

Building an Enterprise Information System using Django

Udi h Bauman, Tikal Knowledge

Executive summary

Django is a popular web-framework based on the Python programming language. In recent years it is gaining popularity all over the world & companies using it report extreme gains, in all aspects. Recent polls in the states, place Django much ahead of any other technology used among new Startups. Large organizations are also endorsing Django, e.g., Google who based their AppEngine Cloud on Django, or NASA that selected Django as the primary application environment for their cloud.

I've been using Django for more than 3 years & think that the reason for its success is quite simple: it let's you write your application business logic in the language most optimal for writing business logic & it takes care of everything else for you (e.g., database access & management, user & security, data-entry UI, & many many more).

This results in extreme productivity. If you search the web, or talk to people who use it, you'll hear many stories of how a project that was supposed to take months, using Java or .Net, with a large team of developers, was eventually developed in a few days (or less) by a single or pair of developers using Django.

The surprising thing is that the end result proves much more robust, scalable & preferment than similar applications developed in more mainstream technologies. Apparently, when a technology is built right, it just works, & this is exactly what you'll hear from companies using Django.

Sounds too good to be true? The result of this tutorial is a production-ready full-featured Project Management application, developed in less than an hour. There aren't many technologies that can offer you that. A sequel tutorial will teach you how to seamlessly deploy this application to cloud computing vendors, such as Google AppEngine or Amazon EC2 & show you how easy it is to add more features, such as: localization, cache, RSS, & almost any web2.0 functionality.

Coal mining is evil

Programming is a bit like coal mining. You find yourself repeating lots of boring tasks over & over again & working many hours monotonously to cut some ugly dirty code.

Do you remember the time in which you actually enjoyed programming? This article is intended to help you learn a technology that can bring that time back. It's called Django & using it is fun. It's fun not just when starting a new project, but also through-out the development life-cycle including when the project is very large & complex.

It's fun by itself, & it's fun when you finish your work in a small fraction of the allocated time, & make your boss and customers grateful.

It's fun because you end up with a fraction of the codebase size you would get in other technologies, which makes the code much easier to maintain, extend & understand. Life is much more fun when the stuff you work with is small & simple to grasp.

The nice thing about it is that it's not just cool & fun for developers, but also compelling business-wise, so this article is also intended for managers too, that would benefit from the resulting product robustness, quality & fast time-to-market (as well as from having happy & productive employees...).

Show me the money

We're going to develop a simple project management system, that allows you to manage projects & their tasks, & present them in a useful visualization.

The functionality will include:

- Users management UI, including roles & permissions
- Projects & Tasks management UI, including search, filtering & recent actions list
- Focus diagram visualization of projects tasks, pointing out the tasks you should start with

What you'll need

Python

Python is a dynamic programming language, endorsed by many cutting-edge organizations such as Google, NASA & MIT due to its simple yet powerful features. Although having the connotations of a mere scripting language, it fully supports Object-Oriented programming, and also Functional programming & Aspect-Oriented programming, which lets you express anything you want, in a most elegant, readable & minimal way. Python is used to create very large & scalable Web applications, such as:

YouTube, FriendFeed, Eve Online, Mahalo, FreeBase & many more.

If you're running a *nix-based machine, such as Mac or Linux, you probably already have Python installed

Otherwise you can download a Python version for your machine from: <http://python.org/download>
The current version of Python is 3.x, but I recommend starting from the previous more supported version: 2.6.x



Just download the installer, run it & follow the installation instructions

To test that you have Python properly installed, open a command-line terminal, & run the command: `python`

If you get the Python interpreter prompt (`>>>`), it means that Python is installed.

Here're some screenshot2 demonstrating common programming idioms in Python. Try run them yourself – run the commands appearing after the `>>>` prompt.

```

C:\>python
Python 2.6.2 (r262:71605, Apr 14 2009, 22:40:02) [MSC v.1500 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello world'
Hello world
>>> 4 + 5 * 6
34
>>> s = 'Python'
>>> t = ' is cool'
>>> s + t
'Python is cool'
>>> s * 3
'PythonPythonPython'
>>> t.strip()
'is cool'
>>> s[0]
'P'
>>> s[3:6]
'hon'
>>> type(s)
<type 'str'>
>>> dir(t)
['_add', '_class', '_contains', '_delattr', '_doc', '_eq', '_format', '_ge', '_getattr', '_getitem', '_getnewargs', '_get
slice', '_gt', '_hash', '_init', '_le', '_len', '_lt', '_mod
d', '_mul', '_ne', '_new', '_reduce', '_reduce_ex', '_repr', '_r
mod', '_rmul', '_setattr', '_sizeof', '_str', '_subclasshook',
'_formatter_field_name_split', '_formatter_parser', '_capitalize', '_center',
'_count', '_decode', '_encode', '_endswith', '_expandtabs', '_find', '_format', '_index',
'_isalnum', '_isalpha', '_isdigit', '_islower', '_isspace', '_istitle', '_isupper',
'_join', '_ljust', '_lower', '_lstrip', '_partition', '_replace', '_rfind', '_rindex',
'_rjust', '_rpartition', '_rsplit', '_rstrip', '_split', '_splitlines', '_startswith',
'_strip', '_swapcase', '_title', '_translate', '_upper', '_zfill']
>>> t.title._doc_
'S.title() -> string\n\nReturn a titlecased version of S, i.e. words start with
uppercase\ncharacters, all remaining cased characters have lowercase.'
>>> "My name is {fname} {lname}".format(fname="Udi", lname="Bauman")
'My name is Udi Bauman'
>>> data = [42, 'foo', 3.14]
>>> data.append('bar')
>>> for item in data:
...     print item
...
42
foo
3.14
bar
bar
>>>

```

```

foo
3.14
bar
>>> immutable_data = (42, 'foo', 3.14)
>>> immutable_data.append('bar')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> dict = {"title": "Django tutorial", "location": "TLU"}
>>> dict["year"] = 2009
>>> for k in dict:
...     print k, "=", dict[k]
...
year = 2009
location = TLU
title = Django tutorial
>>> for k, v in dict.items():
...     print "<key>={val}".format(key=k, val=v)
...
year=2009
location=TLU
title=Django tutorial
>>> i = 0
>>> while i < 5:
...     print i
...     i += 1
...
0
1
2
3
4
>>> temp = 26
>>> if temp <= 0:
...     print "freezing"
... elif temp < 15:
...     print "cold"
... else:
...     print "hot"
...
hot
>>> data = [13, -2, 9]
>>> data2 = [x*2 for x in data if x > 0]
>>> data2
[26, 18]
>>> def factorial(n):
...     return n * factorial(n-1) if n > 1 else 1
>>> factorial(5)
120
>>>

```

To learn more on Python, I recommend these resources:

- Dive into Python - a great book for programmers, available for free in: <http://diveintopython.org>
- Python 101 videos from the PyCon: <http://us.pycon.org/2009/tutorials/schedule/2AM7/>

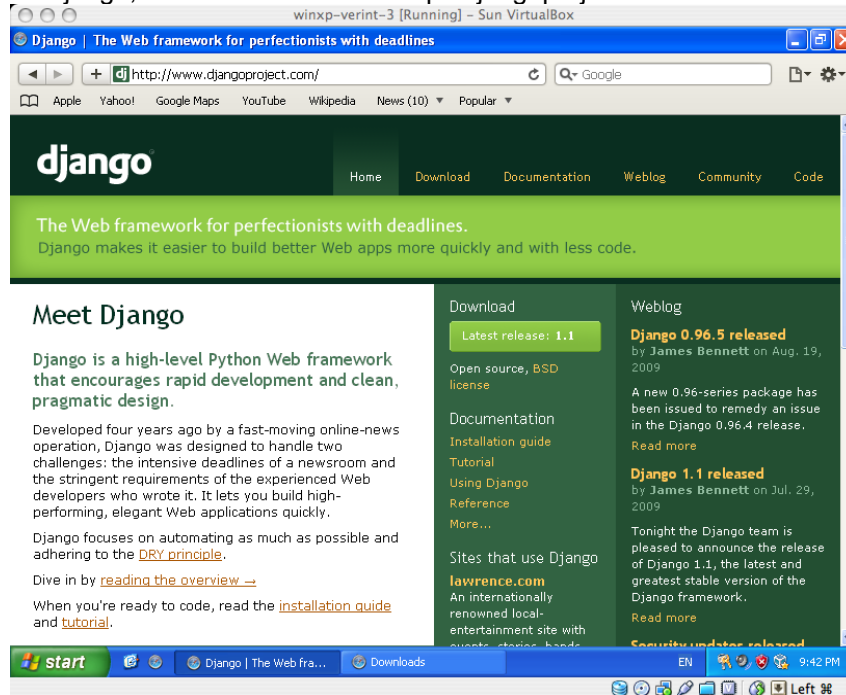
Django

Django is the leading Python web framework, and the technology subject of this article.

It was developed 5 years ago to support very fast development of web applications, by

- taking care of many tasks & aspects repeating in almost any web application
- Emphasizing a clean MVC design
- allowing you to focus on just the logic specific to your application

To install Django, 1st download it from: <http://djangoproject.com>



Current version of Django is 1.1. From my experience, it is normally safe to work with the latest development version, without waiting for official releases, because the code-base is kept very stable even in the development version-control version

To install Django:

Extract the archive downloaded:

- on windows download an archive tool such as WinRAR to do it (<http://rarlabs.com/download.htm>)
- on Mac & Linux, just double-click the archive file

open a command-line terminal to that folder, & run the command:

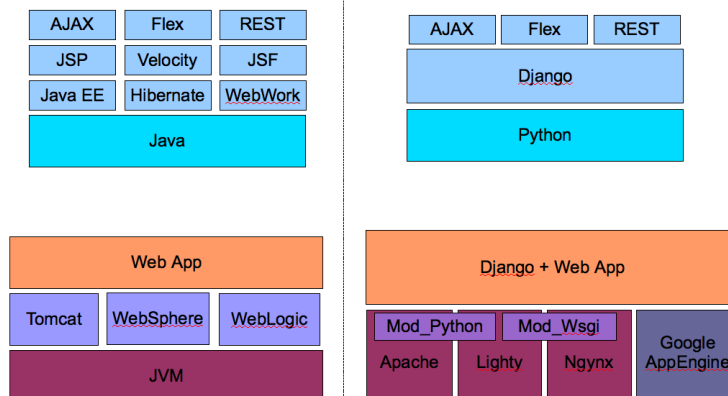
```
python setup.py install
```

To test that Django is installed, run the following command, in a command-line terminal:

```
django-admin.py
```

If you don't get an error message, it means that Django is properly installed. If the command isn't found, you may need to add "<Django folder>/django/bin" to your PATH environment variable.

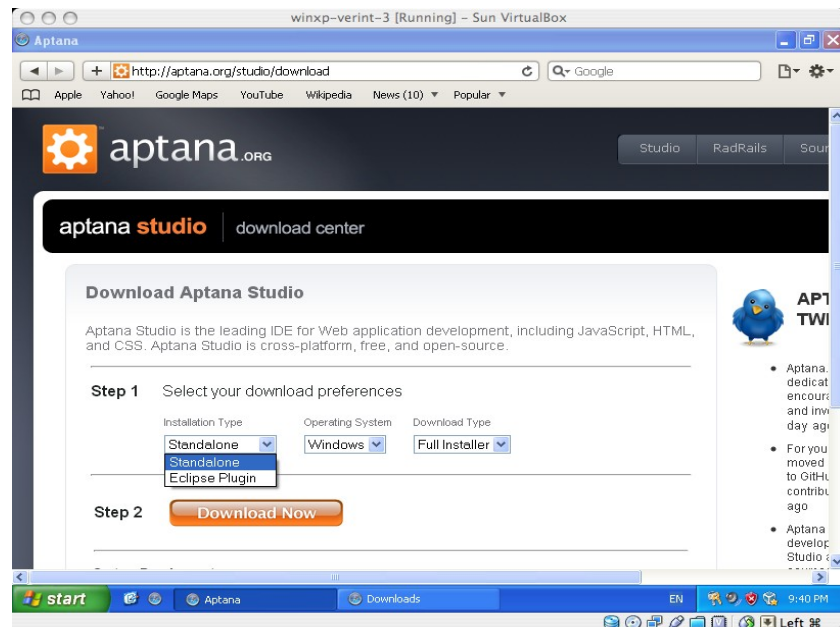
If you're coming from Java, here's a diagram to help you compare Django with equivalent Java frameworks & deployment options:



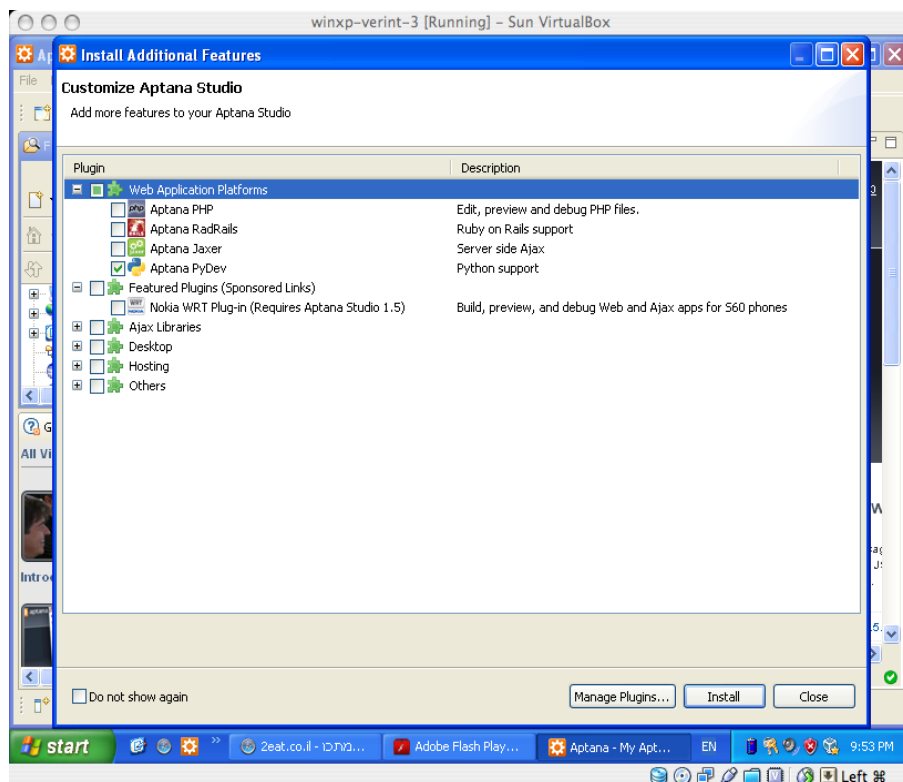
- To learn more on Django, I recommend the following resources:
- The Django tutorial: <http://docs.djangoproject.com/en/dev/intro/tutorial01/>
- The Django book: <http://www.djangobook.com/en/2.0/>

Aptana

Aptana is an IDE for dynamic languages, based on the Eclipse platform. Actually it is can also run inside Eclipse itself, as a plugin, so if you're already working with Eclipse, you can just install it over it. There isn't much difference between the 2. You can download Aptana from: <http://aptana.com/download>



After you run the installer file, you can run Aptana from the applications menu. You'll then be asked to choose which dynamic languages to support - choose Aptana PyDev:



To learn more on Aptana, try the video tutorials, appearing in the application side bar.

Starting the project

Let's start building our Project Management application.

To head off, we'll need to run the Django management command to start a new project folder.

Open a command line terminal in the folder where you'd like to have the project folder, e.g.:

```
c:\dev
```

Run the following command-line to create a project named "myprojects":

```
django-admin.py startproject myprojects
```

(or however you would like to name your project)

This will create a folder to contain the application, e.g.:

```
c:\dev\myprojects
```

The folder will contain the basic Django configuration & managements files, to be explained in more detail later:

- settings.py - The main project configuration file
- urls.py - The URL's configuration file
- manage.py - The script for running the project's management commands

Pluggable Apps

Django projects are designed to be modular & encourage re-use of self & 3rd-party modules.

All code must belong to logical sub-projects or modules, called: Pluggable Apps. Each is responsible for a subset of the application functionality, consisting of its own data model, UI & services

It is very easy to re-use your Pluggable Apps in other projects, or plug 3rd-party Apps into your project

To create a Pluggable App for our application:

Change directory in the terminal to the application folder created, e.g., c:\dev\myprojects

Run the following command-line to create a pluggable-app named "tasks":

```
python manage.py startapp tasks
```

This will create a folder for our pluggable app, under the project folder.

The folder will contain the default files composing a Pluggable App:

- models.py - The file in which we'll define the pluggable app data model
- views.py - The file in which we'll define the pluggable app views
- tests.py - The file in which we'll define our unit-tests

Now, let's open the project in Aptana, to work on our application:

Start the Aptana IDE

From the File menu, choose New > project

Choose PyDev > PyDev project & click on Next

Enter the project name, e.g., myprojects

Uncheck the "Use default" checkbox

Browse the folder you created for the project, e.g.

```
c:\dev\myprojects
```

Make sure a Python version matching the one you installed is selected, e.g., Python 2.6

If no interpreter is listed in the Interpreter drop down list, click the link below it to configure an interpreter

In the dialog window that opens, click on "Auto config"

Click OK

Uncheck the "Create src folder" checkbox

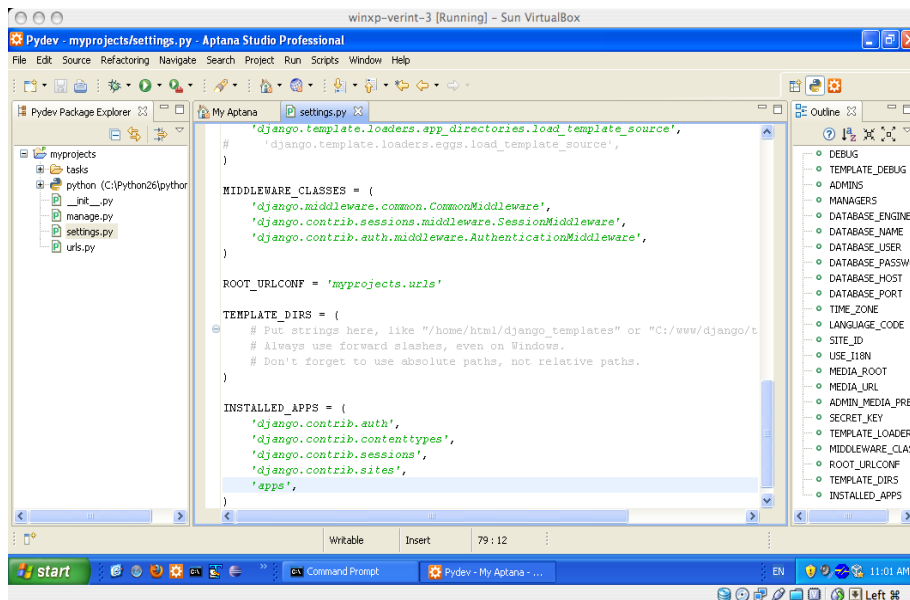
Click on "Finish"

To plug the pluggable app we created into our application, we'll need to add it to the list of installed apps in the project configuration:

In the project explorer, expand the project root, & double-click the file settings.py in the project root folder

Scroll down till you find the variable: INSTALLED_APPS

Add another entry in the list of apps: 'tasks'



The

other pluggable apps that already appear in the list are built-in apps, that are included by default in every new project

Running the development server

We haven't yet written anything in our app, but it will be nice to run it already, just to see that the configuration is right.

Open a command-line terminal in the project root folder

Run the command:

```
python manage.py runserver
```

This runs the development server with our application

By default it serves the application in port 8000, so make sure it's not caught.

To run in a different port, e.g., 8123, run:

```
python manage.py runserver localhost:8123
```

If you want your app to be available in your network, add your machine name or IP, e.g.:

```
python manage.py 10.123.45.6:80
```

If all goes well, & you don't have errors, you should get the output:

```
Development server is running at http://127.0.0.1:8000/
```

Now point your browser to that address, e.g.

```
http://localhost:8000
```

You should see a page saying:

```
It worked!
```

which is the default home page for a Django web site.

Modeling the application data

Enough preparations, let's start coding.

We'll start with our application data model.

We'll need to define 2 entities: Project & Task

In Django, data model is defined by simply creating the OO classes (business objects) that describe our domain.

Because the actual application database will be derived from this OO model, the classes attributes usually contain metadata that affects the database definition, such as:

- whether an attribute is nullable
- whether an attribute is unique
- whether an index should be created for some attribute

The application data model is defined in the file models.py in our tasks app

Locate the file in the Aptana package explorer

Double-click it

Let's define the Project entity first:

Here's how you create a class in Python called "Project", that extend a class called models.Model:

```
class Project(models.Model):
```

[...]

Adding attributes to the entity is done like this:

```
class Project(models.Model):
    name = models.CharField(unique=True, max_length=500)
    description = models.TextField(max_length=4000, null=True, blank=True)
    is_active = models.BooleanField(default=True)
    leader = models.ForeignKey(User, help_text="Person leading this project")
    priority = models.IntegerField(choices=((0, "Urgent"), (1, "High"), (2, "Medium"), (3, "Low")))
    last_update = models.DateTimeField(auto_now=True)
```

The entity attributes are declared as class members, & assigned values according to their data type & metadata, e.g.:

```
name = models.CharField(unique=True, max_length=500)
```

Note that the parameters are usually passed together with their name (parameter=value)

This allows a function call to specify parameters in any order, & also pass just a subset of the parameters (if they have default values)

Note that unlike other programming languages, Python requires you to maintain consistent indentation of blocks (instead of using other means to demarcate them, such as {})

In our example, the contents of the Project class is in a block, consisting of all lines following the class header, indented consistently

The code convention is to use 4 spaces as a unit of indentation, but you can use tab if you really like to.

The "leader" attribute is a bit special as it serves as relation to another entity, of class User. In order for this to work, we must add an import of the User class, which arrives from a different pluggable app called: auth

Add in the top of the file the import line:

```
from django.contrib.auth.models import User
```



If you're wondering what's the data structure passed to the choices parameter:

In Python there are 3 main types of collections:

lists - ordered list of objects of any type

e.g., [1, 3, "a"]

tuples - immutable lists

e.g., (1, 3, "a")

maps - key-value maps (dictionaries). Both keys & values can be of any type

e.g., {"a": 1, "b": 2, "c": 3}

The collection of choices to a field is a tuple of tuples each containing 2 values:

((0, "Urgent"), (1, "High"), (2, "Medium"), (3, "Low"))

The field will contain the 1st value in each tuple, but Django will present to the user the 2nd value.

We can also add some methods to the entity.

For now let's just add a method to describe a project as a string (similar to "toString" in java)

Such method in Python is called: `__str__`, or `__unicode__`

The `__` means that the method is an internal system method, & not part of the object API

```
class Project(models.Model):
    name = models.CharField(unique=True, max_length=500)
    description = models.TextField(max_length=4000, null=True, blank=True)
    is_active = models.BooleanField(default=True)
    leader = models.ForeignKey(User, help_text="Person leading this project")
    priority = models.IntegerField(choices=((0, "Urgent"), (1, "High"), (2, "Medium"), (3, "Low")))
    last_update = models.DateTimeField(auto_now=True)
```

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



Functions in python look like this:

```
def add(x, y):
    return x + y
```

We could also add documentation (similar to JavaDoc):

```
def add(x, y):
    """
    Adds the 2 given numbers.
    >>> add(3, 4)
```

```

7
"""
return x + y

```

Note: the last 2 lines in the documentation were copied from the python interactive shell. Adding them to a method's documentation enables Python to automatically unit-test the method: it runs the lines starting with ">>>" & verifies that the resulting output is as appearing in the following line

An object method in python is a function appearing inside a class, & having as 1st parameter a variable, "self", serving as reference to the current instance of the object

This is different than languages such as Java, in which the reference "this" is always available, without explicitly declaring it as a parameter to methods. When invoking methods though, you don't need to provide this parameter.

Moving on, let's define also the Task entity:

```

class Task(models.Model):
    project = models.ForeignKey(Project)
    name = models.CharField(unique=True, max_length=500)
    description = models.TextField(max_length=4000, null=True, blank=True)
    assigned_to = models.ForeignKey(User, help_text="Person assigned with this task")
    due_in = models.DateTimeField(null=True, blank=True)
    done = models.BooleanField(default=False)
    dependencies = models.ManyToManyField('self', null=True, blank=True, help_text="Which tasks must be
completed before starting this task")
    simplicity = models.IntegerField(choices=((1, "1 - Very complex/hard"), (2, "2"), (3, "3"), (4, "4"), (5, "5"),
(6, "6"), (7, "7"), (8, "8"), (9, "9"), (10, "10 - Very simple/easy")),
help_text="Estimate the ease of implementation of this task")
    value = models.IntegerField(choices=((1, "1 - Unimportant"), (2, "2"), (3, "3"), (4, "4"), (5, "5"),
(6, "6"), (7, "7"), (8, "8"), (9, "9"), (10, "10 - Very important")),
help_text="Estimate the importance of implementing this task")
    last_update = models.DateTimeField(auto_now=True)

    def __unicode__(self):
        return self.name

```

Note the dependencies field which is a many-to-many relation to some entity, in this case, the same entity

This means that any task can be dependent on several other tasks

Configuring the UI

So far we defined our application data model, now we're moving on the UI.

We're going to let Django do most of the UI for us, but we'd like to control it with some metadata

Django arrives with a pluggable app called "Admin" which creates on the fly end-user quality UI

This saves a lot of development time by automating the repetitive task of creating data entry UI for every project

In order to use the admin pluggable app, we need to add it to the list of installed apps:

Open the settings.py file from the project root folder

Scroll to the list of installed apps

Add 'django.contrib.admin' as an entry before the 'tasks' entry:

```

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'tasks',
)

```

To do that let's add another file to the tasks pluggable app, called: admin.py

right-click the tasks folder in the package explorer in Aptana

Choose New > File

Enter "admin.py" as name

Click finish

The file should first import the admin app & the models defined in models.py

```
from django.contrib import admin
```

```
from models import *
```

It then needs define some UI metadata to control the UI of each entity

Let's add UI metadata for the Project entity, it's done by adding a class extending `admin.ModelAdmin`:

```
class ProjectAdmin(admin.ModelAdmin):
    list_display = ('name', 'is_active', 'leader', 'priority', 'last_update')
    search_fields = ('name', 'description')
    list_filter = ('is_active', 'leader', 'priority')
    ordering = ('priority',)
```

Note: in Python, tuples containing just 1 element must end with a comma, e.g.
`list_filter = ('status',)`

The metadata contains mainly lists of fields, & the behavior that will apply to them, e.g.

- `list_display` - determines which fields will appear in a table listing objects
- `search_fields` - determines which fields will participate in textual search of objects
- `list_filter` - determines which filters will appear, for field values
- `ordering` - determines the fields by which a list of objects will be sorted by default

Similarly let's define the UI metadata for the Task entity:

```
class TaskAdmin(admin.ModelAdmin):
    list_display = ('name', 'project', 'done', 'assigned_to', 'due_in', 'simplicity', 'value', 'last_update')
    search_fields = ('name', 'description')
    list_filter = ('done', 'project', 'assigned_to',)
    date_hierarchy = 'due_in'
```

The `date_hierarchy` setting will organize our tasks according in a date hierarchy (years, months, days), according to the tasks `due_in` field

In order for the admin app to use the defined metadata, we need to register the metadata classes with the models classes

Add at the bottom of the file the following lines:

```
admin.site.register(Project, ProjectAdmin)
admin.site.register(Task, TaskAdmin)
```

Optionally, we may want to be able to edit a project's tasks within the project page (allowing us to add tasks directly when editing a project)

To do that let's add another UI metadata class defining a task edited within the project page

Add before the class `ProjectAdmin`:

```
class TaskInline(admin.TabularInline):
    model = Task
    extra = 5
```

`extra` defines how many empty lines will be created to fill new tasks

Now add a line to the `ProjectAdmin` class, saying which in-line forms will be included in it (just the one we defined for `Task`):

```
class ProjectAdmin(admin.ModelAdmin):
    list_display = ('name', 'is_active', 'leader', 'priority', 'last_update')
    search_fields = ('name', 'description')
    list_filter = ('is_active', 'leader', 'priority')
    ordering = ('priority',)
    inlines = (TaskInline,)
```

Configuring the URL's

Unlike some other technologies, in which the URL of web pages isn't logical but rather reflects the specific web site implementation, Django let's you design the URL's of your application pages, decoupled from their implementation

To do that you configure the URL's in a file called `urls.py`

The project root folder contains this file, & you can also have a `urls.py` file in each pluggable app, to define it's own URL's

For now, let's edit the URL's of the whole project:

Edit the file `urls.py` in the project root folder

Uncomment the lines required for the admin app:

```
from django.contrib import admin
```

```
admin.autodiscover()
```

Uncomment the line including the admin URL's:

```
(r'^admin/', include(admin.site.urls)),
```

You should end up with the following configuration:

```
from django.conf.urls.defaults import *
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns("",
    (r'^admin/doc/', include('django.contrib.admin.urls')),
    (r'^admin/', include(admin.site.urls)),
)
```

We'll explain the contents of the file in detail later

Also, let's add a redirect handler for the root URL (/), that will redirect us to the admin pluggable app (/admin/)

```
('^$', 'django.views.generic.simple.redirect_to', {'url': '/admin/'}),
```

Sync'ing the database

At this stage our application is already ready for initial usage, except for 1 thing: it has no database to work against.

The database configuration is done in the global project configuration file:

Open the file settings.py from the project root folder

Find the lines starting with DATABASE_

If you have some database server somewhere, you can create your database there, & configure the project to work against it

To make things simpler, let's use a database arriving with Python, called SQLite

This database is created by Django automatically

It is saved in a simple file

To configure it enter as the value of DATABASE_ENGINE:

```
DATABASE_ENGINE = 'sqlite3'
```

Create a subfolder under the project folder called db

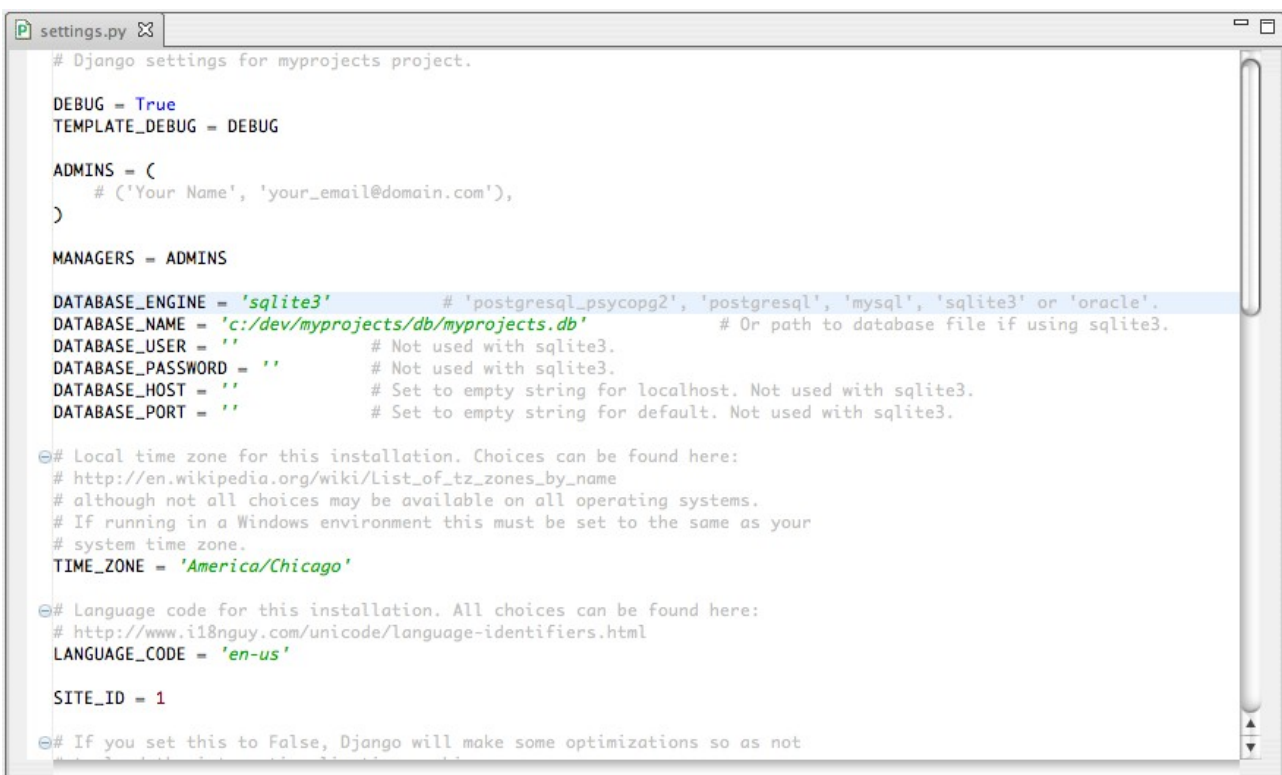
Inside this folder, we'll have a file containing our database, e.g.: myprojects.db

No need to create this file, Django will do it for us

Add the path to this file as the value of DATABASE_NAME, e.g.:

```
DATABASE_NAME = 'c:\dev\myprojects\db\myprojects.db'
```

Since we're using SQLite, there's no need to define the other database configuration attributes (host, port, user & password)



```
settings.py
# Django settings for myprojects project.

DEBUG = True
TEMPLATE_DEBUG = DEBUG

ADMINS = (
    # ('Your Name', 'your_email@domain.com'),
)

MANAGERS = ADMINS

DATABASE_ENGINE = 'sqlite3' # 'postgresql_psycopg2', 'postgresql', 'mysql', 'sqlite3' or 'oracle'.
DATABASE_NAME = 'c:/dev/myprojects/db/myprojects.db' # Or path to database file if using sqlite3.
DATABASE_USER = '' # Not used with sqlite3.
DATABASE_PASSWORD = '' # Not used with sqlite3.
DATABASE_HOST = '' # Set to empty string for localhost. Not used with sqlite3.
DATABASE_PORT = '' # Set to empty string for default. Not used with sqlite3.

# Local time zone for this installation. Choices can be found here:
# http://en.wikipedia.org/wiki/List_of_tz_zones_by_name
# although not all choices may be available on all operating systems.
# If running in a Windows environment this must be set to the same as your
# system time zone.
TIME_ZONE = 'America/Chicago'

# Language code for this installation. All choices can be found here:
# http://www.i18nguy.com/unicode/language-identifiers.html
LANGUAGE_CODE = 'en-us'

SITE_ID = 1

# If you set this to False, Django will make some optimizations so as not
```

Save & close the file

Now, let's ask Django to create the database

Switch to the command-line terminal in the project root folder

Stop the development server, if it's running, by pressing Ctrl+C

Run the following command:

```
python manage.py syncdb
```

If there are no errors, Django will create the database & all tables

You'll be asked for a default admin user details

Now run the development server again

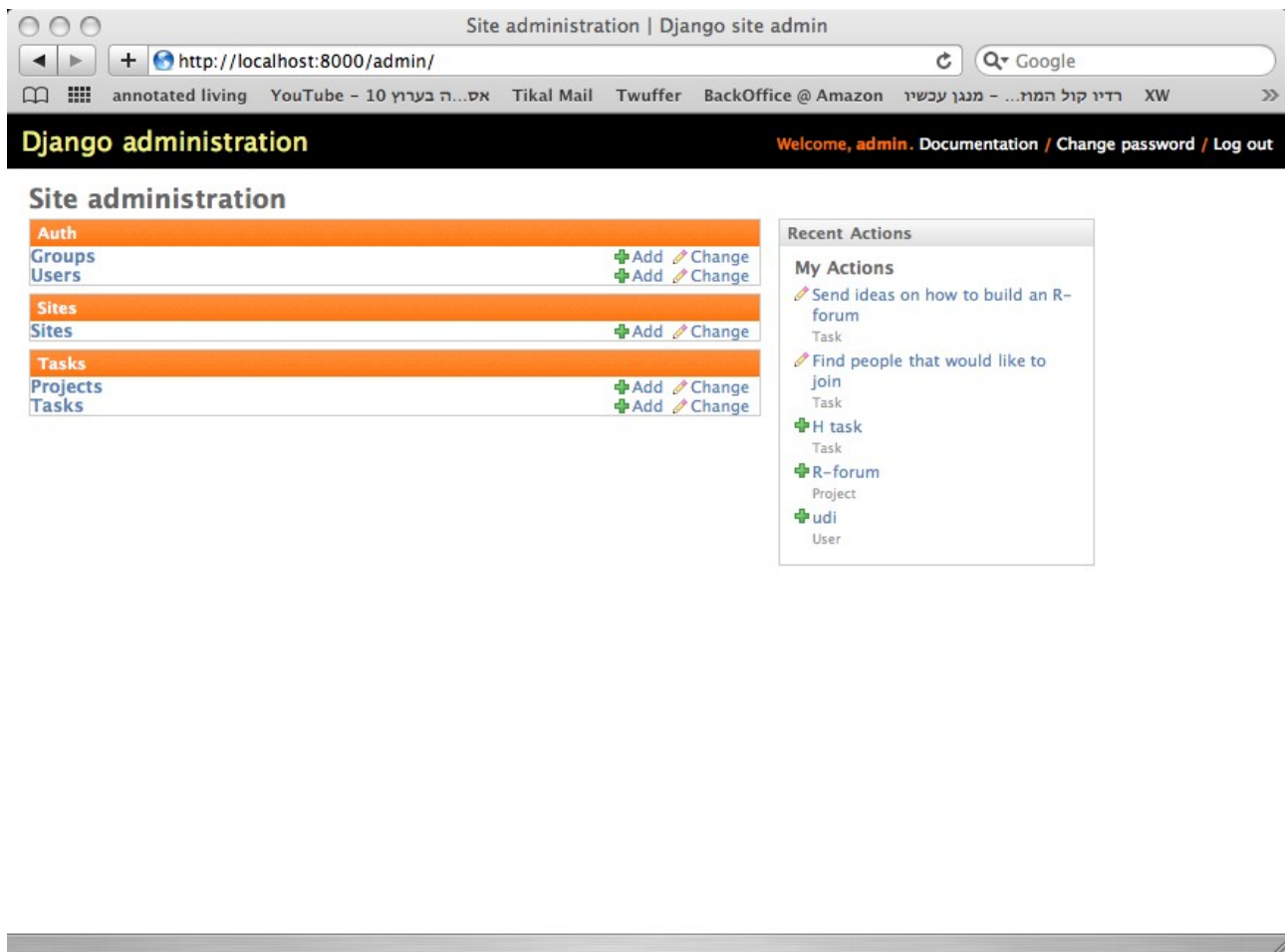
```
python manage.py runserver
```

Point your browser to the application again:

```
http://localhost:8000/admin
```

Fill the admin credentials you chose when running syncdb

You're supposed to see the UI Django generated for us



Click the "Add" link next to Projects to add a 1st project

As you can see, Django manages & abstracts the database for us, according to our logical OO data model.

This BTW is a difference from other RAD frameworks (such as Rails) which are database - as opposed to model - driven

We'll see later how this abstraction continues with the Django ORM

This includes managing the evolution of the database schema, using built-in & pluggable apps
In order to continue the tutorial, please add a few projects & tasks, to have some sample data to work with

Adding project health indicators

Let's add to the project list some health indicators - summary of how the project tasks stand against their due time

Open the admin.py file in the tasks folder

This file defines the UI metadata, controlling the UI Django creates for us

Add to the display_list of Project another field, called 'health_indicator'

```
class ProjectAdmin(admin.ModelAdmin):
    list_display = ('name', 'is_active', 'leader', 'priority', 'health_indicator', 'last_update')
    search_fields = ('name', 'description')
    list_filter = ('is_active', 'leader', 'priority')
    ordering = ('priority',)
    inlines = (TaskInline,)
```

This field isn't an actual model field, but rather a function that we will now write, that generates the HTML code to be presented for each object in that column

The function should be inside the class defining the Project UI metadata

This function receives as parameter a reference to the current Project object displayed in the current row, & returns an HTML, e.g.:

```
def health_indicator(self, obj):
    return "Project " + obj.name + " is healthy"
```

Adds this to the ProjectAdmin class, & browse to the Projects list to see the result

What our function really needs to do is to summarize the tasks status of the project

To do so, we would like to loop on the list of tasks of the current project & calculate their health

If you remember the list of attributes in the Project entity does not contain a list of tasks.

The Task entity does have a reference to Project

What we need to add is an attribute in that relation specifying that we want to have a reverse attribute of Project pointing to its tasks

This attribute is called related_name & should be specified like this:

```
class Task(models.Model):
    project = models.ForeignKey(Project, related_name="tasks")
    name = models.CharField(unique=True, max_length=500)
    description = models.TextField(max_length=4000, null=True, blank=True)
    assigned_to = models.ForeignKey(User, help_text="Person assigned with this task")
    due_in = models.DateTimeField(null=True, blank=True)
    done = models.BooleanField(default=False)
    dependencies = models.ManyToManyField('self', null=True, blank=True,
    help_text="Which tasks must be completed before starting this task")
    simplicity = models.IntegerField(choices=((1, "1 - Very complex/hard"), (2, "2"), (3, "3"), (4, "4"), (5, "5"),
    (6, "6"), (7, "7"), (8, "8"), (9, "9"), (10, "10 - Very simple/easy")),
    help_text="Estimate the ease of implementation of this task")
    value = models.IntegerField(choices=((1, "1 - Unimportant"), (2, "2"), (3, "3"), (4, "4"), (5, "5"), (6, "6"),
    (7, "7"), (8, "8"), (9, "9"), (10, "10 - Very important")),
    help_text="Estimate the importance of implementing this task")

    last_update = models.DateTimeField(auto_now=True)
```

Now we can loop over the tasks of an project like this:

```
def health_indicator(self, obj):
    for task in obj.tasks.all():
        # count tasks that are done / on-track / late
```

Let's have our health indicator column show a count of tasks in the project that are done, on-track & late.

How will we know whether a task is on-track?

if it's due_in field contains a date later than today, or if it's done

To easily calculate this, let's add a method to the Task entity

Open the models.py file in the tasks folder

Find the Task class

Add this method:

```
def is_on_track(self):
    import datetime
    return self.done or (self.due_in != None and self.due_in > datetime.datetime.now())
```

Now we can classify the tasks according to their status, in the health_indicator function:

```
def health_indicator(self, obj):
    tasks_status = {"done": 0, "on-track": 0, "late": 0}
    for task in obj.tasks.all():
        if task.done:
            tasks_status["done"] = tasks_status["done"] + 1
        elif task.is_on_track():
```

```

tasks_status["on-track"] = tasks_status["on-track"] + 1
else:
    tasks_status["late"] = tasks_status["late"] + 1
return ", ".join(["%d-%s" % (count, status) for status, count in tasks_status.items() if count > 0])

```



What have we done here?

tasks_status is a map of status name -> count of tasks

we loop on the project tasks by calling: obj.tasks.all()

tasks.all() is a QuerySet object that translates to an SQL select statement, selecting all tasks that belong to the project

We then want to generate a string concatenating the status names & counts, if the count is larger than 0, e.g., 1-done,5-on-track

In python to format a string such as "1-done", given 2 variables count & status, you write: "%d-%s" % (count, status)

tasks_status is a map mapping status to counts. tasks_status.items() is a list of pairs, e.g., [("done", 1), ("on-track", 5), ("late", 0)]

This is a list of tuples

Python offers a nice feature to map 1 list to another, called list comprehension.

Here's a simple example:

```
list1 = [3, 5, 17, 12]
```

```
list2 = [x*2 for x in list1 if x > 10]
```

```
list2
```

```
[34, 24]
```

If we have a list such as:

```
[("done", 1), ("on-track", 5), ("late", 0)]
```

And we want a list such as:

```
["1-done", "5-on-track"]
```

We can do that using this list comprehension:

```
["%d-%s" % (count, status) for status, count in tasks_status.items() if count > 0]
```

Once we have this list, we want to create a comma-delimited string out of it, which is done like this:

```
", ".join(["%d-%s" % (count, status) for status, count in tasks_status.items() if count > 0])
```

We could also add nice colors:

```

def health_indicator(self, obj):
    tasks_status = {"done": 0, "on-track": 0, "late": 0}
    colors = {"done": "green", "on-track": "silver", "late": "red"}
    for task in obj.tasks.all():
        if task.done:
            tasks_status["done"] = tasks_status["done"] + 1
        elif task.is_on_track():
            tasks_status["on-track"] = tasks_status["on-track"] + 1
        else:
            tasks_status["late"] = tasks_status["late"] + 1
    return ", ".join(["<span style='background-color: %s; color: white;'>%d-%s</span>" %
        (colors[status], count, status)
        for status, count in tasks_status.items()
        if count > 0])

```

Also, in order to add HTML to table column (which is prevented by default, for security reasons), we'll need to add to the health_indicator column a permission to include tags:
health_indicator.allow_tags = True

Select project to change | Django site admin

http://localhost:8000/admin/tasks/project/

Django administration Welcome, admin. Documentation / Change password / Log out

Home > Tasks > Projects

Select project to change Add project +

Search: [] Go

Action: [] Go

<input type="checkbox"/>	Name	Is active	Leader	Priority	Health indicator	Tasks	Last update
<input type="checkbox"/>	R-forum	<input checked="" type="checkbox"/>	udi	Low	1-late, 1-on-track, 1-done	List Focus diagram	Sept. 17, 2009, 3:31 p.m.

1 project

Filter

- By is active
 - All
 - Yes
 - No
- By leader
 - All
 - admin
 - udi
- By priority
 - All
 - Urgent
 - High
 - Medium
 - Low

The Tasks focus view

Next we would like to show a nice visualization of the list of tasks, that would suggest us which tasks should be addressed first, according to their given simplicity & value

We'll have an HTML page that shows a simple chart, that shows the tasks according to these dimensions

What we need to do is:

- Configure a URL for this page
- Create a view function, that prepares the model for this page, i.e., the list of tasks in the selected project
- Create a template that renders the page
- Add to it a graph, using some visualization framework
- Add a link to this page from the list of projects

Configure a URL for this page

Although we could define all URL's in the main urls.py file, in the project root folder, it's not a best-practice, because it will make it harder to re-use our tasks pluggable-app.

The best-practice is to have a separate urls.py for each pluggable app, that configure its own URL's

Create a file called urls.py inside the tasks folder

In the Aptana project explorer, right-click the tasks folder, choose New > File, & enter the name: urls.py

Here's how our urls.py should look like:

```
from django.conf.urls.defaults import *
```

```
urlpatterns = patterns("",
    (r'^project/(?P<project_id>\d+)/focus', 'tasks.views.tasks_focus'),
)
```

The regular expression contains a number variable: `project_id`, which will be passed as a parameter to the view function

The file is a simple map of URL's, represented as regular-expressions, & functions that handle these URL's

The URL we'll have for this page will be:

```
/tasks/project/2/focus/
```

Our view function will be called `tasks_focus` & will be inside the `views.py` file under the `tasks` folder

Now, we need to refer to this URL's configuration file from the central URL's configuration file

Open the file `urls.py` from the root project folder

Add into the `urlpatterns` list the line:

```
(r'^tasks/', include('myprojects.tasks.urls')),
```

This means that any URL starting with `'/tasks/'` will be resolved using the URL's configuration file in the `tasks` pluggable-app

Review again the `urls.py` file under the `tasks` folder, & observe that the regex for our page starts with the prefix `'project/'`, while the URL we've designed starts with `'/tasks/project/'`.

This is because the `'/tasks/'` prefix is used for all URL's in this URL's configuration file.

The view function

Open the `views.py` file under the `tasks` folder

This file contains functions that prepare the model for the actual views our application will render

In Django the MVC pattern is actually MTV:

Model

the data

View

serves to prepare the model to be included in the rendered page

Template

does the actual rendering of pages

The simplest view we could do, that just returns an HTML string looks like this:

```
from django.http import HttpResponse
```

```
def tasks_focus(request, project_id):
```

```
    return HttpResponse("Tasks focus for project %d" % int(project_id))
```

Write it in the `views.py` file

Point your browser to:

```
http://localhost:8000/tasks/project/1/focus/
```

We assume that you have created a 1st project, with id 1

What we really want to do is to fetch the tasks for the given project

First, let's read the project entity, to display it's name in the page title

This is normally done like this:

```
from django.http import HttpResponse, Http404
```

```
from models import *
```

```
def tasks_focus(request, project_id):
```

```
    try:
```

```
        project = Project.objects.get(pk=int(project_id))
```

```
    except:
```

```
        return Http404("Couldn't find project with id %s" % project_id)
```

```
    return HttpResponse("Tasks focus for project %d" % int(project_id))
```

`Project.objects.get` is the Django ORM way to retrieve an entity by it's primary key

It throws an exception when the key isn't found, which we catch & return an HTTP 404

This is because we're trying to follow REST best-practices that treat data as resources

Django has a shortcut to save us from all this code:

```
from django.http import HttpResponse
```

```
from django.shortcuts import get_object_or_404
```

```
from models import *
```

```
def tasks_focus(request, project_id):
```

```
    project = get_object_or_404(Project, pk=int(project_id))
```

```
return HttpResponseRedirect("Tasks focus for project %d" % int(project_id))
```

Getting the project tasks is easy:

```
tasks = project.tasks.all()
```

Finally, we want to render a template of the page to the HTTP response, providing it the model we've fetched from the database

This is done using another Django shortcut:

```
from django.http import HttpResponseRedirect
from django.shortcuts import get_object_or_404, render_to_response
from models import *

def tasks_focus(request, project_id):
    project = get_object_or_404(Project, pk=int(project_id))
    tasks = project.tasks.all()
    return render_to_response("tasks_focus.html", locals())
```

render_to_response receives 2 parameters:

- the name of a template files, see next section for more information

- a map of model variables

We're using the Python locals() function to return a map of all variables assigned in the current function scope (in the form variable-name -> variable value)

In our case, the map returned by locals will contain 2 variables: project & tasks

That's all for our view, now let's write our template

Create a template

Django comes with its own Template Engine, allowing Web designers to author any text file, rendering an output view, in which we embed model variables.

Features include:

- Templates define named blocks of content

- Templates support inheritance: a template can extend other templates, & just override some of its ancestors blocks

 - This is very useful to factor common layout & presentation in base templates

- Templates limit - by-design - inclusion of business logic, besides simple conditions & loops

 - This is in order to enforce clean MVC design

- There are however many filters that let you modify the rendering of model variables

- & if you really want - it's quite easy to write your own template tags, that can generate any logic you want

Django looks up templates in a set of folders listed in the configuration

- 1st, create a folder under the tasks folder, named templates

- Open the settings.py file in the project root folder

- Scroll to the variable: TEMPLATE_DIRS

- Add to the variable value the absolute path to the folder you've just created, followed by a comma

 - e.g.

```
TEMPLATE_DIRS = (
    'c:/dev/myprojects/tasks/templates/',
)
```

Save & close the settings file

Note: the tips section at the end of this article, will explain how to avoid using absolute path here

Create a file inside the templates folder, called: tasks_focus.html

Open the file in Aptana, & type:

```
<h1>Tasks focus for project {{ project.name }}</h1>
```

This is how you use a model variable (project), that was provided by the view function we created earlier. This is similar to writing <%= variable %> in JSP.

We can access the model variables fields & methods, and also apply various filters on it,

e.g., truncate the number of words or change case.

Save the file & point your browser to:

```
http://localhost:8000/tasks/project/1/focus/
```

As mentioned, Django supports inheritance

Let's have our template extend the base template of the admin pluggable app. Add this to the template file:

```
{% extends "admin/base_site.html" %}
```

```
{% block title %}Project {{ project.name }} - Tasks focus{% endblock %}

{% block content %}
<h1>Tasks focus for project {{ project.name }}</h1>

{% endblock %}
```

If we just needed a list of tasks, we could have do that like this:

```
{% extends "admin/base_site.html" %}

{% block title %}Project {{ project.name }} - Tasks focus{% endblock %}

{% block content %}
<h1>Tasks focus for project {{ project.name }}</h1>

{% for task in tasks %}
<li><a href="/admin/tasks/task/{{ task.id }}">{{ task.name }}</a>
{% endfor %}

{% endblock %}
```

Django also supports a reverse function, that decouples the link from our URL's configuration

Add a diagram

What we really want is to have a diagram, showing our tasks by dimensions, so that we'll see on which tasks should our focus be

We'll use Google Charts API, which have few benefits:

- Work well on all browsers
- Don't require us to include & maintain any dependency libraries in our project
- Very expressive & powerful

Browse to the Google Charts API, from which we would like to use the Scatter chart.

<http://code.google.com/apis/chart/>

We add the code that generates the chart - a simple HTML img tag - to our template, & customize it a bit

```

```

Save & browse again to the page:

<http://localhost:8000/tasks/project/1/focus/>

This contains however just sample data. We want to use the real tasks data arriving from the view function.

Instead of writing complex logic in the template, we prefer to prepare all that's needed in the view function, even if it gets a bit coupled with the presentation logic

So let's replace the hard-coded sample data with variables that we'll prepare in the view function:

```

```

And now add the calculation of these variables to the view function:

```
tasks_simplicity_data = ",".join(["%d" % (10*task.simplicity) for task in tasks])
tasks_value_data = ",".join(["%d" % (10*task.value) for task in tasks])
```

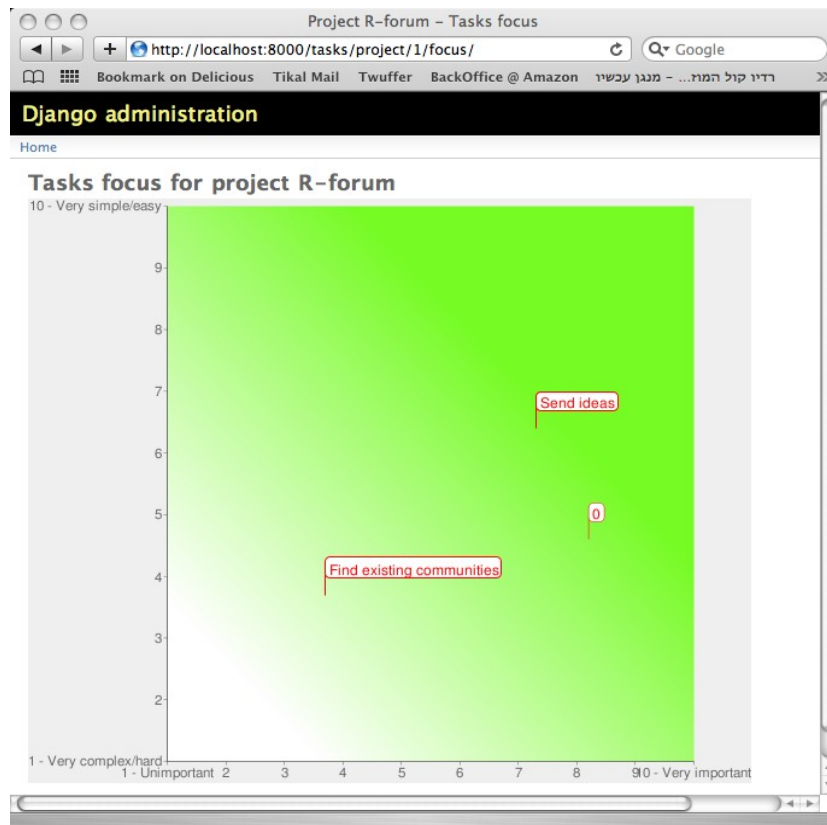
```
tasks_label_data = "|".join(["f%s,FF0000,0,%d,12" % (task.name, idx) for idx, task in
enumerate(tasks)])
```

We multiplied the simplicity & value values by 10, to match the default scale of the charts API (0-100)

We used a trick of looping over a collection & getting both the current item & its index, using: for idx, task in enumerate(tasks): ...

Save the template & refresh the page in the browser

<http://localhost:8000/tasks/project/1/focus/>



Django has an important design principle called DRY - Don't repeat yourself, which applies mainly to avoiding duplication

Our code breaks this principle, because the chart axes scales is defined both in our model fields metadata, & in our charts.

Let's quickly fix that

Open the models.py file, & extract the choices metadata into a variable:

```
SIMPLICITY_CHOICES = ((1, "1 - Very complex/hard"), (2, "2"), (3, "3"), (4, "4"), (5, "5"),
                      (6, "6"), (7, "7"), (8, "8"), (9, "9"), (10, "10 - Very simple/easy"))
```

```
VALUE_CHOICES = ((1, "1 - Unimportant"), (2, "2"), (3, "3"), (4, "4"), (5, "5"),
                 (6, "6"), (7, "7"), (8, "8"), (9, "9"), (10, "10 - Very important"))
```

```
class Task(models.Model):
    project = models.ForeignKey(Project, related_name="tasks")
    name = models.CharField(unique=True, max_length=500)
    description = models.TextField(max_length=4000, null=True, blank=True)
    assigned_to = models.ForeignKey(User, help_text="Person assigned with this task")
    due_in = models.DateTimeField(null=True, blank=True)
    done = models.BooleanField(default=False)
    dependencies = models.ManyToManyField('self', null=True, blank=True, help_text="Which tasks must be
    completed before starting this task")
    simplicity = models.IntegerField(choices=SIMPLICITY_CHOICES, help_text="Estimate the ease of
    implementation of this task")
    value = models.IntegerField(choices=VALUE_CHOICES, help_text="Estimate the importance of
    implementing this task")
    last_update = models.DateTimeField(auto_now=True)
```

```

models.py
from django.db import models
from django.contrib.auth.models import User

class Project(models.Model):
    name = models.CharField(unique=True, max_length=500)
    description = models.TextField(max_length=4000, null=True, blank=True)
    is_active = models.BooleanField(default=True)
    leader = models.ForeignKey(User, help_text="Person leading this project")
    priority = models.IntegerField(choices=((0, "Urgent"), (1, "High"), (2, "Medium"), (3, "Low")))
    last_update = models.DateTimeField(auto_now=True)

    def __unicode__(self):
        return self.name

SIMPLICITY_CHOICES = ((1, "1 - Very complex/hard"), (2, "2"), (3, "3"), (4, "4"), (5, "5"),
                     (6, "6"), (7, "7"), (8, "8"), (9, "9"), (10, "10 - Very simple/easy"))

VALUE_CHOICES = ((1, "1 - Unimportant"), (2, "2"), (3, "3"), (4, "4"), (5, "5"),
                 (6, "6"), (7, "7"), (8, "8"), (9, "9"), (10, "10 - Very important"))

class Task(models.Model):
    project = models.ForeignKey(Project, related_name="tasks")
    name = models.CharField(unique=True, max_length=500)
    description = models.TextField(max_length=4000, null=True, blank=True)
    assigned_to = models.ForeignKey(User, help_text="Person assigned with this task")
    due_in = models.DateTimeField(null=True, blank=True)
    done = models.BooleanField(default=False)
    dependencies = models.ManyToManyField('self', null=True, blank=True, help_text="Which tasks must be completed before starting this")
    simplicity = models.IntegerField(choices=SIMPLICITY_CHOICES, help_text="Estimate the ease of implementation of this task")
    value = models.IntegerField(choices=VALUE_CHOICES, help_text="Estimate the importance of implementing this task")
    last_update = models.DateTimeField(auto_now=True)

    def __unicode__(self):
        return self.name

    def is_on_track(self):
        import datetime
        return self.done or (self.due_in != None and self.due_in > datetime.datetime.now())

```

Now, create variables based on this metadata in our view function:

```

def tasks_focus(request, project_id):
    project = get_object_or_404(Project, pk=int(project_id))
    tasks = project.tasks.all()
    tasks_simplicity_data = ",".join(["%d" % (10*task.simplicity) for task in tasks])
    tasks_value_data = ",".join(["%d" % (10*task.value) for task in tasks])
    tasks_label_data = "|".join(["%s,FF0000,0,%d,12" % (task.name, idx) for idx, task in enumerate(tasks)])
    simplicity_scale = "|".join([label for value, label in SIMPLICITY_CHOICES])
    value_scale = "|".join([label for value, label in VALUE_CHOICES])
    return render_to_response("tasks_focus.html", locals())

```

```

views.py
from django.http import HttpResponse
from django.shortcuts import get_object_or_404, render_to_response
from models import *

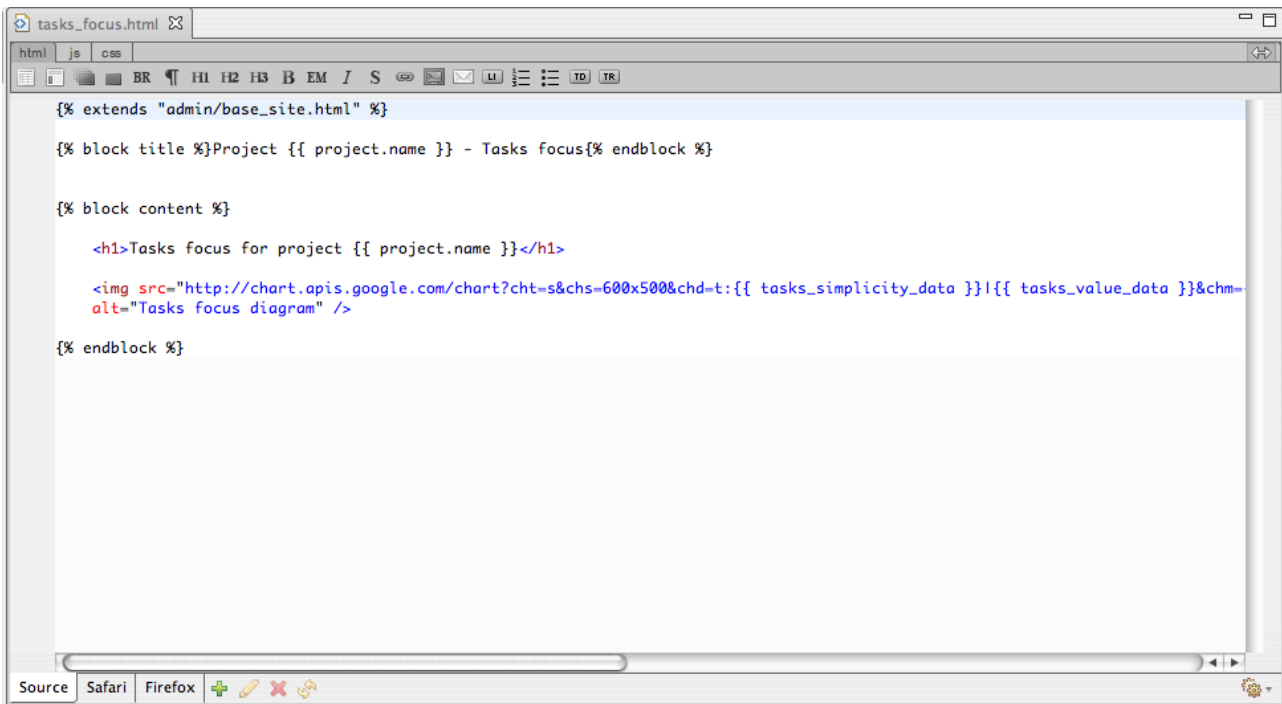
def tasks_focus(request, project_id):
    project = get_object_or_404(Project, pk=int(project_id))
    tasks = project.tasks.all()
    tasks_simplicity_data = ",".join(["%d" % (10*task.simplicity) for task in tasks])
    tasks_value_data = ",".join(["%d" % (10*task.value) for task in tasks])
    tasks_label_data = "|".join(["%s,FF0000,0,%d,12" % (task.name, idx) for idx, task in enumerate(tasks)])
    simplicity_scale = "|".join([label for value, label in SIMPLICITY_CHOICES])
    value_scale = "|".join([label for value, label in VALUE_CHOICES])
    return render_to_response("tasks_focus.html", locals())

```

And use these variables in the template:

```

```



The screenshot shows a web browser window with the address bar displaying 'tasks_focus.html'. The browser's developer tools are open, showing the source code of the page. The code is a Django template that extends 'admin/base_site.html'. It defines a block title as 'Project {{ project.name }} - Tasks focus' and a content block. The content block contains an h1 tag for the project name and an img tag that uses the same Google Chart API as shown in the previous code block. The browser's status bar at the bottom shows 'Source' and navigation icons for Safari and Firefox.

Add a link to this page from the list of projects

Open the admin.py file under the tasks folder

Find the list_display variable inside the ProjectAdmin class

Add a column called: tasks

```
list_display = ('name', 'is_active', 'leader', 'priority', 'health_indicator', 'tasks', 'last_update')
```

Add a function that will render the column for each row

```
def tasks(self, obj):
    return """
        <a href="/admin/tasks/task/?project=%d">List</a> |
        <a href="/tasks/project/%d/focus/">Focus diagram</a>
    """ % (obj.id, obj.id)
```

```

admin.py
from django.contrib import admin
from models import *

class TaskInline(admin.TabularInline):
    model = Task
    extra = 5

class ProjectAdmin(admin.ModelAdmin):
    list_display = ('name', 'is_active', 'leader', 'priority', 'health_indicator', 'tasks', 'last_update')
    search_fields = ('name', 'description')
    list_filter = ('is_active', 'leader', 'priority')
    ordering = ('priority',)
    inlines = (TaskInline,)

    def health_indicator(self, obj):
        tasks_status = {"done": 0, "on-track": 0, "late": 0}
        colors = {"done": "green", "on-track": "silver", "late": "red"}
        for task in obj.tasks.all():
            if task.done:
                tasks_status["done"] = tasks_status["done"] + 1
            elif task.is_on_track():
                tasks_status["on-track"] = tasks_status["on-track"] + 1
            else:
                tasks_status["late"] = tasks_status["late"] + 1
        return ", ".join(["<span style='background-color: %s; color: white;'>%d</span>" % (colors[status], count, status)
                          for status, count in tasks_status.items() if count > 0])

    def tasks(self, obj):
        return "<a href='/admin/tasks/task/?project=%d'>List</a> | <a href='/tasks/project/%d/focus/'>Focus diagram</a>" % (obj.id, obj.id)

    health_indicator.allow_tags = True
    tasks.allow_tags = True

class TaskAdmin(admin.ModelAdmin):
    list_display = ('name', 'project', 'done', 'assigned_to', 'due_in', 'simplicity', 'value', 'last_update')
    search_fields = ('name', 'description')
    list_filter = ('done', 'project', 'assigned_to',)
    date_hierarchy = 'due_in'

admin.site.register(Project, ProjectAdmin)
admin.site.register(Task, TaskAdmin)

```

""" in Python allows you to add a String value that spans across multiple lines
 Configure this task to enable HTML tags:
 tasks.allow_tags = True
 Point your browser to the projects list page to see the result:
<http://localhost:8000/admin/tasks/project/>

Changing the data model

If you'd like to modify the data model you defined in this tutorial, you'll need to run the syncdb command again
 However, if you already have data, you'll need to run a reset command first
 python manage.py tasks
 where tasks is the name of the pluggable app whose schema you wish to change
 This will remove all data from the tables of this pluggable app
 If you don't want to lose your data, you can use the fixtures mechanism (used for unit-testint in Django) to migrate the old data to the new schema
 This assumes you only added new fields, who aren't mandatory
 If this isn't the case, you'll need to add a pluggable app for data model evolution, such as south or django-evolution
 This apps automatically migrate your data when you change your data model
 Create a folder under the tasks folder, named fixtures
 Before you do the data model change, dump the pluggable app data, by running the command:
 python manage.py dumpdata --format=xml tasks >tasks/fixtures/initial_data.xml
 you can use some other format, such as json, if you prefer to
 Do the data model change, then reset & syncdb:
 python manage.py reset tasks
 python manage.py syncdb
 The syncdb command will detect the data dump in tasks/fixtures/initial_data.xml, & will load it into the new schema

Summary

So, you've seen what it takes to create a useful enterprise information system using Django. If you indeed followed all steps, & nothing went wrong in your code/environment, you should have the app running on your machine

Note that apart from using a development server, which isn't a production server, this information system is production/end-users ready, not some uncomplete demo/prototype
In a sequel tutorial I intend to show how to deploy & evolve your information system

- Add localization

 - Translate the application UI to another language

- Add unit-tests, which are extremely easy in Django

- Deploy to either local Apache server, or to cloud computing vendors

 - Google AppEngine

 - Amazon EC2

Note that you can take the tasks Pluggable App that we've built, & add it to any other Django projects, & you'll seamlessly get the Project Management functionality inside that other project.