**A quick tutorial about server-side using webapp2**

**User account management**

The main interface when dealing with authentication is webapp2\_extras.auth. This module leverages the rest of webapp2’s infrastructure to offer us a simple way to manage user authentication.

**The default User model**

In particular, it relies on:

* **Webapp2\_extras.security** to handle password hashing (so that passwords are never stored in clear text) and random string generation.
* **Webapp2\_extras.sessions** to identify requests coming from the same user as part of a client-server conversation.

Since we’re implementing our own user database (which includes a lot of extra fields, such as province, city, phone), we’ll need to define a custom model class to represent users in our app. The auth framework described here works under the assumption that this model class defines the following instance method:

* Get\_id(self)
  + As well as the following class methods (all built-in except for the last one):
    - Get\_by\_auth\_token(cls, user\_id, token)
      * Returns a user object based on a User\_id and a token
    - Get\_by\_auth\_password(cls, auth\_id, password)
      * Returns a user object, validating password
    - Create\_auth\_token(cls, user\_id)
      * Creates a new authentication token for a given user id
    - Delete\_auth\_token(cls, user\_id, token)
      * Delete a given token
    - Get\_by\_auth\_token(cls, user\_Id, token, subject=’auth’)
      * For password reset

**Extending the default User model**

Webapp2 already contains a reference User model for GAE that uses NDB for storage. It is an Expando model – that means it can include properties that were not specified as part of the class definition but are added at run-time, so we can add more specific user info for our app. Queries should still be fast.The following is our customized User model (not complete).

*class* User(Webapp2User):  
  
 first\_name = ndb.StringProperty(required=True)  
 phone1 = ndb.StringProperty(required=True)  
 phone2 = ndb.StringProperty(required=True)  
 phone3 = ndb.StringProperty(required=True)  
 province = ndb.StringProperty(required=True)  
 city = ndb.StringProperty(required=True)

Since the User class is clearly defined, so I used the first four methods that’s already provided in webapp2, and only implemented Get\_by\_auth\_token(cls, user\_Id, token, subject=’auth’) in user.py like below:

@classmethod  
*def* get\_by\_auth\_token(cls, *user\_id*, *token*, *subject*='auth'):  
 """Returns a user object based on a user ID and token  
  
 :param user\_id:  
 The user\_id of the requesting user  
 :param token:  
 The token string to be verified  
 :returns  
 A tuple ``(User, timestamp)``, with a user object and  
 the token timestamp, or ``(None, None)`` if both were not found.  
 """  
  
 token\_key = cls.token\_model.get\_key(*user\_id*, *subject*, *token*)  
 user\_key = ndb.Key(cls, *user\_id*)  
 # Use get\_multi() to save a RPC call  
 valid\_token, user = ndb.get\_multi([token\_key, user\_key])  
 *if* valid\_token *and* user:  
 timestamp = int(time.mktime(valid\_token.created.timetuple()))  
 *return* user, timestamp  
  
 *return* None, None

**Setting up the configuration**

We’ll need to set some properties as follows on the bottom of main.py

config = {  
 'webapp2\_extras.auth': {  
 'user\_model': User,  
 'user\_attributes': ['first\_name', 'phone1', 'phone2', 'phone3', 'province', 'city']  
 },  
 'webapp2\_extras.sessions':{  
 'secret\_key': 'YOUR\_SECRET\_KEY'  
 }  
}

* User\_model: the name of the customized User model class we described earilier
* User\_attributes: a list of attributes in the User model that will be cached in the session, ideally, frequently accessed properties should be stored here. The full User model will be accessible by querying the datastore.
* Secret\_Key: the key used to secure the hash signature calculation for session cookies.

**Creating a base handler class**

RequestHandler is the default handler used in webapp2. It has built-in methods such as dispatch(), error(), abort(), redirect() for request handling.

Before writing the actual handlers that will implement the business logic sign up and authentication users, we will group some utility functions in a base handler class, which will be extended by all the following handler classes.

This will ensure that all handlers will inherit a set of useful utility functions and properties to access user data and infrastructure classes, but also ensures that all session data is properly saved on each request.

*class* BaseHandler(webapp2.RequestHandler):  
 @webapp2.cached\_property  
 *def* auth(self):  
 """Shortcut to access the auth instance as a property"""  
 *return* auth.get\_auth()  
  
 @webapp2.cached\_property  
 *def* user\_info(self):  
 """Shortcut to access a subset of the user attributes that are stored  
 in the session  
  
 The list of attributes to store in the session is specified in  
 config['webapp2\_extras.auth']['user\_attributes'].  
 :returns  
 A dictionary with most user information  
 """  
 *return* self.auth.get\_user\_by\_session()  
  
 @webapp2.cached\_property  
 *def* user(self):  
 """Shortcut to access the current logged-in user.  
  
 Unlike user\_info, it fetches information from the persistence layer and  
 returns an instance of the underlying model.  
 :returns  
 The instance of the user model associated to the signed-in user.  
 """  
 user = self.user\_info  
 *return* self.user\_model.get\_by\_id(user['user\_id']) *if* user *else* None  
  
 @webapp2.cached\_property  
 *def* user\_model(self):  
 """Return the implementation of the user model.  
  
 It is consistent with config['webapp2\_extras.auth']['user\_model'], if set.  
 """  
 *return* self.auth.store.user\_model  
  
 @webapp2.cached\_property  
 *def* session(self):  
 """Shortcut to access the current session."""  
 *return* self.session\_store.get\_session(backend="datastore")  
  
 *def* render\_template(self, view\_filename, *params*=None):  
 *if not params*:  
 params = {}  
 user = self.user\_info  
 *params*['user'] = user  
 path = os.path.join(os.path.dirname(\_\_file\_\_), view\_filename)  
 self.response.out.write(template.render(path, *params*))  
  
 *def* display\_message(self, *message*):  
 """Utility function to display a template with a simple message"""  
 params = {  
 'message': *message* }  
 self.render\_template('message.html', params)  
  
 # this is needed for webapp2 sessions to work  
 *def* dispatch(self):  
 # get a session for webapp2 sessions to work  
 self.session\_store = sessions.get\_store(request=self.request)  
 *try*:  
 # Dispatch the request  
 webapp2.RequestHandler.dispatch(self)  
 *finally*:  
 # Save all sessions  
 self.session\_store.save\_sessions(self.response)

Note: think of @webapp2.cached\_property as some property that needs to be saved for next use.

**Registration: create new users**

To create new users, we can use the create\_user() method that is built-in in webapp2.

*class* CreateUser(BaseHandler):  
 *def* get(self):  
 self.render\_template('create\_user.html')  
  
 *def* post(self):  
 first\_name = self.request.get('firstName')  
 last\_name = self.request.get('lastName')  
 email = self.request.get('email')  
 password = self.request.get('password')  
 phone1 = self.request.get('phone1')  
 phone2 = self.request.get('phone2')  
 phone3 = self.request.get('phone3')  
 province = self.request.get('province')  
 city = self.request.get('city')  
  
 unique\_properties = ['email\_address']  
 user\_data = self.user\_model.create\_user(email, unique\_properties, email\_address=email,  
 first\_name=first\_name, password\_raw=password, phone1=phone1, phone2=phone2, phone3=phone3,  
 province=province, city=city, last\_name=last\_name, verified=False)  
 *if not* user\_data[0]: # user\_data is a tuple  
 self.display\_message('Unable to create user for email %s because of \  
 duplicate keys %s' % (email, user\_data[1]))  
 *return* user = user\_data[1]  
 user\_id = user.get\_id()  
  
 token = self.user\_model.create\_signup\_token(user\_id)  
  
 verification\_url = self.uri\_for('verification', type='v', user\_id=user\_id,  
 signup\_token=token, \_full=True)  
  
 msg = 'Send an email to user in order to verify their address. \  
 They will be able to do so by visiting <a href="{url}">{url}</a>'  
  
 self.display\_message(msg.format(url=verification\_url))

Note: GET and POST are two methods that are often used in sending requests. They are very similar. But usually we are using GET to get some data/property we need back to us, while POST is mostly used for sending such requests as updating the data in the database. So in this case, create\_user is more of type POST, because we want to send all the info about this new user and create a new record in the database. The GET method here is pretty much just get the html page to show the login screen ( in the browser).

Create\_user will accept the following parameters:

* An authentication id: the token (in our case, email) users will use to identify themselves when trying to access our app. While users can have multiple authentication ids, only one is allowed at creation.
* A list of unique properties: (in our case, email) if it is specified, webapp2 will not allow us to create new users if others share the same values for the properties in this list.

unique\_properties = ['email\_address']

* Name/value pairs that are gping to be set as properties of the resulting User model.

**Login and logout**

The get\_user\_by\_password() method can be used to retrieve a user by their credentials. In addition to the user credentials, the method accepts some additional parameters. The one we care about is **remember**: when set to True, the cookie used to identify the session is saved as persistent and the browser will keep it even after the user will close its window.

*class* SignIn(BaseHandler):  
  
 *def* get(self):  
 self.\_serve\_page()  
  
 *def* post(self):  
 # d = json.loads(self.request.body)  
 # user\_email = d['email']  
 # password = d['password']  
 user\_email = self.request.get('email')  
 password = self.request.get('password')  
 *try*:  
 u = self.auth.get\_user\_by\_password(user\_email, password, remember=True,  
 save\_session=True)  
 # self.redirect(self.uri\_for('home'))  
 # self.response.out.write(d)  
 *except* (InvalidAuthIdError, InvalidPasswordError) *as* e:  
 logging.info('Sign-in failed for user %s because of %s', user\_email, type(e))  
 self.\_serve\_page(True)  
 # listings = Listing.query(Listing.lister\_email == user\_email).fetch()  
  
  
 *def* \_serve\_page(self, *failed*=False):  
 user\_email = self.request.get('email')  
 params = {  
 'user\_email': user\_email,  
 'failed': *failed* }  
 self.render\_template('sign\_in.html', params)

The implementation above renders the login page when the request comes via GET and processes the credentials upon POST. When authentication fails it renders the login page and passes the username to the template so that the corresponding field can be pre-filled.

Signout is simplier: it is sufficient to get rid of the user session.

*class* LogoutHandler(BaseHandler):  
 *def* get(self):  
 self.auth.unset\_session()  
 self.redirect(self.uri\_for('home'))

**Email confirmation and password reset**

Signup tokens are one of the undocumented features of webapp2, but they can be quite handy when implementing a flow to confirming email addresses or recover passwords.

The webapp2 uses authentication tokens to identify users after they logged in: they are meant to be securely by a client and the server and exchanged when a client needs to prove its identity.

This mechanism can be generalized to handle email confirmations and password resets: when websites send us an activation link after a registration, the url usually contain their equivalent of signup tokens.

Webapp2 sets a **subject** property for each of the tokens it generates, so the only difference between auth token and signup token is the value for that property. So setting a different value for that property allows us to partition tokens by their purpose: we can then implement useful features as deleting all the password reset tokens that have not been used in 48 hours.

This is a verification handler that is able to process email verification links:

*class* VerificationHandler(BaseHandler):  
 *def* get(self, *\*args*, *\*\*kwargs*):  
 user = None  
 user\_id = *kwargs*['user\_id']  
 signup\_token = *kwargs*['signup\_token']  
 verification\_type = *kwargs*['type']  
  
 user, ts = self.user\_model.get\_by\_auth\_token(int(user\_id), signup\_token, 'signup')  
  
 *if not* user:  
 logging.info('Could not find any user with id "%s" signup token "%s"',  
 user\_id, signup\_token)  
 self.abort(404)  
  
 # store user data in the session  
 self.auth.set\_session(self.auth.store.user\_to\_dict(user), remember=True)  
  
 *if* verification\_type == 'v': #remove signup token, we don't want users to come back with an old link  
 self.user\_model.delete\_signup\_token(user.get\_id(), signup\_token)  
  
 *if not* user.verified:  
 user.verified = True  
 user.put()  
  
 self.display\_message('User email address has been verified.')  
 *return  
 elif* verification\_type == 'p':  
 # supply user to the page  
 params = {  
 'user': user,  
 'token': signup\_token  
 }  
 self.render\_template('resetpassword.html', params)  
 *else*:  
 logging.info('verification type not supported.')  
 self.abort(404)

Ideally, we may want to use a different subject for email confirmation and password reset tokens.

**Ensure users are logged in**

We need to decide whether uses are allowed to access certain resources depending on if they’re logged in or not.

The following decorator can be used to annotate handler methods that require user logged in.

*def* user\_required(*handler*):  
 """  
 Decorator that checks if there's a user associated with the current session.  
 Will also fail if there's no session present.  
 """  
 *def* check\_signin(*self*, *\*args*, *\*\*kwargs*):  
 auth = *self*.auth  
 *if not* auth.get\_user\_by\_session():  
 *self*.redirect(*self*.uri\_for('signin'), abort=True)  
 *else*:  
 *return* handler(*self*, \**args*, \*\**kwargs*)  
  
 *return* check\_signin

Just placing @user\_required above the function header will ensure that anonymous users will be directed to a login page when attempting to go through the annotated handler method.

*class* AuthenticatedHandler(BaseHandler):  
 @user\_required  
 *def* get(self):  
 self.render\_template('authenticated.html')

**Listings**

**Show all the listings for a user**

Now it should be pretty straightforward to deal with listings. All the listings that belongs to a specific user would bound with the user email. (More fields in listing should be added later on.). The GET method currently get all the listings of the user.

*class* ShowListings(webapp2.RequestHandler):  
 *def* get(self):  
 #***TODO: Remove testing account, should pass in user email as parameter*** user = User.query().get()  
 listings = Listing.query(Listing.lister\_email == user.email\_address).fetch()  
 path = os.path.join(os.path.dirname(\_\_file\_\_), 'show\_listings.html')  
 *for* listing *in* listings:  
 template\_values = {  
 'lister\_email': listing.lister\_email,  
 'bedrooms': listing.bedrooms,  
 'sqft': listing.sqft,  
 'bathrooms': listing.bathrooms,  
 'price': listing.price,  
 'description': listing.description,  
 'isPublished': listing.isPublished,  
 'province': listing.province,  
 'city': listing.city,  
 'images': listing.images #***TODO: store image URLs from blobstore to a list*** }  
 self.response.out.write(template.render(path, template\_values))

**Create a new listing**

The GET method is simply get the html page ( in the browser for back-end testing) for user inputs. The POST method is similar to create\_user, what it does is to get all the information from what the user typed, and generate a new listing that belongs to the current user (with email as the key).

*class* CreateListing(webapp2.RequestHandler):  
 *def* get(self):  
 template\_values = {  
 # 'first\_name': user2.first\_name,  
 # 'last\_name': user2.last\_name  
 }  
 # test = json.dumps(template\_values)  
 path = os.path.join(os.path.dirname(\_\_file\_\_), 'create\_listing.html')  
 self.response.out.write(template.render(path, template\_values))  
  
 *def* post(self):  
  
 #***TODO: Remove testing account*** user = User.query().get()  
 lister\_email = user.email\_address  
 bedrooms = int(self.request.get('bedrooms'))  
 sqft = int(self.request.get('sqft'))  
 bathrooms = int(self.request.get('bathrooms'))  
 price = int(self.request.get('price'))  
 description = self.request.get('description')  
 isPublished = self.request.get('isPublished') != ''  
 province = self.request.get('province')  
 city = self.request.get('city')  
 images = self.request.get('images')  
  
 listing = Listing(lister\_email=lister\_email, bedrooms=bedrooms, sqft=sqft, bathrooms=bathrooms,  
 price=price, description=description, isPublished=isPublished, province=province,  
 city=city, images=images)  
 key = Listing.build\_key(lister\_email, bedrooms, sqft, bathrooms, price, description, province, city)  
 listing.key = key  
 listing.put()  
  
 self.response.out.write('<h1>New listing created!</h1>')