CHAPTER 4

Building a REST API

It's been over a decade since Roy Fielding, an American computer scientist and one of the principal authors of the HTTP specification, introduced REpresentational State Transfer (REST) as an architectural style. Over the years, REST has gained momentum thanks to its popularity for building web services.

By being stateless, and creating unique identifiers for allowing caching, layering, and scalability, REST APIs make use of existing HTTP verbs (GET, POST, PUT, and DELETE) to create, update, and remove new resources. The term REST is often abused to describe any URL that returns JSON instead of HTML. What most users do not realize is that to be a RESTful architecture the web service must satisfy formal constraints. In particular, the application must be separated into a client-server model and the server must remain completely stateless. No client context may be stored on the server and resources should also be uniquely and uniformly identified. The client also should be able to navigate the API and transition state through the use of links and metadata in the resource responses. The client should not assume the existence of resources or actions other than a few fixed entry points, such as the root of the API.

In this chapter, we'll walk through how to utilize the power of RESTful architecture with Django.

Django and REST

Combining Django with REST is a common practice for creating data-rich websites. There are numerous reusable applications in the Django community to help you apply REST's strict principles when building an API. In recent years the two most popular are `django-tastypie` and `django-rest-framework`. Both have support for creating resources from ORM and non-ORM data, pluggable authentication and permissions, and support for a variety of serialization methods, including JSON, XML, YAML, and HTML.

Additional packages with comparisons can be found here: *https://www.djangopackages.com/grids/g/api/*.

In this chapter we will be leveraging `django-rest-framework` to help build our API architecture. One of the best features of this Django reusable application is support for creating self-documenting and web-browsable APIs.

Filter translation from the query parameters to the ORM isn't built into `django-rest-framework` like it is for `django-tastypie`. However, `django-rest-framework` has pluggable filter backends, and it is easy to integrate with `django-filter` to provide that feature.

We'll start our project by installing all of the necessary dependencies. `Markdown` is used by the browsable API views to translate the docstrings into pages for help when users are viewing the API data.

```
hostname $ pip install djangorestframework Markdown django-filter
```

Now that we have our requirements installed, let's focus on modeling a REST API for a task board.

Scrum Board Data Map

Data modeling is an important first step to any project. Let's take a moment to list all of the points we'll need to consider when creating our models:

- A task board is commonly used in Scrum-style development to manage tasks in the current sprint.
- Tasks are moved from the backlog into the set of tasks to be completed.
- Various states can be used to track the progress along the way, such as "in progress" or "in testing."
- The preceding tasks must be part of a task for it to be in a "completed" state.

One way for visual learners to start understanding how the data interacts is through a visual map, as shown in Figure 4-1.

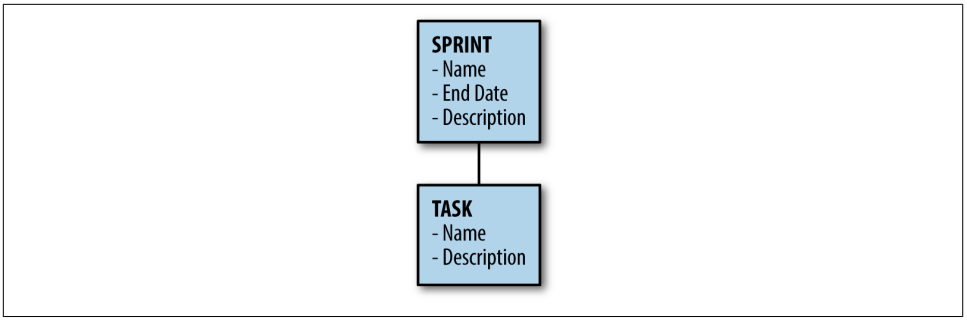

Figure 4-1. Scrum board data map

As you can see, there are multiple steps we'll need to consider when laying out our project. With the aforementioned definitions in place, we can begin laying out the initial project structure.

Initial Project Layout

We'll begin creating our initial project layout by running the base `startproject` command. While running the command, we'll also be passing in a project named `scrum` and a single app named `board`.

```
hostname $ django-admin.py startproject scrum
hostname $ cd scrum
hostname $ python manage.py startapp board
```

Your project's folder structure will be as follows:

```
scrum/
    manage.py
    scrum/
        __init__.py
        settings.py
        urls.py
        wsgi.py
    board/
        migrations/
            __init__.py
        __init__.py
        admin.py
        models.py
        tests.py
        views.py
```

The `startapp` template has evolved in recent Django versions, and the output might be different if you are using a different version of Django. *admin.py* was added to the template in Django 1.6 and the *migrations* folder was added in 1.7.

Let's move on to configuring our *settings.py* file to work with `django-rest-framework` and its inherited settings.

Project Settings

When creating this new Django project, we need to update the default project settings (in *settings.py* in the *scrum* folder) to incorporate `django-rest-framework`, as well as to reduce the defaults from the previous chapters. Also, since the server will not maintain any client state, the `contrib.session` references can be removed. This will break usage of the default Django admin, which means those references can be removed as well.

```
...
INSTALLED_APPS = (                                          ❶
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.staticfiles',
    # Third party apps
    'rest_framework',
    'rest_framework.authtoken',
    # Internal apps
    'board',
)
...
MIDDLEWARE_CLASSES = (                                      ❷
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
)
...
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'scrum',
    }
}
...
```

❶ These changes remove `django.contrib.admin`, `django.contrib.sessions`, and `django.contrib.messages` from the `INSTALLED_APPS`. The new `INSTALLED_APPS` include `rest_framework` and `rest_framework.authtoken` as well as the *`board`* app, which will be created in this chapter.

❷ `django.contrib.sessions.middleware.SessionMiddleware`, `django.contrib.auth.middleware.AuthenticationMiddleware`, `django.contrib.auth.middleware.SessionAuthenticationMiddleware`, and `django.contrib.messages.middleware.MessageMiddleware` are part of the defaults for `startproject` but have been removed from the `MIDDLEWARE_CLASSES` since these applications are no longer installed.

Instead of the default SQLite database, this project will use PostgreSQL. This change requires installing `psycopg2`, the Python driver for PostgreSQL. We'll need to create the database as well.

```
hostname $ pip install psycopg2
hostname $ createdb -E UTF-8 scrum
```

These database settings assume PostgreSQL is listening on the default Unix socket and allows ident or trust authentication for the current user. These settings may need to be adjusted for your server configuration. See Django's documentation for information on these settings.

With the Django admin removed from `INSTALLED_APPS`, the references can also be removed from the *scrum/urls.py* file.

If you are an OS X user, we recommend installing Postgres via Homebrew, a package manager specifically for OS X. To learn more and find instructions on how to install Homebrew, see *http://brew.sh/*.

```
from django.conf.urls import include, url

from rest_framework.authtoken.views import obtain_auth_token


urlpatterns = [
    url(r'^api/token/', obtain_auth_token, name='api-token'),
]
```

As you can see, all of the existing patterns have been removed and replaced with a single URL for `rest_framework.authtoken.views.obtain_auth_token`. This serves as the view for exchanging a username and password combination for an API token. We'll also be using the `include` import in the next steps of the project.

No Django Admin?

Some Django developers or users may have a hard time imagining administering a Django project without the Django admin. In this application it has been removed because the API doesn't use or need Django's session management, which is required to use the admin. The browsable API, which will be provided by the `django-rest-framework`, can serve as a simple replacement for the admin.

You may find that you still need to maintain the Django admin due to other applications that rely on it. If that's the case, you can preserve the admin and simply have two sets of settings for the project: one for the API and one for the admin. The admin settings would restore the session and messages apps along with the authentication middleware. You would also need another set of root URLs that include the admin URLs. With these configurations, some servers would serve the admin part of the site and others would serve the API. Given that in a larger application these will likely have very different scaling needs and concerns, this serves as a clean approach to achieving Django admin access.

Models

With the basic structure of the project in place, we can now start on the data modeling portion of our application. At the top level, tasks will be broken up into sprints, which will have an optional name and description, and a unique end date. Add these models to the *board/models.py* file:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _


class Sprint(models.Model):
    """Development iteration period."""

    name = models.CharField(max_length=100, blank=True, default='')
    description = models.TextField(blank=True, default='')
    end = models.DateField(unique=True)

    def __str__(self):
        return self.name or _('Sprint ending %s') % self.end
```

While this book is written for Python 3.3+, Django provides a `python_2_unicode_compatible` decorator that could be used to make this model compatible with both Python 2.7 and Python 3. More information on Python 3 porting and compatibility tips can be found in Django's documentation.

We will also need to build a task model for the list of tasks within a given sprint. Tasks have a name, optional description, association with a sprint (or a backlog in which they're stored), and a user assigned to them, and include a start, end, and due date.

We should also note that tasks will have a handful of states:

- Not Started
- In Progress
- Testing
- Done

Let's add this list of `STATUS_CHOICES` into our data models (*board/models.py*):

```
from django.conf import settings
from django.db import models
from django.utils.translation import ugettext_lazy as _

...
class Task(models.Model):
    """Unit of work to be done for the sprint."""

    STATUS_TODO = 1
    STATUS_IN_PROGRESS = 2
    STATUS_TESTING = 3
    STATUS_DONE = 4

    STATUS_CHOICES = (
        (STATUS_TODO, _('Not Started')),
        (STATUS_IN_PROGRESS, _('In Progress')),
        (STATUS_TESTING, _('Testing')),
        (STATUS_DONE, _('Done')),
    )

    name = models.CharField(max_length=100)
    description = models.TextField(blank=True, default='')
    sprint = models.ForeignKey(Sprint, blank=True, null=True)
    status = models.SmallIntegerField(choices=STATUS_CHOICES, default=STATUS_TODO)
    order = models.SmallIntegerField(default=0)
    assigned = models.ForeignKey(settings.AUTH_USER_MODEL, null=True, blank=True)
    started = models.DateField(blank=True, null=True)
    due = models.DateField(blank=True, null=True)
    completed = models.DateField(blank=True, null=True)

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

The user reference uses `settings.AUTH_USER_MODEL` to allow support for swapping out the default `User` model. While this project will use the default `User`, the `board` app is designed to be as reusable as possible. More information on customizing and referencing the `User` model can be found at *https://docs.djangoproject.com/en/1.7/topics/auth/customizing/#referencing-the-user-model*.

These are the two models needed for the project, but you can see that there are some clear limitations in this data model. One issue is that this tracks sprints only for a single project and assumes that all system users are involved with the project. The task states are also fixed in the task model, making those states unusable in the sprint model. There is also no support here for customizing the task workflow.

These limitations are likely acceptable if the application will be used for a single project, but obviously would not work if the intention were to build this task board as a software as a service (SaaS) product.

Since we've already written our models, you'll need to run the `makemigrations` and `migrate` commands to have Django update the database properly.

```
hostname $ python manage.py makemigrations board
Migrations for 'board':
  0001_initial.py:
    - Create model Sprint
    - Create model Task
hostname $ python manage.py migrate
Operations to perform:
  Synchronize unmigrated apps: rest_framework, authtoken
  Apply all migrations: contenttypes, auth, board
Synchronizing apps without migrations:
  Creating tables...
    Creating table authtoken_token
  Installing custom SQL...
  Installing indexes...
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying board.0001_initial... OK
```

Now we'll create a superuser, with the username of your choosing, using the `createsuperuser` command. For use in future examples, we will assume you create the user with the username *demo* and the password *test*.

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

Prior to Django 1.7 there was no built-in support for migrated model schemas. Django could create tables only using a single command called `syncdb`. If you are using a version of Django prior to 1.7, you should run `syncdb` instead of the aforementioned commands. `syncdb` will also prompt you to create a superuser and you should create one as outlined in the preceding instructions.

Designing the API

With the models in place, our focus can now shift to the API itself. As far as the URL structure, the API we want to build will look like this:

```
/api/
    /sprints/
        /<id>/
    /tasks/
        /<id>/
    /users/
        /<username>/
```

It's important to consider how clients will navigate this API. Clients will be able to issue a GET request to the API root */api/* to see the sections below it. From there the client will be able to list sprints, tasks, and users, as well as create new ones. As the client dives deeper into viewing a particular sprint, it will be able to see the tasks associated with the sprint. The hypermedia links between the resources enable the client to navigate the API.

We've chosen not to include the version number in the URLs for the API. While it is included in many popular APIs, we feel that the approach for versioning that aligns best with RESTful practices is to use different content types for different API versions. However, if you really want a version number in your API, feel free to add it.

Sprint Endpoints

Building resources tied to Django models with `django-rest-framework` is easy with ViewSets. To build the ViewSet for the */api/sprints/*, we should describe how the model should be serialized and deserialized by the API. This is handled by serializers and created in *board/serializers.py*.

```
from rest_framework import serializers

from .models import Sprint


class SprintSerializer(serializers.ModelSerializer):

    class Meta:
```

```
        model = Sprint
        fields = ('id', 'name', 'description', 'end', )
```

In this simple case, all of the fields are exposed via the API.

Let's create the ViewSet in *board/views.py*.

```
from rest_framework import viewsets

from .models import Sprint
from .serializers import SprintSerializer


class SprintViewSet(viewsets.ModelViewSet):
    """API endpoint for listing and creating sprints."""

    queryset = Sprint.objects.order_by('end')
    serializer_class = SprintSerializer
```

As you can see, the `ModelViewSet` provides the scaffolding needed for the create, read, update, delete (CRUD) operations using the corresponding HTTP verbs. The defaults for authentication, permissions, pagination, and filtering are controlled by the `REST_FRAMEWORK` settings dictionary if not set on the view itself.

The available settings and their defaults are all described in the Django REST framework documentation.

This view will be explicit about its settings. Since the remaining views will need these defaults as well, they will be implemented as a mixin class in *board/views.py*.

```
from rest_framework import authentication, permissions, viewsets

from .models import Sprint
from .serializers import SprintSerializer


class DefaultsMixin(object):                                          ❶
    """Default settings for view authentication, permissions,
    filtering and pagination."""

    authentication_classes = (
        authentication.BasicAuthentication,
        authentication.TokenAuthentication,
    )
    permission_classes = (
        permissions.IsAuthenticated,
    )
    paginate_by = 25
```

```
    paginate_by_param = 'page_size'
    max_paginate_by = 100


class SprintViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating sprints."""

    queryset = Sprint.objects.order_by('end')
    serializer_class = SprintSerializer
```

❶ `DefaultsMixin` will be one of the base classes for the API view classes to define these options.

Now let's add in some authentication for the user permissions. The authentication will use either HTTP basic auth or a token-based authentication. Using basic auth will make it easy to use the browsable API via a web browser. This example does not have fine-grained permissions, since we are working on the assumption that all users in the system are a part of the project. The only permission requirement is that the user is authenticated.

Task and User Endpoints

We need the tasks to be exposed on their own endpoint. Similar to our sprint endpoints, we will start with a serializer and then a ViewSet. First we'll create the serializer in *board/serializers.py*.

```
from rest_framework import serializers

from .models import Sprint, Task

...
class TaskSerializer(serializers.ModelSerializer):

    class Meta:
        model = Task
        fields = ('id', 'name', 'description', 'sprint', 'status', 'order',
            'assigned', 'started', 'due', 'completed', )
```

While this looks fine at a glance, there are some problems with writing the serializer this way. The `status` will show the number rather than the text associated with its state. We can easily address this by adding another field `status_display` to *board/serializers.py* that shows the status text.

```
from rest_framework import serializers

from .models import Sprint, Task

...
class TaskSerializer(serializers.ModelSerializer):
```

```
    status_display = serializers.SerializerMethodField('get_status_display')    ❶

    class Meta:
        model = Task
        fields = ('id', 'name', 'description', 'sprint',
            'status', 'status_display', 'order',
            'assigned', 'started', 'due', 'completed', )

    def get_status_display(self, obj):
        return obj.get_status_display()
```

❶ `status_display` is a read-only field to be serialized that returns the value of the `get_status_display` method on the serializer.

The second issue with our serializer is that `assigned` is a foreign key to the `User` model. This displays the user's primary key, but our URL structure expects to reference users by their username. We can address this by using the `SlugRelatedField` in *board/serializers.py*.

```
...
class TaskSerializer(serializers.ModelSerializer):

    assigned = serializers.SlugRelatedField(
        slug_field=User.USERNAME_FIELD, required=False)
    status_display = serializers.SerializerMethodField('get_status_display')

    class Meta:
        model = Task
        fields = ('id', 'name', 'description', 'sprint',
            'status', 'status_display', 'order',
            'assigned', 'started', 'due', 'completed', )

    def get_status_display(self, obj):
        return obj.get_status_display()
```

Finally, we need to create a serializer for the `User` model. Now, let's take a moment to remember that the `User` model might be swapped out for another and that the intent of our application is to make it as reusable as possible. We will need to use the `get_user_model` Django utility in *board/serializers.py* to create this switch in a clean way.

```
from django.contrib.auth import get_user_model

from rest_framework import serializers

from .models import Sprint, Task

User = get_user_model()

...
class UserSerializer(serializers.ModelSerializer):
```

```
    full_name = serializers.CharField(source='get_full_name', read_only=True)

    class Meta:
        model = User
        fields = ('id', User.USERNAME_FIELD, 'full_name', 'is_active', )
```

This serializer assumes that if a custom `User` model is used, then it extends from `django.contrib.auth.models.CustomUser`, which will always have a `USERNAME_FIELD` attribute, `get_full_name` method, and `is_active` attribute. Also note that since `get_full_name` is a method, the field in the serializer is marked as read-only.

With the serializers in place, the ViewSets for the tasks and users can be created in *board/views.py*.

```
from django.contrib.auth import get_user_model

from rest_framework import authentication, permissions, viewsets

from .models import Sprint, Task
from .serializers import SprintSerializer, TaskSerializer, UserSerializer


User = get_user_model()

...
class TaskViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating tasks."""

    queryset = Task.objects.all()
    serializer_class = TaskSerializer


class UserViewSet(DefaultsMixin, viewsets.ReadOnlyModelViewSet):
    """API endpoint for listing users."""

    lookup_field = User.USERNAME_FIELD
    lookup_url_kwarg = User.USERNAME_FIELD
    queryset = User.objects.order_by(User.USERNAME_FIELD)
    serializer_class = UserSerializer
```

Both are very similar to the `SprintViewSet`, but there is a notable difference in the `UserViewSet` in that it extends from `ReadOnlyModelViewSet` instead. As the name implies, this does not expose the actions to create new users or to edit existing ones through the API. `UserViewSet` changes the lookup from using the ID of the user to the username by setting `lookup_field`. Note that `lookup_url_kwarg` has also been changed for consistency.

Connecting to the Router

At this point the `board` app has the basic data model and view logic in place, but it has not been connected to the URL routing system. `django-rest-framework` has its own URL routing extension for handling ViewSets, where each ViewSet is registered with the router for a given URL prefix. This will be added to a new file in *board/urls.py*.

```
from rest_framework.routers import DefaultRouter

from . import views


router = DefaultRouter()
router.register(r'sprints', views.SprintViewSet)
router.register(r'tasks', views.TaskViewSet)
router.register(r'users', views.UserViewSet)
```

Finally, this router needs to be included in the root URL configuration in *scrum/urls.py*.

```
from django.conf.urls import include, url

from rest_framework.authtoken.views import obtain_auth_token

from board.urls import router


urlpatterns = [
    url(r'^api/token/', obtain_auth_token, name='api-token'),
    url(r'^api/', include(router.urls)),
]
```

We have the basic model, view, and URL structure of our Scrum board application, and can now create the remainder of our RESTful application.

Linking Resources

One important constraint of a RESTful application is hypermedia as the engine of application state (HATEOAS). With this constraint, the RESTful client should be able to interact with the application through the hypermedia responses generated by the server; that is, the client should be aware only of few fixed endpoints to the server. From those fixed endpoints, the client should discover the resources available on the server through the use of descriptive resource messages. The client must be able to interpret the server responses and separate the resource data from metadata, such as links to related resources.

How does this translate to our current resources? What useful links could the server provide between these resources? To start, each resource should know its own URL. Sprints should also provide the URL for their related tasks and backlog. Tasks that have been assigned to a user should provide a link to the user resource. Users should provide

a way to get all tasks assigned to that user. With these in place, the API client should be able to answer the most common questions while navigating the API.

To give clients a uniform place to look for these links, each resource will have a `links` section in the response containing the relevant links. The easiest way to start is for the resources to link back to themselves, as shown here in *board/serializers.py*.

```
from django.contrib.auth import get_user_model

from rest_framework import serializers
from rest_framework.reverse import reverse                              ❶

from .models import Sprint, Task


User = get_user_model()


class SprintSerializer(serializers.ModelSerializer):

    links = serializers.SerializerMethodField('get_links')              ❷

    class Meta:
        model = Sprint
        fields = ('id', 'name', 'description', 'end', 'links', )

    def get_links(self, obj):                                           ❸
        request = self.context['request']
        return {
            'self': reverse('sprint-detail',
                kwargs={'pk': obj.pk},request=request),
        }


class TaskSerializer(serializers.ModelSerializer):

    assigned = serializers.SlugRelatedField(
        slug_field=User.USERNAME_FIELD, required=False)
    status_display = serializers.SerializerMethodField('get_status_display')
    links = serializers.SerializerMethodField('get_links')              ❹

    class Meta:
        model = Task
        fields = ('id', 'name', 'description', 'sprint',
            'status', 'status_display', 'order',
            'assigned', 'started', 'due', 'completed', 'links', )

    def get_status_display(self, obj):
        return obj.get_status_display()

    def get_links(self, obj):                                           ❺
        request = self.context['request']
```

```
            return {
                'self': reverse('task-detail',
                    kwargs={'pk': obj.pk}, request=request),
            }


    class UserSerializer(serializers.ModelSerializer):

        full_name = serializers.CharField(source='get_full_name', read_only=True)
        links = serializers.SerializerMethodField('get_links')                    ❻

        class Meta:
            model = User
            fields = ('id', User.USERNAME_FIELD, 'full_name',
                'is_active', 'links', )

        def get_links(self, obj):                                                 ❼
            request = self.context['request']
            username = obj.get_username()
            return {
                'self': reverse('user-detail',
                    kwargs={User.USERNAME_FIELD: username}, request=request),
            }
```

❶ This is a new import for `rest_framework.reverse.reverse`.

❷ ❹ ❻ Each serializer has a new read-only `links` field for the response body.

❸ ❺ ❼ To populate the `links` value, each serializer has a `get_links` method to build the related links.

Each resource now has a new `links` field that returns the dictionary returned by the `get_links` method. For now there is only a single key in the dictionary, called `"self"`, which links to the details for that resource. `get_links` doesn't use the standard `reverse` from Django, but rather a modification that is built into `django-rest-framework`. Unlike Django's `reverse`, this will return the full URI, including the hostname and protocol, along with the path. For this, `reverse` needs the current request, which is passed into the serializer context by default when we're using the standard ViewSets.

A task assigned to a sprint should point back to its sprint. You can also link from a task to its assigned user by reversing the URL if there is a user assigned, as shown here in *board/serializers.py*.

```
    ...
    class TaskSerializer(serializers.ModelSerializer):
    ...
        def get_links(self, obj):
            request = self.context['request']
            links = {
                'self': reverse('task-detail',
```

```
                kwargs={'pk': obj.pk}, request=request),
            'sprint': None,
            'assigned': None
        }
        if obj.sprint_id:
            links['sprint'] = reverse('sprint-detail',
                kwargs={'pk': obj.sprint_id}, request=request)
        if obj.assigned:
            links['assigned'] = reverse('user-detail',
                kwargs={User.USERNAME_FIELD: obj.assigned}, request=request)
        return links
```

Linking from a sprint or user to the related tasks will require filtering, which will be added in a later section.

Testing Out the API

With the models and views in place for the API, it is time to test the API. We'll explore this first by using the browser, then through the Python shell.

Using the Browsable API

Having a visual tool and creating a browsable API is one of the best features of `django-rest-framework`. While you may choose to disable it in your production system, it is a very powerful way to explore the API. In particular, using the browsable API helps you think about how a client will navigate through the API by following links given in the responses.

The client should be able to enter the API through a fixed endpoint and explore the API through the resource representations returned by the server. The first step is to get the development server running using `runserver`:

```
hostname $ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
June 11, 2014 - 00:46:41
Django version 1.7, using settings 'scrum.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

You should now be able to visit *http://127.0.0.1:8000/api/* in your favorite browser to view the API, as seen in Figure 4-2.

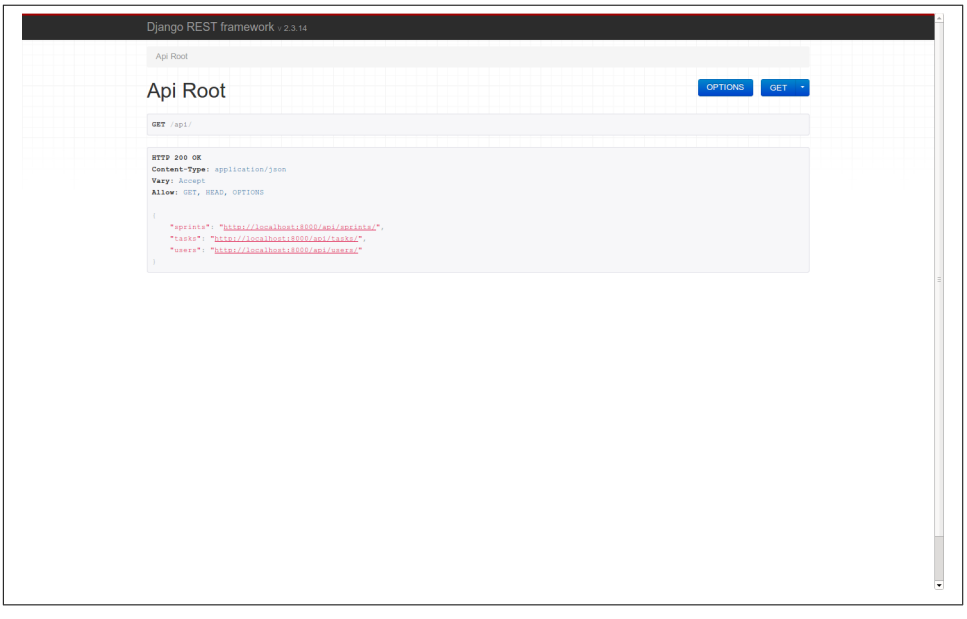

Figure 4-2. django-rest-framework API home page

As you can see, this serves as a nice visual tool for viewing our newly made API. It shows the response from issuing a GET to */api/*. It's a simple response that shows the available top-level resources defined by the API. While the API root doesn't require authentication, all of the subresources do; when you click a link, the browser will prompt you for a username and password for HTTP auth. You can use the username and password created previously.

```
HTTP 200 OK
Vary: Accept
Content-Type: application/json
Allow: GET, HEAD, OPTIONS

{
    "sprints": "http://localhost:8000/api/sprints/",
    "tasks": "http://localhost:8000/api/tasks/",
    "users": "http://localhost:8000/api/users/"
}
```

Clicking on the link for sprints will show the resource that lists all the available sprints. At this point none have been created, so the response is empty. However, at the bottom of the page is a form that allows for creating a new sprint. Create a new sprint with the name "Something Sprint," description "Test," and end date of any date in the future. The

date should be given in ISO format, YYYY-MM-DD. Clicking the POST button shows the successful response with the 201 status code.

```
HTTP 201 CREATED
Vary: Accept
Content-Type: application/json
Location: http://localhost:8000/api/sprints/1/
Allow: GET, POST, HEAD, OPTIONS

{
    "id": 1,
    "name": "Something Sprint",
    "description": "Test",
    "end": "2020-12-31",
    "links": {
        "self": "http://localhost:8000/api/sprints/1/"
    }
}
```

Refreshing the sprint list shows the sprint in the results, as illustrated in Figure 4-3.

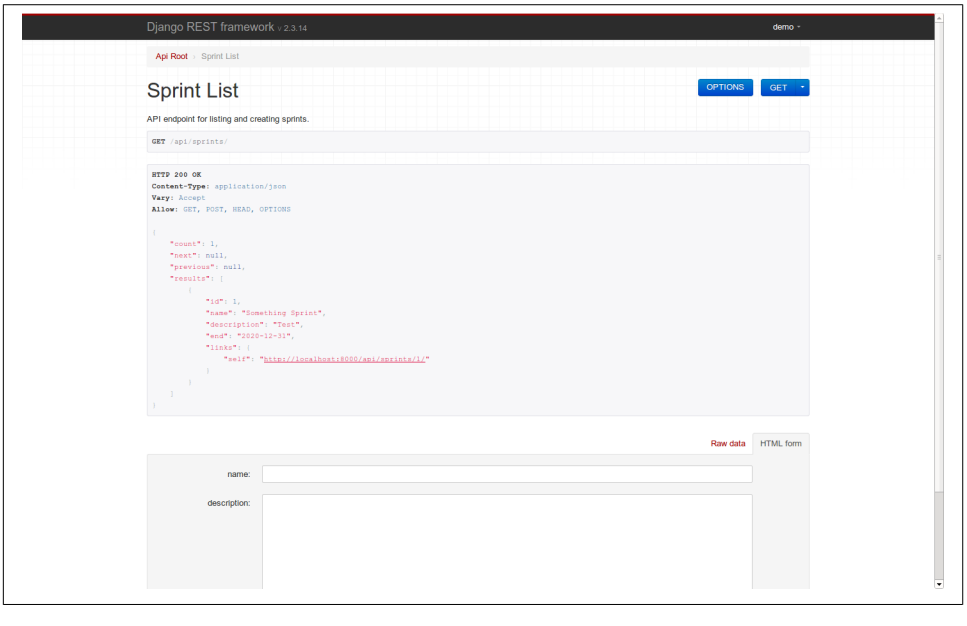

Figure 4-3. Sprints API screenshot

Navigating back to the API root at */api/*, we can now explore the tasks. Similarly to the sprints, clicking the tasks link from the API root shows a listing of all tasks, but there are none yet. Using the form at the bottom, we can create a task. For this task we will use the name "First Task" and the sprint will be the previously created sprint. Again, the API gives a 201 status with the details of the newly created task.

```
HTTP 201 CREATED
Vary: Accept
Content-Type: application/json
Location: http://localhost:8000/api/tasks/1/
Allow: GET, POST, HEAD, OPTIONS

{
    "id": 1,
    "name": "First Task",
    "description": "",
    "sprint": 1,
    "status": 1,
    "status_display": "Not Started"
    "order": 0,
    "assigned": null,
    "started": null,
    "due": null,
    "completed": null,
    "links": {
        "assigned": null,
        "self": "http://localhost:8000/api/tasks/1/",
        "sprint": "http://localhost:8000/api/sprints/1/"
    }
}
```

Refreshing the task list shows the task in the results, as seen in Figure 4-4.

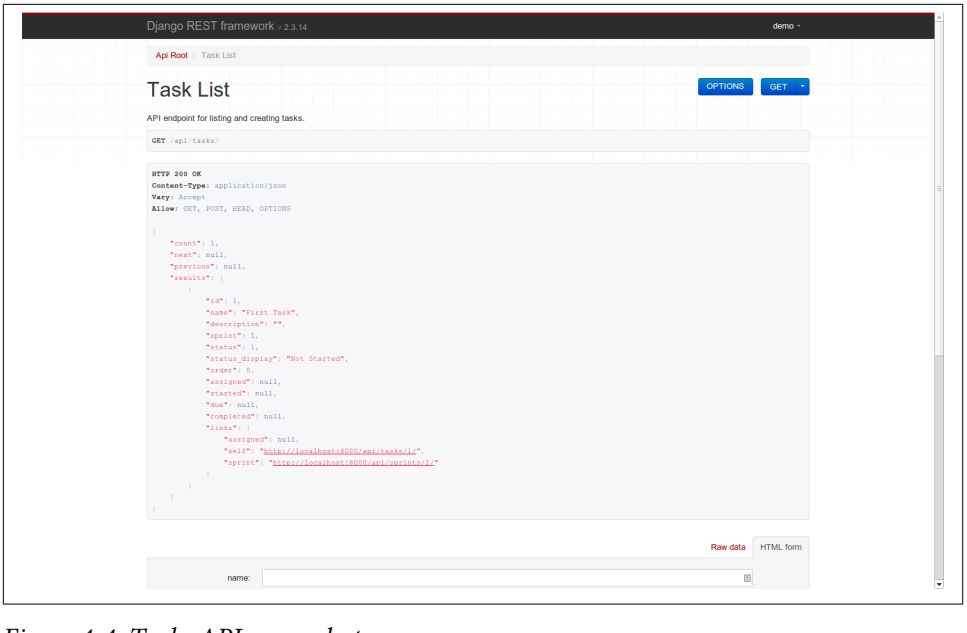

Figure 4-4. Tasks API screenshot

With our basic infrastructure in place for our browsable API, let's add some filtering to further organize the structured page data.

Adding Filtering

As previously mentioned, `django-rest-framework` has support for using various filter backends. We can enable these in all resources by adding them to the `DefaultsMixin` in *board/views.py*.

```
...
from rest_framework import authentication, permissions, viewsets, filters    ❶
...
class DefaultsMixin(object):
    """Default settings for view authentication, permissions,
    filtering and pagination."""

    authentication_classes = (
        authentication.BasicAuthentication,
        authentication.TokenAuthentication,
    )
    permission_classes = (
        permissions.IsAuthenticated,
    )
    paginate_by = 25
    paginate_by_param = 'page_size'
    max_paginate_by = 100
    filter_backends = (                                                      ❷
        filters.DjangoFilterBackend,
        filters.SearchFilter,
        filters.OrderingFilter,
    )
...
```

❶ This adds `filters` to the import list.

❷ `DefaultsMixin` now defines the list of available `filter_backends`, which will enable these for all of the existing ViewSets.

We configure the `SearchFilter` by adding a `search_fields` attribute to each **`ViewSet`**. We configure the `OrderingFilter` by adding a list of fields, which can be used for ordering the `ordering_fields`. This snippet from *board/views.py* demonstrates.

```
...
class SprintViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating sprints."""

    queryset = Sprint.objects.order_by('end')
    serializer_class = SprintSerializer
    search_fields = ('name', )                                               ❶
    ordering_fields = ('end', 'name', )                                      ❷
```

```
class TaskViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating tasks."""

    queryset = Task.objects.all()
    serializer_class = TaskSerializer
    search_fields = ('name', 'description', )                          ❸
    ordering_fields = ('name', 'order', 'started', 'due', 'completed', )  ❹


class UserViewSet(DefaultsMixin, viewsets.ReadOnlyModelViewSet):
    """API endpoint for listing users."""

    lookup_field = User.USERNAME_FIELD
    lookup_url_kwarg = User.USERNAME_FIELD
    queryset = User.objects.order_by(User.USERNAME_FIELD)
    serializer_class = UserSerializer
    search_fields = (User.USERNAME_FIELD, )                            ❺
```

❶ ❸ ❺ `search_fields` are added to all of the views to allow searching on the given list of fields.

❷ ❹ Sprints and tasks are made orderable in the API using the `ordering_fields`. Users will always be ordered by their username in the API response.

Since there is only one task with the name "First Task," searching for "foo" via *http://localhost:8000/api/tasks/?search=foo* will yield no results, while seaching for "first" with *http://localhost:8000/api/tasks/?search=first* will.

To handle additional filtering of the task, we can make use of the `DjangoFilterBackend`. This requires defining a `filter_class` on the `TaskViewSet`. The `filter_class` attribute should be a subclass of `django_filters.FilterSet`. This will be added in a new file, *board/forms.py*.

```
import django_filters

from .models import Task


class TaskFilter(django_filters.FilterSet):

    class Meta:
        model = Task
        fields = ('sprint', 'status', 'assigned', )
```

This is the most basic use of `django-filter`; it builds the filter set based on the model definition. Each field defined in `TaskFilter` will translate into a query parameter, which the client can use to filter the result set. First, this must be associated with the `TaskViewSet` in *board/views.py*.

```
...
from .forms import TaskFilter                                          ❶
from .models import Sprint, Task
from .serializers import SprintSerializer, TaskSerializer, UserSerializer

...
class TaskViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating tasks."""

    queryset = Task.objects.all()
    serializer_class = TaskSerializer
    filter_class = TaskFilter                                          ❷
    search_fields = ('name', 'description', )
    ordering_fields = ('name', 'order', 'started', 'due', 'completed', )
...
```

❶ ❷ Here the new `TaskFilter` class is imported and assigned to the `TaskViewSet.filter_class`.

With this in place, the client can filter on the `sprint`, `status`, or `assigned` user. However, the sprint for a task isn't required and this filter won't allow for selecting tasks without a sprint. In our current data model, a task that isn't currently assigned a sprint would be considered a backlog task. To handle this, we can add a new field to the `TaskFilter` in *board/forms.py*.

```
import django_filters

from .models import Task


class NullFilter(django_filters.BooleanFilter):
    """Filter on a field set as null or not."""

    def filter(self, qs, value):
        if value is not None:
            return qs.filter(**{'%s__isnull' % self.name: value})
        return qs


class TaskFilter(django_filters.FilterSet):

    backlog = NullFilter(name='sprint')

    class Meta:
        model = Task
        fields = ('sprint', 'status', 'assigned', 'backlog', )
```

This will make *http://localhost:8000/api/tasks/?backlog=True* return all tasks that aren't assigned to a sprint. Another issue with `TaskFilter` is that the users referenced by `assigned` are referenced by the `pk`, while the rest of the API uses the `username` as a

unique identifier. We can address this by changing the field used by the underlying `ModelChoiceField` in *board/forms.py*.

```
import django_filters

from django.contrib.auth import get_user_model                    ❶

from .models import Task


User = get_user_model()                                           ❷


class NullFilter(django_filters.BooleanFilter):
    """Filter on a field set as null or not."""

    def filter(self, qs, value):
        if value is not None:
            return qs.filter(**{'%s__isnull' % self.name: value})
        return qs


class TaskFilter(django_filters.FilterSet):

    backlog = NullFilter(name='sprint')

    class Meta:
        model = Task
        fields = ('sprint', 'status', 'assigned', 'backlog', )

    def __init__(self, *args, **kwargs):                          ❸
        super().__init__(*args, **kwargs)
        self.filters['assigned'].extra.update(
            {'to_field_name': User.USERNAME_FIELD})
```

❶ ❷ Fetch a reference to the installed `User` model as we've done in other modules.

❸ Update the `assigned` filter to use the `User.USERNAME_FIELD` as the field reference rather than the default `pk`.

With this change in place, the tasks assigned to the one and only *demo* user can now be retrived using *http://localhost:8000/api/tasks/?assigned=demo* rather than *http://localhost:8000/api/tasks/?assigned=1*.

Sprints could also use some more complex filtering. Clients might be interested in sprints that haven't ended yet or will end in some range. For this, we can create a `SprintFilter` in *board/forms.py*.

```
...
from .models import Task, Sprint
...
```

```
class SprintFilter(django_filters.FilterSet):

    end_min = django_filters.DateFilter(name='end', lookup_type='gte')
    end_max = django_filters.DateFilter(name='end', lookup_type='lte')

    class Meta:
        model = Sprint
        fields = ('end_min', 'end_max', )
```

And then we relate it to the `SprintViewSet` in a similar fashion in *board/views.py*.

```
...
from .forms import TaskFilter, SprintFilter
...

class SprintViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating sprints."""

    queryset = Sprint.objects.order_by('end')
    serializer_class = SprintSerializer
    filter_class = SprintFilter                                    ❶
    search_fields = ('name', )
    ordering_fields = ('end', 'name', )
```

❶ As with the `TaskViewSet`, the `SprintViewSet` now defines a `filter_class` attribute using the new `SprintFilter`.

http://localhost:8000/api/sprints/?end_min=2014-07-01 will show all sprints that ended after July 1, 2014, and *http://localhost:8000/api/sprints/?end_max=2014-08-01* will show all sprints that ended before August 1, 2014. These can be combined to limit sprints to a given date range.

Since the views for sprints and tasks support filtering, you can create links to link a sprint to its related tasks and users to their tasks by modifying *board/serializers.py*.

```
...

class SprintSerializer(serializers.ModelSerializer):
...
    def get_links(self, obj):
        request = self.context['request']
        return {
            'self': reverse('sprint-detail',
                kwargs={'pk': obj.pk}, request=request),
            'tasks': reverse('task-list',
                request=request) + '?sprint={}'.format(obj.pk),
        }
...
class UserSerializer(serializers.ModelSerializer):
...
    def get_links(self, obj):
```

```
        request = self.context['request']
        username = obj.get_username()
        return {
            'self': reverse('user-detail',
                kwargs={User.USERNAME_FIELD: username}, request=request),
            'tasks': '{}?assigned={}'.format(
                reverse('task-list', request=request), username)
        }
```

With our filters in place, and to continue the benefits of using this browsable interface, let's build out some validations to secure the state of our data.

Adding Validations

While using the frontend interface can be useful, it doesn't take much exploration through the API to realize that it has some problems. In particular, it allows changes to things that probably should not be changed. It also makes it possible to create a sprint that has already ended. These are all problems within the serializers.

Up until this point, our focus has been on how the model data is serialized into a dictionary to later become JSON, XML, YAML, and so on for the client. No work has been done yet on changing how the client's request of that markup is translated back into creating a model instance. For a typical Django view, this would be handled by a `Form` or `ModelForm`. In `django-rest-framework`, this is handled by the serializer. Not surprisingly, the API is similar to those of Django's forms. In fact, the serializer fields make use of the existing logic in Django's form fields.

One thing the API should prevent is creating sprints that have happened prior to the current date and time. To handle this, the `SprintSerializer` needs to check the value of the end date submitted by the client. Each serializer field has a `validate_<field>` hook that is called to perform additional validations on the field. Again, this parallels the `clean_<field>` in Django's forms. This should be added to `SprintSerializer` in *board/serializers.py*.

```
from datetime import date                                          ❶

from django.contrib.auth import get_user_model
from django.utils.translation import ugettext_lazy as _            ❷

from rest_framework import serializers
from rest_framework.reverse import reverse

from .models import Sprint, Task


User = get_user_model()


class SprintSerializer(serializers.ModelSerializer):
```

```
    links = serializers.SerializerMethodField('get_links')

    class Meta:
        model = Sprint
        fields = ('id', 'name', 'description', 'end', 'links', )

    def get_links(self, obj):
        request = self.context['request']
        return {
            'self': reverse('sprint-detail',
                kwargs={'pk': obj.pk}, request=request),
            'tasks': reverse('task-list',
                request=request) + '?sprint={}'.format(obj.pk),
        }

    def validate_end(self, attrs, source):                              ❸
        end_date = attrs[source]
        new = not self.object
        changed = self.object and self.object.end != end_date
        if (new or changed) and (end_date < date.today()):
            msg = _('End date cannot be in the past.')
            raise serializers.ValidationError(msg)
        return attrs
...
```

❶ ❷ These are new imports for `datetime` from the standard library and `ugettext_lazy` to make the error messages translatable.

❸ `validate_end` checks that the end date is greater than or equal to the current date for newly created sprints or any sprint that is being updated.

To see this in action, we can attempt to create a historical sprint on *http://localhost:8000/api/sprints/*. You should now get a 400 BAD REQUEST response.

```
HTTP 400 BAD REQUEST
Vary: Accept
Content-Type: application/json
Allow: GET, POST, HEAD, OPTIONS

{
    "end": [
        "End date cannot be in the past."
    ]
}
```

The check for the current object ensures that this validation will apply only to new objects and existing objects where the end date is being changed. That will allow the name of a past sprint to be changed but not its end date.

Similar to this issue, there is also no validation for creating and editing tasks. Tasks can be added to sprints that are already over and can also have the completed date set without being marked as done. Also, the tasks can have their start date set without being started.

Validating conditions that require more than one field is handled in the `validate` method, which parallels the `clean` method for forms, as shown in this snippet from *board/serializers.py*.

```
...
class TaskSerializer(serializers.ModelSerializer):
...
    def validate_sprint(self, attrs, source):                          ❶
        sprint = attrs[source]
        if self.object and self.object.pk:
            if sprint != self.object.sprint:
                if self.object.status == Task.STATUS_DONE:
                    msg = _('Cannot change the sprint of a completed task.')
                    raise serializers.ValidationError(msg)
                if sprint and sprint.end < date.today():
                    msg = _('Cannot assign tasks to past sprints.')
                    raise serializers.ValidationError(msg)
        else:
            if sprint and sprint.end < date.today():
                msg = _('Cannot add tasks to past sprints.')
                raise serializers.ValidationError(msg)
        return attrs

    def validate(self, attrs):                                         ❷
        sprint = attrs.get('sprint')
        status = int(attrs.get('status'))
        started = attrs.get('started')
        completed = attrs.get('completed')
        if not sprint and status != Task.STATUS_TODO:
            msg = _('Backlog tasks must have "Not Started" status.')
            raise serializers.ValidationError(msg)
        if started and status == Task.STATUS_TODO:
            msg = _('Started date cannot be set for not started tasks.')
            raise serializers.ValidationError(msg)
        if completed and status != Task.STATUS_DONE:
            msg = _('Completed date cannot be set for uncompleted tasks.')
            raise serializers.ValidationError(msg)
        return attrs
...
```

❶ `validate_sprint` ensures that the sprint is not changed after the task is completed and that tasks are not assigned to sprints that have already been completed.

❷ `validate` ensures that the combination of fields makes sense for the task.

Since these validations are handled only by the serializers, not the models, they won't be checked if other parts of the project are changing or creating these models. If the Django admin is enabled, sprints in the past can still be added; likewise, if the project adds some import scripts, those could still add historical sprints.

With our validations in place, let's take a look at how we can use a Python client to browse our RESTful API and see how to capture the data with Python.

Using a Python Client

The browsable API is a simple way to explore the API and ensure that the resources link together in a way that makes sense. However, it gives the same experience as how a programmatic client would use the API. To understand how a developer would use the API, we can write a simple client in Python using the popular `requests` library. First, it needs to be installed with `pip`:

```
hostname $ pip install requests
```

In one terminal run the server with `runserver`, and in another start the Python interactive shell. Similar to the browser, we can start by fetching the root of the API at *http://localhost:8000/api/*:

```
hostname $ python
>>> import requests
>>> import pprint
>>> response = requests.get('http://localhost:8000/api/')
>>> response.status_code
200
>>> api = response.json()
>>> pprint.pprint(api)
{'sprints': 'http://localhost:8000/api/sprints/',
 'tasks': 'http://localhost:8000/api/tasks/',
 'users': 'http://localhost:8000/api/users/'}
```

The API root lists the subresources below it for the sprints, tasks, and users. In the current configuration this view does not require any authentication, but the remaining resources do. Continuing in the shell, attempting to access a resource without authentication will return a 401 error:

```
>>> response = requests.get(api['sprints'])
>>> response.status_code
401
```

We can authenticate the client by passing the username and password as the `auth` argument:

```
>>> response = requests.get(api['sprints'], auth=('demo', 'test'))
>>> response.status_code
200
```

Remember, this example assumes that there is a user with the username *demo* and password *test*. You would have set these up during the creation of the database tables in the section "Models" on page 66.

Using this *demo* user, we'll create a new sprint and add some tasks to it. Creating a sprint requires sending a POST request to the sprint's endpoint, giving a name and end date to the sprint.

```
>>> import datetime
>>> today = datetime.date.today()
>>> two_weeks = datetime.timedelta(days=14)
>>> data = {'name': 'Current Sprint', 'end': today + two_weeks}
>>> response = requests.post(api['sprints'], data=data, auth=('demo', 'test'))
>>> response.status_code
201
>>> sprint = response.json()
>>> pprint.pprint(sprint)
{'description': '',
 'end': '2014-08-31',
 'id': 2,
 'links': {'self': 'http://localhost:8000/api/sprints/2/',
           'tasks': 'http://localhost:8000/api/tasks/?sprint=2'},
 'name': 'Current Sprint'}
```

With the sprint created, we can now add tasks associated with it. The URL for the sprint defines a unique reference for it, and that will be passed to the request to create a task.

```
>>> data = {'name': 'Something Task', 'sprint': sprint['id']}
>>> response = requests.post(api['tasks'], data=data, auth=('demo', 'test'))
>>> response.status_code
201
>>> task = response.json()
>>> pprint.pprint(task)
{'assigned': None,
 'completed': None,
 'description': '',
 'due': None,
 'id': 2,
 'links': {'assigned': None,
           'self': 'http://localhost:8000/api/tasks/2/',
           'sprint': 'http://localhost:8000/api/sprints/2/'},
 'name': 'Something Task',
 'order': 0,
 'sprint': 2,
 'started': None,
 'status': 1,
 'status_display': 'Not Started'}
```

We can update the tasks by sending a PUT with the new task data to its URL. Let's update the status and start date and assign the task to the *demo* user.

```
>>> task['assigned'] = 'demo'
>>> task['status'] = 2
>>> task['started'] = today
>>> response = requests.put(task['links']['self'],
        ...data=task, auth=('demo', 'test'))
>>> response.status_code
200
>>> task = response.json()
>>> pprint.pprint(task)
{'assigned': 'demo',
 'completed': None,
 'description': '',
 'due': None,
 'id': 2,
 'links': {'assigned': 'http://localhost:8000/api/users/demo/',
           'self': 'http://localhost:8000/api/tasks/2/',
           'sprint': 'http://localhost:8000/api/sprints/2/'},
 'name': 'Something Task',
 'order': 0,
 'sprint': 2,
 'started': '2014-08-17',
 'status': 2,
 'status_display': 'In Progress'}
```

Notice that throughout this process the URLs were not built by the client but rather were given by the API. Similar to using the browsable API, the client did not need to know how the URLs were constructed. The client explores the API and finds that the resources and logic were about parsing the information out of the server responses and manipulating them. This makes the client much easier to maintain.

Next Steps

With the REST API in place, and some basic understanding of how to use the API with a Python client, we can now move on to using the API with a client written in JavaScript. The next chapter will explore building a single-page web application using Backbone.js on top of this REST API.