The Flask Mega-Tutorial Part XXIII: Application Programming Interfaces (APIs) (/post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis)

May 8 2018

Posted by **Miguel Grinberg (/author/Miguel Grinberg)** under **Flask (/category/Flask)**, **Python (/category/Python)**, **REST (/category/REST)**, **Programming (/category/Programming)**.

This is the twenty third and last installment of the Flask Mega-Tutorial series, in which I'm going to tell you how to extend microblog with an application programming interface (or API) that clients can use to work with the application in a more direct way than the traditional web browser workflow.

For your reference, below is a list of the articles in this series.

- Chapter 1: Hello, World! (/post/the-flask-mega-tutorial-part-i-hello-world)
- Chapter 2: Templates (/post/the-flask-mega-tutorial-part-ii-templates)
- Chapter 3: Web Forms (/post/the-flask-mega-tutorial-part-iii-web-forms)
- Chapter 4: Database (/post/the-flask-mega-tutorial-part-iv-database)
- Chapter 5: User Logins (/post/the-flask-mega-tutorial-part-v-user-logins)
- Chapter 6: Profile Page and Avatars (/post/the-flask-mega-tutorial-part-vi-profile-page-and-avatars)
- Chapter 7: Error Handling (/post/the-flask-mega-tutorial-part-vii-error-handling)
- Chapter 8: Followers (/post/the-flask-mega-tutorial-part-viii-followers)
- Chapter 9: Pagination (/post/the-flask-mega-tutorial-part-ix-pagination)
- Chapter 10: Email Support (/post/the-flask-mega-tutorial-part-x-email-support)
- Chapter 11: Facelift (/post/the-flask-mega-tutorial-part-xi-facelift)
- Chapter 12: Dates and Times (/post/the-flask-mega-tutorial-part-xii-dates-and-times)
- Chapter 13: I18n and L10n (/post/the-flask-mega-tutorial-part-xiii-i18n-and-l10n)
- Chapter 14: Ajax (/post/the-flask-mega-tutorial-part-xiv-ajax)
- Chapter 15: A Better Application Structure (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure)
- Chapter 16: Full-Text Search (/post/the-flask-mega-tutorial-part-xvi-full-text-search)
- Chapter 17: Deployment on Linux (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux)
- Chapter 18: Deployment on Heroku (/post/the-flask-mega-tutorial-part-xviii-deployment-on-heroku)

- Chapter 19: Deployment on Docker Containers (/post/the-flask-mega-tutorial-part-xix-deployment-on-docker-containers)
- Chapter 20: Some JavaScript Magic (/post/the-flask-mega-tutorial-part-xx-some-javascript-magic)
- Chapter 21: User Notifications (/post/the-flask-mega-tutorial-part-xxi-user-notifications)
- Chapter 22: Background Jobs (/post/the-flask-mega-tutorial-part-xxii-background-jobs)
- Chapter 23: Application Programming Interfaces (APIs) (/post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis) (this article)

Note 1: If you are looking for the legacy version of this tutorial, it's here (/post/the-flask-mega-tutorial-part-i-hello-world-legacy).

Note 2: If you would like to support my work on this blog, or just don't have patience to wait for weekly articles, I am offering the complete version of this tutorial packaged as an ebook or a set of videos. For more information, visit courses.miguelgrinberg.com (https://courses.miguelgrinberg.com).

All the functionality that I built so far for this application is meant for one specific type of client: the web browser. But what about other types of clients? If I wanted to build an Android or iOS app, for example, I have two main ways to go about it. The easiest solution would be to build a simple app with just a web view component that fills the entire screen, where the Microblog website is loaded, but this would offer little benefit over opening the application in the device's web browser. A better solution (though much more laborious) would be to build a native app, but how can this app interact with a server that only returns HTML pages?

This is the problem area where Application Programming Interfaces (or APIs) can help. An API is a collection of HTTP routes that are designed as low-level entry points into the application. Instead of defining routes and view functions that return HTML to be consumed by web browsers, APIs allow the client to work directly with the application's *resources*, leaving the decision of how to present the information to the user entirely to the client. For example, an API in Microblog could provide user and blog post information to the client, and it could also allow the user to edit an existing blog post, but only at the data level, without mixing this logic with HTML.

If you study all the routes currently defined in the application, you will notice that there are a few that could fit the definition of API I used above. Did you find them? I'm talking about the few routes that return JSON, such as the */translate* route defined in Chapter 14 (/post/the-flask-mega-tutorial-part-xiv-ajax). This is a route that takes a text, source and destination languages, all given in JSON format in a `POST` request. The response to this

request is the translation of that text, also in JSON format. The server only returns the requested information, leaving the client with the responsibility to present this information to the user.

While the JSON routes in the application have an API "feel" to them, they were designed to support the web application running in the browser. Consider that if a smartphone app wanted to use these routes, it would not be able to because they require a logged in user, and the log in can only happen through an HTML form. In this chapter I'm going to show how to build APIs that do not rely on the web browser and make no assumptions about what kind of client connects to them.

The GitHub links for this chapter are: Browse (https://github.com/miguelgrinberg/microblog/tree/v0.23), Zip (https://github.com/miguelgrinberg/microblog/archive/v0.23.zip), Diff (https://github.com/miguelgrinberg/microblog/compare/v0.22...v0.23).

REST as a Foundation of API Design

Some people may strongly disagree with my statement above that */translate* and the other JSON routes are API routes. Others may agree, with the disclaimer that they consider them a badly designed API. So what are the characteristics of a well designed API, and why aren't the JSON routes in that category?

You may have heard the term REST API (https://en.wikipedia.org/wiki/Representational_state_transfer). REST, which stands for Representational State Transfer, is an architecture proposed by Dr. Roy Fielding in his doctoral dissertation (http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm). In his work, Dr. Fielding presents the six defining characteristics of REST in a fairly abstract and generic way.

There is no authoritative specification for REST besides Dr. Fielding's dissertation, and this leaves a lot of details to be interpreted by the reader. The topic of whether a given API complies with REST or not is often the source of heated debates between REST "purists", who believe that a REST API must observe all six characteristics and do so in a very specific way, versus the REST "pragmatists", who take the ideas presented by Dr. Fielding in his dissertation as guidelines or recommendations. Dr. Fielding himself sides with the purist camp, and has provided some additional insight into his vision in blog posts and online comments.

The vast majority of APIs currently implemented adhere to a "pragmatic" REST implementation. This includes most of the APIs from the "big players", such as Facebook, GitHub, Twitter, etc. There are very few public APIs that are unanimously considered to be pure REST, because most APIs miss certain implementation details that purists consider must-haves. In spite of the strict views Dr. Fielding and other REST purists have on what is or isn't a REST API, it is common in the software industry to refer to REST in the pragmatic sense.

To give you an idea of what's in the REST dissertation, the following sections describe the six principles enumerated by Dr. Fielding.

Client-Server

The client-server principle is fairly straightforward, as it simply states that in a REST API the roles of the client and the server should be clearly differentiated. In practice, this means that the client and the server are in separate processes that communicate over a transport, which in the majority of the cases is the HTTP protocol over a TCP network.

Layered System

The layered system principle says that when a client needs to communicate with a server, it may end up connected to an intermediary and not the actual server. The idea is that for the client, there should be absolutely no difference in the way it sends requests if not connected directly to the server, in fact, it may not even know if it is connected to the target server or not. Likewise, this principle states that a server may receive client requests from an intermediary and not the client directly, so it must never assume that the other side of the connection is the client.

This is an important feature of REST, because being able to add intermediary nodes allows application architects to design large and complex networks that are able to satisfy a large volume of requests through the use of load balancers, caches, proxy servers, etc.

Cache

This principle extends the layered system, by indicating explicitly that it is allowed for the server or an intermediary to cache responses to requests that are received often to improve system performance. There is an implementation of a cache that you are likely familiar with: the one in all web browsers. The web browser cache layer is often used to avoid having to request the same files, such as images, over and over again.

For the purposes of an API, the target server needs to indicate through the use of *cache controls* if a response can be cached by intermediaries as it travels back to the client. Note that because for security reasons APIs deployed to production must use encryption, caching is usually not done in an intermediate node unless this node *terminates* the SSL connection, or performs decryption and re-encryption.

Code On Demand

This is an optional requirement that states that the server can provide executable code in responses to the client. Because this principle requires an agreement between the server and the client on what kind of executable code the client is able to run, this is rarely used in APIs. You would think that the server could return JavaScript code for web browser clients to execute, but REST is not specifically targeted to web browser clients. Executing JavaScript, for example, could introduce a complication if the client is an iOS or Android device.

Stateless

The stateless principle is one of the two at the center of most debates between REST purists and pragmatists. It states that a REST API should not save any client state to be recalled every time a given client sends a request. What this means is that none of the mechanisms that are common in web development to "remember" users as they navigate through the pages of the application can be used. In a stateless API, every request needs to include the information that the server needs to identify and authenticate the client and to carry out the request. It also means that the server cannot store any data related to the client connection in a database or other form of storage.

If you are wondering why REST requires stateless servers, the main reason is that stateless servers are very easy to scale, all you need to do is run multiple instances of the server behind a load balancer. If the server stores client state things get more complicated, as you have to figure out how multiple servers can access and update that state, or alternatively ensure that a given client is always handled by the same server, something commonly referred to as *sticky sessions*.

If you consider again the */translate* route discussed in the chapter's introduction, you'll realize that it cannot be considered *RESTful*, because the view function associated with that route relies on the `@login_required` decorator from Flask-Login, which in turn stores the logged in state of the user in the Flask user session.

Uniform Interface

The final, most important, most debated and most vaguely documented REST principle is the uniform interface. Dr. Fielding enumerates four distinguishing aspects of the REST uniform interface: unique resource identifiers, resource representations, self-descriptive messages, and hypermedia.

Unique resource identifiers are achieved by assigning a unique URL to each resource. For example, the URL associated with a given user can be */api/users/<user-id>*, where *<user-id>* is the identifier assigned to the user in the database table's primary key. This is reasonably well implemented by most APIs.

The use of resource representations means that when the server and the client exchange information about a resource, they must use an agreed upon format. For most modern APIs, the JSON format is used to build resource representations. An API can choose to support multiple resource representation formats, and in that case the *content negotiation* options in the HTTP protocol are the mechanism by which the client and the server can agree on a format that both like.

Self-descriptive messages means that requests and responses exchanged between the clients and the server must include all the information that the other party needs. As a typical example, the HTTP request method is used to indicate what operation the client wants the server to execute. A `GET` request indicates that the client wants to retrieve information about a resource, a `POST` request indicates the client wants to create a new resource, `PUT` or `PATCH` requests define modifications to existing resources, and `DELETE` indicates a request to remove a resource. The target resource is indicated as the request URL, with additional information provided in HTTP headers, the query string portion of the URL or the request body.

The hypermedia requirement is the most polemic of the set, and one that few APIs implement, and those APIs that do implement it rarely do so in a way that satisfies REST purists. Since the resources in an application are all inter-related, this requirement asks that those relationships are included in the resource representations, so that clients can discover new resources by traversing relationships, pretty much in the same way you discover new pages in a web application by clicking on links that take you from one page to the next. The idea is that a client could enter an API without any previous knowledge about the resources in it, and learn about them simply by following hypermedia links. One of the aspects that complicate the implementation of this requirement is that unlike HTML and XML, the JSON format that is commonly used for resource representations in APIs does not define a standard way to include links, so you are forced to use a custom structure, or one of the proposed JSON extensions that try to address this gap, such as JSON-API (http://jsonapi.org/), HAL (http://stateless.co/hal_specification.html), JSON-LD (https://json-ld.org/) or similar.

Implementing an API Blueprint

To give you a taste of what is involved in developing an API, I'm going to add one to Microblog. This is not going to be a complete API, I'm going to implement all the functions related to users, leaving the implementation of other resources such as blog posts to the reader as an exercise.

To keep things organized, and following the structure I described in Chapter 15 (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure), I'm going to create a new blueprint that will contain all the API routes. So let's begin by creating the directory where this blueprint will live:

```
(venv) $ mkdir app/api
```

The blueprint's *__init__.py* file creates the blueprint object, similar to the other blueprints in the application:

app/api/__init__.py: API blueprint constructor.

```
from flask import Blueprint

bp = Blueprint('api', __name__)

from app.api import users, errors, tokens
```

You probably remember that it is sometimes necessary to move imports to the bottom to avoid circular dependency errors. That is the reason why the *app/api/users.py*, *app/api/errors.py* and *app/api/tokens.py* modules (that I'm yet to write) are imported after the blueprint is created.

The meat of the API is going to be stored in the *app/api/users.py* module. The following table summarizes the routes that I'm going to implement:

HTTP Method	Resource URL	Notes
GET	*/api/users/<id>*	Return a user.
GET	*/api/users*	Return the collection of all users.
GET	*/api/users/<id>/followers*	Return the followers of this user.
GET	*/api/users/<id>/followed*	Return the users this user is following.
POST	*/api/users*	Register a new user account.
PUT	*/api/users/<id>*	Modify a user.

For now I'm going to create a skeleton module with placeholders for all these routes:

app/api/users.py: User API resource placeholders.

```
from app.api import bp

@bp.route('/users/<int:id>', methods=['GET'])
def get_user(id):
    pass

@bp.route('/users', methods=['GET'])
def get_users():
    pass

@bp.route('/users/<int:id>/followers', methods=['GET'])
def get_followers(id):
    pass

@bp.route('/users/<int:id>/followed', methods=['GET'])
def get_followed(id):
    pass

@bp.route('/users', methods=['POST'])
def create_user():
    pass

@bp.route('/users/<int:id>', methods=['PUT'])
def update_user(id):
    pass
```

The *app/api/errors.py* module is going to define a few helper functions that deal with error responses. But for now, I'm also going to use a placeholder that I will fill out later:

app/api/errors.py: Error handling placeholder.

```
def bad_request():
    pass
```

The *app/api/tokens.py* is the module where the authentication subsystem is going to be defined. This is going to provide an alternative way for clients that are not web browsers to log in. For now, I'm going to write a placeholder for this module as well:

app/api/tokens.py: Token handling placeholder.

```
def get_token():
    pass

def revoke_token():
    pass
```

The new API blueprint needs to be registered in the application factory function:

app/__init__.py: Register API blueprint with the application.

```
# ...

def create_app(config_class=Config):
    app = Flask(__name__)

    # ...

    from app.api import bp as api_bp
    app.register_blueprint(api_bp, url_prefix='/api')

    # ...
```

Representing Users as JSON Objects

The first aspect to consider when implementing an API is to decide what the representation of its resources is going to be. I'm going to implement an API that works with users, so a representation for my user resources is what I need to decide on. After some brainstorming, I came up with the following JSON representation:

```
{
    "id": 123,
    "username": "susan",
    "password": "my-password",
    "email": "susan@example.com",
    "last_seen": "2017-10-20T15:04:27Z",
    "about_me": "Hello, my name is Susan!",
    "post_count": 7,
    "follower_count": 35,
    "followed_count": 21,
    "_links": {
        "self": "/api/users/123",
        "followers": "/api/users/123/followers",
        "followed": "/api/users/123/followed",
        "avatar": "https://www.gravatar.com/avatar/..."
    }
}
```

Many of the fields are directly coming from the user database model. The `password` field is special in that it is only going to be used when a new user is registered. As you remember from Chapter 5 (/post/the-flask-mega-tutorial-part-v-user-logins), user passwords are not stored in the database, only a hash is, so password are never returned. The `email` field is also treated specially, because I don't want to expose the email addresses of users. The `email` field is only going to be returned when users request their own entry, but not when they retrieve entries from other users. The `post_count`, `follower_count` and `followed_count` fields are "virtual" fields that do not

exist as fields in the database, but are provided to the client as a convenience. This is a great example that demonstrates that a resource representation does not need to match how the actual resource is defined in the server.

Note the `_links` section, which implements the hypermedia requirements. The links that are defined include links to the current resource, the list of users that follow this user, the list of users followed by the user, and finally a link to the user's avatar image. In the future, if I decide to add posts to this API, a link to the list of posts by the user should be included here as well.

One nice thing about the JSON format is that it always translates to a representation as a Python dictionary or list. The `json` package from the Python standard library takes care of converting the Python data structures to and from JSON. So to generate these representations, I'm going to add a method to the `User` model called `to_dict()`, which returns a Python dictionary:

app/models.py: User model to representation.

```
from flask import url_for
# ...

class User(UserMixin, db.Model):
    # ...

    def to_dict(self, include_email=False):
        data = {
            'id': self.id,
            'username': self.username,
            'last_seen': self.last_seen.isoformat() + 'Z',
            'about_me': self.about_me,
            'post_count': self.posts.count(),
            'follower_count': self.followers.count(),
            'followed_count': self.followed.count(),
            '_links': {
                'self': url_for('api.get_user', id=self.id),
                'followers': url_for('api.get_followers', id=self.id),
                'followed': url_for('api.get_followed', id=self.id),
                'avatar': self.avatar(128)
            }
        }
        if include_email:
            data['email'] = self.email
        return data
```

This method should be mostly self-explanatory, the dictionary with the user representation I settled on is simply generated and returned. As I mentioned above, the `email` field needs special treatment, because I want to include the email only when users request their own data. So I'm using the `include_email` flag to determine if that field gets included in the representation or not.

Note how the `last_seen` field is generated. For date and time fields, I'm going to use the ISO 8601 (https://en.wikipedia.org/wiki/ISO_8601) format, which Python's `datetime` can generate through the `isoformat()` method. But because I'm using naive `datetime` objects that are UTC but do not have the timezone recorded in their state, I need to add the `Z` at the end, which is ISO 8601's timezone code for UTC.

Finally, see how I implemented the hypermedia links. For the three links that point to other application routes I use `url_for()` to generate the URLs (which currently point to the placeholder view functions I defined in *app/api/users.py*). The avatar link is special because it is a Gravatar URL, external to the application. For this link I use the same `avatar()` method that I've used to render the avatars in web pages.

The `to_dict()` method converts a user object to a Python representation, which will then be converted to JSON. I also need look at the reverse direction, where the client passes a user representation in a request and the server needs to parse it and convert it to a `User` object. Here is the `from_dict()` method that achieves the conversion from a Python dictionary to a model:

app/models.py: Representation to User model.

```
class User(UserMixin, db.Model):
    # ...

    def from_dict(self, data, new_user=False):
        for field in ['username', 'email', 'about_me']:
            if field in data:
                setattr(self, field, data[field])
        if new_user and 'password' in data:
            self.set_password(data['password'])
```

In this case I decided to use a loop to import any of the fields that the client can set, which are `username`, `email` and `about_me`. For each field I check if I there is a value provided in the `data` argument, and if there is I use Python's `setattr()` to set the new value in the corresponding attribute for the object.

The `password` field is treated as a special case, because it isn't a field in the object. The `new_user` argument determines if this is a new user registration, which means that a `password` is included. To set the password in the user model, I call the `set_password()` method, which creates the password hash.

Representing Collections of Users

In addition to working with single resource representations, this API is going to need a representation for a group of users. This is going to be the format used when the client requests the list of users or followers, for example. Here is the representation for a collection of users:

```
{
    "items": [
        { ... user resource ... },
        { ... user resource ... },
        ...
    ],
    "_meta": {
        "page": 1,
        "per_page": 10,
        "total_pages": 20,
        "total_items": 195
    },
    "_links": {
        "self": "http://localhost:5000/api/users?page=1",
        "next": "http://localhost:5000/api/users?page=2",
        "prev": null
    }
}
```

In this representation, `items` is the list of user resources, each defined as described in the previous section. The `_meta` section includes metadata for the collection that the client might find useful in presenting pagination controls to the user. The `_links` section defines relevant links, including a link to the collection itself, and the previous and next page links, also to help the client paginate the listing.

Generating the representation of a collection of users is tricky because of the pagination logic, but the logic is going to be common to other resources I may want to add to this API in the future, so I'm going to implement this representation in a generic way that I can then apply to other models. Back in Chapter 16 (/post/the-flask-mega-tutorial-part-xvi-full-text-search) I was in a similar situation with full-text search indexes, another feature that I wanted to implement generically so that it can be applied to any models. The solution that I used was to implement a `SearchableMixin` class that any models that need a full-text index can inherit from. I'm going to use the same idea for this, so here is a new mixin class that I named `PaginatedAPIMixin`:

app/models.py: Paginated representation mixin class.

```
class PaginatedAPIMixin(object):
    @staticmethod
    def to_collection_dict(query, page, per_page, endpoint, **kwargs):
        resources = query.paginate(page, per_page, False)
        data = {
            'items': [item.to_dict() for item in resources.items],
            '_meta': {
                'page': page,
                'per_page': per_page,
                'total_pages': resources.pages,
                'total_items': resources.total
            },
            '_links': {
                'self': url_for(endpoint, page=page, per_page=per_page,
                                **kwargs),
                'next': url_for(endpoint, page=page + 1, per_page=per_page,
                                **kwargs) if resources.has_next else None,
                'prev': url_for(endpoint, page=page - 1, per_page=per_page,
                                **kwargs) if resources.has_prev else None
            }
        }
        return data
```

The `to_collection_dict()` method produces a dictionary with the user collection representation, including the `items`, `_meta` and `_links` sections. You may need to review the method carefully to understand how it works. The first three arguments are a Flask-SQLAlchemy query object, a page number and a page size. These are the arguments that determine what are the items that are going to be returned. The implementation uses the `paginate()` method of the query object to obtain a page worth of items, like I did with posts in the index, explore and profile pages of the web application.

The complicated part is in generating the links, which include a self-reference and the links to the next and previous pages. I wanted to make this function generic, so I could not, for example, use `url_for('api.get_users', id=id, page=page)` to generate the self link. The arguments to `url_for()` are going to be dependent on the particular collection of resources, so I'm going to rely on the caller passing in the `endpoint` argument the view function that needs to be sent to `url_for()`. And because many routes have arguments, I also need to capture additional keyword arguments in `kwargs`, and pass those to `url_for()` as well. The `page` and `per_page` query string argument are given explicitly because these control pagination for all API routes.

This mixin class needs to be added to the `User` model as a parent class:

app/models.py: Add PaginatedAPIMixin to User model.

```
class User(PaginatedAPIMixin, UserMixin, db.Model):
    # ...
```

In the case of collections I'm not going to need the reverse direction because I'm not going to have any routes that require the client to send lists of users.

Error Handling

The error pages that I defined in Chapter 7 (/post/the-flask-mega-tutorial-part-vii-error-handling) are only appropriate for a user that is interacting with the application using a web browser. When an API needs to return an error, it needs to be a "machine friendly" type of error, something that the client application can easily interpret. So in the same way I defined representations for my API resources in JSON, now I'm going to decide on a representation for API error messages. Here is the basic structure that I'm going to use:

```
{
    "error": "short error description",
    "message": "error message (optional)"
}
```

In addition to the payload of the error, I will use the status codes from the HTTP protocol to indicate the general class of the error. To help me generate these error responses, I'm going to write the `error_response()` function in *app/api/errors.py*:

app/api/errors.py: Error responses.

```
from flask import jsonify
from werkzeug.http import HTTP_STATUS_CODES

def error_response(status_code, message=None):
    payload = {'error': HTTP_STATUS_CODES.get(status_code, 'Unknown error')}
    if message:
        payload['message'] = message
    response = jsonify(payload)
    response.status_code = status_code
    return response
```

This function uses the handy `HTTP_STATUS_CODES` dictionary from Werkzeug (a core dependency of Flask) that provides a short descriptive name for each HTTP status code. I'm using these names for the `error` field in my error representations, so that I only need to worry about the numeric status code and the optional long description. The `jsonify()` function returns a Flask `Response` object with a default status code of 200, so after I create the response, I set the status code to the correct one for the error.

The most common error that the API is going to return is going to be the code 400, which is the error for "bad request". This is the error that is used when the client sends a request that has invalid data in it. To make it even easier to generate this error, I'm going

to add a dedicated function for it that only requires the long descriptive message as an argument. This is the `bad_request()` placeholder that I added earlier:

app/api/errors.py: Bad request responses.

```
# ...

def bad_request(message):
    return error_response(400, message)
```

User Resource Endpoints

The support that I need to work with JSON representations of users is now complete, so I'm ready to start coding the API endpoints.

Retrieving a User

Let's start with the request to retrieve a single user, given by `id`:

app/api/users.py: Return a user.

```
from flask import jsonify
from app.models import User

@bp.route('/users/<int:id>', methods=['GET'])
def get_user(id):
    return jsonify(User.query.get_or_404(id).to_dict())
```

The view function receives the `id` for the requested user as a dynamic argument in the URL. The `get_or_404()` method of the query object is a very useful variant of the `get()` method you have seen before, that also returns the object with the given `id` if it exists, but instead of returning `None` when the `id` does not exist, it aborts the request and returns a 404 error to the client. The advantage of `get_or_404()` over `get()` is that it removes the need to check the result of the query, simplifying the logic in view functions.

The `to_dict()` method I added to `User` is used to generate the dictionary with the resource representation for the selected user, and then Flask's `jsonify()` function converts that dictionary to JSON format to return to the client.

If you want to see how this first API route works, start the server and then type the following URL in your browser's address bar:

```
http://localhost:5000/api/users/1
```

This should show you the first user, rendered in JSON format. Also try to use a large `id` value, to see how the `get_or_404()` method of the SQLAlchemy query object triggers a 404 error (I will later show you how to extend the error handling so that these errors are also returned in JSON format).

To test this new route, I'm going to install HTTPie (https://httpie.org/), a command-line HTTP client written in Python that makes it easy to send API requests:

```
(venv) $ pip install httpie
```

I can now request information about the user with a `id` of `1` (which is probably yourself) with the following command:

```
(venv) $ http GET http://localhost:5000/api/users/1
HTTP/1.0 200 OK
Content-Length: 457
Content-Type: application/json
Date: Mon, 27 Nov 2017 20:19:01 GMT
Server: Werkzeug/0.12.2 Python/3.6.3

{
    "_links": {
        "avatar": "https://www.gravatar.com/avatar/993c...2724?d=identicon&s=128",
        "followed": "/api/users/1/followed",
        "followers": "/api/users/1/followers",
        "self": "/api/users/1"
    },
    "about_me": "Hello! I'm the author of the Flask Mega-Tutorial.",
    "followed_count": 0,
    "follower_count": 1,
    "id": 1,
    "last_seen": "2017-11-26T07:40:52.942865Z",
    "post_count": 10,
    "username": "miguel"
}
```

Retrieving Collections of Users

To return the collection of all users, I can now rely on the `to_collection_dict()` method of `PaginatedAPIMixin`:

app/api/users.py: Return the collection of all users.

```
from flask import request

@bp.route('/users', methods=['GET'])
def get_users():
    page = request.args.get('page', 1, type=int)
    per_page = min(request.args.get('per_page', 10, type=int), 100)
    data = User.to_collection_dict(User.query, page, per_page, 'api.get_users')
    return jsonify(data)
```

For this implementation I first extract `page` and `per_page` from the query string of the request, using the defaults of 1 and 10 respectively if they are not defined. The `per_page` has additional logic that caps it at 100. Giving the client control to request really large pages is not a good idea, as that can cause performance problems for the server. The `page` and `per_page` arguments are then passed to the `to_collection_query()` method, along with the query, which in this case is simply `User.query`, the most generic query that returns all users. The last argument is `api.get_users`, which is the endpoint name that I need for the three links that are used in the representation.

To test this endpoint with HTTPie, use the following command:

```
(venv) $ http GET http://localhost:5000/api/users
```

The next two endpoints are the ones that return the follower and followed users. These are fairly similar to the one above:

app/api/users.py: Return followers and followed users.

```
@bp.route('/users/<int:id>/followers', methods=['GET'])
def get_followers(id):
    user = User.query.get_or_404(id)
    page = request.args.get('page', 1, type=int)
    per_page = min(request.args.get('per_page', 10, type=int), 100)
    data = User.to_collection_dict(user.followers, page, per_page,
                                   'api.get_followers', id=id)
    return jsonify(data)

@bp.route('/users/<int:id>/followed', methods=['GET'])
def get_followed(id):
    user = User.query.get_or_404(id)
    page = request.args.get('page', 1, type=int)
    per_page = min(request.args.get('per_page', 10, type=int), 100)
    data = User.to_collection_dict(user.followed, page, per_page,
                                   'api.get_followed', id=id)
    return jsonify(data)
```

Since these two routes are specific to a user, they have the `id` dynamic argument. The `id` is used to get the user from the database, and then to provide the `user.followers` and `user.followed` relationship based queries to `to_collection_dict()`, so hopefully now you can see why spending a little bit of extra time and designing this method in a generic way really pays off. The last two arguments to `to_collection_dict()` are the endpoint name, and the `id`, which the method is going to take as an extra keyword argument in `kwargs`, and then pass it to `url_for()` when generating the links section of the representation.

Similar to the previous example, you can exercise these two routes with HTTPie as follows:

```
(venv) $ http GET http://localhost:5000/api/users/1/followers
(venv) $ http GET http://localhost:5000/api/users/1/followed
```

I should note that thanks to hypermedia, you do not need to remember these URLs, as they are included in the `_links` section of the user representation.

Registering New Users

The `POST` request to the */users* route is going to be used to register new user accounts. You can see the implementation of this route below:

app/api/users.py: Register a new user.

```
from flask import url_for
from app import db
from app.api.errors import bad_request

@bp.route('/users', methods=['POST'])
def create_user():
    data = request.get_json() or {}
    if 'username' not in data or 'email' not in data or 'password' not in data:
        return bad_request('must include username, email and password fields')
    if User.query.filter_by(username=data['username']).first():
        return bad_request('please use a different username')
    if User.query.filter_by(email=data['email']).first():
        return bad_request('please use a different email address')
    user = User()
    user.from_dict(data, new_user=True)
    db.session.add(user)
    db.session.commit()
    response = jsonify(user.to_dict())
    response.status_code = 201
    response.headers['Location'] = url_for('api.get_user', id=user.id)
    return response
```

This request is going to accept a user representation in JSON format from the client, provided in the request body. Flask provides the `request.get_json()` method to extract the JSON from the request and return it as a Python structure. This method returns `None` if JSON data isn't found in the request, so I can ensure that I always get a dictionary using the expression `request.get_json() or {}`.

Before I can use the data I need to ensure that I've got all the information, so I start by checking that the three mandatory fields are included. These are `username`, `email` and `password`. If any of those are missing, then I use the `bad_request()` helper function from the *app/api/errors.py* module to return an error to the client. In addition to that check, I need to make sure that the `username` and `email` fields are not already used by another user, so for that I try to load a user from the database by the username and emails provided, and if any of those return a valid user, I also return an error back to the client.

Once I've passed the data validation, I can easily create a user object and add it to the database. To create the user I rely on the `from_dict()` method in the `User` model. The `new_user` argument is set to `True`, so that it also accepts the `password` field which is normally not part of the user representation.

The response that I return for this request is going to be the representation of the new user, so `to_dict()` generates that payload. The status code for a `POST` request that creates a resource should be 201, the code that is used when a new entity has been created. Also, the HTTP protocol requires that a 201 response includes a `Location` header that is set to the URL of the new resource.

Below you can see how to register a new user from the command line through HTTPie:

```
(venv) $ http POST http://localhost:5000/api/users username=alice password=dog \
    email=alice@example.com "about_me=Hello, my name is Alice!"
```

Editing Users

The last endpoint that I'm going to use in my API is the one that modifies an existing user:

app/api/users.py: Modify a user.

```
@bp.route('/users/<int:id>', methods=['PUT'])
def update_user(id):
    user = User.query.get_or_404(id)
    data = request.get_json() or {}
    if 'username' in data and data['username'] != user.username and \
            User.query.filter_by(username=data['username']).first():
        return bad_request('please use a different username')
    if 'email' in data and data['email'] != user.email and \
            User.query.filter_by(email=data['email']).first():
        return bad_request('please use a different email address')
    user.from_dict(data, new_user=False)
    db.session.commit()
    return jsonify(user.to_dict())
```

For this request I receive a user `id` as a dynamic part of the URL, so I can load the designated user and return a 404 error if it is not found. Note that there is no authentication yet, so for now the API is going to allow users to make changes to the accounts of any other users. This is going to be addressed later when authentication is added.

Like in the case of a new user, I need to validate that the `username` and `email` fields provided by the client do not collide with other users before I can use them, but in this case the validation is a bit more tricky. First of all, these fields are optional in this request, so I need to check that a field is present. The second complication is that the client may

be providing the same value, so before I check if the username or email are taken I need to make sure they are different than the current ones. If any of these validation checks fail, then I return a 400 error back to the client, as before.

Once the data has been validated, I can use the `from_dict()` method of the `User` model to import all the data provided by the client, and then commit the change to the database. The response from this request returns the updated user representation to the user, with a default 200 status code.

Here is an example request that edits the `about_me` field with HTTPie:

```
(venv) $ http PUT http://localhost:5000/api/users/2 "about_me=Hi, I am Miguel"
```

API Authentication

The API endpoints I added in the previous section are currently open to any clients. Obviously they need to be available to registered users only, and to do that I need to add *authentication* and *authorization*, or "AuthN" and "AuthZ" for short. The idea is that requests sent by clients provide some sort of identification, so that the server knows what user the client is representing, and can verify if the requested action is allowed or not for that user.

The most obvious way to protect these API endpoints is to use the `@login_required` decorator from Flask-Login, but that approach has some problems. When the decorator detects a non-authenticated user, it redirects the user to a HTML login page. In an API there is no concept of HTML or login pages, if a client sends a request with invalid or missing credentials, the server has to refuse the request returning a 401 status code. The server can't assume that the API client is a web browser, or that it can handle redirects, or that it can render and process HTML login forms. When the API client receives the 401 status code, it knows that it needs to ask the user for credentials, but how it does that is really not the business of the server.

Tokens In the User Model

For the API authentication needs, I'm going to use a *token* authentication scheme. When a client wants to start interacting with the API, it needs to request a temporary token, authenticating with a username and password. The client can then send API requests passing the token as authentication, for as long as the token is valid. Once the token expires, a new token needs to be requested. To support user tokens, I'm going to expand the `User` model:

app/models.py: Support for user tokens.

```
import base64
from datetime import datetime, timedelta
import os

class User(UserMixin, PaginatedAPIMixin, db.Model):
    # ...
    token = db.Column(db.String(32), index=True, unique=True)
    token_expiration = db.Column(db.DateTime)

    # ...

    def get_token(self, expires_in=3600):
        now = datetime.utcnow()
        if self.token and self.token_expiration > now + timedelta(seconds=60):
            return self.token
        self.token = base64.b64encode(os.urandom(24)).decode('utf-8')
        self.token_expiration = now + timedelta(seconds=expires_in)
        db.session.add(self)
        return self.token

    def revoke_token(self):
        self.token_expiration = datetime.utcnow() - timedelta(seconds=1)

    @staticmethod
    def check_token(token):
        user = User.query.filter_by(token=token).first()
        if user is None or user.token_expiration < datetime.utcnow():
            return None
        return user
```

With this change I'm adding a `token` attribute to the user model, and because I'm going to need to search the database by it I make it unique and indexed. I also added `token_expiration`, which has the date and time at which the token expires. This is so that a token does not remain valid for a long period of time, which can become a security risk.

I created three methods to work with these tokens. The `get_token()` method returns a token for the user. The token is generated as a random string that is encoded in base64 so that all the characters are in the readable range. Before a new token is created, this method checks if a currently assigned token has at least a minute left before expiration, and in that case the existing token is returned.

When working with tokens it is always good to have a strategy to revoke a token immediately, instead of only relying on the expiration date. This is a security best practice that is often overlooked. The `revoke_token()` method makes the token currently assigned to the user invalid, simply by setting the expiration date to one second before the current time.

The `check_token()` method is a static method that takes a token as input and returns the user this token belongs to as a response. If the token is invalid or expired, the method returns `None`.

Because I have made changes to the database, I need to generate a new database migration and then upgrade the database with it:

```
(venv) $ flask db migrate -m "user tokens"
(venv) $ flask db upgrade
```

Token Requests

When you write an API you have to consider that your clients are not always going to be web browsers connected to the web application. The real power of APIs comes when standalone clients such as smartphone apps, or even browser-based single page applications can have access to backend services. When these specialized clients need to access API services, they begin by requesting a token, which is the counterpart to the login form in the traditional web application.

To simplify the interactions between client and server when token authentication is used, I'm going to use a Flask extension called Flask-HTTPAuth (https://flask-httpauth.readthedocs.io/). Flask-HTTPAuth is installed with pip:

```
(venv) $ pip install flask-httpauth
```

Flask-HTTPAuth supports a few different authentication mechanisms, all API friendly. To begin, I'm going to use HTTP Basic Authentication (https://en.wikipedia.org/wiki/Basic_access_authentication), in which the client sends the user credentials in a standard Authorization (https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization) HTTP Header. To integrate with Flask-HTTPAuth, the application needs to provide two functions: one that defines the logic to check the username and password provided by the user, and another that returns the error response in the case of an authentication failure. These functions are registered with Flask-HTTPAuth through decorators, and then are automatically called by the extension as needed during the authentication flow. You can see the implementation below:

app/api/auth.py: Basic authentication support.

```
from flask_httpauth import HTTPBasicAuth
from app.models import User
from app.api.errors import error_response

basic_auth = HTTPBasicAuth()

@basic_auth.verify_password
def verify_password(username, password):
    user = User.query.filter_by(username=username).first()
    if user and user.check_password(password):
        return user

@basic_auth.error_handler
def basic_auth_error(status):
    return error_response(status)
```

The `HTTPBasicAuth` class from Flask-HTTPAuth is the one that implements the basic authentication flow. The two required functions are configured through the `verify_password` and `error_handler` decorators respectively.

The verification function receives the username and password that the client provided and returns the authenticated user if the credentials are valid or `None` if not. To check the password I rely on the `check_password()` method of the `User` class, which is also used by Flask-Login during authentication for the web application. The authenticated user will be available as `basic_auth.current_user()`, so that it can be used in the API view functions.

The error handler function returns a standard error response generated by the `error_response()` function in *app/api/errors.py*. The `status` argument is the HTTP status code, which in the case of invalid authentication is going to be 401. The 401 error is defined in the HTTP standard as the "Unauthorized" error. HTTP clients know that when they receive this error the request that they sent needs to be resent with valid credentials.

Now I have basic authentication support implemented, so I can add the token retrieval route that clients will invoke when they need a token:

app/api/tokens.py: Generate user tokens.

```
from flask import jsonify
from app import db
from app.api import bp
from app.api.auth import basic_auth

@bp.route('/tokens', methods=['POST'])
@basic_auth.login_required
def get_token():
    token = basic_auth.current_user().get_token()
    db.session.commit()
    return jsonify({'token': token})
```

This view function is decorated with the `@basic_auth.login_required` decorator from the `HTTPBasicAuth` instance, which will instruct Flask-HTTPAuth to verify authentication (through the verification function I defined above) and only allow the function to run when the provided credentials are valid. The implementation of this view function relies on the `get_token()` method of the user model to produce the token. A database commit is issued after the token is generated to ensure that the token and its expiration are written back to the database.

If you try to send a POST request to the tokens API route, this is what happens:

```
(venv) $ http POST http://localhost:5000/api/tokens
HTTP/1.0 401 UNAUTHORIZED
Content-Length: 30
Content-Type: application/json
Date: Mon, 27 Nov 2017 20:01:00 GMT
Server: Werkzeug/0.12.2 Python/3.6.3
WWW-Authenticate: Basic realm="Authentication Required"

{
    "error": "Unauthorized"
}
```

The HTTP response includes the 401 status code, and the error payload that I defined in my `basic_auth_error()` function. Here is the same request, this time including basic authentication credentials:

```
(venv) $ http --auth <username>:<password> POST http://localhost:5000/api/tokens
HTTP/1.0 200 OK
Content-Length: 50
Content-Type: application/json
Date: Mon, 27 Nov 2017 20:01:22 GMT
Server: Werkzeug/0.12.2 Python/3.6.3

{
    "token": "pC1Nu9wwyNt8VCj1trWilFdFI276AcbS"
}
```

Now the status code is 200, which is the code for a successful request, and the payload includes a newly generated token for the user. Note that when you send this request you will need to replace `<username>:<password>` with your own credentials. The username and password need to be provided with a colon as separator.

Protecting API Routes with Tokens

The clients can now request a token to use with the API endpoints, so what's left is to add token verification to those endpoints. This is something that Flask-HTTPAuth can also handle for me. I need to create a second authentication instance based on the `HTTPTokenAuth` class, and provide a token verification callback:

app/api/auth.py: Token authentication support.

```
# ...
from flask_httpauth import HTTPTokenAuth

# ...
token_auth = HTTPTokenAuth()

# ...

@token_auth.verify_token
def verify_token(token):
    return User.check_token(token) if token else None

@token_auth.error_handler
def token_auth_error(status):
    return error_response(status)
```

When using token authentication, Flask-HTTPAuth uses a `verify_token` decorated function, but other than that, token authentication works in the same way as basic authentication. My token verification function uses `User.check_token()` to locate the user that owns the provided token and return it. As before, a `None` return causes the client to be rejected.

To protect API routes with tokens, the `@token_auth.login_required` decorator needs to be added:

app/api/users.py: Protect user routes with token authentication.

```
from flask import abort
from app.api.auth import token_auth

@bp.route('/users/<int:id>', methods=['GET'])
@token_auth.login_required
def get_user(id):
    # ...

@bp.route('/users', methods=['GET'])
@token_auth.login_required
def get_users():
    # ...

@bp.route('/users/<int:id>/followers', methods=['GET'])
@token_auth.login_required
def get_followers(id):
    # ...

@bp.route('/users/<int:id>/followed', methods=['GET'])
@token_auth.login_required
def get_followed(id):
    # ...

@bp.route('/users', methods=['POST'])
def create_user():
    # ...

@bp.route('/users/<int:id>', methods=['PUT'])
@token_auth.login_required
def update_user(id):
    if token_auth.current_user().id != id:
        abort(403)
    # ...
```

Note that the decorator is added to all the API view functions except `create_user()`, which obviously cannot accept authentication since the user that will request the token needs to be created first. Also note how the `PUT` request that modifies users has an additional check that prevents a user from trying to modify another user's account. If I find that the requested user id does not match the id of the authenticated user, then I return a 403 error response, which indicates that the client does not have permission to carry out the requested operation.

If you send a request to any of these endpoints as shown previously, you will get back a 401 error response. To gain access, you need to add the `Authorization` header, with a token that you received from a request to */api/tokens*. Flask-HTTPAuth expects the token to be sent as a "bearer" token, which isn't directly supported by HTTPie. For basic authentication with username and password, HTTPie offers a `--auth` option, but for tokens the header needs to be explicitly provided. Here is the syntax to send the bearer token:

```
(venv) $ http GET http://localhost:5000/api/users/1 \
    "Authorization:Bearer pC1Nu9wwyNt8VCj1trWilFdFI276AcbS"
```

Revoking Tokens

The last token related feature that I'm going to implement is the token revocation, which you can see below:

app/api/tokens.py: Revoke tokens.

```
from app.api.auth import token_auth

@bp.route('/tokens', methods=['DELETE'])
@token_auth.login_required
def revoke_token():
    token_auth.current_user().revoke_token()
    db.session.commit()
    return '', 204
```

Clients can send a `DELETE` request to the */tokens* URL to invalidate the token. The authentication for this route is token based, in fact the token sent in the `Authorization` header is the one being revoked. The revocation itself uses the helper method in the `User` class, which resets the expiration date on the token. The database session is committed so that this change is written to the database. The response from this request does not have a body, so I can return an empty string. A second value in the return statement sets the status code of the response to 204, which is the code to use for successful requests that have no response body.

Here is an example token revocation request sent from HTTPie:

```
(venv) $ http DELETE http://localhost:5000/api/tokens \
    Authorization:"Bearer pC1Nu9wwyNt8VCj1trWilFdFI276AcbS"
```

API Friendly Error Messages

Do you recall what happened early in this chapter when I asked you to send an API request from the browser with an invalid user URL? The server returned a 404 error, but this error was formatted as the standard 404 HTML error page. Many of the errors the API might need to return can be overriden with JSON versions in the API blueprint, but there are some errors handled by Flask that still go through the error handlers that are globally registered for the application, and these continue to return HTML.

The HTTP protocol supports a mechanism by which the client and the server can agree on the best format for a response, called *content negotiation*. The client needs to send an `Accept` header with the request, indicating the format preferences. The server then looks at the list and responds using the best format it supports from the list offered by the client.

What I want to do is modify the global application error handlers so that they use content negotiation to reply in HTML or JSON according to the client preferences. This can be done using the `request.accept_mimetypes` object from Flask:

app/errors/handlers.py: Content negotiation for error responses.

```
from flask import render_template, request
from app import db
from app.errors import bp
from app.api.errors import error_response as api_error_response

def wants_json_response():
    return request.accept_mimetypes['application/json'] >= \
        request.accept_mimetypes['text/html']

@bp.app_errorhandler(404)
def not_found_error(error):
    if wants_json_response():
        return api_error_response(404)
    return render_template('errors/404.html'), 404

@bp.app_errorhandler(500)
def internal_error(error):
    db.session.rollback()
    if wants_json_response():
        return api_error_response(500)
    return render_template('errors/500.html'), 500
```

The `wants_json_response()` helper function compares the preference for JSON or HTML selected by the client in their list of preferred formats. If JSON rates higher than HTML, then I return a JSON response. Otherwise I'll return the original HTML responses based on templates. For the JSON responses I'm going to import the `error_response` helper function from the API blueprint, but here I'm going to rename it to `api_error_response()` so that it is clear what it does and where it comes from.