses/{cheese.slug}'
```

Run the cheese model tests:

```
coverage run -m pytest
```

All the tests should pass and look something like this:

```
(venv) everycheese a&lt;&gt; coverage run -m pytest
Test session starts (platform: darwin, Python 3.8.2,
pytest 5.3.5, pytest-sugar 0.9.3)
django: settings: config.settings.test (from option)
rootdir: /Users/drg/projects/everycheese, ini file: pytest.ini
plugins: django-3.9.0, Faker-4.1.0, sugar-0.9.3,
django-test-plus-1.4.0
collecting ...
everycheese/cheeses/tests/test_models.py √√ 18% [■]
everycheese/users/tests/test_forms.py √ 27% [■]
everycheese/users/tests/test_models.py √ 36% [■]
everycheese/users/tests/test_urls.py √√√ 64% [■■■]
```

```
everycheese/users/tests/test_views.py ✓✓✓✓ 100% [██████████]
```

```
Results (0.52s):
```

```
    11 passed
```

53.2 Commit the Changes

Let's save our changes:

```
git status  
git add -A  
git commit -m "Add test for get_absolute_url() method"  
git push origin master
```

54 Test All the Cheese Views

54.1 What to Test?

In a previous chapter, we added a `creator` field to the `Cheese` model which was a simple `ForeignKey`. This is just standard Django ORM usage. The Django core codebase already has tests for `ForeignKey`, so we don't need to test that.

There's no other new model code. If we had added any model methods, we would have to write tests for them.

We did define a new view: `CheeseCreateView` with a new URL pattern. It would be nice to test both of those, the view and the URL pattern. It's also a good idea to have basic tests on our other views and URL patterns for completeness.

54.2 Start With Imports

In our text editor, create `cheeses/tests/test_views.py`. This is where we'll be putting all our cheese view tests.

Let's import all the things we'll need for the tests. Put this at the top of the module:

```
import pytest
from pytest_django.asserts import assertContains

from django.urls import reverse

from .factories import CheeseFactory
from ..models import Cheese
from ..views import (
    CheeseListView,
    CheeseDetailView
)
pytestmark = pytest.mark.django_db
```

The first import is the `pytest` library. Then we import common utilities from Django. Finally, we import Cheese models and view from our EveryCheese project.

In this mix, we import the `CheeseFactory` so we can more easily write tests for cheese.

Finally, we set `pytestmark`, which connects our tests to the database.

54.3 The First Cheese View Tests

We like to start our tests by stubbing out a simple test function for each view. These stubs are a great place to start when you're not sure what view tests need to be

written. All we'll do at first is check that the views are returning valid 200 HTTP status codes²⁵ for `GET` requests.

54.3.1 Testing the Cheese List

For the list view test, pass in the name string for the list view. That's `cheeses:list`. Add this test to `test_views.py` under the imports. The test has been commented heavily so we know what's going on:

```
def test_good_cheese_list_view_expanded(rf):
    # Determine the URL
    url = reverse("cheeses:list")
    # rf is pytest shortcut to django.test.RequestFactory
    # We generate a request as if from a user accessing
    #   the cheese list view
    request = rf.get(url)
    # Call as_view() to make a callable object
    # callable_obj is analogous to a function-based view
    callable_obj = CheeseListView.as_view()
    # Pass in the request into the callable_obj to get an
    #   HTTP response served up by Django
    response = callable_obj(request)
    # Test that the HTTP response has 'Cheese List' in the
    #   HTML and has a 200 response code
    assertContains(response, 'Cheese List')
```

²⁵<https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/200>

Try it out!

```
coverage run -m pytest
```

Results should look similar to:

```
Test session starts (platform: darwin, Python 3.8.2, pytest 5.3.5)
django: settings: config.settings.test (from option)
rootdir: /Users/drg/projects/everycheese, ini file: pytest.ini
plugins: django-3.9.0, Faker-4.1.0, sugar-0.9.3
collecting ...
everycheese/cheeses/tests/test_models.py √√    17% [■]
everycheese/cheeses/tests/test_views.py √       25% [■■]
everycheese/users/tests/test_forms.py √        33% [■■■]
everycheese/users/tests/test_models.py √       42% [■■■■]
everycheese/users/tests/test_urls.py √√√     67% [■■■■■■]
everycheese/users/tests/test_views.py √√√√   100% [■■■■■■■■■■]
```

Results (0.64s):

12 passed



assertContains also checks HTTP Status

A more accurate name for the `assertContains` test case would be `assertContainsAndCheckStatus`. If we read the docs for `assertContains` we see it also confirms

a 200 HTTP status code. For more details, reference the documentation on `assertContains` in <https://docs.djangoproject.com/en/3.1/topics/testing/tools/>.

Typically this test is written in much shorter format, but we expanded it as far as we could in order to understand exactly what's going on. Here's how this test is typically written:

```
def test_good_cheese_list_view(rf):
    # Get the request
    request = rf.get(reverse("cheeses:list"))

    # Use the request to get the response
    response = CheeseListView.as_view()(request)

    # Test that the response is valid
    assertContains(response, 'Cheese List')
```

Add this test example to the previous one and run it again (yes, we're running two functions with similar tests):

```
coverage run -m pytest
```

Tests are now 13 and should all pass.

54.3.2 A Basic Cheese Detail Test

Similarly, we can test the `CheeseDetailView` by using `CheeseFactory` to generate some test data. Let's add this to our `test_views.py` module. Since individual cheeses have a URL based on their slug, our use of the `reverse()` function is expanded to use the cheese slug as an argument. We also pass in the cheese slug as part of the callable object. Here's what our 14th test looks like:

```
def test_good_cheese_detail_view(rf):
    # Order some cheese from the CheeseFactory
    cheese = CheeseFactory()
    # Make a request for our new cheese
    url = reverse("cheeses:detail",
                  kwargs={'slug': cheese.slug})
    request = rf.get(url)

    # Use the request to get the response
    callable_obj = CheeseDetailView.as_view()
    response = callable_obj(request, slug=cheese.slug)
    # Test that the response is valid
    assertContains(response, cheese.name)
```



Cheese doesn't have slugs

It goes without saying that Cheese doesn't have slugs. And when eating cheese, we advise against

mixing it with slugs. There's a near-infinite selection of other things to mix with cheese. Indeed we suggest you start with more palatable options to partner with cheese.

Okay, enough about slugs, let's run the tests!

```
coverage run -m pytest
```

54.3.3 A Basic Cheese Create Test

The create view doesn't need a keyword argument. But it's a bit trickier because you can't access it unless you're logged in, due to the `LoginRequiredMixin` you used.

- You need to first set up a user so that you can log in as that user.
- In the test, the `user` fixture provides that user.

In `test_views.py`, underneath the line that says “import `pytest`”, add this new line:

```
from .factories import UserFactory
```

54.4 Write Our First Fixture

Rather than write `user = UserFactory()` in each test function, we're going to create a test fixture. It's quite easy to do, just put this underneath the imports inside of `test_views.py`:

```
@pytest.fixture
def user():
    return UserFactory()
```

54.5 Write the Cheese Create Test View

It's time to write our view! Add the following to the bottom of the test module:

```
def test_good_cheese_create_view(client, user):
    # Make the client authenticate
    client.force_login(user)
    # Specify the URL of the view
    url = reverse("cheeses:add")
    # Use the client to make the request
    response = client.get(url)
    # Test that the response is valid
    assert response.status_code == 200
```

This puts us at 15 tests! Run the tests yet again:

```
coverage run -m pytest
```

At this point, we've stubbed out a test for each cheese view. That's a good start. These tests are better than nothing. But we can do better.

54.6 Really Test the Cheese List View

What does a list view do? It lists its associated objects. A good thing to test is whether the list view response contains a couple of object's names.

For that, we'll need to add a few kinds of cheese. Then look inside the Cheese List response and check that the names of the cheeses can be found.

```
def test_cheese_list_contains_2_cheeses(rf):
    # Let's create a couple cheeses
    cheese1 = CheeseFactory()
    cheese2 = CheeseFactory()

    # Create a request and then a response
    #   for a list of cheeses
    request = rf.get(reverse('cheeses:list'))
    response = CheeseListView.as_view()(request)

    # Assert that the response contains both cheese names
    #   in the template.
```

```
assertContains(response, cheese1.name)
assertContains(response, cheese2.name)
```

As always, run the tests (which should number at 16 now):

```
coverage run -m pytest
```

54.7 Test the Cheese Detail View

A detail view shows detailed data for 1 object, in this case cheese. Let's write a test that checks the response contents:

```
def test_detail_contains_cheese_data(rf):
    cheese = CheeseFactory()
    # Make a request for our new cheese
    url = reverse("cheeses:detail",
                  kwargs={'slug': cheese.slug})
    request = rf.get(url)
    # Use the request to get the response
    callable_obj = CheeseDetailView.as_view()
    response = callable_obj(request, slug=cheese.slug)
    # Let's test our Cheesy details!
    assertContains(response, cheese.name)
    assertContains(response, cheese.get_firmness_display())
    assertContains(response, cheese.country_of_origin.name)
```

Run the tests again, which should now be up at 17 now:

```
coverage run -m pytest
```

54.8 Test the Cheese Create View

A create view should create an object correctly. Upon POST, `CheeseCreateView` should redirect to the detail page for the created cheese. The creator of that cheese should be the user who was logged in and submitted the cheese.

We can test for this as follows:

```
def test_cheese_create_form_valid(client, user):
    # Authenticate the user
    client.force_login(user)
    # Submit the cheese add form
    form_data = {
        "name": "Paski Sir",
        "description": "A salty hard cheese",
        "firmness": Cheese.Firmness.HARD,
    }
    url = reverse("cheeses:add")
    response = client.post(url, form_data)

    # Test the results for redirect
    assert response.status_code == 302
```

```
# Get the cheese based on the name
cheese = Cheese.objects.get(name="Paski Sir")

# Test that the cheese matches our form
assert cheese.description == "A salty hard cheese"
assert cheese.firmness == Cheese.Firmness.HARD
assert cheese.creator == user
```

Run the tests, all 18 of them!

```
coverage run -m pytest
```

54.9 Commit the Changes

Let's save our changes:

```
git status
git add -A
git commit -m "Write tests for all cheese views"
git push origin master
```

54.10 Conclusion

In this chapter we wrote a bunch of tests, some to increase test coverage, and others to ensure that code already covered had more accurate tests. In other words, we added depth to our tests.

Keep in mind that 100% test coverage is only as meaningful as the tests that cover the code. It is possible to write meaningless tests that increase test coverage of code but don't account for checking the validity of the results of that code.

For example, while we came into this chapter with the code for the cheese list and detail views list at 100% coverage, whether or not cheese data showed up in their templates was not yet checked. We addressed that in this chapter, adding depth to our tests.

Keep in mind we're not necessarily done. It may not be obvious yet, but there are ways we can improve the tests in this chapter. It's not uncommon for experienced coders to come back to tests they wrote years, months, weeks, or even days earlier and find ways they could make them better.

And that's okay. it's just the nature of being a software developer writing tests.

55 Test All the Cheese URL Patterns

We like to test each of our URL patterns. Regular expressions can be error-prone. It's nice to have tests in place just to verify that they behave as expected.

55.1 Add the Imported Cheese

Create the module `cheeses/tests/test_urls.py`. Import `reverse` and `resolve` from `django.urls`, which we'll need to test URL patterns backwards and forwards:

```
import pytest

from django.urls import reverse, resolve

from .factories import CheeseFactory

pytestmark = pytest.mark.django_db
```

55.2 Write Our Second Fixture

In this chapter we're going to need an instantiated cheese a few times. Just put this underneath the imports inside of `test_urls.py`:

```
@pytest.fixture  
def cheese():  
    return CheeseFactory()
```

We'll use this later in this chapter.

55.3 Test the Cheese List URL Pattern

When we test URL patterns, it's good to test them backwards and forwards:

- Reversing the view name should give us the absolute URL.
- Resolving the absolute URL should give us the view name.

Applying these concepts to testing the cheese list URL pattern, add the following to `test_urls.py`:

```
def test_list_reverse():  
    """cheeses:list should reverse to /cheeses/. """  
    assert reverse('cheeses:list') == '/cheeses/'  
  
def test_list_resolve():  
    """/cheeses/ should resolve to cheeses:list. """  
    assert resolve('/cheeses/').view_name == 'cheeses:list'
```

Run the tests again, which now number at twenty:

```
coverage run -m pytest
```

What we like about writing these tests is that it provides us more than just proof that we have good URLs, and protects us from supplying “404 page not found” errors to users. Specifically, it helps us understand how Django resolves URLs.

55.4 Test the Add Cheese URL Pattern

The tests for the add cheese URL pattern are very similar to those for the cheese list. That’s because for both URL patterns there are no slugs, primary keys, or other arguments to pass in.

Add these tests to `test_urls.py`:

```
def test_add_reverse():
    """cheeses:add should reverse to /cheeses/add/."""
    assert reverse('cheeses:add') == '/cheeses/add/'

def test_add_resolve():
    """/cheeses/add/ should resolve to cheeses:add."""
    assert resolve('/cheeses/add/').view_name == 'cheeses:add'
```

Run the cheese URL tests, which should number at 22 now:

```
coverage run -m pytest
```

55.5 Test the Cheese Detail URL Pattern

Here's where things get slightly fancier. The cheese detail URL pattern takes in a cheese slug. Remember how we created a cheese fixture earlier for testing purposes? Now's the time to use it. We just include it as the argument in the test functions.

Add these tests to `cheeses/tests/test_urls.py`:

```
def test_detail_reverse(cheese):
    """cheeses:detail reverses to /cheeses/cheese_slug/."""
    url = reverse('cheeses:detail',
                  kwargs={'slug': cheese.slug})
    assert url == f'/cheeses/{cheese.slug}/'

def test_detail_resolve(cheese):
    """/cheeses/cheese_slug/ resolves to cheeses:detail."""
    url = f'/cheeses/{cheese.slug}/'
    assert resolve(url).view_name == 'cheeses:detail'
```

Run the cheese URL tests, which will number 24 now:

```
coverage run -m pytest
```

Remember in the previous chapter when we used the `rf` and `user` arguments in our test functions? Those were fixtures just like our new `cheese` fixture.

In any case, we've tested all our cheese URL patterns. That wasn't so bad. URL patterns are relatively simple to test.

55.6 Commit the Changes

Let's save our changes:

```
git status  
git add -A  
git commit -m "Write tests for all cheese URL patterns"  
git push origin master
```

56 Adding a CheeseUpdateView and Recycling a Form

So far, EveryCheese has a way to create new cheeses. However, it lacks a way for end-users to update existing cheeses. If we're a regular user entering a new cheese and we make a mistake, we have no way to fix it.

In this chapter, we're going to add an update view to the cheeses app.

56.1 Add the CheeseUpdateView

The first step is to go to `cheeses/views.py` and import the generic `UpdateView` class. We can combine this import with the existing imports:

```
from django.views.generic import (
    ListView,
    DetailView,
    CreateView,
    UpdateView
)
```

At the bottom of the file, define `cheeseUpdateView`:

```
class CheeseUpdateView(UpdateView):
    model = Cheese
    fields = [
        'name',
        'description',
        'firmness',
        'country_of_origin'
    ]
    action = "Update"
```

Notice how the definition is very similar to that of `cheeseCreateView`.

56.2 Wire in the URL Pattern

Add the following URL pattern to `cheeses/urls.py`. We put it before the `detail` and after the `add` views:

```
path(
    route='<slug:slug>/update/',
    view=views.CheeseUpdateView.as_view(),
    name='update'
),
```

The route here is equivalent to that of the `cheeseDetailView` plus `update/` at the end.

56.3 Try It in the Browser

Start `runserver` if it's not already running.

Go back to the browser. From the cheese list, click on any cheese to go to its cheese detail page. Then add `update/` to the end of the URL and load the page.

That should bring up the Add Cheese form. The difference from before is that the form fields should already be populated with data:

The screenshot shows a web browser window titled "Add Cheese". The address bar displays the URL "127.0.0.1:8000/cheeses/havarti/update/". The page content is a form titled "Add Cheese". The "Name of Cheese*" field contains "Havarti". The "Description" field contains "A mild, buttery cheese that comes in loaves or blocks.". The "Firmness*" field contains "Semi-Soft". The "Country of Origin" field contains "Denmark". At the bottom of the form is a "Save" button.

Figure 57: Pre-populated Add Cheese Form

Try making a change to the description and country of origin. Save the changes.

We should see our changes reflected in the cheese detail page. Great! Right?

However, check the URL bar we'll notice something weird. The slug has a "-2" appended to it.

Even more noticeable is that if we go to <http://127.0.0.1:8000/cheeses/> we can see that there are now 2 kinds of cheese with that name. What just happened?!?

56.4 What Happened?!?

By default, both `cheesecreateView` and `cheeseUpdateView` share the same `cheese_form.html` template.

The problem is that our current `cheese_form.html` always sends the user to the `cheesecreateView`, even if we're doing it from the `cheeseUpdateView`.

Let's fix that, making the form template flexible enough to handle both `cheesecreateView` and `cheeseUpdateView` views. Open `templates/cheeses/cheese_form.html` in your text editor.

56.4.1 Using View Properties In Templates

Right now the `h1` element of `templates/cheeses/cheese_form.html` and the form button are hardcoded as:

```
<h1>Add Cheese</h1>
...
<button type="submit" class="btn btn-primary">Save</button>
```

This makes the form confusing because adding cheese isn't the same as updating cheese. Let's fix it by replacing the current code in `{% block content %}` with the following more flexible approach:

```
{% block content %}
{% with action=view.action|default:"Add" %}
<h1>{{ action }} Cheese</h1>

<form method="post" action="{% url 'cheeses:add' %}">
    {% csrf_token %}
    {{ form|crispy }}
    <button type="submit" class="btn btn-primary">
        {{ action }}
    </button>
</form>
{% endwith %}
{% endblock content %}
```

Let's break down what we just did:

1. We used the Django template tag `{% with %}` to set the `view.action` view method under the simpler name

of `action`, and default it to “Add”. This only lasts for the duration of the request, and can only be used within one code block.

2. We used the `action` variable in both the `h1` and `submit` `form`.

If we refresh the page, we’ll see that the `Add` is gone and replaced by `Update`. So where did the `view.action` variable come from? Let’s dig in and find out!

1. In the section above titled [Add the CheeseUpdateView](#), the `CheeseUpdateView` included a property called `action` and was set with the value of `"Update"`.
2. Inside a template rendered by a view, we can call attributes and properties of the view, referencing it under the easily remembered object name of `view`.
3. The `action` attribute of the view specifies what we are doing. Because we used clear language, it’s not just easier to debug, we can use it as a internationalizable value in our project.

56.4.2 Smart Titling

Right now the title element of `templates/cheeses/cheese_form.html` is hardcoded as:

```
{% block title %}Add Cheese{% endblock title %}
```

Let's make it a more flexible title by leveraging in the `view.action` property. In our template, change the title block to be:

```
{% block title %}  
  {{ view.action|default:"Add" }} Cheese  
{% endblock title %}
```

We just removed the word `Add` with `{{ view.action }}`, which calls the `action` property on the view used in a request. Refresh the page and if we check the browser tab the title of the page has changed.



Why didn't we wrap the whole page in `with`?

Unfortunately, Django's `{% with %}` template tag doesn't work across template blocks.

56.4.3 Form Posting to Itself

Right now the form method and action are defined as:

```
<form method="post" action="{% url 'cheeses:add' %}">
```

Using `action=".,"` as below is a commonly-used trick to reuse a form template for both create and update views. Replace the above with:

```
<form method="post" action=".,">
```

Now the form page posts to itself in both cases, eliminating the need to create 2 forms with hardcoded actions.

56.5 Link to the Update Form

It would be nice if every cheese detail page had an Update button, with the button going to the update form for the cheese.

Open `templates/cheeses/cheese_detail.html` in your text editor. Right below the `<h2>` tags in the `{% block content %}`, add a button link to the cheese detail form:

```
<p>
  <a class="btn btn-primary"
    href="{% url 'cheeses:update' cheese.slug %}"
    role="button">
    Update
  </a>
</p>
```

56.6 Try It in the Browser

Go to the Cheese Detail page for any cheese.

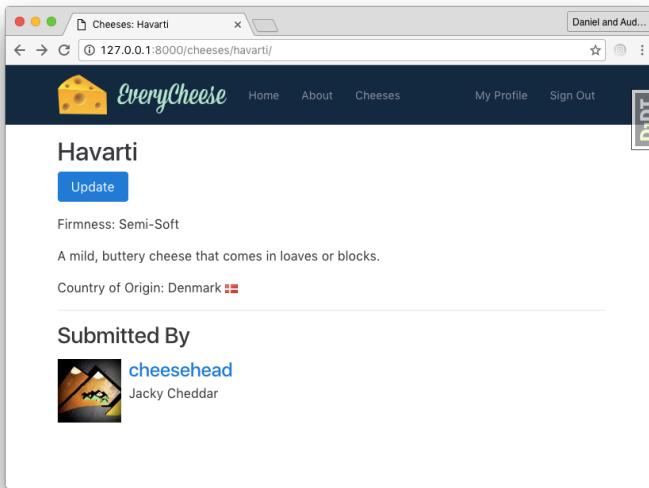


Figure 58: Cheese Detail With Update Button

Click Update to make changes to a cheese. Then see what happens. Compare this with creating a cheese.

56.7 Make Login Required

Earlier we imported `LoginRequiredMixin` so that we could use it in `cheeseCreateView`. Add it to `cheeseUpdateView` as well:

```
class CheeseUpdateView(LoginRequiredMixin, UpdateView):
```

Order matters here with multiple inheritance. Make sure it's put to the left of `UpdateView`.

This will prevent anonymous users from modifying existing cheeses.

56.8 Commit the Changes

Finally, we save our work:

```
git status  
git add -A  
git commit -m "Add cheese update view, modify template"  
git push origin master
```

57 Test the Cheese Forms and Update View

We've implemented a `CheeseUpdateView`, and we've made changes to the `cheese_form.html` template used by both `CheesecreateView` and `CheeseUpdateView`. Now's a good time to check on the tests and write more.

57.1 Refactoring our Cheese Fixture

Before we add more tests, let's do some refactoring. The cheese fixture that we created in the last chapter was so useful, let's add it to other Cheesy tests.

To do this, open up `cheeses/tests/factories.py` and under the other imports at the top bring in the `pytest` library. Also, copy in the cheese fixture:

```
import pytest

@pytest.fixture
def cheese():
    return CheeseFactory()
```

Now, let's open up `cheeses/tests/test_views.py` and bring our new cheese fixture. We'll do it by importing the cheese

fixture right after `cheeseFactory` import. That means we should have a line like this at the top of the module:

```
from .factories import CheeseFactory, cheese
```

Once that's in place, in `cheeses/tests/test_views.py` find the test function named `test_good_cheese_detail_view`. Change the top three lines from this:

```
def test_good_cheese_detail_view(rf):
    # Order some cheese from the CheeseFactory
    cheese = CheeseFactory()
```

to this, where a `cheese` argument has been added to the function and the call to `CheeseFactory` has been removed:

```
def test_good_cheese_detail_view(rf, cheese):
```

Now run all 24 tests again:

```
coverage run -m pytest
```

All tests should pass. Not only that, but we've reduced line count and added clarity to the tests. Yeah!

Now, go and add the `cheese` argument to the `test_detail_contains_cheese_data` function, removing the use of `cheeseFactory` in it to reduce line count and complexity:

The other tests in this module either need more than one cheese or create new cheeses on their own.

57.2 Test the Add Cheese Page Title

Since we added an if/else block to the page title in `cheese_form.html`, it's good to test that both branches work.

Add this 25th test to `cheeses/tests/test_views.py`:

```
def test_cheese_create_correct_title(client, user):
    """Page title for CheeseCreateView should be Add Cheese."""
    # Authenticate the user
    client.force_login(user)
    # Call the cheese add view
    response = client.get(reverse("cheeses:add"))
    # Confirm that 'Add Cheese' is in the rendered HTML
    assertContains(response, "Add Cheese")
```

This test checks that the web page corresponding to `CheeseCreateView` contains the string `Add Cheese`.

57.3 Test That CheeseUpdateView Is a Good View

So we can test it, add `CheeseUpdateView` to the imports:

```
from ..views import (
    CheeseCreateView,
    CheeseListView,
    CheeseDetailView,
    CheeseUpdateView
)
```

Then, following the same pattern established in previous chapters, we start testing `CheeseUpdateView`. We define a 26th test case in `test_views.py`, comment it extensively, and call it `test_good_cheese_update_view()`.

```
def test_good_cheese_update_view(client, user, cheese):
    # Authenticate the user
    client.force_login(user)
    # Get the URL
    url = reverse("cheeses:update",
                  kwargs={"slug": cheese.slug})
    # Fetch the GET request for our new cheese
    response = client.get(url)
    # Test that the response is valid
    assertContains(response, "Update Cheese")
```

Again, `test_good_cheese_update_view()` is just a start. It gives us some peace of mind, especially as EveryCheese grows in new and unexpected directions.

57.4 Test That Cheese Updates Correctly

Remember, the point of `CheeseUpdateView` is to give us a form page for modifying cheeses that already exist in the database.

A good thing to test is whether we can send a POST request to `CheeseUpdateView` with data modifying an existing cheese. Add this 27th test to `test_views.py` module:

```
def test_cheese_update(client, user, cheese):
    """POST request to CheeseUpdateView updates a cheese
    and redirects.

    """
    # Authenticate the user
    client.force_login(user)
    # Make a request for our new cheese
    form_data = {
        "name": cheese.name,
        "description": "Something new",
        "firmness": cheese.firmness,
    }
    url = reverse("cheeses:update",
                  kwargs={"slug": cheese.slug})
    response = client.post(url, form_data)

    # Check that the cheese has been changed
```

```
cheese.refresh_from_db()  
assert cheese.description == "Something new"
```

Notice how we call `refresh_from_db()` to update the `cheese` instance with the latest values from the database. This is needed because the database was modified, the old `cheese` object is stale until refreshed.

57.5 Run the Tests And Confirm 100% Test Coverage

```
coverage run -m pytest  
coverage html
```

Once both commands have been run, open `everycheese/htmfcov/index.html` with your browser to confirm that all modules have full test coverage.

57.6 Commit the Changes

It's time to save our changes:

```
git status  
git add -A  
git commit -m "Test the cheese forms and update view"  
git push origin master
```

58 EveryCheese is the Foundation!

Congratulations! We've reached the end of the tutorial part of this book, but this isn't the end for EveryCheese.

The EveryCheese project is built off solid Django principles, the same ones that we've espoused in our **Two Scoops of Django** book series for over seven years. Like the `helloworldjango` project, we recommend keeping EveryCheese as a reference project to see how to write tests, modify views, use choices fields, and other techniques used in production Django projects.

Want to learn more? Read on!

58.I Take the Live, Online Version of this book!

Periodically we teach the live, interactive, online version of this book. To find out more, please visit <https://www.feldroy.com/pages/courses>.

58.2 Level Up With Two Scoops of Django

If you feel you are past the point of tutorials and ready to build real projects, then you are ready for Two Scoops of Django.

In Two Scoops of Django we'll introduce you to various tips, tricks, patterns, code snippets, and techniques that we've picked up over the years. Two Scoops of Django is NOT for beginners just starting out. Rather, it is meant for people who are at least building their first real project.

<https://www.feldroy.com/products/two-scoops-of-django-3-x>

58.3 Other Django Books

You can find a list of high quality Django books, including more tutorials, at <https://wsvincent.com/best-django-books/>

58.4 Giving Us Feedback

If you have any feedback, the absolute best place to give it to us is on our official issue tracker at <https://github.com/feldroy/django-crash-course/issues>.

Part I

Aftermatter

59 Troubleshooting

This appendix is for helping readers resolve problems that may come up during A Wedge of Django.

59.1 Troubleshooting Conda Installations on Pre-Catalina OSX

If `conda list` fails, then enter these two commands, replacing `THEUSERNAME` with our Mac username:

```
source /Users/THEUSERNAME/miniconda3/bin/activate  
conda init
```

59.2 Troubleshooting Conda Installations on Catalina or higher OSX

```
source /Users/THEUSERNAME/miniconda3/bin/activate  
conda init zsh
```

Reference:

<https://docs.conda.io/projects/conda/en/latest/user-guide/install/macos.html>

If that still doesn't work, we recommend reading
<https://towardsdatascience.com/how-to-successfully-install-anaconda-on-a-mac-and-actually-get-it-to-work-53ce18025f97>

59.3 Troubleshooting PostgreSQL: Database EveryCheese Already Exists and/or Role myuser Already Exists

Some students may run into these warnings if they've run the commands of the [How to Create It](#) section more than one time.

If able to execute the commands of the [Run the Migrations](#) section without errors, these warning messages can be ignored, but if errors are obtained, here are some things to correct them.



Be very careful here!

The commands below erase data, and should only be executed on your local development machine and NEVER on a production server.

Option 1: Recreate the database

To recreate the database and leave it in the initial state, that is, create the database but not create the tables, we can use the `reset_db` command from `django-extensions`. Then it is possible to recreate the tables using the `migrate` command.

```
python manage.py reset_db  
python manage.py migrate
```

Option 2: Manually delete the database and database user, and after, recreate them

Deleting the `everycheese` database:

On Linux:

```
sudo -u postgres dropdb -U postgres --if-exists everycheese
```

On Mac:

```
dropdb --if-exists everycheese
```

Deleting the `myuser` database user:

On Linux:

```
sudo -u postgres dropuser -U postgres --if-exists myuser
```

On Mac:

```
dropuser --if-exists myuser
```

To recreate the database, the database user and the tables, run the commands of the [How to Create It](#) and [Run the Migrations](#).

59.4 Troubleshooting PostgreSQL: Role Does Not Exist

Some students may run into this if they're on GNU/Linux, and if they haven't given our local development user a role yet. To handle this error:

```
createdb: could not connect to database template1: FATAL:  
role "audreyr" does not exist'
```

Now create a role for the local user. Start the `psql` prompt as the `postgres` user, and create a role with the `CREATEDB` and `LOGIN` attributes. Change `audreyr` to our own username:

```
psql -U postgres
postgres=# CREATE ROLE audreyr WITH CREATEDB LOGIN;
postgres=# \q
```

The username is what prints out when we type `whoami` at the command line.

Try `createdb everycheese` again. With this role we shouldn't need to create a Postgres user or password specific to `everycheese`. This means there is no need to edit our settings to include a new database-specific username or password.

59.5 Troubleshooting PostgreSQL: Cannot Create Database

If we get an error about not having permission to create a database, we can try enabling trust authentication for all local users in PostgreSQL. Edit our `pg_hba.conf` (e.g. `/etc/postgresql/13/main/pg_hba.conf`) to have this line:

```
local all all trust
```

Make sure that there's no other line that cancels this line out, such as `local all all md5`. Then restart PostgreSQL:

```
sudo service postgresql restart
```

Try `createdb everycheese` again.

59.6 Troubleshooting Psycopg2

If we get the error `pg_config executable not found`, it most likely means that psycopg2 can't find where we installed Postgres on our system. Try adding our Postgres path to our PATH variable.

59.7 Troubleshooting GCC Errors on the Mac

If you get this error that looks similar to this on the Mac:

```
error: command 'gcc' failed with exit status 69
```

That could mean **XCode** development tools haven't been fully installed yet due to licensing. Apple insists Mac users agree to their license. To ensure we've met their requirements, type this on the command-line:

```
sudo xcodebuild -license
```

Go ahead and agree to the license, then run `pip install -r requirements/local.txt` again.

59.8 Troubleshooting the MS Visual Studio Build Tools Error

Windows users might get an error that says `Need MS Visual Studio Build Tools 2014" error.` If that's the case, installing VS Tools won't help. Instead, run `pip` again with this install option:

```
pip install rcssmin --install-option="--without-c-extensions"
```

59.9 Navbar Isn't Dark

For a while the django-crash-starter project had a light colored navbar which doesn't match images in the book. If we created our project before this was corrected, find the `everycheese/templates/base.html` file and open it. Replace these two values:

```
navbar-light bg-light
```

with

```
navbar-dark bg-dark
```

Refresh the page and we should see the corrected dark navbar.

60 Acknowledgements

This book was not written in a vacuum. We would like to express our thanks to everyone who had a part in putting it together.

Contributors are listed in the order that their bug reports, tips, and other submissions were addressed.

60.1 Tech Reviewers

It is often said that a good technical book is impossible without great technical editing. The book would not be in the shape it is without the three following people:

60.1.1 Christian Hetmann

A lifelong learner, Christian Hetmann is a degreed German engineer working for a Swedish furniture retailer in transport logistics. He's passionate about documenting and modeling of processes and concepts in logistics, automation and robotics. Christian enjoys watching movies and doing computer things with his two sons.

60.1.2 Enrique Matías Sánchez

Enrique is an engineer who loves strong-flavored cheeses, software development, system administration

and everything in between. When he is not at his job as a Python programmer at the University of Zaragoza in Spain, he can be found hiking the mountains nearby or traveling the world. He shares a black cat with his lovely girlfriend, another cheese-lover.

Enrique also provided our Linux installation instructions. Any innaccuracies or problems in that section are due to our inability to copy and paste his writing.

60.1.3 Fabio C. Barrionuevo da Luz

Fabio is a Brazilian engineer working for Feldroy. He likes to focus on producing the highest quality code that runs on all platforms. He enjoys open source and is a Cook-iecutter and Cookiecutter Django core developer. Fabio is a creator and maintainer of <http://pythonclub.com.br/>. When not programming, Fabio enjoys reading science fiction, playing the guitar, and spending time with his wife.

Fabio created the new book building system at Feldroy, for which we are very grateful.

60.2 Contributors to 3.x Beta

We could not have survived without our beta readers. Their attention to detail cannot be understated:

Carla Breden, Enrique Matías Sánchez, Christian Hetmann, Derrick Kearney, Joe Talarovich, Justin Kanusek,

Magnus Vaughan, Ivange Larry, Bob Earl, Og Maciel, Doug Gibbs, Mark Pors, Romack Natividad, Fábio C. Barrionuevo da Luz, Rohith PR, Toby Bettridge, Matt Winslow, Jonn Doe, Ben McNeill, Leigh Michael Forrest, Kevin J. Murray, Miguel Pachas, Marc Richter, Trevor Pierce, Carlos Johnson, Kim Roberts, Pedro Queiroga, Nigel Finch, Aaron C. Hall, Heitor Chang, Leon Lan, Chris Dorsman, Romain Sommerard, Joan Eliot, Tim Wilson, Ken Murray, Kevin Semilla, Phebe Polk, Mark Sevelj, Adonis Geronimo, Ivo Grondman, Jason Schvach, Marc Schwartz, Paolo Brocco, Patrick Endres, Jeff Robinson, Michael E. Marino, Elvio Severino, John Beeler, and Chris Sederqvist

60.3 Contributors to 3.x Alpha

Our alpha readers were incredibly patient and understanding. Working through the first version of any programming tutorial book can be quite a challenge! We are so very grateful to:

Nnenanya Obinna Kingsley, Oluwole Majiyagbe, Paul Muston, Adam Johnson, David Nugent, Diederik de Vries, Chuma Umenze, Perry Mertz, Derrick Kearney, Justin Kanusek, Modasser Billah, Christian Hetmann, Og Maciel, and Joe Talarovich

If your name is not on these lists but should be, please send us an email so we can make corrections!

60.4 Changelog

<https://github.com/feldroy/django-crash-course/blob/master/changelog.md>

60.5 Typesetting

We originally wrote **A Wedge of Django** using ReStructuredText and compiled using Sphinx. It was moved later to GitHub-Flavored Markdown and hosted on a custom corporate training platform. For this book the material was converted to R Markdown and an extended version of Bookdown²⁶ created and maintained by Daniel Roy Greenfeld, Audrey Roy Greenfeld, and Fabio C. Barrionuevo da Luz.

²⁶<https://bookdown.org/>