```
from datetime import datetime
from flask import Blueprint, render_template, abort, request, redirect
from flask import url_for
from metasrht.audit import audit_log
from metasrht.auth import allow_registration, user_valid, prepare_user
from metasrht.auth import is_external_auth, set_user_password, set_user_email
from metasrht.auth.builtin import hash_password, check_password
from metasrht.auth_validation import validate_password
from metasrht.blueprints.security import metrics as security_metrics
from metasrht.blueprints.profile import change_email_GET
from metasrht.graphql import Client, GraphQLClientGraphQLMultiError
from metasrht.totp import totp
from metasrht.types import User, UserType
from metasrht.types import UserAuthFactor, FactorType, PGPKey
from metasrht.webhooks import UserWebhook
from prometheus_client import Counter
from srht.app import csrf_bypass, session
from srht.config import cfg, get_global_domain, config, get_origin
from srht.database import db
from srht.graphql import InternalAuth, has_error
from srht.oauth import current_user, login_user, logout_user
from srht.validation import Validation
from urllib.parse import urlparse

try:
    # This file is kept private to prevent spammers from reading it to
    # understand how to circumvent our spam prevention mechanisms.
    with open("/etc/abuse.py") as f:
        try:
            exec(f.read())
        except Exception as ex:
            print("Error loading abuse.py", ex)
            raise
except:
    def is_abuse(valid):
        return False

auth = Blueprint('auth', __name__)

origin = cfg("meta.sr.ht", "origin")
owner_name = cfg("sr.ht", "owner-name")
owner_email = cfg("sr.ht", "owner-email")
site_name = cfg("sr.ht", "site-name")
onboarding_redirect = cfg("meta.sr.ht::settings", "onboarding-redirect")
site_key_id = cfg("mail", "pgp-key-id", None)

metrics = type("metrics", tuple(), {
    c.describe()[0].name: c
    for c in [
        Counter("meta_registrations", "Number of new user registrations"),
        Counter("meta_confirmations", "Number of account confirmations"),
        Counter("meta_logins_failed", "Number of failed logins"),
        Counter("meta_logins_success", "Number of successful logins"),
        Counter("meta_logouts", "Number of sessions logged out"),
        Counter("meta_pw_resets", "Number of password resets completed"),
    ]
})

# Returns return_to if it points within the SouceHut installation,
# "/" otherwise.
def validate_return_url(return_to):
    if return_to and return_to[0] == "/":
        # Relative path => valid.
        return return_to
    parsed = urlparse(return_to)
    if parsed.hostname is None:
        # Unparsable host in an absolute URL => invalid.
        return "/"
    # Absolute URLs are only valid if they point within the global domain...
    global_domain = cfg("sr.ht", "global-domain", None)
    if global_domain and parsed.hostname.endswith(global_domain):
```

```
72          return return_to
73      # ... or a configured sr.ht service.
74      srht_sites = [ s for s in config if s.endswith(".sr.ht") ]
75      for site in srht_sites:
76          origin = get_origin(site, True, None)
77          if origin and return_to.startswith(origin):
78              return return_to
79      return "/"
80
81  def issue_reset(user):
82      client = Client(InternalAuth.anonymous())
83      client.request_password_reset(user.email)
84      return render_template("forgot.html", done=True)
85
86  @auth.route("/")
87  def index():
88      if current_user:
89          return redirect(url_for("profile.profile_GET"))
90      return render_template("index.html")
91
92  @auth.route("/register")
93  def register():
94      if current_user:
95          return redirect("/")
96      if cfg("meta.sr.ht::billing", "enabled") != "yes":
97          return redirect(url_for("auth.register_step2_GET"))
98      return render_template("register.html", site_key=site_key_id)
99
100 @auth.route("/register", methods=["POST"])
101 def register_POST():
102     is_open = allow_registration()
103
104     valid = Validation(request)
105     payment = valid.require("payment")
106     if not valid.ok:
107         abort(400)
108     payment = payment == "yes"
109     session["payment"] = payment
110
111     return redirect(url_for("auth.register_step2_GET"))
112
113 @auth.route("/register/step2")
114 def register_step2_GET():
115     payment = session.get("payment", "no")
116     if current_user:
117         return redirect("/")
118     return render_template("register-step2.html",
119             site_key=site_key_id, payment=payment)
120
121 @auth.route("/register/step2", methods=["POST"])
122 def register_step2_POST():
123     if current_user:
124         abort(400)
125     is_open = allow_registration()
126     payment = session.get("payment", False)
127
128     valid = Validation(request)
129     username = valid.require("username", friendly_name="Username")
130     email = valid.require("email", friendly_name="Email address")
131     password = valid.require("password", friendly_name="Password")
132     pgp_key = valid.optional("pgpKey", default=None)
133     if not pgp_key:
134         pgp_key = None
135
136     if not valid.ok:
137         return render_template("register-step2.html", is_open=is_open,
138                 site_key=site_key_id, payment=payment, **valid.kwargs), 400
139
140     if is_abuse(valid):
141         return redirect("/registered")
142
143     allow_plus_in_email = valid.optional("allow-plus-in-email")
```

```
144     if "+" in email and allow_plus_in_email != "yes":
145         return render_template("register-step2.html", is_open=is_open,
146                 site_key=site_key_id, payment=payment, **valid.kwargs), 400
147 
148     client = Client(InternalAuth.anonymous())
149     with valid:
150         client.register_account(email, username, password, pgp_key)
151 
152     if not valid.ok:
153         return render_template("register-step2.html", is_open=is_open,
154                 site_key=site_key_id, payment=payment, **valid.kwargs), 400
155 
156     metrics.meta_registrations.inc()
157     return redirect("/registered")
158 
159 @auth.route("/registered")
160 def registered():
161     return render_template("registered.html")
162 
163 @auth.route("/confirm-account/<token>")
164 def confirm_account(token):
165     client = Client(InternalAuth.anonymous())
166 
167     try:
168         resp = client.confirm_registration(token)
169     except GraphQLClientGraphQLMultiError as err:
170         if has_error(err, "ERR_INVALID_EMAIL_TOKEN"):
171             abort(404)
172         raise
173 
174     user = User.query.get(resp.user.id)
```

```
175         assert user is not None
176         login_user(user, set_cookie=True)
177 
178         metrics.meta_confirmations.inc()
179 
180         payment = session.pop("payment", False)
181         if payment and cfg("meta.sr.ht::billing", "enabled") == "yes":
182             return redirect(url_for("billing.billing_setup_GET"))
183         else:
184             return redirect(onboarding_redirect)
185 
186 @auth.route("/login")
187 def login_GET():
188     if current_user:
189         return redirect("/")
190     return_to = request.args.get('return_to')
191     context = session.get("login_context")
192     return render_template("login.html",
193             return_to=return_to,
194             login_context=context)
195 
196 def get_challenge(factor):
197     if factor.factor_type == FactorType.totp:
198         return redirect("/login/challenge/totp")
199     abort(500)
200 
201 @auth.route("/login", methods=["POST"])
202 def login_POST():
203     if current_user:
204         return redirect("/")
205     valid = Validation(request)
206 
207     username = valid.require("username", friendly_name="Username")
208     password = valid.require("password", friendly_name="Password")
209     return_to = valid.optional("return_to", "/")
210 
211     if not valid.ok:
212         return render_template("login.html", **valid.kwargs), 400
213 
214     user_valid(valid, username, password)
215 
216     if not valid.ok:
217         metrics.meta_logins_failed.inc()
218         print(f"{datetime.utcnow()} Login attempt failed for {username}")
219         return render_template("login.html",
220             username=username,
221             valid=valid)
222 
223     user = prepare_user(username)
224     valid.expect(user.user_type != UserType.pending,
225             "Your account is pending confirmation. Please check your inbox, or reach out to support if you did not
226 receive an email.")
227     valid.expect(user.user_type != UserType.suspended,
228             f"Your account is suspended: {user.suspension_notice}. Contact support.")
229     if not valid.ok:
230         return render_template("login.html", **valid.kwargs), 400
231 
232     factors = (UserAuthFactor.query
233         .filter(UserAuthFactor.user_id == user.id)).all()
234 
235     session.pop("login_context", None)
236     if any(factors):
237         session['extra_factors'] = [f.id for f in factors]
238         session['authorized_user'] = user.id
239         session['challenge_type'] = 'login'
240         session['return_to'] = return_to
241         return get_challenge(factors[0])
242 
243     login_user(user, set_cookie=True)
244     print("session_login = True")
245     session["session_login"] = True
246     audit_log("logged in")
```

```
247         print(f"Logged in account: {user.username} ({user.email})")
248         db.session.commit()
249         metrics.meta_logins_success.inc()
250         return_to = validate_return_url(return_to)
251         return redirect(return_to)
252
253     @auth.route("/login/challenge/totp")
254     def totp_challenge_GET():
255         user = session.get('authorized_user')
256         if not user:
257             return redirect("/login")
258         challenge_type = session.get('challenge_type')
259         return render_template("totp-challenge.html", challenge_type=challenge_type)
260
261     @auth.route("/login/challenge/totp", methods=["POST"])
262     def totp_challenge_POST():
263         user_id = session.get('authorized_user')
264         factors = session.get('extra_factors')
265         challenge_type = session.get('challenge_type')
266         return_to = session.get('return_to') or '/'
267         if not user_id or not factors:
268             return redirect("/login")
269         valid = Validation(request)
270
271         code = valid.require("code")
272         if not valid.ok:
273             return render_template("totp-challenge.html",
274                 return_to=return_to, valid=valid)
275
276         code = code.replace(" ", "")
277         try:
278             code = int(code)
279         except:
280             valid.error(
281                     "This TOTP code is invalid (expected a number)", field="code")
282         if not valid.ok:
283             return render_template("totp-challenge.html",
284                 return_to=return_to, valid=valid)
285
286         factor = UserAuthFactor.query.get(factors[0])
287         secret = factor.secret.decode('utf-8')
288
289         valid.expect(totp(secret, code),
290                 'The code you entered is incorrect.', field='code')
291
292         user = User.query.get(user_id)
293         if not valid.ok:
294             print(f"{challenge_type} attempt failed (TOTP) for " +
295                 f"{user.username} ({user.email})")
296             return render_template("totp-challenge.html",
297                 valid=valid, return_to=return_to)
298
299         factors = factors[1:]
300         if len(factors) != 0:
301             return get_challenge(UserAuthFactor.query.get(factors[0]))
302
303         session.pop('authorized_user', None)
304         session.pop('extra_factors', None)
305         session.pop('challenge_type', None)
306         session.pop('return_to', None)
307
308         if challenge_type == "login":
309             login_user(user, set_cookie=True)
310             session["session_login"] = True
311             audit_log("logged in")
312             print(f"Logged in account: {user.username} ({user.email})")
313             db.session.commit()
314             metrics.meta_logins_success.inc()
315             return_to = validate_return_url(return_to)
316             return redirect(return_to)
317         elif challenge_type == "reset":
318             return issue_reset(user)
```

```
319         elif challenge_type == "disable_totp":
320             db.session.delete(factor)
321             audit_log("Disable TOTP", details="Disabled two-factor authentication",
322                     email=True, subject=f"TOTP has been disabled for your {cfg('sr.ht', 'site-name')} account",
323                     email_details="2FA via TOTP was disabled")
324             db.session.commit()
325             security_metrics.meta_totp_disabled.inc()
326             return redirect(return_to)
327         else:
328             raise NotImplemented
329
330     @auth.route("/login/challenge/totp-recovery")
331     def totp_recovery_GET():
332         user = session.get('authorized_user')
333         if not user:
334             return redirect("/login")
335         factors = session.get('extra_factors')
336         factor = UserAuthFactor.query.get(factors[0])
337         supported = factor.extra is not None
338         return render_template("totp-recovery.html", supported=supported)
339
340     @auth.route("/login/challenge/totp-recovery", methods=["POST"])
341     def totp_recovery_POST():
342         user_id = session.get('authorized_user')
343         factors = session.get('extra_factors')
344         challenge_type = session.get('challenge_type')
345         return_to = session.get('return_to') or '/'
346         if not user_id or not factors:
347             return redirect("/login")
348         valid = Validation(request)
349
350         code = valid.require('recovery-code')
351         if not valid.ok:
352             return render_template("totp-recovery.html",
353                 return_to=return_to, supported=True, **valid.kwargs)
354
355         factor = UserAuthFactor.query.get(factors[0])
356         is_valid = False
357         for h in factor.extra:
358             if check_password(code, h):
359                 is_valid = True
360                 break
361         valid.expect(is_valid, "Incorrect recovery code", field="recovery-code")
362         if not valid.ok:
363             return render_template("totp-recovery.html",
364                 return_to=return_to, supported=True, **valid.kwargs)
365
366         user = User.query.get(user_id)
367
368         db.session.delete(factor)
369         audit_log("TOTP recovery code used", user=user, email=True,
370                 subject=f"A recovery code was used for your {cfg('sr.ht', 'site-name')} account",
371                 email_details="Two-factor authentication recovery code used")
372         session["notice"] = "TOTP has been disabled for your account."
373         db.session.commit()
374
375         factors = factors[1:]
376         if len(factors) != 0:
377             return get_challenge(UserAuthFactor.query.get(factors[0]))
378
379         session.pop('authorized_user', None)
380         session.pop('extra_factors', None)
381         session.pop('return_to', None)
382         session.pop('challenge_type', None)
383
384         if challenge_type == "login":
385             login_user(user, set_cookie=True)
386             session["session_login"] = True
387             audit_log("logged in")
388             print(f"Logged in account: {user.username} ({user.email})")
389             db.session.commit()
390             metrics.meta_logins_success.inc()
```

```
391             return_to = validate_return_url(return_to)
392             return redirect(return_to)
393         elif challenge_type == "reset":
394             return issue_reset(user)
395         elif challenge_type == "disable_totp":
396             security_metrics.meta_totp_disabled.inc()
397             return redirect(return_to)
398         else:
399             raise NotImplemented
400
401 @auth.route("/logout")
402 def logout():
403     if current_user:
404         audit_log("logged out")
405         logout_user()
406         db.session.commit()
407         metrics.meta_logouts.inc()
408     if request.args.get("return_to"):
409         return_to = validate_return_url(request.args["return_to"])
410         return redirect(return_to)
411     return redirect("/login")
412
413 @auth.route("/forgot")
414 def forgot_GET():
415     return render_template("forgot.html")
416
417 @auth.route("/forgot", methods=["POST"])
418 def forgot_POST():
419     valid = Validation(request)
420     email = valid.require("email", friendly_name="Email")
421     if not valid.ok:
422         return render_template("forgot.html", **valid.kwargs)
423     user = User.query.filter(User.email == email).first()
424     valid.expect(user, "No account found with this email address.")
425     valid.expect(not user or user.user_type != UserType.admin,
426             "You can't reset the password of an admin.")
427     valid.expect(not user or user.user_type != UserType.pending,
428             f"Your account has not been confirmed. Please contact support via {cfg('sr.ht', 'owner-email')} if you
429 did not receive a confirmation email.")
430     if not valid.ok:
431         return render_template("forgot.html", **valid.kwargs)
432
433     factors = (UserAuthFactor.query
434         .filter(UserAuthFactor.user_id == user.id)).all()
435     if any(factors):
436         session['extra_factors'] = [f.id for f in factors]
437         session['authorized_user'] = user.id
438         session['challenge_type'] = 'reset'
439         return get_challenge(factors[0])
440
441     return issue_reset(user)
442
443 @auth.route("/reset-password/<token>")
444 def reset_GET(token):
445     return render_template("reset.html")
446
447 @auth.route("/reset-password/<token>", methods=["POST"])
448 def reset_POST(token):
449     valid = Validation(request)
450     password = valid.require("password", friendly_name="Password")
451     if not valid.ok:
452         return render_template("reset.html", valid=valid)
453     validate_password(valid, password)
454     if not valid.ok:
455         return render_template("reset.html", valid=valid)
456
457     client = Client(InternalAuth.anonymous())
458
459     try:
460         resp = client.confirm_password_change(token, password)
461     except GraphQLClientGraphQLMultiError:
462         return render_template("reset.html", invalid_token=True)
```

```
463
464     user = User.query.get(resp.user.id)
465     assert user is not None
466
467     session["session_login"] = True
468     session["notice"] = "Your password has been changed."
469     login_user(user, set_cookie=True)
470
        metrics.meta_pw_resets.inc()
        return redirect("/")
```