THE UNIVERSITY OF WESTERN AUSTRALIA

Topic 11: Authentication

CITS3403 Agile Web Development

Reading: The Flask Mega-Tutorial
Miguel Grinberg
Chapter 5

Semester 1, 2023

Secure web apps

THE UNIVERSITY OF WESTERN AUSTRALIA

- Security is a primary concern for anyone developing web applications.
- Data access must be controlled, passwords must be validated securely, and users just be able to trust the information presented to them.
- Complete security is very hard to achieve and beyond the scope of this unit, but basic authentication is relatively easy.
- An interesting case study of internet security is *Anonymous*'s attack on HBGary:

arstechnica.com/tech-policy/2011/02/anonymous-speaks-the-inside-story-of-the-hbgary-hack/

Web security basics

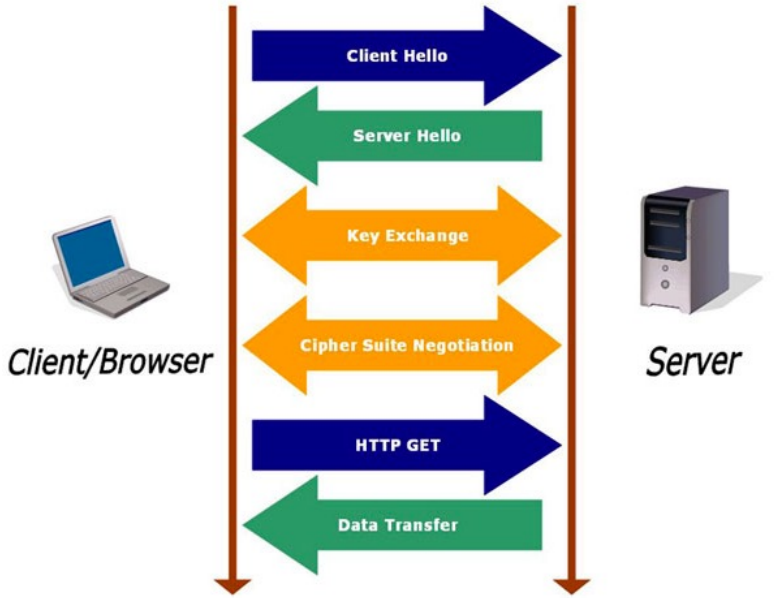
THE UNIVERSITY OF WESTERN AUSTRALIA

Web security makes use of the following basic concepts

- Public Key Encryption (eg RSA)
  - A public-private key is 2 functions *pub* and *priv* so that *x* = *priv(pub(x))* and given that you know *pub*, *priv* is hard to work out.
  - Public Key Encryption can be used for authentication. I can compute and publish *pub(x)* and only someone who knows *priv* can tell me what *x* is.
  - Public Key Encryption can be used for digital signatures. The pair *(x, priv(x))* can be verified by anyone, but only created by some who knows *priv.*
  - Key distribution. A random key *x* can be generated and *pub(x)* can be sent to someone who knows *priv*. Then the pair knows *x,* but no body else does (even if they have been eaves dropping
- Hashing (eg MD5)
  - Secure hashing computes a large number from a stream of data, in such a way that it's very difficult to fabricate data with a certain hash.
  - Different to hashing used for Hash tables etc.

Secure web session

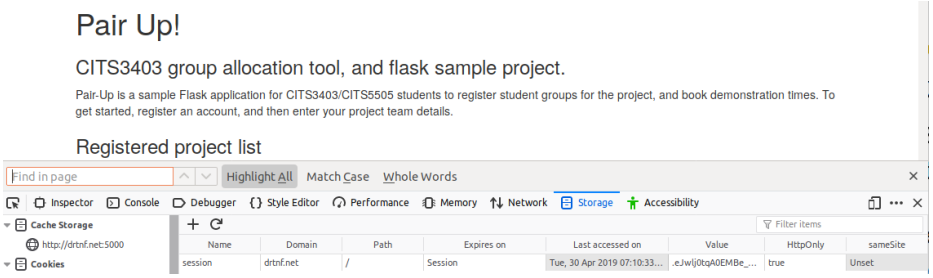
- HTTP is stateless, so the server does not remember the client.
- For a secure session, every request needs to be authenticated... thankfully there are protocols to help here.
- SSL (secure sockets layer) wraps up the public key encryption process to enable a secure transaction.
- To use SSL we need to use the HTTPS protocol, which requires a signed certificate to allows users to trust the server.
- This prevents anyone from intercepting traffic from reading its contents.



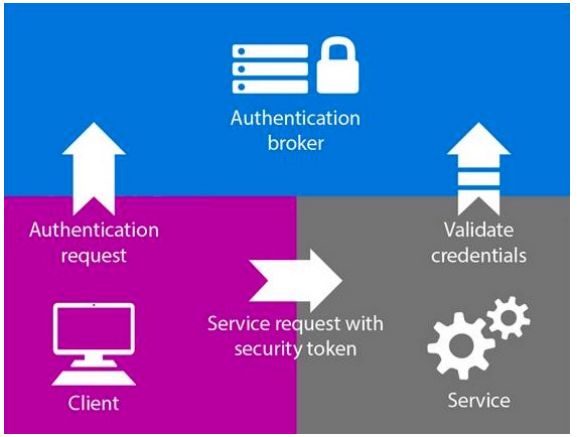
Cookies and Tokens

- Web session security is managed through cookies and tokens.
- Cookies are packets of data stored in the browser.
  - Session cookies can record a users interaction with a site, persistent remain in your browser and allow sites to track your browsing habits.
  - Cookies consist of a name, a value and a set of attribute value pairs (e.g. expiration).
  - Cookies can be created and managed through javascript: `document.cookie="trackme: false";`
  - Cookies are sent from the server to the browser:

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: theme=light
Set-Cookie: sessionToken=abc123; Expires=Wed, 09 Jun 2021 10:18:14 GMT
…
```

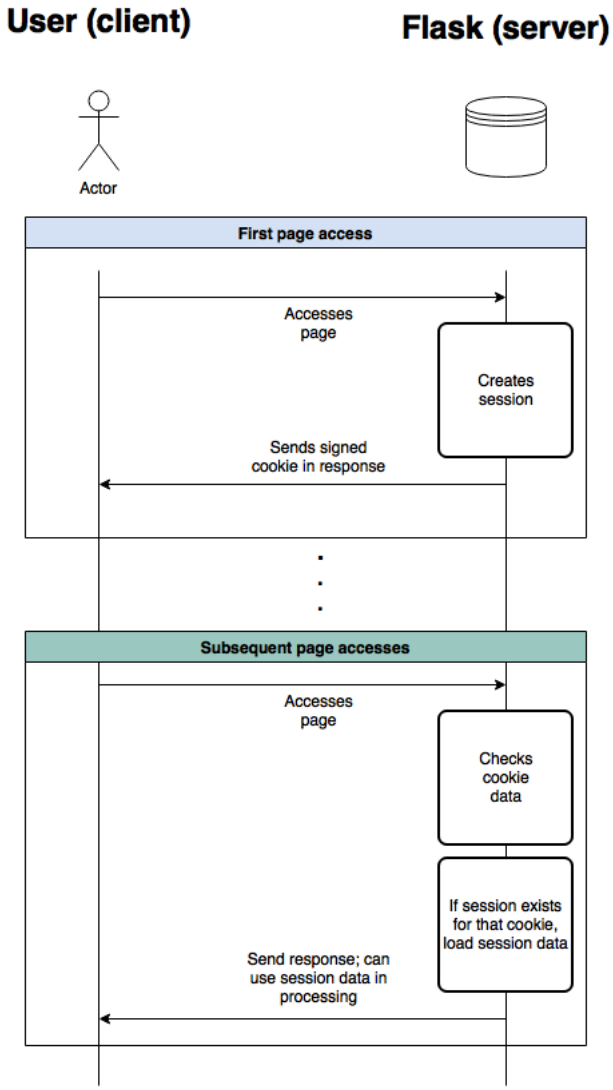


- Authentication tokens allow you to store user privaleges in JWT, (JSON web tokens), or other formats.
- These tokens, once granted are submitted with web-requests to verify identities.



Authentication and session management

- To manage users, a database can store user data and password hashes. Unverified users are required to enter login details before a secure session commences.
- When we verify a users credentials against the database, the application can remember that all requests associated with that session come from the specified user.
- Therefore HTTP is stateless, but the application is not, and it can track the state (and authority) of every user that is logged in.
- Flask provides some basic tools for session management.



Elements of Web-security

THE UNIVERSITY OF WESTERN AUSTRALIA

- Web security depends on trust. There are several elements to this:
  1. The web server needs to be confident that someone accessing data is authorised.
  2. The user needs to know that the site they are visiting is the one they intend to.
  3. Both the server and the client need to be confident that no one in the middle is accessing unauthorised data.
- 2 is typically handled by browsers, and 3 is achieved with https (week 11). In this lecture we'll focus on 1.
- To track a users identity we need to have them register so we can associate a user name with them.
- When someone uses an application a *session* is maintained via a variable held by the web-browser.
- When someone logs in they provide a password. This is salted and hashed to provide a digest which can compared to a hash in a database (keeping the password secure).
- Once the user is authenticated, they will be be served there requested pages, and their id will be a parameter of the requests.

Adding authentication to Flask apps

- In previous lectures we have looked at the MVC architecture, and linked in a simple database which contains a table of users.
- To add in authentication, we need every user to have a unique user id and a password, but we only want to store password hashes.
- The python package *werkzeug* (a part of flask) can handle the hashing.

```
>>> from werkzeug.security import generate_password_hash
>>> hash = generate_password_hash('foobar')
>>> hash
'pbkdf2:sha256:50000$vT9fkZM8$04dfa35c6476acf7e788a1b5b3c35e217c78dc04539d295f011f01f18cd2175f'
```

```
>>> from werkzeug.security import check_password_hash
>>> check_password_hash(hash, 'foobar')
True
>>> check_password_hash(hash, 'barfoo')
False
```

app/models.py: Password hashing and verification

```
from werkzeug.security import generate_password_hash, check_password_hash

# ...

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

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)
```

- The password management can now be added to the User model, using werkzeug to generate and verify hashes.

Flask Login Manager

- `Flask-Login` is a package that will automatically track secure sessions. It requires the User model to implement a number of methods and properties for checking if the user is authenticated, what their id is, etc.
- This functionality can be achieved by using the `UserMixin`, which implements those methods for you.
- As Flask-Login is agnostic to the database or ORM, we need to tell flask how to load a user.
- The decorator `@login.user_loader` is for the method that maps an id to a user.

```
from app import db, login
from werkzeug.security import generate_password_hash, check_password_hash
from flask_login import UserMixin

#allows login to get student from database, given id
#will be stored as current_user?
@login.user_loader
def load_student(id):
  return Student.query.get(int(id))

#student database will be prepopulated with
# student numbers, firstname, surname, CITS3403 boolean
#students can add/edit pin and project id
class Student(UserMixin, db.Model):
  __tablename__='students'
  id = db.Column(db.String(8), primary_key = True)#prepopulate
  first_name = db.Column(db.String(64))#prepopulate
  surname = db.Column(db.String(64))#prepopulate
  prefered_name = db.Column(db.String(64))#defaults to first_name if empty
  cits3403 = db.Column(db.Boolean)#prepopulate
  password_hash = db.Column(db.String(128))#overkill to hash a four digit
pin, but included for learning.
  project_id = db.Column(db.Integer, db.ForeignKey('projects.project_id'),
nullable=True) #assigned when project registered

#  def registered(number):
#    student = Student.query.filter_by(number = id.data).first()
#    if student.password_hash is not None:
#        raise validationError('Student is already registered')
#    return True

  def set_password(self, password):
    self.password_hash = generate_password_hash(password)

  def check_password(self, password):
    return check_password_hash(self.password_hash, password)
```

Using Flask-Login

- `Flask-Login` provides a methods `login_user`, `logout_user` and a variable `current_user` (possibly anonymous) to manage sessions.
- `login_user` will set current user to the specified user model.
- `current_user` has a method `is_authenticated` to check if they have provided login credentials.
- We also use a decorator `@login_required` from `Flask-Login` to label the routes that require a login.
- Finally, in `app/__init__.py`, an instance of `LoginManager` is created, and the `login_view` is set to the route login.

```python
class StudentController():

  def login():
    form = LoginForm()
    if form.validate_on_submit(): #will return false for a get request
      student = Student.query.filter_by(id=form.student_number.data).first()
      if student is None or not student.check_password(form.pin.data):
        flash('invalid username or data')
        return redirect(url_for('login'))
      login_user(student, remember=form.remember_me.data)
      next_page = request.args.get('next')
      if not next_page or url_parse(next_page).netloc !='':
        next_page = 'index'
      return redirect(url_for(next_page))
    return render_template('login.html',title="Sign in", form = form)

  def logout():
    logout_user()
    return redirect(url_for('index'))
```

```python
from flask import render_template, flash, redirect, url_for
from app import app, db
from flask_login import current_user, login_user, logout_user, login_required
from app.controllers import StudentController, ProjectController
from flask import request
from werkzeug.urls import url_parse

@app.route('/login', methods=['GET','POST'])
def login():
  if not current_user.is_authenticated:
    return StudentController.login()
  return redirect(url_for('index'))

@app.route('/logout')
def logout():
  return StudentController.logout()

@app.route('/register', methods=['GET','POST'])
def register():
  return StudentController.register()


@app.route('/new_project', methods=['GET','POST'])
@login_required
def new_project():
  if not current_user.is_authenticated:
    return redirect(url_for('login'))
  return ProjectController.new_project()
```

```python
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from flask_login import LoginManager

app = Flask(__name__)
app.config.from_object('config.TestingConfig')
db = SQLAlchemy(app)
migrate = Migrate(app, db)
login = LoginManager(app)
login.login_view = 'login'

from app import routes,models
```

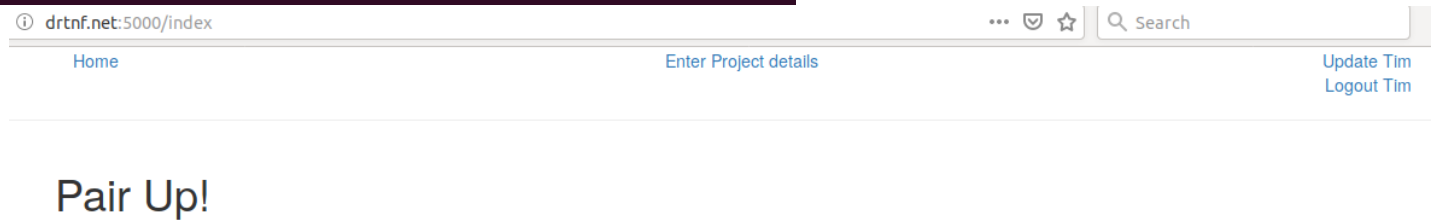
app/__init__.py

Updating the views

- We can now use the current_user variable in the templates we have built.
- The current_user properties can be used to guard components of the web page that you only want logged in users to see, or to personalise the web page.

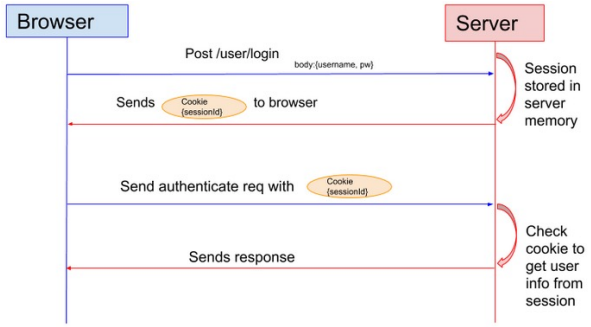
```
<h3>Registered project list</h3>
<table class='table table-striped table-bordered'>
  <tr>
    <th>Project Team</th>
    <th>Project Description</th>
    <th>Demo location</th>
    <th>Demo time</th>
    {% if not current_user.is_anonymous %}
      <th>Action</th>
    {% endif %}
  </tr>
  {% for p in projects%}
    <tr>
      <td>{{p['team']}}</td>
      <td>{{p['description']}}</td>
      <td>{{p['lab']}}</td>
      <td>{{p['time']}}</td>
      {% if not current_user.is_anonymous %}
        <td>
        {% if p['project_id']== current_user.project_id %}
          <a href='{{url_for("delete_project") }}'><span class="glyphicon glyphicon.garbage">delete</span></a>
          <a href='{{ url_for("edit_project") }}'><span class="glyphicon glyphicon.pencil">edit</span></a>
        {% endif %}
        </td>
      {% endif %}
  </tr>
  {% endfor %}
</table>
```

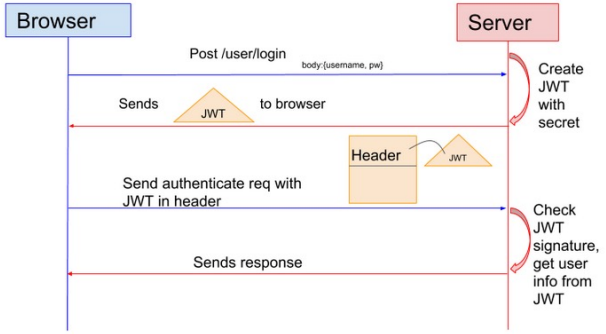
```
<body>
  <div class='container'>
    <div class='col-sm-4'>
    <a href='{{ url_for("index") }}'>Home</a>
    </div>
    {%if current_user.is_anonymous %}
    <div class='col-sm-4'></div>
    <div class='col-sm-4 text-right'>
      <a href='{{ url_for("login") }}'>Login</a><br/>
      <a href='{{ url_for("register") }}'>Register</a>
    </div>
    {% else %}
      {% if current_user.project_id == None %}
        <div class='col-sm-4 text-center'>
          <a href='{{ url_for("new_project") }}'>Enter Project details</a>
        </div>
      {% else %}
        <div class='col-sm-4 text-center'>
          <a href='{{ url_for("edit_project") }}'>Edit Project details</a>
        </div>
      {% endif %}
        <div class='col-sm-4 text-right'>
          <a href='{{ url_for("register") }}'>Update {{ current_user.prefered_name }}
</a><br/>
          <a href='{{ url_for("logout") }}'>Logout {{ current_user.prefered_name }}</
a>
        </div>
    {%endif %}
  </div>
  <hr>
  {% with messages = get_flashed_messages() %}
  {% if messages %}
  <ul>
    {% for message in messages %}
    <li> {{ message }} </li>
    {% endfor %}
  </ul>
  {% endif %}
  {% endwith %}
  <div class='container'>
  <h1>Pair Up!<h1>
      <h3>CITS3403 group allocation tool, and flask sample project.</h3>

      {% block content%}
      {% endblock %}
```



Alternative authentication methods

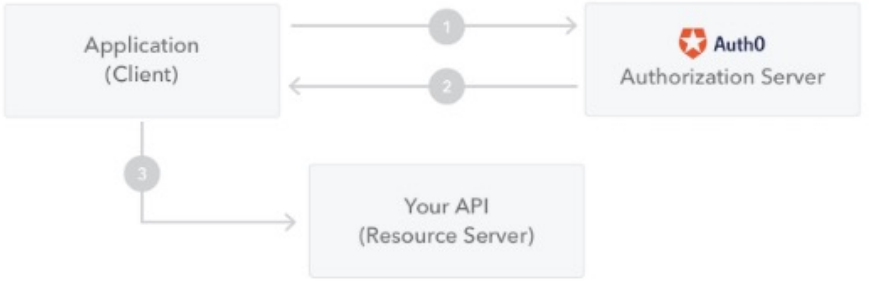
- This type of authentication works well for web based sessions, but has a number of drawbacks.
  - It requires the application to track all user sessions which may not scale well.
  - HTTP requests are sent in plain text, and passwords should never be transmitted or stored in plain text.
  - The web is not only access through a browser, so how can we authenticate without sessions?
- JSON Web Tokens provide an alternative where a token is granted to a client, and the client must submit that token with every request.
- An added bonus is that using OAuth, external providers can check and grant tokens.





OAuth and JWT

- OAuth (now OAuth2) was developped by Twitter to allow applications to authenticate and interact with Twitter, without requiring repeated logins.
- Oauth verifies a users identitiy and provides a JSON Web Token (JWT). This contains a header, a payload and a signature in compressed JSON.
- This header describes the encryption type, the payload typically provides some user identifier, an expiry and issuer information, and the signature is a secure hash, proving the token wasn't tampered with.
- You can configure flask to serve JWT tokens to clients, and verify those tokens, rather than checking session cookies with the flask_oauth module.
- However, you can also have 3rd parties like Google and Twitter, provide the tokens and do the validation.
- This saves you having to manage senesitive user data.



1. The application or client requests authorization to the authorization server. This is performed through one of the different authorization flows. For example, a typical OpenID Connect compliant web application will go through the `/oauth/authorize` endpoint using the authorization code flow.
2. When the authorization is granted, the authorization server returns an access token to the application.
3. The application uses the access token to access a protected resource (like an API).

Authentication with passwords and tokens

- The web application used session based authentication, but there is no such session cookie for a REST API.
- Instead a token is granted to the user when they provide credentials, and requests augmented with that token user are assumed to come from the user.
- g is a context object that comes with each HTTP request

```
1  from flask import g
2  from flask_httpauth import HTTPBasicAuth
3  from app.models import Student
4  from app.api.errors import error_response
5  from flask_httpauth import HTTPTokenAuth
6
7  basic_auth = HTTPBasicAuth()
8  token_auth = HTTPTokenAuth()
9
10 #password required for granting tokens
11 @basic_auth.verify_password
12 def verify_password(student_number, pin):
13   student = Student.query.get(student_number)
14   if student is None:
15     return Flase
16   g.current_user = student
17   return student.check_password(pin)
18
19 @basic_auth.error_handler
20 def basic_auth_error():
21   return error_response(401)
22
23 #token auth below
24 @token_auth.verify_token
25 def verify_token(token):
26   g.current_user = Student.check_token(token) if token else None
27   return g.current_user is not None
28
29 @token_auth.error_handler
30 def token_auth_error():
31   return error_response(401)
32
app/api/auth.py                                    1,1
```

Authentication with passwords and tokens

THE UNIVERSITY OF WESTERN AUSTRALIA

- The HTTPBasicAuth module is for verifying passwords in a request, which will grant a token.
- Then the HTTPTokenAuth can do token based authentication
- We need to update our User models so that the temporary token is kept in the database, as well as the password hash.
- When making changes to the models, remember to upgrade and migrate the changes to the database

```
26   project_id = db.Column(db.Integer, db.ForeignKey('projects.project_id'),nul
   lable=True)
27   #token authetication for api
28   token = db.Column(db.String(32), index=True, unique = True)
29   token_expiration=db.Column(db.DateTime)
30
31   def set_password(self, password):
32     self.password_hash = generate_password_hash(password)
33
34   def check_password(self, password):
35     return check_password_hash(self.password_hash, password)
36
37   ###Token support methods for api
38
39   def get_token(self, expires_in=3600):
40     now = datetime.utcnow()
41     if self.token and self.token_expiration > now + timedelta(seconds=60):
42       return self.token
43     self.token = base64.b64encode(os.urandom(24)).decode('utf-8')
44     self.token_expiration = now+timedelta(seconds=expires_in)
45     db.session.add(self)
46     return self.token
47
48   def revoke_token(self):
49     self.token_expiration = datetime.utcnow() - timedelta(seconds=1)
50
51   @staticmethod
52   def check_token(token):
53     student = Student.query.filter_by(token=token).first()
54     if student is None or student.token_expiration < datetime.utcnow():
55       return None
56     return student
57
app/models.py                                          54,1          13%
```

Interacting with the REST API

- To interact with a REST API, you can use a browser for GET requests, but others are not trivial
- The python package `HTTPie` can be used to send requests and receive responses.
- There are also graphical user interfaces, such as Postman for sending, receiving and testing in HTTP

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

```
(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"
}
```

```
(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": "pClNu9wwyNt8VCj1trWilFdFI276AcbS"
}
```

```
(venv) $ http GET http://localhost:5000/api/users/1 \
    "Authorization:Bearer pClNu9wwyNt8VCj1trWilFdFI276AcbS"
```