# timApp/messaging/MIT-license.txt

```
Copyright 2021 Hannamari Heiniluoma, Kristian KÃ¤yhty,
               Tomi Lundberg and Tuuli Veini

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

# timApp/messaging/init.py

# timApp/messaging/messagelist/init.py

# timApp/messaging/messagelist/emaillist.py

```python
from dataclasses import dataclass
from typing import List, Optional
from urllib.error import HTTPError

from mailmanclient import Client, MailingList, Domain, Member

from timApp.messaging.messagelist.listoptions import ListOptions, mailman_archive_policy_correlate, ArchiveType, \
    ReplyToListChanges
from timApp.messaging.messagelist.messagelist_models import MessageListModel
from timApp.tim_app import app
from timApp.user.user import User
from timApp.util.flask.requesthelper import NotExist, RouteException
from timApp.util.logger import log_warning, log_info, log_error
from tim_common.marshmallow_dataclass import class_schema


@dataclass
class MailmanConfig:
    MAILMAN_URL: Optional[str]
    MAILMAN_USER: Optional[str]
    MAILMAN_PASS: Optional[str]
    MAILMAN_UI_LINK_PREFIX: Optional[str]

    def __bool__(self) -> bool:
        return bool(self.MAILMAN_URL and self.MAILMAN_USER and self.MAILMAN_PASS and self.MAILMAN_UI_LINK_PREFIX)


```

```
config: MailmanConfig = class_schema(MailmanConfig)().load(app.config, unknown="EXCLUDE")
_client = Client(config.MAILMAN_URL, config.MAILMAN_USER, config.MAILMAN_PASS) if config else None
"""A client object to utilize Mailman's REST API. If this is None, mailmanclient-library has not been configured for
use. """

# Test mailmanclient's initialization on TIM's boot up.
if not _client:
    log_warning("No mailman configuration found, no email support will be enabled.")
else:
    log_info("Mailman connection configured.")


def log_mailman(err: HTTPError, optional_message: str = "") -> None:
    """Log potentially troublesome Mailman activity when mailmanclient raises HTTPErrors.

    mailmanclient library raises liburl HTTPError messages for non 2xx status code responses for backward
    compatibility. Some of these indicate actual trouble that should be investigated, and some are just information
    about the operation (e.g. subscribe() method for MailinList objects raises HTTPError with a code 409 if the
    address we are trying to subscribe already exists on the list).

    The general idea is to use log_error() for 5Xxx status codes and log_warning() for 4xx status codes. Technically
    3xx codes should not happen for TIM, but they get log_info().

    :param err: The HTTPError to be logged.
    :param optional_message: Optional, additional message for logging.
    """
    code_category = err.code // 100
    # If an optional message is given, it sets a slightly different format for the message.
    if optional_message:
        log_message = f"Mailman log: {optional_message}; Mailman returned status code {err}"
    else:
        log_message = f"Mailman log: Mailman returned status code {err}"
    # Check for 5xx, 4xx and 3xx status codes, they are logged with different severity.
    if code_category == 5:
        log_error(log_message)
    elif code_category == 4:
        log_warning(log_message)
    elif code_category == 3:
        log_info(log_message)
    else:
        log_warning(f"Mailman log: Error that should not have happened, happened: {err}")


def verify_mailman_connection() -> None:
    """Verifies if the connection to Mailman is possible. Aborts if connection is not possible due to connection not
    being configured in the first place. Used for operations that are meaningless without configured connection to
```

```
74      Mailman. """
75      if not _client:
76          raise NotExist("No connection to Mailman configured.")
77
78
79  def check_mailman_connection() -> bool:
80      """Checks if the connection to Mailman is possible. Used for operations where the ↩
        ability to connect to Mailman
81      is optional, and other meaningful operations can resume in case the connection is ↩
        not available.
82
83      :return: True if connection is possible, i.e. _client is configured. Otherwise ↩
        False.
84      """
85      if not _client:
86          return False
87      return True
88
89
90  def set_notify_owner_on_list_change(mlist: MailingList, on_change_flag: bool) -> None:
91      """Set email list's notify owner on list change change flag.
92
93      :param mlist: Email list where the flag is set.
94      :param on_change_flag: For True, set the notification flag on and then the ↩
        changes on a list will send
95      notifications from Mailman. For False, set the flag off and then the email list ↩
        will not send notifications
96      from Mailman.
97      """
98      try:
99          mlist.settings["admin_notify_mchanges"] = on_change_flag
100         mlist.settings.save()
101     except HTTPError as e:
102         log_mailman(e, "In set_notify_owner_on_list_change()")
103         raise
104
105
106 def delete_email_list(list_to_delete: MailingList, permanent_deletion: bool = False) ↩
        -> None:
107     """Delete a mailing list.
108
109     :param permanent_deletion: If True, then the list is permanently gone. If False, ↩
        perform a soft deletion.
110     :param list_to_delete: MailingList object of the email list to be deleted.
111     """
112     try:
113         if permanent_deletion:
114             list_to_delete.delete()
115         else:
116             # Soft deletion.
117             freeze_list(list_to_delete)
118     except HTTPError as e:
119         log_mailman(e, "In delete_email_list()")
120         raise
121
122
123 def remove_email_list_membership(member: Member, permanent_deletion: bool = False) -> ↩
        None:
124     """Remove membership from an email list.
```

```
125
126     :param member: The membership to be terminated on a list.
127     :param permanent_deletion: If True, unsubscribes the user from the list ↩
        permanently. If False, membership is
128     "deleted" in a soft manner by removing delivery and send rights, conforming to ↩
        TIM's policy in deleting objects.
129     Membership is kept, but emails from member aren't automatically let through nor ↩
        does the member receive mail from
130     the list.
131     """
132     try:
133         if permanent_deletion:
134             member.unsubscribe()
135         else:
136             set_email_list_member_send_status(member, False)
137             set_email_list_member_delivery_status(member, False)
138     except HTTPError as e:
139         log_mailman(e, "In remove_email_list_membership()")
140         raise
141
142
143 def set_default_templates(email_list: MailingList) -> None:
144     """Set default templates for email list.
145
146     Sometimes Mailman gets confused how mail is supposed to be interpreted, and with ↩
        some  email client's different
147     interpretation with Mailman's email coding templates (e.g. list information ↩
        footers) may appear as attachments.
148     We fix it by setting header and footer for all new lists explicitly.
149
150     :param: email_list: The email list we are setting templates for.
151     """
152
153     # Build the URI in case because not giving one is interpreted by Mailman as ↩
        deleting the template form the list.
154     try:
155
156         list_id = email_list.rest_data['list_id']
157         template_base_uri = "http://localhost/postorius/api/templates/list/" \
158                             f"{list_id}"
159         footer_uri = f"{template_base_uri}/list:member:regular:footer"
160         header_uri = f"{template_base_uri}/list:member:regular:header"
161
162         # These templates don't actually exist, but non-existing templates are ↩
        substituted with empty strings and that
163         # should still fix the broken coding leading to attachments issue.
164         email_list.set_template("list:member:regular:footer", footer_uri)
165         email_list.set_template("list:member:regular:header", header_uri)
166     except HTTPError as e:
167         log_mailman(e, "In set_default_templates()")
168         raise
169
170
171 def set_email_list_archive_policy(email_list: MailingList, archive: ArchiveType) -> ↩
        None:
172     """Set email list's archive policy.
173
174     :param email_list: Email list
175     :param archive: What type of archiving is set for message list, and what that ↩
```

```
    means for an email list.
176     """
177     try:
178         mlist_settings = email_list.settings
179         mm_policy = mailman_archive_policy_correlate[archive]
180         mlist_settings["archive_policy"] = mm_policy
181         mlist_settings.save()  # This needs to be the last line, otherwise changes ↩
    won't take effect.
182     except HTTPError as e:
183         log_mailman(e, "In set_email_list_archive_policy()")
184         raise
185
186
187 def create_new_email_list(list_options: ListOptions, owner: User) -> None:
188     """Create a new email list with proper initial options set.
189
190     :param owner: Who owns this list.
191     :param list_options: Options for message lists, here we use the options necessary ↩
    for email list creation.
192     :return:
193     """
194     if list_options.domain:
195         verify_name_availability(list_options.name, list_options.domain)
196     else:
197         log_warning("Tried to create an email list without selected domain part.")
198         raise RouteException("Tried to create an email list without selected domain ↩
    part.")
199
200     try:
201         domain: Domain = _client.get_domain(list_options.domain)
202         email_list: MailingList = domain.create_list(list_options.name)
203
204         set_default_templates(email_list)
205
206         if list_options.list_description:
207             set_email_list_description(email_list, list_options.list_description)
208         if list_options.list_info:
209             set_email_list_info(email_list, list_options.list_info)
210
211         # settings-attribute acts like a dict. Set default settings.
212         mlist_settings = email_list.settings
213
214         # Make sure lists aren't advertised by accident by defaulting to not ↩
    advertising them. Owner switches
215         # advertising on if they so choose.
216         mlist_settings["advertised"] = False
217         # Ownerss / moderators don't get automatic notifications from changes on ↩
    their message list. Owner switches
218         # this on if necessary.
219         mlist_settings["admin_notify_mchanges"] = False
220         # Turn off automatic welcome and goodbye messages.
221         mlist_settings["send_welcome_message"] = False
222         mlist_settings["send_goodbye_message"] = False
223         # Set content filtering on, so lists can set pass_extensions on and off. ↩
    Because allowing attachments is not
224         # on by default, add pass_extensions value to block attachments.
225         mlist_settings["filter_content"] = True
226         mlist_settings["pass_extensions"] = ['no_extension']
227
```

```
228         # This is to force Mailman generate archivers into its db. It fixes a race ↩
    condition, where creating a new list
229         # without proper engineer interface procedures might make duplicate archiver ↩
    rows in to db, while Mailman's code
230         # expects there to be only one archiver row (which results in the db code ↩
    breaking and the list becoming
231         # unusable, at least without manual db fixing). This might be unnecessary at ↩
    some point in time in the future if
232         # the condition is remedied in Mailman Core, but since this line is only ↩
    needed once on list creation it might
233         # be good enough to just leave as is.
234         _ = dict(email_list.archivers)
235 
236         set_email_list_archive_policy(email_list, list_options.archive)
237 
238         # This needs to be the last line aften changing settings, because no changes ↩
    to settings take effect until
239         # save() method is called.
240         mlist_settings.save()
241 
242         # All lists created through TIM need an owner, and owners need email ↩
    addresses to control their lists on
243         # Mailman.
244         email_list.add_owner(owner.email)
245         # Add owner automatically as a member of a list, so they receive the posts on ↩
    the list.
246         email_list.subscribe(owner.email, display_name=owner.real_name, ↩
    pre_approved=True, pre_verified=True,
247                              pre_confirmed=True)
248     except HTTPError as e:
249         log_mailman(e, "In create_new_email_list()")
250         raise
251 
252 
253 def get_list_ui_link(listname: str, domain: Optional[str]) -> Optional[str]:
254     """Get a link for a list to use for advanced email list options and moderation.
255 
256     The function assumes that Mailman uses Postorius as its web-UI. There exists no ↩
    guarantee that other web-UIs would
257     use the exact form for their links. If Postorius is changed to some other web-UI, ↩
    this needs to be updated.
258 
259     :param listname: The list we are getting the UI link for.
260     :param domain: Domain for the list.
261     :return: A Hyperlink for list web UI on the Mailman side. If there is no email ↩
    list for (parameter domain is None)
262     then return None. Return None if no connection to Mailman is configured.
263     """
264     try:
265         if domain is None or not config.MAILMAN_UI_LINK_PREFIX:
266             return None
267         if _client is None:
268             return None
269         mail_list = _client.get_list(f"{listname}@{domain}")
270         # Get the list's list id, which is basically its address/name but '@' ↩
    replaced with a dot.
271         list_id: str = mail_list.rest_data["list_id"]
272         # Build the hyperlink.
273         link = f"{config.MAILMAN_UI_LINK_PREFIX}{list_id}"
```

```
274         return link
275     except HTTPError as e:
276         log_mailman(e, "In get_list_ui_link()")
277         raise RouteException("Connection to Mailman failed while getting list's UI ↩
        link.")
278
279
280 def set_email_list_description(mlist: MailingList, new_description: str) -> None:
281     """Set mailing list's (short) description.
282
283     :param mlist: Email list we wish to set description for.
284     :param new_description: A new (short) description for the email list.
285     """
286     try:
287         mlist.settings["description"] = new_description
288         mlist.settings.save()
289     except HTTPError as e:
290         log_mailman(e, "In set_email_list_description()")
291         raise
292
293
294 def set_email_list_info(mlist: MailingList, new_info: str) -> None:
295     """Set email list's info, A.K.A. long description.
296
297     :param mlist: Email list where the info is set.
298     :param new_info: New info for the email list.
299     """
300     try:
301         mlist.settings["info"] = new_info
302         mlist.settings.save()
303     except HTTPError as e:
304         log_mailman(e, "In set_email_list_info()")
305         raise
306
307
308 def get_email_list_by_name(list_name: str, list_domain: str) -> MailingList:
309     """Get email list by name.
310
311     :param list_name: List's name.
312     :param list_domain: A domain we use to search an email list.
313     :return: Return email list as an MailingList object.
314     """
315     try:
316         mlist = _client.get_list(fqdn_listname=f"{list_name}@{list_domain}")
317         return mlist
318     except HTTPError as e:
319         log_mailman(e, "In get_email_list_by_name()")
320         raise
321
322
323 def add_email(mlist: MailingList, email: str, email_owner_pre_confirmation: bool, ↩
        real_name: Optional[str],
324               send_right: bool = True, delivery_right: bool = False) -> None:
325     """Add a new email to a email list.
326
327     :param mlist: Email list where a new member is being added.
328     :param email: The email being added. Email address has to be validated before ↩
        calling this function.
329     :param email_owner_pre_confirmation: Whether the email's owner has to confirm ↩
```

```
    them joining an email list. For True,
330     no confirmation is needed by the email's owner. For False, Mailman send's a confirmation mail for email's owner to
331     join the list.
332     :param real_name: Name associated with the email.
333     :param send_right: Whether email can send mail to the list. For True, then can send messages and for False they
334     can't.
335     :param delivery_right: Whether email list delivers mail to email. For True, mail is delivered to email. For False,
336     no mail is delivered.
337     """
338     # TODO: The email_owner_pre_confirmation flag was made available for caller with the idea that when inviting to
339     #  message list would be implemented, this would control whether the new user was invited or directly added.
340     #  However, inviting will most likely be implemented with other means than Mailman's invite system, so the flag
341     #  might be unnecessary. When the invite to a messge list is implemented, check if this is still needed or if the
342     #  flag is given the same constant value of True as the other two flags for pre_something.
343
344     # We use pre_verify flag, because we assume email adder knows the address they are adding in. Otherwise they have
345     # to verify themselves for Mailman through an additional verification process. Note that this is different from
346     # confirming to join a list. We use pre_approved flag, because we assume that who adds an email to a list also
347     # wants that email onto a list. Otherwise they would have to afterwards manually moderate their subscription
348     # request.
349     try:
350         new_member = mlist.subscribe(email,
351                                      pre_verified=True,
352                                      pre_approved=True,
353                                      pre_confirmed=email_owner_pre_confirmation,
354                                      display_name=real_name)
355         # Set member's send and delivery rights to email list.
356         set_email_list_member_send_status(new_member, send_right)
357         set_email_list_member_delivery_status(new_member, delivery_right)
358     except HTTPError as e:
359         if e.code == 409:
360             # With code 409, Mailman indicates that the member is already in the list. We assume that a member has
361             # been soft removed previously, and is now re-added to email list. Set send and delivery rights.
362             member = get_email_list_member(mlist, email)
363             set_email_list_member_send_status(member, send_right)
364             set_email_list_member_delivery_status(member, delivery_right)
365         else:
366             log_mailman(e, "In add_email()")
367             raise
368
369
370 def set_email_list_member_send_status(member: Member, status: bool) -> None:
371     """ Change user's send status on an email list. Send right / status is changed by changing the member's
372     moderation status.
```

```
373
374     This function can fail if connection to Mailman is lost.
375
376     :param member: Member who is having their send status changed.
377     :param status: A value that determines if option is 'enabled' or 'disabled'. If ↩
        True, the member can send messages
378     (past moderation) to the email list. If False, then all email from member will be ↩
        rejected.
379     """
380     try:
381         if status:
382             member.moderation_action = "accept"
383         else:
384             member.moderation_action = "reject"
385         member.save()
386     except HTTPError as e:
387         log_mailman(e, "In set_email_list_member_send_status()")
388         raise
389
390
391 def set_email_list_member_delivery_status(member: Member, status: bool, by_moderator: ↩
        bool = True) -> None:
392     """Change email list's member's delivery status on a list.
393
394     This function can fail if connection to Mailman is lost.
395
396     :param member: Member who is having their delivery status changed.
397     :param status: If True, then this member receives email list's messages. If ↩
        false, member does not receive messages
398     from the email list.
399     :param by_moderator: Who initiated the change in delivery right. If True, then ↩
        the change was initiated by a
400     moderator or owner of a list. If False, then the change was initiated by the ↩
        member themselves.
401     """
402     try:
403         if status:
404             member.preferences["delivery_status"] = "enabled"
405         else:
406             if by_moderator:
407                 member.preferences["delivery_status"] = "by_moderator"
408             else:
409                 member.preferences["delivery_status"] = "by_user"
410         member.preferences.save()
411     except HTTPError as e:
412         log_mailman(e, "In set_email_list_member_delivery_status()")
413         raise
414
415
416 def verify_emaillist_name_requirements(name_candidate: str, domain: str) -> None:
417     """Check email list's name requirements. General message list name requirement ↩
        checks are assumed to be passed
418     at this point and that those requirements encompass email list name requirements.
419
420     :param name_candidate: A possible name for an email list name to check.
421     :param domain: Domain where name availability is to be checked.
422     """
423     verify_name_availability(name_candidate, domain)
424     verify_reserved_names(name_candidate)
```

```
425
426
427 def verify_name_availability(name_candidate: str, domain: str) -> None:
428     """Search for a name from the pool of used email list names.
429
430     Raises a RouteException if no connection was ever established with the Mailman ↩
      server via mailmanclient.
431
432     :param domain: Domain to search for lists, which then are used to check name ↩
      availability.
433     :param name_candidate: The name to search for.
434     """
435     try:
436         checked_domain = _client.get_domain(domain)
437         mlists: List[MailingList] = checked_domain.get_lists()
438         fqdn_name_candidate = f"{name_candidate}@{domain}"
439         for name in [mlist.fqdn_listname for mlist in mlists]:
440             if fqdn_name_candidate == name:
441                 raise RouteException("Name is already in use.")
442     except HTTPError as e:
443         log_mailman(e, "In check_name_availability()")
444         raise
445
446
447 def verify_reserved_names(name_candidate: str) -> None:
448     """Check a name candidate against reserved names, e.g. postmaster.
449
450     Raises a RouteException if the name candidate is a reserved name. If name is not ↩
      reserved, the method completes
451     silently.
452
453     If reserved names are not configured, we assume that there are no reserved names.
454
455     :param name_candidate: The name to be compared against reserved names.
456     """
457     reserved_names = app.config.get("RESERVED_NAMES")
458     if reserved_names and name_candidate in reserved_names:
459         raise RouteException(f"Name '{name_candidate}' is a reserved name and cannot ↩
      be used.")
460
461
462 def get_email_list_member(mlist: MailingList, email: str) -> Member:
463     """Get a Member object with an email address from a MailingList object (i.e. from ↩
      an email list).
464
465     :param mlist: MailingList object, the email list in question.
466     :param email: Email used to find the member in an email list.
467     :return: Return a Member object belonging to an email address on an email list.
468     """
469     try:
470         member = mlist.get_member(email)
471         return member
472     except HTTPError as e:
473         log_mailman(e, "In get_email_list_member()")
474         raise
475
476
477 def set_email_list_unsubscription_policy(email_list: MailingList, ↩
      can_unsubscribe_flag: bool) -> None:
```

```
    """Set the unsubscription policy of an email list.

    :param email_list: The email list where the policy is to be set.
    :param can_unsubscribe_flag: For True, then set the policy as 'confirm_then_moderate'. For False, set the policy as
    'confirm'.
    """
    # mailmanclient exposes an 'unsubscription_policy' setting, that follows SubscriptionPolicy enum values, see
    # https://gitlab.com/mailman/mailman/-/blob/master/src/mailman/interfaces/mailinglist.py for details.
    try:
        if can_unsubscribe_flag:
            email_list.settings["unsubscription_policy"] = "confirm"
        else:
            email_list.settings["unsubscription_policy"] = "confirm_then_moderate"
        email_list.settings.save()
    except HTTPError as e:
        log_mailman(e, "In set_email_list_unsubscription_policy()")
        raise


def set_email_list_subject_prefix(email_list: MailingList, subject_prefix: str) -> None:
    """Set the subject prefix for an email list.

    :param email_list: Email list where the subject prefix is to be set.
    :param subject_prefix: The prefix set for email list's subject.
    """
    try:
        email_list.settings["subject_prefix"] = subject_prefix
        email_list.settings.save()
    except HTTPError as e:
        log_mailman(e, "In set_email_list_subject_prefix()")
        raise


def set_email_list_only_text(email_list: MailingList, only_text: bool) -> None:
    """Set email list to only text mode. Affects new email sent to list and HyperKitty archived messages.

    :param email_list: Email list which is to be set into text only mode.
    :param only_text: A boolean flag controlling list rendering mode. For True, the list is in an only text mode.
    For False, the list is not on an only text mode, and other rendering (e.g. HTML) is allowed.
    """
    try:
        email_list.settings["convert_html_to_plaintext"] = only_text
        # The archive_rendering_mode setting mainly has an effect on HyperKitty. If the archiver on Mailman is something
        # else, this might have no effect. It is still exposed on mailmanclient, so it should not be harmful either.
        if only_text:
            email_list.settings["archive_rendering_mode"] = "text"
        else:
            email_list.settings["archive_rendering_mode"] = "markdown"
        email_list.settings.save()
    except HTTPError as e:
```

```
528         log_mailman(e, "In set_email_list_only_text()")
529         raise
530 
531 
532 def set_email_list_allow_nonmember(email_list: MailingList, ↩
    non_member_message_pass_flag: bool) -> None:
533     """Set email list's non member (message pass) action.
534 
535     :param email_list: The email list where the non member message pass action is set.
536     :param non_member_message_pass_flag: For True, set the default non member ↩
    moderation action as 'accept'. For False,
537     set the default non member moderation action as 'hold'
538     """
539     try:
540         if non_member_message_pass_flag:
541             email_list.settings["default_nonmember_action"] = "accept"
542         else:
543             email_list.settings["default_nonmember_action"] = "hold"
544         email_list.settings.save()
545     except HTTPError as e:
546         log_mailman(e, "In set_email_list_non_member_message_pass()")
547         raise
548 
549 
550 def set_email_list_allow_attachments(email_list: MailingList, allow_attachments_flag: ↩
    bool) -> None:
551     """Set email list allowed attachments.
552 
553     :param email_list: The email list where allowed attachment extensions are set.
554     :param allow_attachments_flag: For True, set all the allowed extensions for an ↩
    email list. If False, set the allowed
555      extensions to an empty list.
556     """
557     try:
558         if allow_attachments_flag:
559             email_list.settings["pass_extensions"] = ↩
    app.config.get("PERMITTED_ATTACHMENTS")
560         else:
561             # There might not be a direct option to disallow all attachments to a ↩
    list. We pass a value we don't expect
562             # to find as a file extension. Then the only type of extension that would ↩
    be allowed is file.no_extensions.
563             email_list.settings["pass_extensions"] = ["no_extensions"]
564         email_list.settings.save()
565     except HTTPError as e:
566         log_mailman(e, "In set_email_list_allow_attachments()")
567         raise
568 
569 
570 def set_email_list_default_reply_type(email_list: MailingList, default_reply_type: ↩
    ReplyToListChanges) -> None:
571     """Set the email list's default reply type, i.e. perform Reply-To munging.
572 
573     :param email_list: The email list where the reply type is set.
574     :param default_reply_type: See ReplyToListChanges and reply_to_munging variable.
575     """
576     try:
577         email_list.settings["reply_goes_to_list"] = default_reply_type.value
578         email_list.settings.save()
```

```
    except HTTPError as e:
        log_mailman(e, "In set_email_list_default_reply_type()")
        raise


def get_domain_names() -> List[str]:
    """Returns a list of all domain names, that are configured for our instance of Mailman.

    :return: A list of possible domain names.
    """
    try:
        domains: List[Domain] = _client.domains
        domain_names: List[str] = [domain.mail_host for domain in domains]
        return domain_names
    except HTTPError as e:
        log_mailman(e, "In get_domain_names()")
        raise


def freeze_list(mlist: MailingList) -> None:
    """Freeze an email list. No posts are allowed on the list after freezing (owner might be an exception).

    Think a course specific email list and the course ends, but mail archive is kept intact for later potential use.
    This stops (or at least mitigates) that the mail archive on that list changes after the freezing.

    :param mlist: The list about the be frozen.
    """
    try:
        for member in mlist.members:
            # All members have their send and delivery rights revoked. We need this, because individual members
            # settings, when set, take precedence over list settings.
            set_email_list_member_delivery_status(member, False, by_moderator=True)
            set_email_list_member_send_status(member, False)

        # Also set list's default moderation actions for good measure.
        mail_list_settings = mlist.settings
        mail_list_settings["default_member_action"] = "reject"
        mail_list_settings["default_nonmember_action"] = "reject"
        mail_list_settings.save()
    except HTTPError as e:
        log_mailman(e, "In freeze_list()")
        raise


def unfreeze_list(mlist: MailingList, msg_list: MessageListModel) -> None:
    """The opposite of freezing a list.

    Sets default values for delivery and send status of each member, depending on message lists options.

    :param msg_list: The message list the email list belongs to.
    :param mlist: The list to bring back into function.
    """
    # Not in use at the moment.
```

```
632     try:
633         mail_list_settings = mlist.settings
634         mail_list_settings["default_member_action"] = "accept"
635         set_email_list_allow_nonmember(mlist, msg_list.non_member_message_pass)
636         mail_list_settings.save()
637     except HTTPError as e:
638         log_mailman(e, "In unfreeze_list()")
639         raise
```

# timApp/messaging/messagelist/listoptions.py

```
1  from dataclasses import dataclass, field
2  from datetime import datetime
3  from enum import Enum
4  from typing import Dict, Optional, List
5
6
7  class Channel(Enum):
8      """The message channels TIM uses and provides for message lists."""
9      TIM_MESSAGE = 'tim_message'
10     EMAIL_LIST = 'email_list'
11
12
13 @dataclass
14 class Distribution:
15     """A class to wrap information about the message channels used by a message list ↩
       or TIM users."""
16     tim_message: bool
17     email_list: bool
18
19
20 class ArchiveType(Enum):
21     """Different supported archive types."""
22     # If you change this, make sure the mapping for Mailman's archive policies is ↩
       also updated at
23     # mailman_archive_policy_correlate. TIM's and Mailman's archive policies aren't a ↩
       one-to-one match.
24
25     # No archiving at all for list. Equals to Mailman's archive policy of 'none'.
26     NONE = 0
27     # Secret archive. Only for owner(and moderators?). No direct correlation with ↩
       Mailman's archive policies.
28     SECRET = 1
29     # For group and its members' eyes only. Equal for Mailman's archive policy of ↩
       'private'.
30     GROUPONLY = 2
31     # Logged in TIM users can see the list.
32     UNLISTED = 3
33     # Completely public archive, people don't have to be logged in to see the ↩
       archive. Equals to Mailman's archive
34     # policy of 'public'.
35     PUBLIC = 4
36
37
38 class ReplyToListChanges(Enum):
39     """Options for email list's own address to be added to the Reply-To header for ↩
       emails that are sent through the
40     list. See reply_to_munging for mapping to Mailman's options."""
41     # Don't meddle with the Reply-To header.
```

```
42      NOCHANGES = "no_munging"
43      # Change the Reply-To header so that the message/email list is preferred(/forced) ↩
        to be included in the replies.
44      ADDLIST = "point_to_list"
45
46
47  @dataclass
48  class ListOptions:
49      """All options regarding message lists."""
50      name: str
51      """The name of the message list. A mandatory value when list options are ↩
        considered."""
52
53      archive: ArchiveType = field(metadata={'by_value': True})
54      """The type of archive policy this list uses."""
55
56      default_reply_type: Optional[ReplyToListChanges] = field(metadata={'by_value': ↩
        True}, default=None)
57      """The default reply type of the list."""
58
59      notify_owners_on_list_change: Optional[bool] = None
60      """A flag that determines if owners of the message list are notified of certain ↩
        changes regarding the list,
61      e.g. a new user joins the list. """
62
63      only_text: Optional[bool] = None
64      """If only pure text is allowed on a list."""
65
66      list_description: Optional[str] = None
67      """A short description of the list and its purpose."""
68
69      list_info: Optional[str] = None
70      """Additional information about the list."""
71
72      email_admin_url: Optional[str] = None
73      """If the message list has an email list associated with it, this is the link to ↩
        Mailman's advanced list
74      controls. """
75
76      tim_users_can_join: Optional[bool] = None
77      """Flag used to determine if TIM users can directly join this list."""
78
79      members_can_unsubscribe: Optional[bool] = None
80      """Flag used to determine if the TIM members of this list can leave the list on ↩
        their own."""
81
82      default_send_right: Optional[bool] = None
83      """The list's default send right for (new) members."""
84
85      default_delivery_right: Optional[bool] = None
86      """The list's default delivery right for (new) members."""
87
88      list_subject_prefix: Optional[str] = None
89      """Messages routed by a message list will have this subject prefix added to them."""
90
91      domain: Optional[str] = None
92      """The domain of the message list, if it has email list associated with it."""
93
94      non_member_message_pass: Optional[bool] = None
```

```
95      """A flag that controls if messages from non members are automatically passed to ↩
        the list."""
96
97      allow_attachments: Optional[bool] = None
98      """A flag controlling if attachments are allowed on the list."""
99
100     distribution: Optional[Distribution] = None
101     """All the message channels the list is using."""
102
103     removed: Optional[datetime] = None
104     """If set, shows the date the message list is set to not be in use. If a user has ↩
        access to the admin document
105     even if this is set, it means that the message list is frozen, but not completely ↩
        deleted. """
106
107
108 @dataclass
109 class MemberInfo:
110     """Wrapper for information about a member on a message list."""
111     name: str
112     username: str
113     sendRight: bool
114     deliveryRight: bool
115     email: str
116     removed: Optional[datetime] = None
117
118
119 @dataclass
120 class GroupAndMembers:
121     """Helper class for querying user group and its members."""
122     groupName: str
123     members: List[MemberInfo]
124
125
126 # A mapping of TIM's archive policies to Mailman's archive policies. Mailman's ↩
        archive policies are listed here:
127 # https://gitlab.com/mailman/mailman/-/blob/master/src/mailman/interfaces/archiver.py
128 mailman_archive_policy_correlate: Dict[ArchiveType, str] = {
129     ArchiveType.NONE: "never",
130     # Secret archive type doesn't exist in Mailman. Because Mailman's private archive ↩
        policy is open for list
131     # member's, we turn Mailman's archiving off and rely solely on TIM's archiving.
132     ArchiveType.SECRET: "never",
133     ArchiveType.GROUPONLY: "private",
134     # Unlisted archive type doesn't exist in Mailman, but closest is setting policy ↩
        as private and provide necessary
135     # archive links from TIM.
136     ArchiveType.UNLISTED: "private",
137     ArchiveType.PUBLIC: "public"
138 }
```

# timApp/messaging/messagelist/mailman_events.py

```
1 # Mailman event handler
2 # Listens to events sent by https://github.com/dezhidki/mailman-rest-events
3
4 import secrets
5 from dataclasses import dataclass
6 from typing import Optional
```

```
7
8  from flask import request, Response, Blueprint
9
10 from timApp.messaging.messagelist.messagelist_models import MessageListModel
11 from timApp.messaging.messagelist.messagelist_utils import parse_mailman_message, ↩
       archive_message
12 from timApp.tim_app import app, csrf
13 from timApp.util.flask.requesthelper import RouteException
14 from timApp.util.flask.responsehelper import ok_response
15 from timApp.util.logger import log_warning
16 from tim_common.marshmallow_dataclass import class_schema
17
18 mailman_events = Blueprint("mailman_events", __name__, url_prefix="/mailman/event")
19
20 AUTH_USER = app.config.get("MAILMAN_EVENT_API_USER") or ""
21 AUTH_KEY = app.config.get("MAILMAN_EVENT_API_KEY") or ""
22 if not AUTH_USER or not AUTH_KEY:
23     log_warning("No mailman event API credentials set! Generating random credentials.")
24     AUTH_USER = secrets.token_urlsafe(32)
25     AUTH_KEY = secrets.token_urlsafe(32)
26
27
28 def check_auth() -> bool:
29     auth = request.authorization
30     return auth is not None and auth.type == "basic" and auth.username == AUTH_USER ↩
       and auth.password == AUTH_KEY
31
32
33 @dataclass
34 class MailmanMessageList:
35     id: str
36     name: str
37     host: str
38
39
40 @dataclass
41 class MailmanMemberAddress:
42     email: str
43     name: Optional[str]  # Names associated with an (member) email addresses are ↩
       optional in Mailman.
44
45
46 @dataclass
47 class MailmanMember:
48     user_id: int
49     address: MailmanMemberAddress
50
51
52 @dataclass
53 class SubscriptionEvent:
54     event: str
55     mlist: MailmanMessageList
56     member: MailmanMember
57
58
59 SubscriptionEventSchema = class_schema(SubscriptionEvent)
60
61
62 @dataclass
```

```
63 class NewMessageEvent:
64     event: str
65     mlist: MailmanMessageList
66     message: dict
67 
68 
69 NewMessageEventSchema = class_schema(NewMessageEvent)
70 
71 EVENTS = {
72     "user_subscribed": SubscriptionEventSchema(),
73     "user_unsubscribed": SubscriptionEventSchema(),
74     "new_message": NewMessageEventSchema()
75 }
76 
77 
78 @mailman_events.route("", methods=["POST"])
79 @csrf.exempt
80 def handle_event() -> Response:
81     """Handle events sent by Mailman."""
82     if not check_auth():
83         return Response(status=401, headers={"WWW-Authenticate": "Basic realm=\"Needs ↩
    auth\""})
84 
85     if not request.is_json:
86         raise RouteException("Body must be JSON")
87 
88     data = request.json
89     if "event" not in data or data["event"] not in EVENTS:
90         raise RouteException("Event not handled")
91 
92     evt = EVENTS[data["event"]].load(data)
93 
94     if isinstance(evt, SubscriptionEvent):
95         if evt.event == "user_subscribed":
96             # TODO: Handle subscription event.
97             pass
98         elif evt.event == "user_unsubscribed":
99             # TODO: Handle unsubscription event.
100            pass
101    # TODO: Check if this message is a duplicate. If it is, then handle (e.g. drop) ↩
    it. How to check if the message
102    #  is a duplicate? If we are checking for a duplicate, should we be counting how ↩
    "manyeth" duplicate the message
103    #  is, so we can e.g. catch if there is a spammer channel that bombards with ↩
    duplicate messages?
104    elif isinstance(evt, NewMessageEvent):
105        handle_new_message(evt)
106 
107    return ok_response()
108 
109 
110 def handle_new_message(event: NewMessageEvent) -> None:
111    """Handles an event raised by a new message.
112 
113    :param event: Contains information about a new message sent to Mailman's list.
114    """
115    m_list_name, _, _ = event.mlist.name.partition("@")
116    message_list = MessageListModel.get_list_by_name_first(m_list_name)
117 
```

```
    if message_list is None:
        raise RouteException("Message list does not exist.")
    if not message_list.email_list_domain == event.mlist.host:
        # If we are here, something is now funky. Message list doesn't have a email list (domain) configured,
        # but messages are directed at it. Not sure what do exactly do here, honestly, except log the event for
        # further investigation.
        log_warning(f"Message list '{message_list.name}' with id '{message_list.id}' doesn't have a domain "
                    f"configured properly. Domain '{event.mlist.host}' was expected.")
        raise RouteException("List not configured properly.")
    parsed_message = parse_mailman_message(event.message, message_list)
    archive_message(message_list, parsed_message)
    # TODO: Relay this message forward, if there are other message channels in use for a message list.
```

# timApp/messaging/messagelist/messagelist_models.py

```
from datetime import datetime
from enum import Enum
from typing import List, Optional, Dict, Any

from sqlalchemy.orm.exc import MultipleResultsFound, NoResultFound  # type: ignore

from timApp.messaging.messagelist.listoptions import ArchiveType, Channel, ReplyToListChanges
from timApp.timdb.sqa import db
from timApp.util.utils import get_current_time


class MemberJoinMethod(Enum):
    """How a user was added to a message list."""
    DIRECT_ADD = 1
    """The owner of the list has just added this member. The member wasn't asked. This is the only join method that
    makes sense for groups. """
    INVITED = 2
    """User was invited and they confirmed joining."""
    JOINED = 3
    """User joined the list on their own."""


class MessageListModel(db.Model):
    """Database model for message lists"""

    __tablename__ = "messagelist"

    id = db.Column(db.Integer, primary_key=True)

    manage_doc_id = db.Column(db.Integer, db.ForeignKey("block.id"))
    """The document which manages a message list."""

    name = db.Column(db.Text)
    """The name of a message list."""

    can_unsubscribe = db.Column(db.Boolean)
    """If a member can unsubscribe from this list on their own."""

```

```
39      email_list_domain = db.Column(db.Text)
40      """The domain used for an email list attached to a message list. If None/null, then message list doesn't have an
41      attached email list. This is a tad silly at this point in time, because JYU TIM only has one domain. However,
42      this allows quick adaptation if more domains are added or otherwise changed in the future. """
43
44      archive = db.Column(db.Enum(ArchiveType))
45      """The archive policy of a message list."""
46
47      notify_owner_on_change = db.Column(db.Boolean)
48      """Should the owner of the message list be notified if there are changes on message list members."""
49
50      description = db.Column(db.Text)
51      """A short description what a message list is about."""
52
53      info = db.Column(db.Text)
54      """Additional information about the message list."""
55
56      removed = db.Column(db.DateTime(timezone=True))
57      """When this list has been marked for removal."""
58
59      default_send_right = db.Column(db.Boolean)
60      """Default send right for new members who join the list on their own."""
61
62      default_delivery_right = db.Column(db.Boolean)
63      """Default delivery right for new members who join the list on their own."""
64
65      tim_user_can_join = db.Column(db.Boolean)
66      """Flag if TIM users can join the list on their own."""
67
68      subject_prefix = db.Column(db.Text)
69      """What prefix message subjects that go through the list get."""
70
71      only_text = db.Column(db.Boolean)
72      """Flag if only text format messages are allowed on a list."""
73
74      default_reply_type = db.Column(db.Enum(ReplyToListChanges))
75      """Default reply type for the list."""
76
77      non_member_message_pass = db.Column(db.Boolean)
78      """Flag if non members messages to the list are passed straight through without moderation."""
79
80      allow_attachments = db.Column(db.Boolean)
81      """Flag if attachments are allowed on the list. The list of allowed attachment file extensions are stored at
82      listoptions.py """
83
84      block = db.relationship("Block", back_populates="managed_messagelist", lazy="select")
85      """Relationship to the document that is used to manage this message list."""
86
87      members = db.relationship("MessageListMember", back_populates="message_list", lazy="select")
88      """All the members of the list."""
89
```

```
90      distribution = db.relationship("MessageListDistribution", ↩
        back_populates="message_list", lazy="select")
91      """The message channels the list uses."""
92
93      @staticmethod
94      def get_list_by_manage_doc_id(doc_id: int) -> 'MessageListModel':
95          m = MessageListModel.query.filter_by(manage_doc_id=doc_id).one()
96          return m
97
98      @staticmethod
99      def get_list_by_name_exactly_one(name: str) -> 'MessageListModel':
100         """Get a message list. Use this when the list is expected to exist.
101
102         Raise NotExist exception, if the message list is not found (or technically if ↩
        multiple ones are found).
103
104         :param name: The name of the list.
105         :return: MessageListModel object, if a MessageListModel with attribute name ↩
        is found.
106         """
107         try:
108             m = MessageListModel.query.filter_by(name=name).one()
109             return m
110         except (MultipleResultsFound, NoResultFound):
111             from timApp.util.flask.requesthelper import NotExist
112             raise NotExist(f"No message list named {name}")
113
114     @staticmethod
115     def get_list_by_name_first(name_candidate: str) -> 'MessageListModel':
116         """Get a message list by its name, if a list with said name exists.
117
118         :param name_candidate: The name of the message list.
119         :return: Return the message list after query by name. Returns at most one ↩
        result or None if no there are hits.
120         """
121         m = MessageListModel.query.filter_by(name=name_candidate).first()
122         return m
123
124     @staticmethod
125     def name_exists(name_candidate: str) -> bool:
126         """Check if given name already exists among message lists.
127
128         :param name_candidate: The name we are checking if it already is already in ↩
        use by another list.
129         """
130         maybe_list = ↩
        MessageListModel.get_list_by_name_first(name_candidate=name_candidate)
131         if maybe_list is None:
132             return False
133         else:
134             return True
135
136     @property
137     def archive_policy(self) -> ArchiveType:
138         return self.archive
139
140     def get_individual_members(self) -> List['MessageListMember']:
141         """Get all the members that are not groups.
142
```

```
143         :return: A list of message list's members, who are individual TIM users (MessageListTimMember objects) or
144         external members (MessageListExternalMember objects).
145         """
146         individuals = []
147         for member in self.members:
148             if not member.is_group():
149                 individuals.append(member)
150         return individuals
151 
152     def get_tim_members(self) -> List['MessageListTimMember']:
153         """Get all members that have belong to a user group, i.e. TIM users and user groups.
154 
155         :return: A list of MessageListTimMember objects.
156         """
157         tim_members = []
158         for member in self.members:
159             if member.is_tim_member():
160                 tim_members.append(member.tim_member)
161         return tim_members
162 
163     def find_member(self, username: Optional[str], email: Optional[str]) -> Optional['MessageListMember']:
164         """Get member of this list. Member can be searched with username and/or email. At least one has to be given. If
165         both are given, username is preferred and is used in a search first.
166 
167         Raises ValueError if used with both name and email parameters as None.
168 
169         :param username: Userame of the member.
170         :param email: Member's email address
171         :return: A message list member, if one is found with given arguments. Otherwise return None.
172         """
173         if not username and not email:
174             raise ValueError
175 
176         for member in self.members:
177             if username and username == member.get_username():
178                 return member
179             if email and email == member.get_email():
180                 return member
181         return None
182 
183 
184 class MessageListMember(db.Model):
185     """Database model for members of a message list."""
186 
187     __tablename__ = "messagelist_member"
188 
189     id = db.Column(db.Integer, primary_key=True)
190 
191     message_list_id = db.Column(db.Integer, db.ForeignKey("messagelist.id"))
192     """What message list a member belongs to."""
193 
194     send_right = db.Column(db.Boolean)
195     """If a member can send messages to a message list. Send right for a user group is meaningless at this point"""
```

```
196
197     delivery_right = db.Column(db.Boolean)
198     """If a member can get messages from a message list. Delivery right for a user ↩
        group is meaningless at this
199     point. """
200
201     membership_ended = db.Column(db.DateTime(timezone=True))
202     """When member's membership on a list ended. This is set when member is removed ↩
        from a list. A value of None means
203     the member is still on the list."""
204
205     join_method = db.Column(db.Enum(MemberJoinMethod))
206     """How the member came to a list."""
207
208     membership_verified = db.Column(db.DateTime(timezone=True))
209     """When the user's joining was verified. If user is added e.g. by a teacher to a ↩
        course's message list,
210     this date is the date teacher added the member. If the member was invited, then ↩
        this is the date they verified
211     their join. """
212
213     member_type = db.Column(db.Text)
214     """Discriminator for polymorhphic members."""
215
216     message_list = db.relationship("MessageListModel", back_populates="members", ↩
        lazy="select", uselist=False)
217     tim_member = db.relationship("MessageListTimMember", back_populates="member", ↩
        lazy="select",
218                                  uselist=False, post_update=True)
219     external_member = db.relationship("MessageListExternalMember", ↩
        back_populates="member", lazy="select",
220                                       uselist=False, post_update=True)
221     distribution = db.relationship("MessageListDistribution", ↩
        back_populates="member", lazy="select")
222
223     __mapper_args__ = {"polymorphic_identity": "member", "polymorphic_on": member_type}
224
225     def is_external_member(self) -> bool:
226         """If this member is an external member to a message list."""
227         if self.external_member:
228             return True
229         return False
230
231     def is_tim_member(self) -> bool:
232         """If this member is a 'TIM member', i.e. a user group. This can be either a ↩
            personal user group or a
233         group. """
234         if self.tim_member:
235             return True
236         return False
237
238     def is_personal_user(self) -> bool:
239         """If this member is an individual user, i.e. a personal user group or an ↩
            external member."""
240         try:
241             gid = self.tim_member.group_id
242         except AttributeError:
243             # External members don't have a group_id attribute.
244             return self.is_external_member()
```

```
245         from timApp.user.usergroup import UserGroup
246         ug = UserGroup.query.filter_by(id=gid).one()
247         return ug.is_personal_group
248 
249     def is_group(self) -> bool:
250         """If this message list member is actually a group of users."""
251         return not self.is_personal_user()
252 
253     def is_active(self) -> bool:
254         """Check if the message list's member is an active member of the list. A member is an active member if they have
255          been verified and have not been removed from the list.
256 
257         :return: True if the member is both verified and not removed. Otherwise returns False.
258         """
259         return self.membership_ended is None and self.is_verified()
260 
261     def is_verified(self) -> bool:
262         """If the member is verified to be on the list. """
263         return self.membership_verified is not None
264 
265     def remove(self, end_time: Optional[datetime] = get_current_time()) -> None:
266         """Shorthand for removing a member out of the group, by setting the membership_ended attribute."""
267         self.membership_ended = end_time
268 
269     def get_email(self) -> str:
270         """The process of obtaining member's address varies depending on if the member
271         is a TIM user or not. Child classes have their own implementation depending on how they obtain the
272         information. This is mostly for helping with types.
273 
274         :return: This particular instance raises NotImplementedError. The supposed return value is the user's email
275         address.
276         """
277         raise NotImplementedError
278 
279 
280 class MessageListTimMember(MessageListMember):
281     """A member of message list who is also a TIM user(group). This can be one person in their own personal user
282     group or this can be e.g. a course's group."""
283 
284     __tablename__ = "messagelist_tim_member"
285 
286     id = db.Column(db.Integer, db.ForeignKey("messagelist_member.id"), primary_key=True)
287 
288     group_id = db.Column(db.Integer, db.ForeignKey("usergroup.id"))
289     """A UserGroup id for a member."""
290 
291     member = db.relationship("MessageListMember", back_populates="tim_member", lazy="select",
292                              uselist=False, post_update=True)
293 
294     user_group = db.relationship("UserGroup", back_populates="messagelist_membership", lazy="select", uselist=False,
295                                  post_update=True)
```

```
296
297     __mapper_args__ = {"polymorphic_identity": "tim_member"}
298
299     def to_json(self) -> Dict[str, Any]:
300         return {
301             "name": self.get_name(),
302             "username": self.get_username(),
303             "email": self.get_email() if self.get_email() is not None else "",
304             "sendRight": self.member.send_right,
305             "deliveryRight": self.member.delivery_right,
306             "removed": self.membership_ended
307         }
308
309     def get_username(self) -> str:
310         """Get the TIM user group's name."""
311         ug = self.user_group
312         return ug.name
313
314     def get_email(self) -> str:
315         """Get TIM user group's email. Email makes sense only for personal user ↩
    groups. Using this method for groups
316         returns an empty string"""
317         if self.is_group():
318             return ""
319         ug = self.user_group
320         user = ug.personal_user
321         return user.email
322
323     def get_name(self) -> str:
324         """Get TIM user's name. For group, this is an empty string. For a user, this ↩
    is their full name."""
325         if not self.is_group():
326             ug = self.user_group
327             user = ug.personal_user
328             return user.pretty_full_name
329         return ""
330
331
332 class MessageListExternalMember(MessageListMember):
333     """A member of message list who is *not* a TIM user. Mainly intended for, but not ↩
    necessary limited to,
334     email-list only usage."""
335
336     __tablename__ = "messagelist_external_member"
337
338     id = db.Column(db.Integer, db.ForeignKey("messagelist_member.id"), primary_key=True)
339
340     email_address = db.Column(db.Text)
341     """Email address of message list's external member."""
342
343     display_name = db.Column(db.Text)
344     """Display name for external user, which in most cases should be the external ↩
    member's address' owner's name."""
345
346     member = db.relationship("MessageListMember", back_populates="external_member", ↩
    lazy="select", uselist=False)
347
348     __mapper_args__ = {"polymorphic_identity": "external_member"}
349
```

```
350     def to_json(self) -> Dict[str, Any]:
351         return {
352             "name": self.get_name(),
353             "username": self.get_username(),
354             "email": self.email_address,
355             "sendRight": self.member.send_right,
356             "deliveryRight": self.member.delivery_right,
357             "removed": self.membership_ended
358         }
359 
360     def get_name(self) -> str:
361         """Get the external member's name, if one has been specified.
362 
363         :return:The display name or an empty string.
364         """
365         return self.display_name if self.display_name is not None else ""
366 
367     def get_email(self) -> str:
368         """Get message list's external member's email.
369 
370         :return: The email address.
371         """
372         return self.email_address
373 
374     def get_username(self) -> str:
375         """External member's don't have usernames, but this is for consistency when using other methods."""
376         return ""
377 
378 
379 class MessageListDistribution(db.Model):
380     """Message list member's chosen distribution channels."""
381 
382     __tablename__ = "messagelist_distribution"
383 
384     id = db.Column(db.Integer, primary_key=True)
385 
386     user_id = db.Column(db.Integer, db.ForeignKey("messagelist_member.id"))
387     """Message list member's id, if this row is about message list member's channel distribution."""
388 
389     message_list_id = db.Column(db.Integer, db.ForeignKey("messagelist.id"))
390     """Message list's id, if this row is about message list's channel distribution."""
391 
392     channel = db.Column(db.Enum(Channel))
393     """Which message channels are used by a message list or a user."""
394 
395     member = db.relationship("MessageListMember", back_populates="distribution", lazy="select", uselist=False)
396     message_list = db.relationship("MessageListModel", back_populates="distribution", lazy="select", uselist=False)
397 
398 
399 class UserEmails(db.Model):
400     """TIM users' additional emails."""
401 
402     __tablename__ = "user_emails"
403 
404     id = db.Column(db.Integer, db.ForeignKey("useraccount.id"), primary_key=True)
```

```
405
406     additional_email = db.Column(db.Text, unique=True)
407     """The users another email."""
408
409     is_primary_email = db.Column(db.Boolean)
410     """Which one of the additional emails is user's primary email address. If none ↩
        are, the email address in users
411     own table is assumed to be primary email address. """
412
413     address_verified = db.Column(db.DateTime(timezone=True))
414     """The user has to verify they are in the possession of the email address."""
415
416
417 class VerificationType(Enum):
418     """Type of verification, used to direct the proper verification action ↩
        afterwards."""
419     LIST_JOIN = 1
420     EMAIL_OWNERSHIP = 2
421
422
423 class Verification(db.Model):
424     """For various pending verifications, such as message list joining and email ↩
        ownership verification."""
425     __tablename__ = "verifications"
426
427     id = db.Column(db.Integer, primary_key=True)
428
429     verification_type = db.Column(db.Enum(VerificationType))
430     """The type of verification, see VerificationType class for details."""
431
432     verification_pending = db.Column(db.DateTime(timezone=True))
433     """When a verification has been added to db, pending sending to a user."""
434
435     verification_link = db.Column(db.Text)
436     """Generated verification link. This is given to the user and once they click on ↩
        it, they are verified (in
437     whatever it was that needed verification)."""
438
439     verified = db.Column(db.DateTime(timezone=True))
440     """When the user used the link to verify."""
```

# timApp/messaging/messagelist/messagelist_utils.py

```
1  import re
2  from dataclasses import dataclass, field
3  from datetime import datetime
4  from email.utils import parsedate_to_datetime
5  from typing import Optional, List, Dict, Tuple
6
7  from mailmanclient import MailingList
8
9  from timApp.auth.accesshelper import has_manage_access
10 from timApp.auth.accesstype import AccessType
11 from timApp.document.create_item import create_document
12 from timApp.document.docentry import DocEntry
13 from timApp.document.docinfo import DocInfo
14 from timApp.document.document import Document
15 from timApp.folder.folder import Folder
16 from timApp.item.block import Block
```

```
17 from timApp.messaging.messagelist.emaillist import get_email_list_by_name, ↩
       set_notify_owner_on_list_change, \
18     set_email_list_unsubscription_policy, set_email_list_subject_prefix, ↩
       set_email_list_only_text, \
19     set_email_list_allow_nonmember, set_email_list_allow_attachments, ↩
       set_email_list_default_reply_type, \
20     add_email, get_email_list_member, remove_email_list_membership, ↩
       set_email_list_member_send_status, \
21     set_email_list_member_delivery_status, set_email_list_description, ↩
       set_email_list_info
22 from timApp.messaging.messagelist.listoptions import ArchiveType, ListOptions, ↩
       ReplyToListChanges
23 from timApp.messaging.messagelist.messagelist_models import MessageListModel, ↩
       Channel, MessageListTimMember, \
24     MessageListExternalMember, MessageListMember
25 from timApp.timdb.sqa import db
26 from timApp.user.user import User
27 from timApp.user.usergroup import UserGroup
28 from timApp.util.flask.requesthelper import RouteException
29 from timApp.util.logger import log_warning
30 from timApp.util.utils import remove_path_special_chars, get_current_time
31
32
33 def verify_messagelist_name_requirements(name_candidate: str) -> None:
34     """Checks name requirements specific for email list.
35
36     If at any point a name requirement check fails, then an exception is raised an ↩
       carried to the client. If all
37     name requirements are met, then succeed silently.
38
39     :param name_candidate: Name to check against naming rules.
40     """
41     # There might become a time when we also check here if name is some message list ↩
       specific reserved name. We
42     # haven't got a source of those reserved names, not including names that already ↩
       exists, so no check at this time.
43     verify_name_rules(name_candidate)
44     verify_name_availability(name_candidate)
45
46
47 def verify_name_availability(name_candidate: str) -> None:
48     """Check if a message list with a given name already exists.
49
50     :param name_candidate: The name to be checked if it already exists.
51     """
52     msg_list_exists = MessageListModel.name_exists(name_candidate)
53     if msg_list_exists:
54         raise RouteException(f"Message list with name {name_candidate} already exists.")
55
56
57 # Regular expression patters used for name rule verification. They are kept here, so ↩
       they are not re-compiled at
58 # every name rule verification. The explanation of the rules is at their usage in ↩
       verify_name_rules function.
59 START_WITH_LOWERCASE_PATTER = re.compile(r"^[a-z]")
60 SEQUENTIAL_DOTS_PATTERN = re.compile(r"\.\.+")
61 PROHIBITED_CHARACTERS_PATTERN = re.compile(r"[^a-z0-9.\-_]")
62 REQUIRED_DIGIT_PATTERN = re.compile(r"\d")
63
```

```
64
65 def verify_name_rules(name_candidate: str) -> None:
66     """Check if name candidate complies with naming rules.
67
68     The function raises a RouteException if naming rule is violated. If this function doesn't raise an exception,
69     then the name candidate follows naming rules.
70
71     :param name_candidate: What name we are checking against the rules.
72     """
73     # Be careful when checking regex rules. Some rules allow a pattern to exist, while prohibiting others. Some
74     # rules prohibit something, but allow other things to exist. If the explanation for a rule is different than
75     # the regex, the explanation is more likely to be correct.
76
77     # Name is within length boundaries.
78     lower_bound = 5
79     upper_bound = 36
80     if not (lower_bound <= len(name_candidate) <= upper_bound):
81         raise RouteException(f"Name is not within length boundaries. Name has to be at least {lower_bound} and at "
82                              f"most {upper_bound} characters long.")
83
84     # Name has to start with a lowercase letter.
85     if not START_WITH_LOWERCASE_PATTER.search(name_candidate):
86         raise RouteException("Name has to start with a lowercase letter.")
87
88     # Name cannot have multiple dots in sequence.
89     if SEQUENTIAL_DOTS_PATTERN.search(name_candidate):
90         raise RouteException("Name cannot have sequential dots.")
91
92     # Name cannot end in a dot.
93     if name_candidate.endswith("."):
94         raise RouteException("Name cannot end in a dot.")
95
96     # Name can have only these allowed characters. This set of characters is an import from Korppi's character
97     # limitations for email list names, and can probably be expanded in the future if desired.
98     #     lowercase letters a - z
99     #     digits 0 - 9
100    #     dot '.'
101    #     hyphen '-'
102    #     underscore '_'
103    # The pattern is a negation of the actual rules.
104    if PROHIBITED_CHARACTERS_PATTERN.search(name_candidate):
105        raise RouteException("Name contains forbidden characters.")
106
107    # Name has to include at least one digit.
108    if not REQUIRED_DIGIT_PATTERN.search(name_candidate):
109        raise RouteException("Name has to include at least one digit.")
110
111
112 @dataclass
113 class EmailAndDisplayName:
114     """Wrapper for parsed email messages containing sender/receiver email and display name."""
115     email_address: str
```

```
116     display_name: str
117 
118     def __repr__(self) -> str:
119         """The representation of an email and display name for an email message is
120 
121         Jane Doe <jane.doe@domain.com>
122 
123         or just
124 
125         <john.doe@domain.com>
126 
127         if no name is associated with the email address.
128         """
129         if self.display_name:
130             return f"{self.display_name} <{self.email_address}>"
131         return f"<{self.email_address}>"
132 
133 
134 @dataclass
135 class BaseMessage:
136     """A unified datastructure for messages TIM handles."""
137     # Meta information about where this message belongs to and where its from. Mandatory values for all messages.
138     message_list_name: str
139     message_channel: Channel = field(metadata={'by_value': True})  # Where the message came from.
140 
141     # Header information. Mandatory values for all messages.
142     sender: EmailAndDisplayName
143     recipients: List[EmailAndDisplayName]
144     subject: str
145 
146     # Message body. Mandatory value for all messages.
147     message_body: str
148 
149     # Email specific attributes.
150     domain: Optional[str] = None
151     reply_to: Optional[EmailAndDisplayName] = None
152 
153     # Timestamp for the message is a mandatory value. If the message comes from an outside source, it should already
154     # have a time stamp. The default value is mostly for messages that would be generated inside TIM. It can also be
155     # set for messages which for some reason don't already have any form of timestamp present.
156     timestamp: datetime = get_current_time()
157 
158 
159 # Path prefixes for documents and folders.
160 MESSAGE_LIST_DOC_PREFIX = "messagelists"
161 MESSAGE_LIST_ARCHIVE_FOLDER_PREFIX = "archives"
162 
163 
164 def create_archive_doc_with_permission(archive_subject: str, archive_doc_path: str, message_list: MessageListModel,
165                                        message: BaseMessage) -> DocEntry:
166     """Create archive document with permissions matching the message list's archive policy.
167 
```

```
168     :param archive_subject: The subject of the archive document.
169     :param archive_doc_path: The path where the archive document should be created.
170     :param message_list: The message list where the message belongs.
171     :param message: The message about to be archived.
172     :return: The archive document.
173     """
174     # Gather owners of the archive document.
175     message_owners: List[UserGroup] = []
176     message_sender = User.get_by_email(message.sender.email_address)
177 
178     # List owners get a default ownership for the messages on a list. This covers the ↩
        archive policy of SECRET.
179     message_owners.extend(get_message_list_owners(message_list))
180 
181     # Who gets to see a message in the archives.
182     message_viewers: List[UserGroup] = []
183 
184     # Gather permissions to the archive doc. The meanings of different archive ↩
        settings are listed with ArchiveType
185     # class.
186     if message_list.archive_policy is ArchiveType.PUBLIC or ArchiveType.UNLISTED:
187         message_viewers.append(UserGroup.get_anonymous_group())
188         if message_sender:
189             message_owners.append(message_sender.get_personal_group())
190     elif message_list.archive_policy is ArchiveType.UNLISTED:
191         message_viewers.append(UserGroup.get_logged_in_group())
192         if message_sender:
193             message_owners.append(message_sender.get_personal_group())
194     elif message_list.archive_policy is ArchiveType.GROUPONLY:
195         message_viewers = [m.user_group for m in message_list.get_tim_members()]
196         if message_sender:
197             message_owners.append(message_sender.get_personal_group())
198 
199     # If we don't provide at least one owner up front, then current user is set as ↩
        owner. We don't want that,
200     # because in this context that is the anonymous user, which raises an error in ↩
        document creation.
201     archive_doc = DocEntry.create(title=archive_subject, path=archive_doc_path, ↩
        owner_group=message_owners[0])
202 
203     # Add the rest of the message owners.
204     if len(message_owners) > 1:
205         archive_doc.block.add_rights(message_owners[1:], AccessType.owner)
206 
207     # Add view rights.
208     archive_doc.block.add_rights(message_viewers, AccessType.view)
209 
210     return archive_doc
211 
212 
213 def archive_message(message_list: MessageListModel, message: BaseMessage) -> None:
214     """Archive a message for a message list.
215 
216     :param message_list: The message list where the archived message belongs.
217     :param message: The message being archived.
218     """
219     # Archive policy of no archiving is a special case, where we abort immediately ↩
        since these won't be archived at all.
220     if message_list.archive_policy is ArchiveType.NONE:
```

```
221         return
222 
223     archive_subject = f"{message.subject}"
224     archive_folder_path = ↩
    f"{MESSAGE_LIST_ARCHIVE_FOLDER_PREFIX}/{remove_path_special_chars(message_list.name)}"
225     archive_doc_path = ↩
    remove_path_special_chars(f"{archive_folder_path}/{archive_subject}-"
226                                                         ↩
    f"{get_current_time().strftime('%Y-%m-%d %H:%M:%S')}")
227 
228     # From the archive folder, query all documents, sort them by created attribute. ↩
    We do this to get the previously
229     # newest archived message, before we create a archive document for newest message.
230     # Archive folder for message list.
231     archive_folder = Folder.find_by_path(archive_folder_path)
232     all_archived_messages = []
233     if archive_folder is not None:
234         all_archived_messages = archive_folder.get_all_documents()
235     else:
236         owners = get_message_list_owners(message_list)
237         Folder.create(archive_folder_path, owner_groups=owners, ↩
    title=f"{message_list.name}")
238 
239     archive_doc = create_archive_doc_with_permission(archive_subject, ↩
    archive_doc_path, message_list, message)
240 
241     # Set header information for archived message.
242     archive_doc.document.add_text(f"""
243 #- {{ .mailheader}}\n
244 [{message.subject}]{{.mailtitle}}\\
245 Sender: {message.sender}\\
246 Recipients: {message.recipients}\\
247 Date: {message.timestamp.strftime('%Y-%m-%d %H:%M:%S')}
248 """)
249 
250     # Set message body for archived message.
251     # TODO: Check message list's only_text flag.
252     archive_doc.document.add_text(f"{message.message_body}")
253 
254     if not len(all_archived_messages):
255         # If the message currently being archived is the very first one, it can't be ↩
    linked to other messages.
256         db.session.commit()
257         return
258 
259     # Linking messages.
260 
261     sorted_messages = sorted(all_archived_messages, key=lambda document: ↩
    document.block.created, reverse=True)
262     # Get all_archived_messages before creating the archive document for the newest ↩
    message, so the previous newest
263     # message is at index 0.
264     previous_doc = sorted_messages[0]
265 
266     if len(all_archived_messages) == 1:
267         # Having only one message in the archive at first before "this newest" is a ↩
    special case, because at that
268         # point the first links to a next message has not been set. Setting link to a ↩
    next message assumes that a
```

```
269         # link to previous-previous message exits in the previous document.
270         set_message_link_next(previous_doc.document, archive_doc.title, archive_doc.url,
271                               archive_init_flag=True)
272     else:
273         set_message_link_next(previous_doc.document, archive_doc.title, archive_doc.url)
274     set_message_link_previous(archive_doc.document, previous_doc.title, ↩
    previous_doc.url)
275 
276     db.session.commit()
277 
278 
279 def set_message_link_next(doc: Document, link_text: str, url_next: str, ↩
    archive_init_flag: bool = False) -> None:
280     """Set links to a document from another document.
281 
282     This function sets a footer link to next archived document and a button to the ↩
    header section with a link to next
283     archived document.
284 
285     :param doc: The document where the link is appended.
286     :param link_text: The text the link gets.
287     :param url_next: The link to another document.
288     :param archive_init_flag: A boolean flag, denoting if doc should be treated as ↩
    the special case of being very first
289     message in the archive when linking is started.
290     """
291 
292     # Also add the directional button to the next document at the start of the header ↩
    section.
293     direct_button = f"[[>]{{.timButton}}]({url_next})"
294     header = doc.get_paragraphs()[0]
295     header_md = header.get_markdown()
296 
297     link_footer = f"[Next Message: {link_text}]({url_next})"
298     if archive_init_flag:
299         # Here we set the very first message's button and link to next. For some ↩
    reason, the trailing whitespace is
300         # needed for the directional button to appear on the document. The text for ↩
    the directional button will be
301         # there regardless, but it won't show on the document. So don't remove the ↩
    trailing whitespace unless you
302         # know that you get the link button to appear on the first archvie document ↩
    some other way.
303         button_first = f"{direct_button}\n\n{header_md}"
304         header.set_markdown(button_first)
305         header.save()
306 
307         footer_text = f"""
308 #- {{ .mailfooter}}\n
309 {link_footer}
310 """
311         doc.add_text(footer_text)
312     else:
313         # Find the index of character combination of ']{.ma'. That denotes the ↩
    closing of the area of class
314         # mailbrowsebuttons defined in the else part of this conditional statement. ↩
    Inject the button to next
315         # document there.
316         limit = header_md.find("]{.ma")
```

```
        new_button_set = f"{header_md[0:limit]}\n{direct_button}{header_md[limit:]}"
        header.set_markdown(new_button_set)
        header.save()

        last_par = doc.get_paragraphs()[-1]
        last_par_md = last_par.get_markdown()
        # Add a \ and a line break after the previous link, then the new link.
        modified_footer = f"""{last_par_md}\\
{link_footer}"""
        last_par.set_markdown(modified_footer)
        last_par.save()


def set_message_link_previous(doc: Document, link_text: str, url_previous: str) -> None:
    """Set links to a document from another document.

    This function sets a footer link to previous archived document and a button to the header section with a link to
    revious archived document.

    :param doc: The document where the link is appended.
    :param link_text: The text the link gets.
    :param url_previous: The link to another document.
    """
    # Add footer for link to previous document.
    link_footer = f"""
#- {{ .mailfooter}}\n
[Previous message: {link_text}]({url_previous})
"""
    doc.add_text(link_footer)
    # Add a directional button at the start of the header section of document.
    direct_button = f"[[[<]{{.timButton}}]({url_previous})]{{.mailbrowsebuttons}}\n\n"
    header = doc.get_paragraphs()[0]
    header.insert_before_md(direct_button)
    header.save()


def parse_mailman_message(original: Dict, msg_list: MessageListModel) -> BaseMessage:
    """Modify an email message sent from Mailman to TIM's universal message format.

    :param original: An email message sent from Mailman.
    :param msg_list: The message list where original is meant to go.
    :return: A BaseMessage object corresponding the original email message.
    """
    # original message is of form specified in https://pypi.org/project/mail-parser/
    visible_recipients: List[EmailAndDisplayName] = []
    maybe_to_addresses = parse_mailman_message_address(original, "to")
    if maybe_to_addresses is not None:
        visible_recipients.extend(maybe_to_addresses)
    maybe_cc_addresses = parse_mailman_message_address(original, "cc")
    if maybe_cc_addresses is not None:
        visible_recipients.extend(maybe_cc_addresses)

    sender: Optional[EmailAndDisplayName] = None
    maybe_from_address = parse_mailman_message_address(original, "from")
    if maybe_from_address is not None:
        # Expect only one sender.
        sender = maybe_from_address[0]
    if sender is None:
```

```
375         # If no sender is found on a message, we don't archive the message.
376         raise RouteException("No sender found in the message.")
377 
378     message = BaseMessage(
379         message_list_name=msg_list.name,
380         domain=msg_list.email_list_domain,
381         message_channel=Channel.EMAIL_LIST,
382 
383         # Header information
384         sender=sender,
385         recipients=visible_recipients,
386         subject=original["subject"],
387 
388         # Message body
389         message_body=original["body"],
390     )
391 
392     # Try parsing the rest of email specific fields.
393     if "reply_to" in original:
394         message.reply_to = original["reply_to"]
395     if "date" in original:
396         try:
397             # At first we except RFC5322 format Date header.
398             message.timestamp = parsedate_to_datetime(original["date"])
399         except (TypeError, ValueError):
400             # Being here means that the date field is not in RFC5322 format. Testing ↩
    has shown that ISO8601 format is
401             # then a likely candidate format for Date header. Try parsing that format.
402             try:
403                 message.timestamp = datetime.fromisoformat(original["date"])
404             except ValueError:
405                 # Being here means that the date field was none of tried formats ↩
    after all. We'll log the format the
406                 # date was in so that it can be fixed.
407                 log_warning(
408                     f"Function parse_mailman_message has encountered a Date header ↩
    format it cannot handle. The "
409                     f"date is of format {original['date']}. Please handle this at ↩
    earliest convenience.")
410     return message
411 
412 
413 def parse_mailman_message_address(original: Dict, header: str) -> ↩
    Optional[List[EmailAndDisplayName]]:
414     """Parse (potentially existing) fields 'from' 'to', 'cc', or 'bcc' from a dict ↩
    representing Mailman's email message.
415     The fields are in lists, with individual list indicies being lists themselves of ↩
    the form
416         ['Display Name', 'email@domain.fi']
417 
418     :param original: Original message.
419     :param header: One of "from", "to", "cc" or "bcc".
420     :return: Return None if the header is not one of "from", "to", "cc" or "bcc". ↩
    Otherwise return a list of
421     EmailAndDisplayName objects.
422     """
423 
424     if header not in ["from", "to", "cc", "bcc"]:
425         return None
```

```
426
427     email_name_pairs: List[EmailAndDisplayName] = []
428
429     if header in original:
430         for email_name_pair in original[header]:
431             new_email_name_pair = EmailAndDisplayName(email_address=email_name_pair[1],
432                                                       display_name=email_name_pair[0])
433             email_name_pairs.append(new_email_name_pair)
434
435     return email_name_pairs
436
437
438 def get_message_list_owners(mlist: MessageListModel) -> List[UserGroup]:
439     """Get the owners of a message list.
440
441     :param mlist: The message list we want to know the owners.
442     :return: A list of owners, as their personal user group.
443     """
444     manage_doc_block = Block.query.filter_by(id=mlist.manage_doc_id).one()
445     return manage_doc_block.owners
446
447
448 def create_management_doc(msg_list_model: MessageListModel, list_options: ↩
        ListOptions) -> DocInfo:
449     """Create management doc for a new message list.
450
451     :param msg_list_model: The message list the management document is created for.
452     :param list_options: Options for creating the management document.
453     :return: Newly created management document.
454     """
455
456     # We'll err on the side of caution and make sure the path is safe for the ↩
        management doc.
457     path_safe_list_name = remove_path_special_chars(list_options.name)
458     path_to_doc = f'/{MESSAGE_LIST_DOC_PREFIX}/{path_safe_list_name}'
459
460     doc = create_document(path_to_doc, list_options.name)
461
462     # We add the admin component to the document.
463     admin_component = """#- {allowangular="true"}
464 <tim-message-list-admin></tim-message-list-admin>
465     """
466     doc.document.add_text(admin_component)
467
468     # Set the management doc for the message list.
469     msg_list_model.manage_doc_id = doc.id
470
471     return doc
472
473
474 def new_list(list_options: ListOptions) -> Tuple[DocInfo, MessageListModel]:
475     """Adds a new message list into the database and creates the list's management doc.
476
477     :param list_options: The list information for creating a new message list. Used ↩
        to carry list's name and archive
478     policy.
479     :return: The management document of the message list.
480     :return: The message list db model.
481     """
```

```
482     msg_list = MessageListModel(name=list_options.name, archive=list_options.archive)
483     db.session.add(msg_list)
484     doc_info = create_management_doc(msg_list, list_options)
485     return doc_info, msg_list
486
487
488 def set_message_list_notify_owner_on_change(message_list: MessageListModel,
489                                             notify_owners_on_list_change_flag: ↩
    Optional[bool]) -> None:
490     """Set the notify list owner on list change flag for a list, and update necessary ↩
    channels with this information.
491
492     If the message list has an email list as a message channel, this will set the ↩
    equilavent flag on the email list.
493
494     :param message_list: The message list where the flag is being set.
495     :param notify_owners_on_list_change_flag: An optional boolean flag. If True, then ↩
    changes on the message list sends
496     notifications to list owners. If False, notifications won't be sent. If None, ↩
    nothing is set.
497     """
498     if notify_owners_on_list_change_flag is None \
499             or message_list.notify_owner_on_change == notify_owners_on_list_change_flag:
500         return
501
502     message_list.notify_owner_on_change = notify_owners_on_list_change_flag
503
504     if message_list.email_list_domain:
505         # Email lists have their own flag for notifying list owners for list changes.
506         email_list = get_email_list_by_name(message_list.name, ↩
    message_list.email_list_domain)
507         set_notify_owner_on_list_change(email_list, message_list.notify_owner_on_change)
508
509
510 def set_message_list_member_can_unsubscribe(message_list: MessageListModel,
511                                             can_unsubscribe_flag: Optional[bool]) -> ↩
    None:
512     """Set the list member's free unsubscription flag, and propagate that setting to ↩
    channels that have own handling
513     of unsubscription.
514
515     If the message list has an email list as a message channel, this will set the ↩
    equilavent flag on the email list.
516
517     :param message_list: Message list where the flag is being set.
518     :param can_unsubscribe_flag: An optional boolean flag. For True, the member can ↩
    unsubscribe on their own. For False,
519      then the member can't unsubscribe from the list on their own. If None, then the ↩
    current value is kept.
520     """
521     if can_unsubscribe_flag is None or message_list.can_unsubscribe == ↩
    can_unsubscribe_flag:
522         return
523     message_list.can_unsubscribe = can_unsubscribe_flag
524
525     if message_list.email_list_domain:
526         # Email list's have their own settings for unsubscription.
527         email_list = get_email_list_by_name(message_list.name, ↩
    message_list.email_list_domain)
```

```
528         set_email_list_unsubscription_policy(email_list, can_unsubscribe_flag)
529
530
531 def set_message_list_subject_prefix(message_list: MessageListModel, subject_prefix: ↩
    Optional[str]) -> None:
532     """Set the message list's subject prefix.
533
534     If the message list has an email list as a message list, then set the subject ↩
    prefix there also.
535
536     Sets one extra space automatically to offset prefix from the actual title.
537
538     :param message_list: The message list where the subject prefix is being set.
539     :param subject_prefix: The prefix set for messages that go through the list. If ↩
    None, then the current value is
540     kept.
541     """
542     if subject_prefix is None or message_list.subject_prefix == subject_prefix:
543         return
544
545     # Add an extra space, if there is none.
546     if not subject_prefix.endswith(" "):
547         subject_prefix = f"{subject_prefix} "
548
549     message_list.subject_prefix = subject_prefix
550
551     if message_list.email_list_domain:
552         email_list = get_email_list_by_name(message_list.name, ↩
    message_list.email_list_domain)
553         set_email_list_subject_prefix(email_list, subject_prefix)
554
555
556 def set_message_list_tim_users_can_join(message_list: MessageListModel, ↩
    can_join_flag: Optional[bool]) -> None:
557     """Set the flag controlling if TIM users can directly join this list.
558
559     Because the behaviour that is controlled by the can_join_flag applies to TIM ↩
    users, there is no message channel
560     specific handling.
561
562     :param message_list: Message list where the flag is being set.
563     :param can_join_flag: An optional boolean flag. If True, then TIM users can ↩
    directly join this list, no moderation
564     needed. If False, then TIM users can't direclty join the message list. If None, ↩
    the current value is kept.
565     """
566     if can_join_flag is None or message_list.tim_user_can_join == can_join_flag:
567         return
568
569     message_list.tim_user_can_join = can_join_flag
570
571
572 def set_message_list_default_send_right(message_list: MessageListModel,
573                                         default_send_right_flag: Optional[bool]) -> ↩
    None:
574     """Set the default message list new member send right flag.
575
576     :param message_list: The message list where the flag is set.
577     :param default_send_right_flag: An optional boolean flag. For True, new members ↩
```

```
    on the list get default send right.
    For False, new members don't get a send right. For None, the current value is kept.
    """
    if default_send_right_flag is None or message_list.default_send_right == default_send_right_flag:
        return
    message_list.default_send_right = default_send_right_flag


def set_message_list_default_delivery_right(message_list: MessageListModel,
                                            default_delivery_right_flag: Optional[bool]) -> None:
    """Set the message list new member default delivery right.

    :param message_list: The message list where the flag is set.
    :param default_delivery_right_flag: An optional boolean flag. For True, new members on the list get default delivery
    right. For False, new members don't automatically get a delivery right. For None, the current value is kept.
    """
    if default_delivery_right_flag is None or message_list.default_delivery_right == default_delivery_right_flag:
        return
    message_list.default_delivery_right = default_delivery_right_flag


def set_message_list_only_text(message_list: MessageListModel, only_text: Optional[bool]) -> None:
    """Set the flag controlling if message list is to accept text-only messages.

    :param message_list: The message list where the flag is to be set.
    :param only_text: An optional boolean flag. For True, the message list is set to text-only mode. For False, the
    message list accepts HTML-based messages. For None, the current value is kept.
    """
    if only_text is None or message_list.only_text == only_text:
        return
    message_list.only_text = only_text

    if message_list.email_list_domain:
        email_list = get_email_list_by_name(message_list.name, message_list.email_list_domain)
        set_email_list_only_text(email_list, only_text)


def set_message_list_non_member_message_pass(message_list: MessageListModel,
                                             non_member_message_pass_flag: Optional[bool]) -> None:
    """Set message list's non member message pass flag.

    :param message_list: The message list where the flag is set.
    :param non_member_message_pass_flag: An optional boolean flag. For True, sources outside the list can send messages
    to this list. If False, messages form sources outside the list will be hold for moderation. For None, the current
    value is kept.
    """
    if non_member_message_pass_flag is None or message_list.non_member_message_pass == non_member_message_pass_flag:
```

```
624         return
625     message_list.non_member_message_pass = non_member_message_pass_flag
626     if message_list.email_list_domain:
627         email_list = get_email_list_by_name(message_list.name, ↩
        message_list.email_list_domain)
628         set_email_list_allow_nonmember(email_list, non_member_message_pass_flag)
629
630
631 def set_message_list_allow_attachments(message_list: MessageListModel, ↩
        allow_attachments_flag: Optional[bool]) -> None:
632     """Set the flag controlling if a message list accepts messages with attachments.
633
634     :param message_list: The message list where the flag is to be set.
635     :param allow_attachments_flag: An optional boolean flag. For True, the list will ↩
        allow a pre-determined set of
636     attachments. For False, no attachments are allowed. For None, the current value ↩
        is kept.
637     """
638     if allow_attachments_flag is None or message_list.allow_attachments == ↩
        allow_attachments_flag:
639         return
640
641     message_list.allow_attachments = allow_attachments_flag
642     if message_list.email_list_domain:
643         email_list = get_email_list_by_name(message_list.name, ↩
        message_list.email_list_domain)
644         set_email_list_allow_attachments(email_list, allow_attachments_flag)
645
646
647 def set_message_list_default_reply_type(message_list: MessageListModel,
648                                         default_reply_type: ↩
        Optional[ReplyToListChanges]) -> None:
649     """Set a value controlling how replies to a message list are steered.
650
651     The reply type is analogous to email lists' operation of "Reply-To munging". ↩
        Reply-To munging is a process where
652     messages sent to list may be subject to having their Reply-To header changed from ↩
        what the sender of the message
653     initially used. This is mainly used (and sometimes abused) to steer conversation ↩
        from announce-only lists (which
654     don't accept posts from anyone except few select individuals) to separate ↩
        discussion lists.
655
656     :param message_list: The message list where the value is to be set.
657     :param default_reply_type: An optional enumeration. For value NOCHANGES the user ↩
        is completely left the control
658     how to respond to messages sent from the list. For value ADDLIST the replies will ↩
        be primarily steered towards
659     the message list. For None, the current value is kept.
660     """
661     if default_reply_type is None or message_list.default_reply_type == ↩
        default_reply_type:
662         return
663
664     message_list.default_reply_type = default_reply_type
665     if message_list.email_list_domain:
666         email_list = get_email_list_by_name(message_list.name, ↩
        message_list.email_list_domain)
667         set_email_list_default_reply_type(email_list, default_reply_type)
```

```
668
669
670 def add_new_message_list_group(msg_list: MessageListModel, ug: UserGroup,
671                                send_right: bool, delivery_right: bool, em_list: ↩
    Optional[MailingList]) -> None:
672     """Add new (user) group to a message list.
673
674     For groups, checks that the adder has at least manage rights to group's admin doc.
675
676     Performs a duplicate check for memberships. A duplicate member will not be added ↩
    again to the list. The process
677     of re-activating a removed member of a list is different. For re-activating an ↩
    already existing member,
678     use set_message_list_member_removed_status function.
679
680     This is a direct add, meaning member's membership_verified attribute is set in ↩
    this function. Use other means to
681     invite members.
682
683     :param msg_list: The message list where the group will be added.
684     :param ug: The user group being added to a message list.
685     :param send_right: Send right for user groups members, that will be added to the ↩
    message list individually.
686     :param delivery_right: Delivery right for user groups members, that will be added ↩
    to the message list individually.
687     :param em_list: An optional email list. If given, then all the members of the ↩
    user group will also be subscribed to
688     the email list.
689     """
690     # Check right to a group. Right checking is not required for personal groups, ↩
    only user groups.
691     if not ug.is_personal_group and not has_manage_access(ug.admin_doc):
692         return
693
694     # Check for membership duplicates.
695     member = msg_list.find_member(username=ug.name, email=None)
696     if member and not member.membership_ended:
697         return
698     # Add the user group as a member to the message list.
699     new_group_member = MessageListTimMember(message_list_id=msg_list.id, group_id=ug.id,
700                                             delivery_right=delivery_right, ↩
    send_right=send_right,
701                                             membership_verified=get_current_time())
702     db.session.add(new_group_member)
703
704     # Add group's individual members to message channels.
705     if em_list is not None:
706         for user in ug.users:
707             # TODO: Search for a set of emails and a primary email here when users' ↩
    additional emails are implemented.
708             user_email = user.email  # In the future, we can search for a set of ↩
    emails and a primary email here.
709             add_email(em_list, user_email, email_owner_pre_confirmation=True, ↩
    real_name=user.real_name,
710                       send_right=send_right, delivery_right=delivery_right)
711
712
713 def add_message_list_external_email_member(msg_list: MessageListModel, ↩
    external_email: str,
```

```
714                                         send_right: bool, delivery_right: bool, em_list: MailingList,
715                                         display_name: Optional[str]) -> None:
716     """Add external member to a message list. External members at this moment only support external members to email
717      lists.
718
719     :param msg_list: Message list where the member is to be added.
720     :param external_email: The email address of an external member to be added to the message list.
721     :param send_right: The send right to the list by the new member.
722     :param delivery_right: The delivery right to the list by the new member.
723     :param em_list: The email list where this external member will be also added, because at this time external members
724     only make sense for an email list.
725     :param display_name: Optional name associated with the external member.
726     """
727     # Check for duplicate members.
728     if msg_list.find_member(username=None, email=external_email):
729         return
730
731     new_member = MessageListExternalMember(email_address=external_email, display_name=display_name,
732                                            delivery_right=delivery_right, send_right=send_right,
733                                            message_list_id=msg_list.id)
734     db.session.add(new_member)
735     add_email(em_list, external_email, email_owner_pre_confirmation=True, real_name=display_name,
736               send_right=send_right, delivery_right=delivery_right)
737
738
739 def sync_message_list_on_add(user: User, new_group: UserGroup) -> None:
740     """On adding a user to a new group, sync the user to user group's message lists.
741
742     :param user: The user that was added to the new_group.
743     :param new_group: The new group that the user was added to.
744     """
745     # TODO: This might become a bottle neck, as adding to group is often done in a loop and every sync is a potential
746     #  call to different message channels (now just Mailman). In order to rid ourselves of that, we might need to
747     #  revamp the syncing. A solution might be a call to (Mailman's) server (sidelining mailmanclient-library) with a
748     #  batch of users we want to add with necessary information, and then let the server handle adding in a loop
749     #  locally.
750
751     # Get all the message lists for the user group.
752     for group_tim_member in new_group.messagelist_membership:
753         group_message_list: MessageListModel = group_tim_member.message_list
754         # Propagate the adding on message list's message channels.
755         if group_message_list.email_list_domain:
756             email_list = get_email_list_by_name(group_message_list.name, group_message_list.email_list_domain)
757             add_email(email_list, user.email, True, user.real_name,
758                       group_tim_member.member.send_right, group_tim_member.member.delivery_right)
759
```

```
760
761 def sync_message_list_on_expire(user: User, old_group: UserGroup) -> None:
762     """On removing a user from a user group, remove the user from all the message ↩
    lists that watch the group.
763
764     :param user: The user who was removed from the user group.
765     :param old_group: The group where the user was removed from.
766     """
767     # TODO: This might become a bottle neck, as removing from group is often done in ↩
    a loop and every sync is a
768     #  potential call to different message channels (now just Mailman). In order to ↩
    rid ourselves of that,
769     #  we might need to revamp the syncing. A solution might be a call to (Mailman's) ↩
    server (sidelining
770     #  mailmanclient-library) with a batch of users we want to add with necessary ↩
    information, and then let the
771     #  server handle removing in a loop locally.
772
773     # Get all the message lists for the user group.
774     for group_tim_member in old_group.messagelist_membership:
775         group_message_list: MessageListModel = group_tim_member.message_list
776         # Propagate the deletion on message list's message channels.
777         if group_message_list.email_list_domain:
778             email_list = get_email_list_by_name(group_message_list.name, ↩
    group_message_list.email_list_domain)
779             email_list_member = get_email_list_member(email_list, user.email)
780             remove_email_list_membership(email_list_member)
781
782
783 def set_message_list_member_removed_status(member: MessageListMember,
784                                            removed: Optional[datetime], email_list: ↩
    Optional[MailingList]) -> None:
785     """Set the message list member's membership removed status.
786
787     :param member: The member who's membership status is being set.
788     :param removed: Member's date of removal from the message list. If None, then the ↩
    member is an active member on the
789     list.
790     :param email_list: An email list belonging to the message list. If None, the ↩
    message list does not have an email
791     list.
792     """
793     if (member.membership_ended is None and removed is None) or ↩
    (member.membership_ended and removed):
794         return
795
796     member.remove(removed)
797     # Remove members from email list or return them there.
798     if email_list:
799         if member.is_group():
800             ug = member.tim_member.user_group
801             ug_members = ug.users
802             for ug_member in ug_members:
803                 mlist_member = get_email_list_member(email_list, ug_member.email)
804                 if removed:
805                     remove_email_list_membership(mlist_member)
806                 else:
807                     # Re-set the member's send and delivery rights on the email list.
808                     set_email_list_member_send_status(mlist_member, member.send_right)
```

```
809                     set_email_list_member_delivery_status(mlist_member, ↩
    member.delivery_right)
810         elif member.is_personal_user():
811             # Make changes to member's status on the email list.
812             mlist_member = get_email_list_member(email_list, member.get_email())
813             # If there is an email list and the member is removed, do a soft removal ↩
    on the email list.
814             if removed:
815                 remove_email_list_membership(mlist_member)
816             else:
817                 # Re-set the member's send and delivery rights on the email list.
818                 set_email_list_member_send_status(mlist_member, member.send_right)
819                 set_email_list_member_delivery_status(mlist_member, ↩
    member.delivery_right)
820
821
822 def set_member_send_delivery(member: MessageListMember, send: bool, delivery: bool,
823                              email_list: Optional[MailingList] = None) -> None:
824     """Set message list member's send and delivery rights.
825
826     :param member: Member who's rights are being set.
827     :param send: Member's new send right.
828     :param delivery: Member's new delivery right.
829     :param email_list: If the message list has email list as one of its message ↩
    channels, set the send and delivery
830      rights there also.
831     :return: None.
832     """
833     # Send right
834     if member.send_right != send:
835         member.send_right = send
836         if email_list:
837             if member.is_personal_user():
838                 mlist_member = get_email_list_member(email_list, member.get_email())
839                 set_email_list_member_send_status(mlist_member, send)
840             elif member.is_group():
841                 # For group, set the delivery status for its members on the email list.
842                 ug = member.tim_member.user_group
843                 ug_members = ug.users  # ug.current_memberships
844                 for ug_member in ug_members:
845                     # user = ug_member.personal_user
846                     email_list_member = get_email_list_member(email_list, ↩
    ug_member.email)
847                     set_email_list_member_send_status(email_list_member, send)
848
849     # Delivery right.
850     if member.delivery_right != delivery:
851         member.delivery_right = delivery
852         if email_list:
853             # If message list has an email list associated with it, set delivery ↩
    rights there.
854             if member.is_personal_user():
855                 mlist_member = get_email_list_member(email_list, member.get_email())
856                 set_email_list_member_delivery_status(mlist_member, delivery)
857             elif member.is_group():
858                 # For group, set the delivery status for its members on the email list.
859                 ug = member.tim_member.user_group
860                 ug_members = ug.users  # ug.current_memberships
861                 for ug_member in ug_members:
```

```
                    # user = ug_member.personal_user
                    email_list_member = get_email_list_member(email_list, ug_member.email)
                    set_email_list_member_delivery_status(email_list_member, delivery)


def set_message_list_description(message_list: MessageListModel, description: Optional[str]) -> None:
    """Set a (short) description to a message list and its associated message channels.

    :param message_list: The message list where the description is set.
    :param description: The new description. If None, keep the current value.
    """
    if description is None or message_list.description == description:
        return
    message_list.description = description
    if message_list.email_list_domain:
        email_list = get_email_list_by_name(message_list.name, message_list.email_list_domain)
        set_email_list_description(email_list, description)


def set_message_list_info(message_list: MessageListModel, info: Optional[str]) -> None:
    """Set a long description (called 'info' on Mailman) to a message list and its associated message channels.

    :param message_list: The message list where the (long) description is set.
    :param info: The new long description. If None, keep the current value.
    """
    if info is None or message_list.info == info:
        return
    message_list.info = info
    if message_list.email_list_domain:
        email_list = get_email_list_by_name(message_list.name, message_list.email_list_domain)
        set_email_list_info(email_list, info)
```

# timApp/messaging/messagelist/routes.py

```
from typing import List, Optional

from flask import Response

from timApp.auth.accesshelper import verify_logged_in, has_manage_access, get_doc_or_abort
from timApp.auth.sessioninfo import get_current_user_object
from timApp.document.docinfo import move_document
from timApp.folder.folder import Folder
from timApp.item.manage import get_trash_folder
from timApp.messaging.messagelist.emaillist import get_email_list_by_name
from timApp.messaging.messagelist.emaillist import get_list_ui_link, create_new_email_list, \
    delete_email_list, verify_emaillist_name_requirements, get_domain_names, verify_mailman_connection
from timApp.messaging.messagelist.listoptions import ListOptions, Distribution, MemberInfo, GroupAndMembers
from timApp.messaging.messagelist.messagelist_models import MessageListModel
from timApp.messaging.messagelist.messagelist_utils import verify_messagelist_name_requirements, new_list, \
```

```
16      set_message_list_notify_owner_on_change, \
17      set_message_list_member_can_unsubscribe, set_message_list_subject_prefix, ↩
        set_message_list_tim_users_can_join, \
18      set_message_list_default_send_right, set_message_list_default_delivery_right, ↩
        set_message_list_only_text, \
19      set_message_list_non_member_message_pass, set_message_list_allow_attachments, ↩
        set_message_list_default_reply_type, \
20      add_new_message_list_group, add_message_list_external_email_member, \
21      set_message_list_member_removed_status, set_member_send_delivery, ↩
        set_message_list_description, \
22      set_message_list_info
23  from timApp.timdb.sqa import db
24  from timApp.user.groups import verify_groupadmin
25  from timApp.user.usergroup import UserGroup
26  from timApp.util.flask.requesthelper import RouteException
27  from timApp.util.flask.responsehelper import json_response, ok_response
28  from timApp.util.flask.typedblueprint import TypedBlueprint
29  from timApp.util.logger import log_error
30  from timApp.util.utils import is_valid_email, get_current_time
31
32  messagelist = TypedBlueprint('messagelist', __name__, url_prefix='/messagelist')
33
34
35  @messagelist.route('/createlist', methods=['POST'])
36  def create_list(options: ListOptions) -> Response:
37      """Handles creating a new message list.
38
39      :param options All options necessary for establishing a new message list.
40      :return: A Response with the list's management doc included. This way the creator ↩
        can re-directed to the list's
41      management page directly.
42      """
43      # Access right checks. The creator of the list has to be a group admin. This ↩
        probably changes in the future.
44      verify_logged_in()
45      verify_groupadmin()
46
47      # Current user is set as the default owner.
48      owner = get_current_user_object()
49
50      # Options object is given directly to new_list, so we don't want to use temporary ↩
        variable for stripped name.
51      options.name = options.name.strip()
52
53      test_name(options.name)
54      manage_doc, message_list = new_list(options)
55
56      if options.domain:
57          verify_mailman_connection()
58          create_new_email_list(options, owner)
59          # Add the domain to a message list only after the email list has been ↩
        created. This way if the list creation
60          # fails, we have indication that the list does not have an email list ↩
        attached to it.
61          message_list.email_list_domain = options.domain
62
63      set_message_list_subject_prefix(message_list, f"[{message_list.name}]")
64
65      db.session.commit()
```

```
66      return json_response(manage_doc)
67
68
69  def test_name(name_candidate: str) -> None:
70      """Check new message list's name candidate's name.
71
72       The name has to meet naming rules, it has to be not already be in use and it cannot be a reserved name. If the
73       function retuns control to its caller, then name is viable to use for a message list. If at some point the name
74       is not viable, then an exception is raised.
75
76      :param name_candidate: The name candidate to check.
77      """
78      normalized_name = name_candidate.strip()
79      name, sep, domain = normalized_name.partition("@")
80      verify_messagelist_name_requirements(name)
81      if sep:
82          # If character '@' is found, we check email list specific name requirements.
83          verify_mailman_connection()
84          verify_emaillist_name_requirements(name, domain)
85
86
87  @messagelist.route("/domains", methods=['GET'])
88  def domains() -> Response:
89      """Send possible domains for a client, if such exists.
90
91      :return: If domains are configured, return them as an array.
92      """
93      verify_mailman_connection()
94      possible_domains: List[str] = get_domain_names()
95      return json_response(possible_domains)
96
97
98  @messagelist.route("/deletelist", methods=['DELETE'])
99  def delete_list(listname: str, permanent: bool) -> Response:
100     """Delete message and  associated message channels.
101
102     :param listname: The list to be deleted.
103     :param permanent: A boolean flag indicating if the deletion is meant to be permanent.
104     :return: A string describing how the operation went.
105     """
106     # Check access rights.
107     verify_logged_in()
108     message_list = MessageListModel.get_list_by_name_exactly_one(listname)
109     if not has_manage_access(message_list.block):
110         raise RouteException("You need at least a manage access to the list in order to do this action.")
111
112     # The amount of docentries a message list's block relationship refers to should be one. If not, something is
113     # terribly wrong.
114     if len(message_list.block.docentries) > 1:
115         log_error(f"Message list '{listname}' has multiple docentries to its block relationship.")
116         raise RouteException("Can't perform deletion at this time. The problem has been logged for admins.")
117
```

```
118     # Perform deletion.
119     if permanent:
120         # If the deletion is (more) permanent, move the admin doc to bin.
121         manage_doc = get_doc_or_abort(message_list.manage_doc_id)
122         trash_folder: Folder = get_trash_folder()
123         move_document(manage_doc, trash_folder)
124     # Set the db entry as removed
125     message_list.removed = get_current_time()
126 
127     if message_list.email_list_domain:
128         # A message list has a domain, so we are also looking to delete an email list.
129         verify_mailman_connection()
130         email_list = get_email_list_by_name(message_list.name, ↩
    message_list.email_list_domain)
131         delete_email_list(email_list, permanent_deletion=permanent)
132 
133     db.session.commit()
134     return ok_response()
135 
136 
137 @messagelist.route("/getlist/<int:document_id>", methods=['GET'])
138 def get_list(document_id: int) -> Response:
139     """Get the information for a message list.
140 
141     :param document_id: ID for message list's admin document.
142     :return: ListOptions with the list's information.
143     """
144     verify_logged_in()
145 
146     message_list = MessageListModel.get_list_by_manage_doc_id(document_id)
147     list_options = ListOptions(
148         name=message_list.name,
149         notify_owners_on_list_change=message_list.notify_owner_on_change,
150         domain=message_list.email_list_domain,
151         archive=message_list.archive,
152         default_reply_type=message_list.default_reply_type,
153         tim_users_can_join=message_list.tim_user_can_join,
154         list_subject_prefix=message_list.subject_prefix,
155         members_can_unsubscribe=message_list.can_unsubscribe,
156         default_send_right=message_list.default_send_right,
157         default_delivery_right=message_list.default_delivery_right,
158         only_text=message_list.only_text,
159         non_member_message_pass=message_list.non_member_message_pass,
160         email_admin_url=get_list_ui_link(message_list.name, ↩
    message_list.email_list_domain),
161         list_info=message_list.info,
162         list_description=message_list.description,
163         allow_attachments=message_list.allow_attachments,
164         distribution=Distribution(email_list=True, tim_message=True),
165         removed=message_list.removed
166     )
167     return json_response(list_options)
168 
169 
170 @messagelist.route("/save", methods=['POST'])
171 def save_list_options(options: ListOptions) -> Response:
172     """Save message list's options.
173 
174     :param options: The options to be saved.
```

```
175     :return: OK response.
176     """
177     # Check access rights.
178     verify_logged_in()
179     message_list = MessageListModel.get_list_by_name_exactly_one(options.name)
180     if not has_manage_access(message_list.block):
181         raise RouteException("You need at least a manange access to the list in order ↩
    to do this action.")
182 
183     if message_list.archive_policy != options.archive:
184         # TODO: If message list changes its archive policy, the members on the list ↩
    need to be notified. Insert
185         #  messaging here.
186         message_list.archive = options.archive
187 
188     set_message_list_description(message_list, options.list_description)
189     set_message_list_info(message_list, options.list_info)
190     set_message_list_notify_owner_on_change(message_list, ↩
    options.notify_owners_on_list_change)
191     set_message_list_member_can_unsubscribe(message_list, ↩
    options.members_can_unsubscribe)
192     set_message_list_subject_prefix(message_list, options.list_subject_prefix)
193     set_message_list_only_text(message_list, options.only_text)
194     set_message_list_allow_attachments(message_list, options.allow_attachments)
195     set_message_list_tim_users_can_join(message_list, options.tim_users_can_join)
196     set_message_list_non_member_message_pass(message_list, ↩
    options.non_member_message_pass)
197     set_message_list_default_send_right(message_list, options.default_send_right)
198     set_message_list_default_delivery_right(message_list, ↩
    options.default_delivery_right)
199     set_message_list_default_reply_type(message_list, options.default_reply_type)
200 
201     db.session.commit()
202     return ok_response()
203 
204 
205 @messagelist.route("/savemembers", methods=['POST'])
206 def save_members(listname: str, members: List[MemberInfo]) -> Response:
207     """Save the state of existing list members, e.g. send and delivery rights.
208 
209     :param listname: The name of the message list where the members will be saved.
210     :param members: The members to be saved.
211     :return: Response for the client. The Response is a simple ok_response().
212     """
213     # Check access rights.
214     verify_logged_in()
215     message_list = MessageListModel.get_list_by_name_exactly_one(listname)
216     if not has_manage_access(message_list.block):
217         raise RouteException("You need at least a manage access to the list to do ↩
    this action.")
218 
219     email_list = None
220     if message_list.email_list_domain:
221         # If there is a domain configured for a list and the Mailman connection is ↩
    not configured, we can't continue
222         # at this time.
223         verify_mailman_connection()
224         email_list = get_email_list_by_name(message_list.name, ↩
    message_list.email_list_domain)
```

```
225
226     for member in members:
227         db_member = message_list.find_member(member.name, member.email)
228         # This if mostly guards against type errors, but what if we legitimely can't ↩
        find them? They are given from
229         # the db in the first place. Is there a reasonable way to communicate this ↩
        state?
230         if db_member:
231             set_member_send_delivery(db_member, member.sendRight, ↩
        member.deliveryRight, email_list=email_list)
232             set_message_list_member_removed_status(db_member, member.removed, ↩
        email_list=email_list)
233     db.session.commit()
234     return ok_response()
235
236
237 def parse_external_member(external_member_candidate: str) -> Optional[List[str]]:
238     """Parse the information of an external member.
239
240     There are two supported ways to give external members. The user can write
241
242         user.userington@domain.fi User Userington
243     or
244         User Userington <user.userington@domain.fi>
245
246     :param external_member_candidate: A string represeting the external member.
247     :return: Return a list of the form [email, name_part_1, name_part_2, ...] if ↩
        parsing was successful. Otherwise
248     return None.
249     """
250     # Split the name candidate to a list for processing.
251     words = external_member_candidate.strip().split(' ')
252
253     # Check if the first word is the email.
254     if is_valid_email(words[0]):
255         return words
256
257     # If the first word was not an email, then check if the email is at the last ↩
        word, in angle brackets.
258     open_bracket_index = words[-1].find("<")
259     closing_bracket_index = words[-1].find(">")
260     if open_bracket_index != -1 and closing_bracket_index != -1:
261         # Remove angle brackets around the email.
262         email = words[-1].strip("<").strip(">")
263         if is_valid_email(email):
264             return_list = [email]
265             return_list.extend(words[0:-1])
266             return return_list
267
268     # If we are here, then no applicable version of external member's information was ↩
        given.
269     return None
270
271
272 @messagelist.route("/addmember", methods=['POST'])
273 def add_member(member_candidates: List[str], msg_list: str, send_right: bool, ↩
        delivery_right: bool) -> Response:
274     """Add new members to a message list.
275
```

```
276     :param member_candidates: Names of member candidates.
277     :param msg_list: The message list where we are trying to add new members.
278     :param send_right: The send right on a list for all the member candidates.
279     :param delivery_right: The delivery right on a list for all the member candidates.
280     :return: OK response.
281     """
282     # Check access right.
283     verify_logged_in()
284     message_list = MessageListModel.get_list_by_name_exactly_one(msg_list)
285     if not has_manage_access(message_list.block):
286         raise RouteException("You need at least a manage access to the list to do this action.")
287 
288     # TODO: Implement checking whether or not users are just added to a list (like they are now) or they are invited
289     #  to a list (requires link generation and other things).
290 
291     em_list = None
292     if message_list.email_list_domain is not None:
293         verify_mailman_connection()
294         em_list = get_email_list_by_name(message_list.name, message_list.email_list_domain)
295 
296     for member_candidate in member_candidates:
297         # For user groups and individual users.
298         ug = UserGroup.get_by_name(member_candidate.strip())
299         if ug is not None:
300             # The name belongs to a user group.
301             add_new_message_list_group(message_list, ug, send_right, delivery_right, em_list)
302 
303         # For external members.
304         # If member candidate is not a user, or a user group, then we assume an external member. Add external members.
305         external_member = parse_external_member(member_candidate)
306         if external_member and em_list:
307             if len(external_member) == 1:
308                 # There is no display name given for external member.
309                 dname = None
310             else:
311                 # Construct an optional display name by joining all the other words given on the line that are not
312                 # the email address. Remove possible white space between the email address and the first part of a
313                 # display name. Other white space within the name is left as is.
314                 dname = ' '.join(external_member[1:]).lstrip()
315             add_message_list_external_email_member(message_list, external_member[0],
316                                                    send_right, delivery_right, em_list, display_name=dname)
317     db.session.commit()
318     return ok_response()
319 
320 
321 @messagelist.route("/getmembers/<list_name>", methods=['GET'])
322 def get_members(list_name: str) -> Response:
323     """Get members belonging to a certain list.
324 
325     :param list_name: The list where we are querying all the members.
326     :return: All the members of a list.
```

```
    """
    verify_logged_in()
    msg_list = MessageListModel.get_list_by_name_exactly_one(list_name)
    if not has_manage_access(msg_list.block):
        raise RouteException("You are not authorized to see the members of this list.")
    list_members = msg_list.members
    return json_response(list_members)


@messagelist.route("/getgroupmembers/<list_name>", methods=['GET'])
def get_group_members(list_name: str) -> Response:
    """View function for getting members of groups that are on a message list.

    :param list_name: Message list.
    :return: All members of groups associated in a message list as a list of GroupAndMembers objects.
    """
    # Check rights.
    verify_logged_in()
    message_list = MessageListModel.get_list_by_name_exactly_one(list_name)
    if not has_manage_access(message_list.block):
        raise RouteException("Only an owner of this list can see the members of this group.")

    # Get group.
    groups = [member for member in message_list.members if member.is_group()]

    # At this point we assume we have a user that is a TIM user group.
    groups_and_members = []
    for group in groups:
        user_group: UserGroup = group.user_group
        # Create a MemberInfo object for every current user in the group. As these are current members of the user
        # group, removed is None.
        group_members = [MemberInfo(name=user.real_name, username=user.name,
                                    sendRight=group.send_right, deliveryRight=group.delivery_right,
                                    removed=None, email=user.email)
                         for user in user_group.users]
        gm = GroupAndMembers(groupName=user_group.name, members=group_members)
        groups_and_members.append(gm)
    return json_response(groups_and_members)
```

# timApp/messaging/timMessage/init.py

# timApp/messaging/timMessage/internalmessage_models.py

```
from enum import Enum
from typing import Dict, Any

from timApp.timdb.sqa import db


class DisplayType(Enum):
    TOP_OF_PAGE = 1
    STICKY = 2


```

```
class InternalMessage(db.Model):
    """A TIM message."""

    __tablename__ = 'internalmessage'

    id = db.Column(db.Integer, primary_key=True)
    """Message identifier."""

    doc_id = db.Column(db.Integer, db.ForeignKey('block.id'), nullable=False)
    """Block identifier."""

    par_id = db.Column(db.Text, nullable=False)
    """Paragraph identifier."""

    can_mark_as_read = db.Column(db.Boolean, nullable=False)
    """Whether the recipient can mark the message as read."""

    reply = db.Column(db.Boolean, nullable=False)
    """Whether the message can be replied to."""

    display_type = db.Column(db.Enum(DisplayType), nullable=False)
    """How the message is displayed."""

    expires = db.Column(db.DateTime)
    """When the message display will disappear."""

    replies_to = db.Column(db.Integer)
    """Id of the message which this messages is a reply to"""

    displays = db.relationship('InternalMessageDisplay', back_populates='message')
    readreceipts = db.relationship('InternalMessageReadReceipt', back_populates='message')
    block = db.relationship('Block', back_populates='internalmessage')

    def to_json(self) -> Dict[str, Any]:
        return {'id': self.id,
                'doc_id': self.doc_id,
                'par_id': self.par_id,
                'can_mark_as_read': self.can_mark_as_read,
                'reply': self.reply,
                'display_type': self.display_type,
                'displays': self.displays,
                'readreceipts': self.readreceipts,
                }


class InternalMessageDisplay(db.Model):
    """Where and for whom a TIM message is displayed."""

    __tablename__ = 'internalmessage_display'

    id = db.Column(db.Integer, primary_key=True)
    """Message display identifier."""

    message_id = db.Column(db.Integer, db.ForeignKey('internalmessage.id'), nullable=False)
    """Message identifier."""

    usergroup_id = db.Column(db.Integer, db.ForeignKey('usergroup.id'))
```

```
    """Who sees the message; if null, displayed for everyone."""

    display_doc_id = db.Column(db.Integer, db.ForeignKey('block.id'))
    """
    Identifier for the document or the folder where the message is displayed. If null,
    the message is displayed globally.
    """

    message = db.relationship('InternalMessage', back_populates='displays')
    usergroup = db.relationship('UserGroup', back_populates='internalmessage_display')
    display_block = db.relationship('Block', back_populates='internalmessage_display')

    def to_json(self) -> Dict[str, Any]:
        return {'id': self.id,
                'message_id': self.message_id,
                'usergroup_id': self.usergroup_id,
                'display_doc_id': self.display_doc_id,
                }


class InternalMessageReadReceipt(db.Model):
    """Metadata about read receipts."""

    __tablename__ = 'internalmessage_readreceipt'

    rcpt_id = db.Column(db.Integer, db.ForeignKey('usergroup.id'), primary_key=True)
    """Message recipient identifier."""

    message_id = db.Column(db.Integer, db.ForeignKey('internalmessage.id'), ↩
    primary_key=True)
    """Message identifier."""

    user_id = db.Column(db.Integer, db.ForeignKey('useraccount.id'))
    """Identifier for the user who marked the message as read."""

    marked_as_read_on = db.Column(db.DateTime)
    """Timestamp for when the message was marked as read."""

    recipient = db.relationship('UserGroup', ↩
    back_populates='internalmessage_readreceipt')
    message = db.relationship('InternalMessage', back_populates='readreceipts')
    user = db.relationship('User', back_populates='internalmessage_readreceipt')

    def to_json(self) -> Dict[str, Any]:
        return {'rcpt_id': self.rcpt_id,
                'message_id': self.message_id,
                'user_id': self.user_id,
                'marked_as_read_on': self.marked_as_read_on,
                }
```

# timApp/messaging/timMessage/routes.py

```
import re
from dataclasses import dataclass
from typing import Optional, List
from datetime import datetime

from flask import Response
from sqlalchemy import tuple_
```

```
8
9  from timApp.document.document import Document
10 from timApp.document.documents import import_document_from_file
11 from timApp.document.viewcontext import default_view_ctx
12 from timApp.item.manage import get_trash_folder
13 from timApp.util.flask.requesthelper import RouteException, NotExist
14 from timApp.auth.accesshelper import verify_logged_in
15 from timApp.auth.accesstype import AccessType
16 from timApp.auth.sessioninfo import get_current_user_object
17 from timApp.auth.accesshelper import verify_edit_access
18 from timApp.document.create_item import create_document
19 from timApp.document.docentry import DocEntry
20 from timApp.document.docinfo import DocInfo
21 from timApp.folder.createopts import FolderCreationOptions
22 from timApp.folder.folder import Folder
23 from timApp.item.item import Item
24 from timApp.messaging.timMessage.internalmessage_models import InternalMessage, ↩
       DisplayType, InternalMessageDisplay, \
25      InternalMessageReadReceipt
26 from timApp.timdb.sqa import db
27 from timApp.user.user import User
28 from timApp.user.usergroup import UserGroup
29 from timApp.util.flask.responsehelper import ok_response, json_response, error_generic
30 from timApp.util.flask.typedblueprint import TypedBlueprint
31 from timApp.util.utils import remove_path_special_chars, static_tim_doc
32
33 timMessage = TypedBlueprint('timMessage', __name__, url_prefix='/timMessage')
34
35
36 @dataclass
37 class MessageOptions:
38     # Options regarding TIM messages
39     messageChannel: bool
40     important: bool
41     isPrivate: bool
42     archive: bool
43     pageList: str
44     readReceipt: bool
45     reply: bool
46     sender: str
47     senderEmail: str
48     repliesTo: Optional[int] = None
49     expires: Optional[datetime] = None
50
51
52 @dataclass
53 class ReplyOptions:
54     archive: bool
55     messageChannel: bool
56     pageList: str
57     recipient: str
58     readReceipt: bool = True
59     repliesTo: Optional[int] = None
60
61
62 @dataclass
63 class MessageBody:
64     messageBody: str
65     messageSubject: str
```

```
66      recipients: List[str]
67
68
69  @dataclass
70  class TimMessageData:
71      # Information about the message sent to browser
72      id: int
73      sender: str
74      doc_id: int
75      par_id: str
76      can_mark_as_read: bool
77      can_reply: bool
78      display_type: int
79      message_body: str
80      message_subject: str
81      recipients: List[str]
82
83
84  @dataclass
85  class TimMessageReadReceipt:
86      rcpt_id: int
87      message_id: int
88      user_id: int
89      marked_as_read_on: datetime
90      can_mark_as_read: bool
91
92
93  @timMessage.route("/get/<int:item_id>", methods=['GET'])
94  def get_tim_messages(item_id: int) -> Response:
95      """
96      Retrieve messages displayed for current based on item id and return them in json ↩
        format.
97
98      :param item_id: Identifier for document or folder where message is displayed
99      :return:
100     """
101     fullmessages = get_tim_messages_as_list(item_id)
102
103     return json_response(fullmessages)
104
105
106 def get_tim_messages_as_list(item_id: int) -> List[TimMessageData]:
107     """
108     Retrieve messages displayed for current user based on item id and return them as ↩
        a list.
109
110     TODO: Once global messages are implemented, verify that user has view access
111      to the document before displaying messages.
112
113     :param item_id: Identifier for document or folder where message is displayed
114     :return:
115     """
116     current_page_obj = DocEntry.query.filter_by(id=item_id).first()
117     if not current_page_obj:
118         current_page_obj = Folder.query.filter_by(id=item_id).first()
119         if not current_page_obj:
120             raise NotExist('No document or folder found')
121
122     parent_paths = current_page_obj.parent_paths()  # parent folders
```

```
123
124     # get message displays shown on current page or in parent folders
125     displays = InternalMessageDisplay.query \
126         .outerjoin(Folder, Folder.id == InternalMessageDisplay.display_doc_id) \
127         .filter((InternalMessageDisplay.usergroup == ↩
    get_current_user_object().get_personal_group())
128                 & ((InternalMessageDisplay.display_block == current_page_obj)
129                    | (tuple_(Folder.location, Folder.name).in_(parent_paths))))
130
131     replies = ↩
    InternalMessage.query.filter(InternalMessage.replies_to.isnot(None)).with_entities(
132         InternalMessage.replies_to).all()
133     replies_to_ids = [a for a, in replies]  # list of messages that have been replied to
134
135     messages = []
136     recipients = []
137     for display in displays:
138         receipt = ↩
    InternalMessageReadReceipt.query.filter_by(rcpt_id=display.usergroup_id,
139                                                                 ↩
    message_id=display.message_id).first()
140         expires = InternalMessage.query.filter_by(id=display.message_id).first()
141         # message is shown if it has not been marked as read or replied to, and has ↩
    not expired
142         if receipt.marked_as_read_on is None and (expires.expires is None or ↩
    expires.expires > datetime.now()):
143             ↩
    messages.append(InternalMessage.query.filter_by(id=display.message_id).first())
144             ↩
    recipients.append(UserGroup.query.filter_by(id=display.usergroup_id).first())
145
146     fullmessages = []
147     for message in messages:
148         document = DocEntry.query.filter_by(id=message.doc_id).first()
149         if not document:
150             return error_generic('Message document not found', 404)
151
152         # Hides the message if the corresponding document has been deleted
153         trash_folder_path = get_trash_folder().path
154         if document.name.startswith(trash_folder_path):
155             continue
156
157         fullmessages.append(TimMessageData(id=message.id,
158                                            sender=document.owners.pop().name,
159                                            doc_id=message.doc_id,
160                                            par_id=message.par_id,
161                                            can_mark_as_read=message.can_mark_as_read,
162                                            can_reply=message.reply,
163                                            display_type=message.display_type,
164                                            ↩
    message_body=Document.get_paragraph(document.document,
165                                                                                ↩
    message.par_id).get_html(
166                                                default_view_ctx),
167                                            message_subject=document.title,
168                                            recipients=recipients))
169
170     return fullmessages
171
```

```
172
173 @timMessage.route("/get_read_receipt/<int:doc_id>", methods=['GET'])
174 def get_read_receipt(doc_id: int) -> Response:
175     """
176     Retrieve read receipt object for the current user and message related to the ↩
    given document id
177
178     :param doc_id: Id of the message document
179     :return:
180     """
181     message = InternalMessage.query.filter_by(doc_id=doc_id).first()
182     receipt = ↩
    InternalMessageReadReceipt.query.filter_by(rcpt_id=get_current_user_object().get_personal_group
183                                                                 message_id=message.id).first()
184     if not receipt:
185         raise NotExist('Read receipt not found')
186
187     receipt_data = TimMessageReadReceipt(rcpt_id=receipt.rcpt_id, message_id=message.id,
188                                          user_id=receipt.user_id, ↩
    marked_as_read_on=receipt.marked_as_read_on,
189                                          can_mark_as_read=message.can_mark_as_read)
190
191     return json_response(receipt_data)
192
193 # Regex pattern for url verification.
194 URL_PATTERN = ↩
    re.compile(r"https?://[a-z0-9.-]*/(show_slide|view|teacher|velp|answers|lecture|review|slide)/"
195
196 @timMessage.route("/url_check", methods=['POST'])
197 def check_urls(urls: str) -> Response:
198     """
199     Checks if given URLS's exist in TIM and that user has right to post TIM message ↩
    to them
200
201     :param urls: Urls where user wishes to post TIM message
202     :return: Shortened urls to show the user in the UI, or an error message
203     """
204     url_list = list(
205         filter(None, urls.splitlines()))  # turn URL string into a list with empty ↩
    values (new lines) removed
206     valid_urls: List[str] = []
207     error_message: str = ""
208     status_code: int
209
210     for url in url_list:
211         url = url.strip()  # remove leading and trailing whitespaces
212         if url.endswith("/"):
213             url = url[:-1]
214         hashtag_index = url.find("#")  # remove anchors
215         if hashtag_index != -1:
216             url = url[:hashtag_index]
217         if URL_PATTERN.search(url):  # check if url matches the TIM urls' pattern
218             shortened_url = URL_PATTERN.sub("", url)
219         else:
220             shortened_url = url
221         document = DocEntry.find_by_path(shortened_url)  # check if url exists in TIM
222         if document is None:
223             document = Folder.find_by_path(shortened_url)
224         if document is None:
```

```
225             error_message = url + " was not found in TIM"
226             status_code = 404
227             break
228         try:  # check if user has permission to edit the url
229             verify_edit_access(document)
230             valid_urls.append(shortened_url)
231         except Exception:
232             error_message = "You don't have permission to post TIM message to " + url
233             status_code = 401
234 
235     if error_message:
236         return json_response({"error": error_message}, status_code)
237     else:
238         valid_urls_string = "\n".join(valid_urls)  # turn URL list into a string again
239         return json_response({"shortened_urls": valid_urls_string}, 200)
240 
241 
242 @timMessage.route("/send", methods=['POST'])
243 def send_tim_message(options: MessageOptions, message: MessageBody) -> Response:
244     return send_message_or_reply(options, message)
245 
246 
247 def send_message_or_reply(options: MessageOptions, message: MessageBody) -> Response:
248     """
249         Creates a new TIM message and saves it to database.
250 
251         :param options: Options related to the message
252         :param message: Message subject, contents and sender
253         :return:
254         """
255     verify_logged_in()
256 
257     tim_message = InternalMessage(can_mark_as_read=options.readReceipt, ↩
        reply=options.reply, expires=options.expires,
258                                   replies_to=options.repliesTo)
259     create_tim_message(tim_message, options, message)
260     db.session.add(tim_message)
261 
262     pages = get_display_pages(options.pageList.splitlines())
263     recipients = get_recipient_users(message.recipients)
264     create_message_displays(tim_message, pages, recipients)
265     if recipients:
266         create_read_receipts(tim_message, recipients)
267 
268     db.session.commit()
269 
270     return ok_response()
271 
272 
273 def create_tim_message(tim_message: InternalMessage, options: MessageOptions, ↩
        message_body: MessageBody) -> DocInfo:
274     """
275     Creates a TIM document for the message to the TIM messages folder at TIM's root.
276 
277     :param tim_message: InternalMessage object
278     :param options: Options related to the message
279     :param message_body: Message subject, contents and list of recipients
280     :return: The created Document object
281     """
```

```
282     sender = get_current_user_object()
283     recipient_users = get_recipient_users(message_body.recipients)
284     message_folder_path = 'messages/tim-messages'
285 
286     message_subject = message_body.messageSubject
287     timestamp = datetime.now()  # add timestamp to document path to make it unique
288     message_path = remove_path_special_chars(f'{timestamp}-{message_subject}')
289 
290     check_messages_folder_path('messages', message_folder_path)
291     message_doc = create_document(f'{message_folder_path}/{message_path}',
292                                   message_subject)
293     message_doc.block.add_rights([sender.get_personal_group()], AccessType.owner)
294     message_doc.block.add_rights(recipient_users, AccessType.view)
295 
296     update_tim_msg_doc_settings(message_doc, sender, message_body)
297     message_par = message_doc.document.add_paragraph(message_body.messageBody)
298     message_doc.document.add_paragraph('<manage-read-receipt></manage-read-receipt>', ↩
      attrs={"allowangular": "true"})
299     tim_message.block = message_doc.block
300     tim_message.par_id = message_par.get_id()
301 
302     if options.important:
303         # Important messages are interpreted as 'sticky' display type
304         tim_message.display_type = DisplayType.STICKY  # TODO actual functionality
305     else:
306         tim_message.display_type = DisplayType.TOP_OF_PAGE  # default display type
307 
308     return message_doc
309 
310 
311 @timMessage.route("/reply", methods=['POST'])
312 def reply_to_tim_message(options: ReplyOptions, messageBody: MessageBody) -> Response:
313     messageOptions = MessageOptions(options.messageChannel, False, True, ↩
      options.archive, options.pageList,
314                                     options.readReceipt, False, ↩
      get_current_user_object().name,
315                                     get_current_user_object().email, options.repliesTo)
316     recipient = User.get_by_name(messageBody.recipients.pop())
317     if recipient:
318         recipient_email = recipient.email
319     else:
320         raise NotExist('Recipient not found')
321 
322     message = MessageBody(messageBody.messageBody, messageBody.messageSubject, ↩
      [recipient_email])
323 
324     return send_message_or_reply(messageOptions, message)
325 
326 
327 @timMessage.route("/mark_as_read", methods=['POST'])
328 def mark_as_read(message_id: int) -> Response:
329     """
330         Marks given message as read in database.
331         Expects that message receiver and marker are the same person.
332 
333         :param message_id: Id of given message
334         :return:
335         """
336     verify_logged_in()
```

```
337
338     marker = get_current_user_object().get_personal_group().id
339
340     read_receipt = InternalMessageReadReceipt.query.filter_by(rcpt_id=marker, ↩
        message_id=message_id).first()
341     if read_receipt is None:
342         raise RouteException
343     read_receipt.user_id = get_current_user_object().id
344     read_receipt.marked_as_read_on = datetime.now()
345     db.session.add(read_receipt)
346     db.session.commit()
347
348     return ok_response()
349
350
351 @timMessage.route("/cancel_read_receipt", methods=['POST'])
352 def cancel_read_receipt(message_id: int) -> Response:
353     """
354     Removes read receipt date and the user who marked it from the database entry.
355
356     :param message_id: Message identifier
357     :return:
358     """
359     verify_logged_in()
360
361     receipt = ↩
        InternalMessageReadReceipt.query.filter_by(rcpt_id=get_current_user_object().get_personal_group
362                                                                 message_id=message_id).one()
363     receipt.user_id = None
364     receipt.marked_as_read_on = None
365     db.session.commit()
366
367     return ok_response()
368
369
370 def get_recipient_users(recipients: List[str]) -> List[UserGroup]:
371     """
372     Finds UserGroup objects of recipients based on their email
373
374     :param recipients: list of recipients' emails
375     :return: list of recipient UserGroups
376     """
377     users = []
378     for rcpt in recipients:
379         user = User.get_by_email(rcpt)
380         if user:
381             users.append(UserGroup.get_by_name(user.name))
382
383     return users
384
385
386 def get_display_pages(pagelist: List[str]) -> List[Item]:
387     """
388     Finds folders and documents based on their paths.
389
390     :param pagelist: list of paths
391     :return: list of folders and documents
392     """
393     pages: List[Item] = []
```

```
394     for page in pagelist:
395         folder = Folder.find_by_path(page)
396         if folder:
397             pages.append(folder)
398             continue
399 
400         doc = DocEntry.find_by_path(page)
401         if doc:
402             pages.append(doc)
403 
404     return pages
405 
406 
407 def check_messages_folder_path(msg_folder_path: str, tim_msg_folder_path: str) -> ↩
        Folder:
408     """
409     Checks if the /messages/tim-messages folder exists and if not, creates it. All users
410      get view access to /messages folder and edit access to /messages/tim-messages ↩
        folder
411      so that documents for sent messages can be created. Also creates the preamble for
412      message documents.
413 
414     :param msg_folder_path: path for /messages
415     :param tim_msg_folder_path: path for /messages/tim-messages
416     :return: /messages/tim-messages folder
417     """
418     msg_folder = Folder.find_by_location(msg_folder_path, 'messages')
419     admin_group = UserGroup.get_admin_group()
420 
421     if not msg_folder:
422         msg_folder = Folder.create(msg_folder_path, admin_group, title='Messages',
423                                    ↩
        creation_opts=FolderCreationOptions(apply_default_rights=True))
424         msg_block = msg_folder.block
425         if msg_block:
426             msg_block.add_rights([UserGroup.get_logged_in_group()], AccessType.view)
427 
428     tim_msg_folder = Folder.find_by_location(tim_msg_folder_path, 'tim-messages')
429 
430     if not tim_msg_folder:
431         tim_msg_folder = Folder.create(tim_msg_folder_path, admin_group, title='TIM ↩
        messages',
432                                        ↩
        creation_opts=FolderCreationOptions(apply_default_rights=True))
433         tim_msg_block = tim_msg_folder.block
434         if tim_msg_block:
435             tim_msg_block.add_rights([UserGroup.get_logged_in_group()], AccessType.edit)
436 
437     tim_msg_preambles = ↩
        Folder.find_by_location(f'{tim_msg_folder_path}/templates/preambles', 'preambles')
438 
439     if not tim_msg_preambles:
440         tim_msg_templates = Folder.create(f'{tim_msg_folder_path}/templates', ↩
        admin_group,
441                                           title='templates',
442                                           ↩
        creation_opts=FolderCreationOptions(apply_default_rights=True))
443         tim_msg_preambles = ↩
        Folder.create(f'{tim_msg_folder_path}/templates/preambles', admin_group,
```

```
                                        title='preambles',
                                        ↩
    creation_opts=FolderCreationOptions(apply_default_rights=True))

        tim_msg_templates_block = tim_msg_templates.block
        if tim_msg_templates_block:
            tim_msg_templates_block.add_rights([UserGroup.get_logged_in_group()], ↩
    AccessType.view)

        tim_msg_preambles_block = tim_msg_preambles.block
        if tim_msg_preambles_block:
            tim_msg_preambles_block.add_rights([UserGroup.get_logged_in_group()], ↩
    AccessType.view)

    preamble_path = f'{tim_msg_folder_path}/templates/preambles/preamble'
    tim_msg_preamble = DocEntry.find_by_path(preamble_path)

    if not tim_msg_preamble:
        tim_msg_preamble = import_document_from_file(
            static_tim_doc('initial/tim_msg_preamble.md'), preamble_path, ↩
    admin_group, title='preamble',
        )

        tim_msg_preamble.block.add_rights([UserGroup.get_logged_in_group()], ↩
    AccessType.view)

    return tim_msg_folder


def update_tim_msg_doc_settings(message_doc: DocInfo, sender: User, message_body: ↩
    MessageBody) -> None:
    """
    Sets the message information into the preamble macros.

    :param message_doc: TIM message document
    :param sender: Sender user
    :param message_body: Message body
    :return:
    """
    s = message_doc.document.get_settings().get_dict().get('macros', {})
    s['subject'] = message_body.messageSubject
    s['sendername'] = sender.name
    s['senderemail'] = sender.email
    s['recipients'] = message_body.recipients

    message_doc.document.add_setting('macros', s)


def create_message_displays(msg: InternalMessage, pages: List[Item], recipients: ↩
    List[UserGroup]) -> None:
    """
    Creates InternalMessageDisplay entries for all recipients and display pages.

    :param msg: Message
    :param pages: List of pages where message is displayed
    :param recipients: List of message recipients
    :return:
    """
    if pages and recipients:
```

```
496         for page in pages:
497             for rcpt in recipients:
498                 display = InternalMessageDisplay(message=msg, usergroup=rcpt, ↩
    display_block=page.block)
499                 db.session.add(display)
500 
501     if pages and not recipients:
502         for page in pages:
503             display = InternalMessageDisplay(message=msg, display_block=page.block)
504             db.session.add(display)
505 
506     if not pages and recipients:
507         for rcpt in recipients:
508             display = InternalMessageDisplay(message=msg, usergroup=rcpt)
509             db.session.add(display)
510 
511     if not pages and not recipients:
512         display = InternalMessageDisplay(message=msg)
513         db.session.add(display)
514 
515 
516 def create_read_receipts(msg: InternalMessage, recipients: List[UserGroup]) -> None:
517     """
518     Create InternalMessageReadReceipt entries for all recipients.
519 
520     :param msg: Message
521     :param recipients: Message recipients
522     :return:
523     """
524     for recipient in recipients:
525         readreceipt = InternalMessageReadReceipt(recipient=recipient, message=msg)
526         db.session.add(readreceipt)
```

# timApp/migrations/versions/0b21714ace95_add_tables_for_i

```
1 """Add tables for internal TIM messages, message displays and read receipts.
2 
3 Revision ID: 0b21714ace95
4 Revises: 11bc200da4f8
5 Create Date: 2021-04-20 11:54:03.737256
6 
7 """
8 
9 # revision identifiers, used by Alembic.
10 revision = '0b21714ace95'
11 down_revision = '11bc200da4f8'
12 
13 from alembic import op
14 import sqlalchemy as sa
15 
16 display_type_enum = sa.Enum('TOP_OF_PAGE', 'STICKY', name='displaytype')
17 
18 
19 def upgrade():
20     # display_type_enum.create(op.get_bind(), checkfirst=True)
21     op.create_table('internalmessage',
22                     sa.Column('id', sa.Integer(), nullable=False),
23                     sa.Column('doc_id', sa.Integer(), nullable=False),
24                     sa.Column('par_id', sa.Text(), nullable=False),
```

```
                    sa.Column('can_mark_as_read', sa.Boolean(), nullable=False),
                    sa.Column('reply', sa.Boolean(), nullable=False),
                    sa.Column('display_type', display_type_enum, nullable=False),
                    sa.ForeignKeyConstraint(['doc_id'], ['block.id'], ),
                    sa.PrimaryKeyConstraint('id')
                    )
    op.create_table('internalmessage_display',
                    sa.Column('id', sa.Integer(), nullable=False),
                    sa.Column('message_id', sa.Integer(), nullable=False),
                    sa.Column('usergroup_id', sa.Integer(), nullable=True),
                    sa.Column('display_doc_id', sa.Integer(), nullable=True),
                    sa.ForeignKeyConstraint(['display_doc_id'], ['block.id'], ),
                    sa.ForeignKeyConstraint(['message_id'], ['internalmessage.id'], ),
                    sa.ForeignKeyConstraint(['usergroup_id'], ['usergroup.id'], ),
                    sa.PrimaryKeyConstraint('id')
                    )
    op.create_table('internalmessage_readreceipt',
                    sa.Column('rcpt_id', sa.Integer(), nullable=False),
                    sa.Column('message_id', sa.Integer(), nullable=False),
                    sa.Column('user_id', sa.Integer(), nullable=False),
                    sa.Column('marked_as_read_on', sa.DateTime(), nullable=True),
                    sa.ForeignKeyConstraint(['rcpt_id'], ['usergroup.id'], ),
                    sa.ForeignKeyConstraint(['message_id'], ['internalmessage.id'], ),
                    sa.ForeignKeyConstraint(['user_id'], ['useraccount.id'], ),
                    sa.PrimaryKeyConstraint('rcpt_id', 'message_id')
                    )


def downgrade():
    op.drop_table('internalmessage_readreceipt')
    op.drop_table('internalmessage_display')
    op.drop_table('internalmessage')
    display_type_enum.drop(op.get_bind())
```

# timApp/migrations/versions/11bc200da4f8__add__message__list_

```
"""Add message list models

Revision ID: 11bc200da4f8
Revises: 0bb9e6d20006
Create Date: 2021-04-15 06:51:33.023232

"""

# revision identifiers, used by Alembic.
revision = '11bc200da4f8'
down_revision = '0bb9e6d20006'

from alembic import op
import sqlalchemy as sa

archive_enum = sa.Enum('NONE', 'SECRET', 'GROUPONLY', 'UNLISTED', 'PUBLIC', ↩
    name='archivetype')
channel_id_enum = sa.Enum('TIM_MESSAGE', 'EMAIL_LIST', name='channel')


def upgrade():
    # archive_enum.create(op.get_bind(), checkfirst=True)
    # channel_id_enum.create(op.get_bind(), checkfirst=True)
```

```
    op.create_table('messagelist',
                    sa.Column('id', sa.Integer(), nullable=False),
                    sa.Column('manage_doc_id', sa.Integer(), nullable=True),
                    sa.Column('name', sa.Text(), nullable=True),
                    sa.Column('can_unsubscribe', sa.Boolean(), nullable=True),
                    sa.Column('archive', archive_enum, nullable=True),
                    sa.Column('notify_owner_on_change', sa.Boolean(), nullable=True),
                    sa.Column('description', sa.Text(), nullable=True),
                    sa.Column('info', sa.Text(), nullable=True),
                    sa.ForeignKeyConstraint(['manage_doc_id'], ['block.id'], ),
                    sa.PrimaryKeyConstraint('id')
                    )
    op.create_table('messagelist_member',
                    sa.Column('id', sa.Integer(), nullable=False),
                    sa.Column('message_list_id', sa.Integer(), nullable=True),
                    sa.Column('send_right', sa.Boolean(), nullable=True),
                    sa.Column('delivery_right', sa.Boolean(), nullable=True),
                    sa.Column('member_type', sa.Text(), nullable=True),
                    sa.ForeignKeyConstraint(['message_list_id'], ['messagelist.id'], ),
                    sa.PrimaryKeyConstraint('id')
                    )
    op.create_table('messagelist_distribution',
                    sa.Column('id', sa.Integer(), nullable=False),
                    sa.Column('channel_id', channel_id_enum, nullable=True),
                    sa.ForeignKeyConstraint(['id'], ['messagelist_member.id'], ),
                    sa.PrimaryKeyConstraint('id')
                    )
    op.create_table('messagelist_external_member',
                    sa.Column('id', sa.Integer(), nullable=False),
                    sa.Column('email_address', sa.Text(), nullable=True),
                    sa.ForeignKeyConstraint(['id'], ['messagelist_member.id'], ),
                    sa.PrimaryKeyConstraint('id'),
                    sa.UniqueConstraint('email_address')
                    )
    op.create_table('messagelist_tim_member',
                    sa.Column('id', sa.Integer(), nullable=False),
                    sa.Column('group_id', sa.Integer(), nullable=True),
                    sa.ForeignKeyConstraint(['group_id'], ['usergroup.id'], ),
                    sa.ForeignKeyConstraint(['id'], ['messagelist_member.id'], ),
                    sa.PrimaryKeyConstraint('id')
                    )


def downgrade():
    op.drop_table('messagelist_tim_member')
    op.drop_table('messagelist_external_member')
    op.drop_table('messagelist_distribution')
    op.drop_table('messagelist_member')
    op.drop_table('messagelist')
    channel_id_enum.drop(op.get_bind())
    archive_enum.drop(op.get_bind())
```

# timApp/migrations/versions/132d3c908b54_remove_unique_c

```
"""Remove unique constraint from external member's email address.

Revision ID: 132d3c908b54
Revises: 5512ad2fb9f2
Create Date: 2021-05-23 14:58:18.367535
```

```
6
7 """
8 from alembic import op
9
10 # revision identifiers, used by Alembic.
11 revision = '132d3c908b54'
12 down_revision = '5512ad2fb9f2'
13
14
15 def upgrade():
16     op.drop_constraint('messagelist_external_member_email_address_key', ↩
      'messagelist_external_member', type_='unique')
17
18
19 def downgrade():
20     op.create_unique_constraint('messagelist_external_member_email_address_key', ↩
      'messagelist_external_member',
21                                 ['email_address'])
```

# timApp/migrations/versions/38e714c6b20d__add__tables__for__u

```
1 """Add tables for user's additional emails and verifications.
2
3 Revision ID: 38e714c6b20d
4 Revises: 0b21714ace95
5 Create Date: 2021-04-22 19:13:55.805398
6
7 """
8
9 # revision identifiers, used by Alembic.
10 revision = '38e714c6b20d'
11 down_revision = '0b21714ace95'
12
13 from alembic import op
14 import sqlalchemy as sa
15
16 verification_type_enum = sa.Enum('LIST_JOIN', 'EMAIL_OWNERSHIP', ↩
      name='verificationtype')
17
18
19 def upgrade():
20     # verification_type_enum.create(op.get_bind(), checkfirst=True)
21     op.create_table('verifications',
22                     sa.Column('id', sa.Integer(), nullable=False),
23                     sa.Column('verification_type', verification_type_enum, ↩
      nullable=True),
24                     sa.Column('verification_pending', sa.DateTime(timezone=True), ↩
      nullable=True),
25                     sa.Column('verification_link', sa.Text(), nullable=True),
26                     sa.Column('verified', sa.DateTime(timezone=True), nullable=True),
27                     sa.PrimaryKeyConstraint('id')
28                     )
29     op.create_table('user_emails',
30                     sa.Column('id', sa.Integer(), nullable=False),
31                     sa.Column('additional_email', sa.Text(), nullable=True),
32                     sa.Column('is_primary_email', sa.Boolean(), nullable=True),
33                     sa.Column('address_verified', sa.DateTime(timezone=True), ↩
      nullable=True),
34                     sa.ForeignKeyConstraint(['id'], ['useraccount.id'], ),
```

```
                    sa.PrimaryKeyConstraint('id'),
                    sa.UniqueConstraint('additional_email')
                    )


def downgrade():
    op.drop_table('user_emails')
    op.drop_table('verifications')
    verification_type_enum.drop(op.get_bind())
```

# timApp/migrations/versions/5512ad2fb9f2_update_message_l

```
"""Update message list related tables.

Revision ID: 5512ad2fb9f2
Revises: c00fceabf513
Create Date: 2021-05-07 08:17:26.542982

The creation date is before the previous migration file, because the development got forked from a common file.
"""

# revision identifiers, used by Alembic.
revision = '5512ad2fb9f2'
down_revision = 'c00fceabf513'

from alembic import op
import sqlalchemy as sa

replytolistchanges_enum = sa.Enum('NOCHANGES', 'ADDLIST', name='replytolistchanges')
memberjoinmethod_enum = sa.Enum('DIRECT_ADD', 'INVITED', 'JOINED', name='memberjoinmethod')


def upgrade():
    replytolistchanges_enum.create(op.get_bind(), checkfirst=True)
    memberjoinmethod_enum.create(op.get_bind(), checkfirst=True)
    op.add_column('messagelist', sa.Column('allow_attachments', sa.Boolean(), nullable=True))
    op.add_column('messagelist', sa.Column('default_delivery_right', sa.Boolean(), nullable=True))
    op.add_column('messagelist', sa.Column('default_reply_type', replytolistchanges_enum, nullable=True))
    op.add_column('messagelist', sa.Column('default_send_right', sa.Boolean(), nullable=True))
    op.add_column('messagelist', sa.Column('non_member_message_pass', sa.Boolean(), nullable=True))
    op.add_column('messagelist', sa.Column('only_text', sa.Boolean(), nullable=True))
    op.add_column('messagelist', sa.Column('subject_prefix', sa.Text(), nullable=True))
    op.add_column('messagelist', sa.Column('tim_user_can_join', sa.Boolean(), nullable=True))
    op.add_column('messagelist_distribution', sa.Column('message_list_id', sa.Integer(), nullable=True))
    op.add_column('messagelist_distribution', sa.Column('user_id', sa.Integer(), nullable=True))
    op.drop_constraint('messagelist_distribution_id_fkey', 'messagelist_distribution', type_='foreignkey')
    op.create_foreign_key(None, 'messagelist_distribution', 'messagelist', ['message_list_id'], ['id'])
```

```
    op.create_foreign_key(None, 'messagelist_distribution', 'messagelist_member', ['user_id'], ['id'])
    op.add_column('messagelist_external_member', sa.Column('display_name', sa.Text(), nullable=True))
    op.add_column('messagelist_member', sa.Column('join_method', memberjoinmethod_enum, nullable=True))


def downgrade():
    op.drop_column('messagelist_member', 'join_method')
    op.drop_column('messagelist_external_member', 'display_name')
    op.drop_constraint(None, 'messagelist_distribution', type_='foreignkey')
    op.drop_constraint(None, 'messagelist_distribution', type_='foreignkey')
    op.create_foreign_key('messagelist_distribution_id_fkey', 'messagelist_distribution', 'messagelist_member', ['id'], ['id'])
    op.drop_column('messagelist_distribution', 'user_id')
    op.drop_column('messagelist_distribution', 'message_list_id')
    op.drop_column('messagelist', 'tim_user_can_join')
    op.drop_column('messagelist', 'subject_prefix')
    op.drop_column('messagelist', 'only_text')
    op.drop_column('messagelist', 'non_member_message_pass')
    op.drop_column('messagelist', 'default_send_right')
    op.drop_column('messagelist', 'default_reply_type')
    op.drop_column('messagelist', 'default_delivery_right')
    op.drop_column('messagelist', 'allow_attachments')
    replytolistchanges_enum.drop(op.get_bind())
    memberjoinmethod_enum.drop(op.get_bind())
```

# timApp/migrations/versions/9f18815ce8cb__remove__not__null_

```
"""Remove not-null constraint from InternalMessageReadReceipt

Revision ID: 9f18815ce8cb
Revises: dd105e441f5f
Create Date: 2021-04-26 09:08:43.497876

"""

# revision identifiers, used by Alembic.
revision = '9f18815ce8cb'
down_revision = 'dd105e441f5f'

from alembic import op
import sqlalchemy as sa


def upgrade():
    op.alter_column('internalmessage_readreceipt', 'user_id',
                    existing_type=sa.INTEGER(),
                    nullable=True)


def downgrade():
    op.alter_column('internalmessage_readreceipt', 'user_id',
                    existing_type=sa.INTEGER(),
                    nullable=False)
```

# timApp/migrations/versions/c00fceabf513__add__column__replie

```
"""Add column replies_to to table internalmessage

Revision ID: c00fceabf513
Revises: f8f14db7dc37
Create Date: 2021-05-10 08:14:04.419147

"""

# revision identifiers, used by Alembic.
revision = 'c00fceabf513'
down_revision = 'f8f14db7dc37'

from alembic import op
import sqlalchemy as sa


def upgrade():
    op.add_column('internalmessage', sa.Column('replies_to', sa.Integer(), nullable=True))


def downgrade():
    op.drop_column('internalmessage', 'replies_to')
```

# timApp/migrations/versions/dd105e441f5f_update_message_l

```
"""Update message list tables.

Revision ID: dd105e441f5f
Revises: 38e714c6b20d
Create Date: 2021-04-22 19:32:49.960011

"""

# revision identifiers, used by Alembic.
revision = 'dd105e441f5f'
down_revision = '38e714c6b20d'

from alembic import op
import sqlalchemy as sa

messagelist_distribution_enum = sa.Enum('TIM_MESSAGE', 'EMAIL_LIST', name='channel')


def upgrade():
    # messagelist_distribution_enum.create(op.get_bind(), checkfirst=True)
    op.add_column('messagelist', sa.Column('email_list_domain', sa.Text(), nullable=True))
    op.add_column('messagelist', sa.Column('removed', sa.DateTime(timezone=True), nullable=True))
    op.add_column('messagelist_distribution', sa.Column('channel', messagelist_distribution_enum, nullable=True))
    op.drop_column('messagelist_distribution', 'channel_id')
    op.add_column('messagelist_member', sa.Column('membership_ended', sa.DateTime(timezone=True), nullable=True))
    op.add_column('messagelist_member', sa.Column('membership_verified', sa.DateTime(timezone=True), nullable=True))


```

```
29 def downgrade():
30     op.drop_column('messagelist_member', 'membership_verified')
31     op.drop_column('messagelist_member', 'membership_ended')
32     op.add_column('messagelist_distribution', sa.Column('channel_id', ↩
       messagelist_distribution_enum,
33                                                         autoincrement=False, ↩
       nullable=True))
34     op.drop_column('messagelist_distribution', 'channel')
35     op.drop_column('messagelist', 'removed')
36     op.drop_column('messagelist', 'email_list_domain')
37     messagelist_distribution_enum.drop(op.get_bind())
```

# timApp/migrations/versions/f8f14db7dc37_add_column_expir

```
1 """add column expires to table internalmessage
2
3 Revision ID: f8f14db7dc37
4 Revises: 9f18815ce8cb
5 Create Date: 2021-05-06 07:50:35.874633
6
7 """
8
9 # revision identifiers, used by Alembic.
10 revision = 'f8f14db7dc37'
11 down_revision = '9f18815ce8cb'
12
13 from alembic import op
14 import sqlalchemy as sa
15
16
17 def upgrade():
18     op.add_column('internalmessage', sa.Column('expires', sa.DateTime(), nullable=True))
19
20
21 def downgrade():
22     op.drop_column('internalmessage', 'expires')
```

# timApp/static/scripts/tim/messaging/MIT-licence.txt

```
1 Copyright 2021 Hannamari Heiniluoma, Kristian KÃ¤yhty,
2                Tomi Lundberg and Tuuli Veini
3
4 Permission is hereby granted, free of charge, to any person obtaining a copy
5 of this software and associated documentation files (the "Software"), to deal
6 in the Software without restriction, including without limitation the rights
7 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
8 copies of the Software, and to permit persons to whom the Software is
9 furnished to do so, subject to the following conditions:
10
11 The above copyright notice and this permission notice shall be included in
12 all copies or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
15 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
16 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
17 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
18 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
19 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
20 THE SOFTWARE.
```

# timApp/static/scripts/tim/messaging/listOptionTypes.ts

```
import {Moment} from "moment";

export interface ListOptions {
    // Keep this updated with ListOptions Python class. That class also provides further information about these
    // variables.
    name: string;
    domain?: string;
    archive?: ArchiveType;
    notify_owners_on_list_change?: boolean;
    list_description?: string;
    list_info?: string;
    only_text?: boolean;
    default_reply_type?: ReplyToListChanges;
    email_admin_url?: string;
    tim_users_can_join?: boolean;
    members_can_unsubscribe?: boolean;
    default_send_right?: boolean;
    default_delivery_right?: boolean;
    list_subject_prefix?: string;
    non_member_message_pass?: boolean;
    allow_attachments?: boolean;
    // distribution?: Channel[];
    distribution?: Distribution;
    removed?: Moment;
}

// Interface to express what message channels the message list has in use.
export interface Distribution {
    tim_message: boolean;
    email_list: boolean;
}

// See class Channel on listoptions.py for explanation.
export enum Channel {
    TIM_MESSAGE = "tim_message",
    EMAIL_LIST = "email_list",
}

// See ReplyToListChanges Python class.
export enum ReplyToListChanges {
    NOCHANGES = "no_munging",
    ADDLIST = "point_to_list",
}

// See ArchiveType class on Python side of things for explanations.
export enum ArchiveType {
    NONE,
    SECRET,
    GROUPONLY,
    UNLISTED,
    PUBLIC,
}

// For proper setting of archive options on UI.
export interface ArchivePolicyNames {
    archiveType: ArchiveType;
```

```
    policyName: string;
}

// Mapping of archive policy enum to explanations given in UI.
export const archivePolicyNames: ArchivePolicyNames[] = [
    {archiveType: ArchiveType.NONE, policyName: "No archiving."},
    {
        archiveType: ArchiveType.SECRET,
        policyName: "Owners only",
    },
    {
        archiveType: ArchiveType.GROUPONLY,
        policyName: "Members only.",
    },
    {
        archiveType: ArchiveType.UNLISTED,
        policyName: "TIM users.",
    },
    {
        archiveType: ArchiveType.PUBLIC,
        policyName: "Public.",
    },
].reverse();

// See MemberInfo Python class for further details.
export interface MemberInfo {
    name: string;
    username: string;
    sendRight: boolean;
    deliveryRight: boolean;
    email: string;
    removed?: Moment;
    // The member's removed status at the time of loading. Used only for display purposes. This is removed on save,
    // since this is not used on server side.
    removedDisplay?: string;
}

// See GroupAndMembers Python class for further details.
export interface GroupAndMembers {
    groupName: string;
    members: MemberInfo[];
}
```

# timApp/static/scripts/tim/messaging/manage-read-receipt.component.scss

```
.manageReadReceipt {
    margin-top: 30px;

    p {
        margin-top: 2px;
        font-size: 80%;
    }
}
```

# timApp/static/scripts/tim/messaging/manage-read-receipt.component.ts

```
import {Component, NgModule, OnInit} from "@angular/core";
import {CommonModule} from "@angular/common";
import {itemglobals} from "tim/util/globals";
import {to2} from "tim/util/utils";
import {HttpClient} from "@angular/common/http";
import {markAsRead} from "tim/messaging/messagingUtils";

@Component({
    selector: "manage-read-receipt",
    template: `
        <ng-container *ngIf="canMarkAsRead">
            <div class="manageReadReceipt">
                <button class="timButton" title="Mark as Read" *ngIf="!markedAsRead" (click)="markAsRead()">
                    Mark as Read
                </button>
                <div class="cancelReadReceipt" *ngIf="markedAsRead">
                    <button class="timButton" title="Cancel Read Receipt" (click)="cancelReadReceipt()">
                        Cancel Read Receipt
                    </button>
                    <p>Cancelling the read receipt re-displays the message on designated TIM pages</p>
                </div>
            </div>
        </ng-container>
    `,
    styleUrls: ["manage-read-receipt.component.scss"],
})
export class ManageReadReceiptComponent implements OnInit {
    markedAsRead: boolean = false;
    receipt: TimMessageReadReceipt | undefined;
    canMarkAsRead: boolean = false;

    constructor(private http: HttpClient) {}

    ngOnInit(): void {
        const docId = itemglobals().curr_item.id;

        void this.getReadReceipt(docId);
    }

    /**
     * Retrieves information about the read receipt from database;
     * displays the read receipt component if marking the message as read
     * is allowed and sets the read receipt status correctly.
     *
     * @param docId Identifier for the message's document
     */
    async getReadReceipt(docId: number) {
        const message = await to2(
            this.http
                .get<TimMessageReadReceipt>(
                    `/timMessage/get_read_receipt/${docId}`
                )
```

```
                .toPromise()
        );

        if (message.ok) {
            this.receipt = message.result;
            this.canMarkAsRead = message.result.can_mark_as_read;
            if (message.result.marked_as_read_on != null) {
                this.markedAsRead = true;
            }
        } else {
            console.error(message.result.error.error);
        }
    }

    /**
     * Marks the message as read and displays the option to cancel read receipt.
     */
    async markAsRead() {
        if (!this.receipt) {
            return;
        }

        const result = await markAsRead(this.http, this.receipt.message_id);
        if (result.ok) {
            this.markedAsRead = true;
        }

        window.location.reload();
    }

    /**
     * Cancels the read receipt and displays the option to mark the message as read.
     */
    async cancelReadReceipt() {
        const result = await to2(
            this.http
                .post("/timMessage/cancel_read_receipt", {
                    message_id: this.receipt?.message_id,
                })
                .toPromise()
        );
        if (!result.ok) {
            console.error(result.result.error.error);
        }

        this.markedAsRead = false;
        window.location.reload();
    }
}

interface TimMessageReadReceipt {
    // Information about the read receipt retrieved from server
    rcpt_id: number;
    message_id: number;
    user_id: number;
    marked_as_read_on: Date;
    can_mark_as_read: boolean;
}

```

```
@NgModule({
    declarations: [ManageReadReceiptComponent],
    exports: [ManageReadReceiptComponent],
    imports: [CommonModule],
})
export class ManageReadReceiptModule {}
```

# timApp/static/scripts/tim/messaging/message-list-admin.component.scss

```
h1, h2 {
  margin-bottom: 1em;
}

div.archive-options {
  margin-top: 2em;
  margin-bottom: 0.2em;
}

.list-archive-policy-header {
  margin-bottom: 0.5em;
  font-weight: bold;
}

input[type='radio'] {
  margin-right: 2em;
}

.list-name {
  font-weight: bold;
}

.owner-address {
  font-weight: bold;
}

.subject-prefix {
  font-weight: bold;
}

.short-description {
  font-weight: bold;
}

.long-description {
  font-weight: bold;
}

label {
  font-weight: normal;
  padding-left: 1em;
}

#add-multiple-members {
  width: 25em;
  height: 5em;
}

```

```
.section {
  padding-bottom: 1.5em;
}

#archive-policy-list {
  list-style-type: none
}

#add-member-feedback {
  padding-top: 1em;
  padding-bottom: 1em;
}

#email-send {
  padding-bottom: 1em;
}

.indented {
  // Indent by uniform amount.
  margin-left: 2em;
}

.indented-more {
  // Some elemenents are sub-elements, those are indented a bit more.
  margin-left: 4em;
}

#domain-indicator {
  padding: 0 0.25em;
}

.member-table-row {
  font-size: 0.8em;
}

tbody {
  // Tables, especially the one showing group's members, might be hundreds of rows ↩
    long. We don't want that to make the
  // table and by extension the page unnecessarily long, so add scrolling to table's ↩
    body.
  overflow: scroll;
  max-height: 40em;
  size: auto;
}
```

# timApp/static/scripts/tim/messaging/message-list-admin.component.ts

```
import {HttpClient} from "@angular/common/http";
import {Component, NgModule, OnInit} from "@angular/core";
import {CommonModule} from "@angular/common";
import {to, to2} from "tim/util/utils";
import {FormsModule} from "@angular/forms";
import {
    archivePolicyNames,
    ArchiveType,
    Distribution,
    GroupAndMembers,
```

```
11      ListOptions,
12      MemberInfo,
13      ReplyToListChanges,
14  } from "tim/messaging/listOptionTypes";
15  import {documentglobals} from "tim/util/globals";
16  import {TimUtilityModule} from "tim/ui/tim-utility.module";
17  import {TableFormModule} from "tim/plugin/tableForm";
18  import moment, {Moment} from "moment";
19  import {showInputDialog} from "tim/ui/showInputDialog";
20  import {InputDialogKind} from "tim/ui/input-dialog.kind";
21  import {$http} from "tim/util/ngimport";
22  import {Users} from "../user/userService";
23
24  @Component({
25      selector: "tim-message-list-admin",
26      template: `
27          <form class="form-horizontal">
28              <h1>Message list management</h1>
29              <h2>{{listname}}</h2>
30              <tim-alert *ngIf="permanentErrorMessage" ↩
    severity="danger">{{permanentErrorMessage}}</tim-alert>
31              <div id="email-send">
32                  <tim-message-send [(recipientList)]="recipients" ↩
    [docId]="getDocId()"></tim-message-send>
33                  <button class="timButton" (click)="openEmail()" ↩
    *ngIf="!recipients">Send message to list</button>
34              </div>
35              <div class="section">
36                  <h3>Options</h3>
37                  <div class="form-group"
38                       title="To be added in front of message subjects for all messages ↩
    that pass through this list.">
39                      <label for="list-subject-prefix" class="subject-prefix ↩
    control-label col-sm-3">Subject
40                          prefix: </label>
41                      <div class="col-sm-9">
42                          <input type="text" name="list-subject-prefix" ↩
    class="form-control"
43                                 [(ngModel)]="listSubjectPrefix">
44                      </div>
45                  </div>
46                  <div class="form-group" *ngIf="domain">
47                      <label for="list-description" class="short-description ↩
    control-label col-sm-3">Email
48                          address: </label>
49                      <div class="col-sm-9" *ngIf="distribution && ↩
    distribution.email_list">
50                          <input type="text" class="form-control" ↩
    name="list-email-address" id="list-email-address"
51                                 [ngModel]="listAddress()" disabled/>
52                      </div>
53                  </div>
54                  <div class="form-group">
55                      <label for="list-description" class="short-description ↩
    control-label col-sm-3">Short
56                          description: </label>
57                      <div class="col-sm-9">
58                          <input type="text" class="form-control" ↩
    name="list-description" id="list-description"
```

```
                                [(ngModel)]="listDescription"/>
                    </div>
                </div>
                <div class="form-group">
                    <label for="list-info" class="long-description control-label col-sm-3">Long description: <br>
                        (Info)</label>
                    <div class="col-sm-9">
                <textarea name="list-info" class="list-info form-control" id="list-info"
                          [(ngModel)]="listInfo"></textarea>
                    </div>
                </div>
                <div *ngIf="archiveOptions && archive">
                    <p class="list-archive-policy-header">Archive policy:</p>
                    <!-- Variable archiveoptions is reversed, so indexing for display has to accommodate. -->
                    <p class="indented">{{archiveOptions[archiveOptions.length - (archive + 1)].policyName}}</p>
                    <!-- Hide radio buttons here, until the changing of archive policy levels is implemented -->
                    <!--
                    <ul id="archive-policy-list">
                        <li *ngFor="let option of archiveOptions">
                            <label for="archive-{{option.archiveType}}">
                            <input
                                    name="items-radio"
                                    type="radio"
                                    id="archive-{{option.archiveType}}"
                                    [value]="option.archiveType"
                                    [(ngModel)]="archive"
                            />
                            {{option.policyName}}</label>
                        </li>
                    </ul>
                    -->
                </div>
                <div class="indented">
                    <label>
                        <input type="checkbox" name="notify-owner-on-list-change" id="notify-owner-on-list-change"
                                [(ngModel)]="notifyOwnerOnListChange"/>
                        Notify owners on list changes (e.g. user subscribes).</label>
                </div>
                <div class="indented">
                    <label for="tim-users-can-join">
                        <input type="checkbox" name="tim-users-can-join" [(ngModel)]="timUsersCanJoin" disabled>
                        TIM users can freely join this list.</label>
                    <div class="indented-more">
                        <label>
                            <input type="checkbox" name="default-send-right" [(ngModel)]="defaultSendRight"
                                   [disabled]="!timUsersCanJoin">
                            Default send right for new members.</label>
                    </div>
                    <div class="indented-more">
                        <label>
                            <input type="checkbox" name="default-delivery-right" [(ngModel)]="defaultDeliveryRight"
```

```
                                [disabled]="!timUsersCanJoin">
                            Default delivery right for new members.</label>
                    </div>
                </div>
                <div class="indented">
                    <label for="can-user-unsubscribe">
                        <input type="checkbox" name="can-user-unsubscribe" [(ngModel)]="canUnsubscribe">
                        Members can unsubscribe from the list on their own.</label>
                </div>
                <div class="indented">
                    <label for="non-members-can-send">
                        <input type="checkbox" name="non-members-can-send" [(ngModel)]="nonMemberMessagePass">
                        Non members can send messages to list.</label>
                </div>
                <div class="indented">
                    <label for="only-text">
                        <input type="checkbox" name="only-text" [(ngModel)]="onlyText">
                        No HTML messages allowed on the list.</label>
                </div>
                <div class="indented">
                    <label for="allow-attachments">
                        <input type="checkbox" name="allow-attachments" [(ngModel)]="allowAttachments">
                        Allow attachments on the list.</label>
                </div>
                <div class="indented">
                    <label>
                        <input type="checkbox" name="list-answer-guidance" [(ngModel)]="listAnswerGuidance">
                        Direct answers to message list.</label>
                </div>
                <div class="indented">
                    <button class="timButton" (click)="saveOptions()">Save options</button>
                    <tim-alert severity="success" *ngIf="saveSuccessMessage">{{saveSuccessMessage}}</tim-alert>
                    <tim-alert severity="danger" *ngIf="saveFailMessage">{{saveFailMessage}}</tim-alert>
                </div>
                <div id="members-section" class="section">
                    <h3>Members</h3>
                    <div>
                        <p class="indented">Instructions:</p>
                        <p class="indented">Add new members by setting each member on their own separate lines. The
                            members are only
                            added after you click the "Add new members" button.</p>
                        <p class="indented">Add individual TIM users by writing their username.</p>
                        <p class="indented">Add a group by writing its name. You need to be the owner of the group for
                            the adding to succeed.</p>
                        <p class="indented">Add an external member (someone who is not a TIM user) by writing their
                            email address
                            (mandatory) and name (optional) either in the form <code>john.doe@domain.fi John Doe</code>
```

```
157                         or <code>Jane Doe &lt;jane.doe@domain.fi&gt;</code></p>
158                     <p class="indented">Send right means that a member's message should not be caught up in a
159                         moderation process.
160                         Delivery right means that the member receives messages sent to the list. For a group, the
161                         send and delivery right affect all the members of a group.</p>
162                 </div>
163                 <div id="add-members-section">
164                     <label class="indented" for="add-multiple-members">Add members</label> <br/>
165                     <textarea class="indented" id="add-multiple-members" name="add-multiple-members"
166                               [(ngModel)]="membersTextField"></textarea>
167                     <div class="indented-more">
168                         <div>
169                             <input type="checkbox" name="new-member-send-right" [(ngModel)]="newMemberSendRight">
170                             <label for="new-member-send-right">New member's send right.</label>
171                         </div>
172                         <div>
173                             <input type="checkbox" name="new-member-delivery-right"
174                                    [(ngModel)]="newMemberDeliveryRight">
175                             <label for="new-member-delivery-right">New member's delivery right.</label>
176                         </div>
177                     </div>
178                     <div class="indented">
179                         <button class="indented-more timButton" (click)="addNewListMember()">Add new
180                             members
181                         </button>
182                     </div>
183                     <div id="member-add-feedback">
184                         <tim-alert *ngIf="memberAddSucceededResponse"
185                                    severity="success">{{memberAddSucceededResponse}}</tim-alert>
186                         <tim-alert *ngIf="memberAddFailedResponse"
187                                    severity="danger">{{memberAddFailedResponse}}</tim-alert>
188                     </div>
189                 </div>
190             </div>
191             <div class="section">
192                 <h3>Current members</h3>
193                 <table>
194                     <thead>
195                     <tr class="member-table-row">
196                         <th>Name</th>
197                         <th>Username</th>
198                         <th>Email</th>
199                         <th>Send right</th>
200                         <th>Delivery right</th>
201                         <th>Membership ended</th>
202                         <th>Removed</th>
203                     </tr>
204                     </thead>
```

```
205                     <tbody>
206                     <tr class="member-table-row" *ngFor="let member of membersList">
207                         <td>{{member.name}}</td>
208                         <td>{{member.username}}</td>
209                         <td>{{member.email}}</td>
210                         <td>
211                             <input type="checkbox" [(ngModel)]="member.sendRight"
212                                    name="member-send-right-{{member.email}}">
213                         </td>
214                         <td>
215                             <input type="checkbox" ↩
    [(ngModel)]="member.deliveryRight"
216                                    name="member-delivery-right-{{member.email}}">
217                         </td>
218                         <td>{{member.removedDisplay}}</td>
219                         <td><input type="checkbox" ↩
    (click)="membershipChange(member)" [ngModel]="!!member.removed"
220                                    name="removed-{{member.email}}"/></td>
221                     </tr>
222                     </tbody>
223                 </table>
224                 <button class="indented timButton" (click)="saveMembers()">Save ↩
    members</button>
225                 <div>
226                     <tim-alert *ngIf="memberSaveSuccessResponse"
227                                ↩
    severity="success">{{memberSaveSuccessResponse}}</tim-alert>
228                     <tim-alert *ngIf="memberSaveFailResponse"
229                                ↩
    severity="danger">{{memberAddFailedResponse}}</tim-alert>
230                 </div>
231             </div>
232         </div>
233         <div class="section" *ngIf="hasGroups">
234             <h3>Show members of a group {{currentGroup}}</h3>
235             <select class="indented" [(ngModel)]="currentGroup" name="usergroups"
236                     (change)="setGroupMembers()">
237                 <option></option>
238                 <option *ngFor="let memberGroup of ↩
    memberGroups">{{memberGroup}}</option>
239             </select>
240             <table *ngIf="currentGroup">
241                 <thead>
242                 <tr class="member-table-row">
243                     <th>Name</th>
244                     <th>Username</th>
245                     <th>Email</th>
246                 </tr>
247                 </thead>
248                 <tbody>
249                 <tr class="member-table-row" *ngFor="let gMember of groupMembers">
250                     <td>{{gMember.name}}</td>
251                     <td>{{gMember.username}}</td>
252                     <td>{{gMember.email}}</td>
253                 </tr>
254                 </tbody>
255             </table>
256         </div>
257         <div class="section">
```

```
258                 <h2>List deletion</h2>
259                 <button class=" indented timButton" (click)="deleteList()">Delete ↩
    List</button>
260             </div>
261             <div class="section">
262                 <h3>Links</h3>
263                 <div class="indented" *ngIf="archiveURL">
264                     <a [href]="archiveURL">List's archive.</a>
265                 </div>
266                 <div class="indented" *ngIf="emailAdminURL">
267                     <a [href]="emailAdminURL">Advanced email list settings (takes to ↩
    Mailman).</a>
268                 </div>
269             </div>
270         </form>
271     `,
272     styleUrls: ["message-list-admin.component.scss"],
273 })
274 export class MessageListAdminComponent implements OnInit {
275     listname: string = "";
276
277     archive?: ArchiveType;
278
279     domain?: string;
280     domains: string[] = [];
281
282     membersTextField?: string;
283     membersList: MemberInfo[] = [];
284
285     urlPrefix: string = "/messagelist";
286
287     archiveOptions = archivePolicyNames;
288
289     notifyOwnerOnListChange?: boolean;
290     timUsersCanJoin?: boolean;
291
292     listInfo?: string;
293     listDescription?: string;
294
295     emailAdminURL?: string;
296     archiveURL?: string;
297
298     canUnsubscribe?: boolean;
299     defaultSendRight?: boolean;
300     defaultDeliveryRight?: boolean;
301     listSubjectPrefix?: string;
302     nonMemberMessagePass?: boolean;
303     onlyText?: boolean;
304     allowAttachments?: boolean;
305     distribution?: Distribution;
306     listReplyToChange?: ReplyToListChanges;
307     listAnswerGuidance?: boolean; // Track above enum value in a checkbox.
308
309     // Flags for new members' rights on the list.
310     newMemberSendRight: boolean = true;
311     newMemberDeliveryRight: boolean = true;
312
313     // Response strings for saving list options.
314     saveSuccessMessage: string = "";
```

```
315     saveFailMessage: string = "";
316
317     // Response strings used in giving feedback to the user on adding new members to ↩
        the message list.
318     memberAddSucceededResponse: string = "";
319     memberAddFailedResponse: string = "";
320
321     // Response strings for saving members' state.
322     memberSaveSuccessResponse: string = "";
323     memberSaveFailResponse: string = "";
324
325     // Permanent error messages that cannot be recovered from, e.g. loading failed ↩
        and reload is needed.
326     permanentErrorMessage?: string;
327
328     // For using tim-message-send component.
329     recipients = "";
330
331     // Timestamp if this message list has been removed.
332     removed?: Moment;
333
334     // If groups are members part of list, these hold information about the members ↩
        of said groups.
335     groupMembers?: MemberInfo[];
336     // Flag if the UI shows the additional group member area.
337     hasGroups: boolean = false;
338     // Names of groups to set as selectable.
339     memberGroups?: string[];
340     // List members which are groups and the members of those groups.
341     groupsAndMembers?: GroupAndMembers[];
342     // Current group to view on the UI. If undefined, shows no group on the UI.
343     currentGroup?: string;
344
345     /**
346      * Modifies the member's removed attribute if the member's state is changed.
347      * @param member Who's membership on the list is changed.
348      */
349     membershipChange(member: MemberInfo) {
350         if (member.removed) {
351             member.removed = undefined;
352         } else {
353             // Set time stamp when the member was removed.
354             member.removed = moment();
355         }
356     }
357
358     /**
359      * The current document's document ID.
360      */
361     getDocId() {
362         return documentglobals().curr_item.id;
363     }
364
365     /**
366      * Build this list's email address, if there is a domain configured. Otherwise ↩
        return an empty string.
367      */
368     listAddress() {
369         if (this.domain) {
```

```
370             return `${this.listname}@${this.domain}`;
371         }
372         return "";
373     }
374 
375     /**
376      * Initialization procedures.
377      */
378     async ngOnInit() {
379         if (Users.isLoggedIn()) {
380             // Get domains.
381             await this.getDomains();
382 
383             // Load message list options.
384             const docId = this.getDocId();
385             const result1 = await this.loadValues(docId);
386 
387             if (result1.ok) {
388                 this.setValues(result1.result);
389             } else {
390                 this.permanentErrorMessage = `Loading list options failed: ↩
    ${result1.result.error.error}`;
391                 // Loading options failed. Short circuit here, no reason to continue.
392                 return;
393             }
394 
395             // Load list members.
396             const result2 = await this.getListMembers();
397 
398             if (result2.ok) {
399                 // TODO order members by name.
400                 this.membersList = result2.result;
401                 // Set the UI value for removed attribute.
402                 for (const member of this.membersList) {
403                     if (member.removed) {
404                         member.removedDisplay = moment(member.removed).format(
405                             "DD.MM.YYYY hh:mm"
406                         );
407                     }
408                 }
409             } else {
410                 this.permanentErrorMessage = `Loading list's members failed: ↩
    ${result2.result.error.error}`;
411             }
412 
413             await this.getGroupMembers();
414         }
415     }
416 
417     /**
418      * Opens the email sending view by adding the list's address to the string of ↩
    recipients.
419      */
420     openEmail() {
421         this.recipients = this.listAddress();
422     }
423 
424     /**
425      * Get domains ccondigured for email list use.
```

```
426      */
427     private async getDomains() {
428         const result = await to2(
429             this.http.get<string[]>(`${this.urlPrefix}/domains`).toPromise()
430         );
431         if (result.ok) {
432             this.domains = result.result;
433             if (!this.domains.length) {
434                 this.domain = this.domains[0];
435             }
436         } else {
437             /* Getting an error here is not a problem, since these domains are not (yet) in use other than displaying
438              * them in the UI. The UI will probably look a bit funky, but it does not affect functionality right now.
439              * In the future, this is a problem if email list could be taken into use after the message list already
440              * exists.
441              */
442         }
443     }
444
445     constructor(private http: HttpClient) {}
446
447     /**
448      * Compile email addresses separated by line breaks into a list
449      * @private
450      */
451     private parseMembers(): string[] {
452         if (!this.membersTextField) {
453             return [];
454         }
455         return this.membersTextField.split("\n").filter((e) => e);
456     }
457
458     /**
459      * Add new members to message list.
460      */
461     async addNewListMember() {
462         const memberCandidates = this.parseMembers();
463         if (memberCandidates.length == 0) {
464             return;
465         }
466         const result = await to2(
467             this.http
468                 .post(`${this.urlPrefix}/addmember`, {
469                     member_candidates: memberCandidates,
470                     msg_list: this.listname,
471                     send_right: this.newMemberSendRight,
472                     delivery_right: this.newMemberDeliveryRight,
473                 })
474                 .toPromise()
475         );
476         if (result.ok) {
477             // Empty the text field.
478             this.membersTextField = undefined;
479             this.memberAddSucceededResponse = "New members added.";
480         } else {
481             this.memberAddFailedResponse = `Adding new members failed:
```

```
    ${result.result.error.error}`;
482         }
483     }
484
485     /**
486      * Get all list members.
487      */
488     async getListMembers() {
489         return to2(
490             this.http
491                 .get<MemberInfo[]>(
492                     `${this.urlPrefix}/getmembers/${this.listname}`
493                 )
494                 .toPromise()
495         );
496     }
497
498     /**
499      * Helper for list deletion.
500      */
501     async deleteList() {
502         // Ask confirmation from the user.
503         await showInputDialog({
504             title: "Confirm list deletion",
505             text: "Confirm you really want to delete this list.",
506             okText: "Delete",
507             isInput: InputDialogKind.ValidatorOnly,
508             validator: async () => {
509                 const result = await to(
510                     $http.delete(`/messagelist/deletelist`, {
511                         params: {
512                             listname: this.listname,
513                             permanent: false, // Only non-permanent deletion at this ↩
    point.
514                         },
515                     })
516                 );
517                 if (result.ok) {
518                     // Deletion was successful. Beam us up.
519                     location.assign("/view/messagelists");
520                     return {ok: true, result: result.result} as const;
521                 } else {
522                     this.permanentErrorMessage = "Deleting the list failed.";
523                     return {
524                         ok: false,
525                         result: result.result.data.error,
526                     } as const;
527                 }
528             },
529         });
530     }
531
532     /**
533      * Get values for message list's options.
534      * @param docID List is defined by its management document, so we get list's ↩
    options and members with it.
535      */
536     async loadValues(docID: number) {
537         return to2(
```

```
538             this.http
539                 .get<ListOptions>(`${this.urlPrefix}/getlist/${docID}`)
540                 .toPromise()
541         );
542     }
543
544     /**
545      * Setting list values after loading.
546      * @param listOptions All the options of the list returned from the server.
547      */
548     setValues(listOptions: ListOptions) {
549         this.listname = listOptions.name;
550         this.archive = listOptions.archive;
551         // Without archive value, there is no reason to continue. Show error and ↩
    short circuit here.
552         if (this.archive == null) {
553             this.permanentErrorMessage =
554                 "Loading the archive value failed. Please reload the page. If ↩
    reloading the page does not fix the " +
555                 "problem, then please contact TIM's support and tell about this error.";
556             return;
557         }
558
559         this.domain = listOptions.domain;
560
561         this.notifyOwnerOnListChange = listOptions.notify_owners_on_list_change;
562
563         this.listInfo = listOptions.list_info;
564         this.listDescription = listOptions.list_description;
565
566         this.emailAdminURL = listOptions.email_admin_url;
567
568         // If some type of archiving exists for the list, provide a link to it.
569         if (this.archive !== ArchiveType.NONE) {
570             this.archiveURL = `/view/archives/${this.listname}`;
571         }
572
573         this.timUsersCanJoin = listOptions.tim_users_can_join;
574
575         this.listSubjectPrefix = listOptions.list_subject_prefix;
576         this.canUnsubscribe = listOptions.members_can_unsubscribe;
577         this.defaultSendRight = listOptions.default_send_right;
578         this.defaultDeliveryRight = listOptions.default_delivery_right;
579         this.nonMemberMessagePass = listOptions.non_member_message_pass;
580         this.onlyText = listOptions.only_text;
581         this.allowAttachments = listOptions.allow_attachments;
582         this.distribution = listOptions.distribution;
583         this.allowAttachments = listOptions.allow_attachments;
584         // Convert enum to boolean for tracking this on a checkbox.
585         if (listOptions.default_reply_type != null) {
586             this.listAnswerGuidance =
587                 listOptions.default_reply_type !== ReplyToListChanges.NOCHANGES;
588             this.listReplyToChange = listOptions.default_reply_type;
589         }
590         this.removed = listOptions.removed;
591         if (this.removed) {
592             this.permanentErrorMessage =
593                 "This message list is not currently in use.";
594         }
```

```
595     }
596
597     /**
598      * Save the list options.
599      */
600     async saveOptions() {
601         // Reset a failed saving message.
602         this.saveFailMessage = "";
603         // There is no reason to send this.removed back to server.
604         const result = await this.saveOptionsCall({
605             name: this.listname,
606             domain: this.domain,
607             list_info: this.listInfo,
608             list_description: this.listDescription,
609             only_text: this.onlyText,
610             // If the checkbox for guiding messages to message list is checked, put ↩
    ADDLIST enum, otherwise NOCHANGES
611             // enum as default_reply_type.
612             default_reply_type: this.listAnswerGuidance
613                 ? ReplyToListChanges.ADDLIST
614                 : ReplyToListChanges.NOCHANGES,
615             notify_owners_on_list_change: this.notifyOwnerOnListChange,
616             archive: this.archive,
617             tim_users_can_join: this.timUsersCanJoin,
618             list_subject_prefix: this.listSubjectPrefix,
619             members_can_unsubscribe: this.canUnsubscribe,
620             default_delivery_right: this.defaultDeliveryRight,
621             default_send_right: this.defaultSendRight,
622             non_member_message_pass: this.nonMemberMessagePass,
623             allow_attachments: this.allowAttachments,
624         });
625         if (result.ok) {
626             this.showTempSaveSuccess();
627         } else {
628             this.saveFailMessage = `Save failed with an error: ↩
    ${result.result.error.error}`;
629         }
630     }
631
632     /**
633      * Helper for list saving to keep types in check.
634      * @param options All the list options the user saves.
635      */
636     private saveOptionsCall(options: ListOptions) {
637         return to2(this.http.post(`/messagelist/save`, {options}).toPromise());
638     }
639
640     /**
641      * Save the lists members' state.
642      */
643     async saveMembers() {
644         const tempMembersList = this.membersList;
645         // Get rid of removedDisplay property for the members being send to server, ↩
    as the server does not need it for
646         // anything.
647         for (const tempMember of tempMembersList) {
648             delete tempMember.removedDisplay;
649         }
650         const resultSaveMembers = await this.saveMembersCall(tempMembersList);
```

```
651         // Give timed feedback to user.
652         if (resultSaveMembers.ok) {
653             this.memberSaveSuccessResponse = "Saving members succeeded!";
654             window.setTimeout(
655                 () => (this.memberSaveSuccessResponse = ""),
656                 5 * 1000
657             );
658         } else {
659             this.memberSaveFailResponse = "Saving members failed.";
660             window.setTimeout(
661                 () => (this.memberSaveFailResponse = ""),
662                 5 * 1000
663             );
664         }
665     }
666 
667     /**
668      * Makes the actual REST call to save the state of list members'.
669      * @param memberList A list of message list members with their information.
670      */
671     saveMembersCall(memberList: MemberInfo[]) {
672         return to2(
673             this.http
674                 .post(`${this.urlPrefix}/savemembers`, {
675                     members: memberList,
676                     listname: this.listname,
677                 })
678                 .toPromise()
679         );
680     }
681 
682     /**
683      * Modify the recipient list for tim-message-send component. Adds the message ↩
    list's email list as the recipient.
684      */
685     recipientList() {
686         if (this.domain) {
687             return `${this.listname}@${this.domain}`;
688         } else {
689             return "";
690         }
691     }
692 
693     /**
694      * Shows a timed save success message.
695      */
696     showTempSaveSuccess() {
697         this.saveSuccessMessage = "Save success!";
698         window.setTimeout(() => (this.saveSuccessMessage = ""), 5 * 1000);
699     }
700 
701     /**
702      * Call for members of a user group.
703      */
704     getGroupMembersCall() {
705         return to2(
706             this.http
707                 .get<GroupAndMembers[]>(
708                     `${this.urlPrefix}/getgroupmembers/${this.listname}`
```

```
                )
                .toPromise()
        );
    }

    /**
     * Get the members of a user group.
     */
    async getGroupMembers() {
        const result = await this.getGroupMembersCall();
        if (result.ok) {
            // TODO Order members by name.
            this.groupsAndMembers = result.result;
            // If there are no groups on this list, then we short circuit here.
            if (this.groupsAndMembers.length === 0) {
                return;
            }
            this.memberGroups = [];
            // Set the groups members are to show on UI.
            this.hasGroups = true;
            // Set the names of all groups to select element.
            for (const gm of this.groupsAndMembers) {
                this.memberGroups.push(gm.groupName);
            }
        } else {
            this.permanentErrorMessage =
                "Loading members of groups failed. Please refresh the browser. If the problem " +
                "persists, please contact TIM's support.";
        }
    }

    /**
     * Helper for setting members of a group to a table.
     */
    setGroupMembers() {
        if (this.currentGroup) {
            // Find the currently selected group where we want to see members.
            const groupAndMembers = this.groupsAndMembers?.find(
                (g) => g.groupName === this.currentGroup
            );
            if (groupAndMembers) this.groupMembers = groupAndMembers.members;
        }
    }
}

@NgModule({
    declarations: [MessageListAdminComponent],
    exports: [MessageListAdminComponent],
    imports: [CommonModule, FormsModule, TimUtilityModule, TableFormModule],
})
export class NewMsgListModule {}
```

# timApp/static/scripts/tim/messaging/message-list-create-dialog.component.scss

```
div.archive-options {
  margin-top: 10px;
```

```
}

input[type='radio'] {
  margin-right: 10px;
}

.list-name {
  font-weight: bold;
  text-align: left !important;
}

label {
  font-weight: normal;
}

tim-loading {
  margin-right: 1em;
}

.archive-list {
  list-style-type: none;
}
```

# timApp/static/scripts/tim/messaging/message-list-create-dialog.component.ts

```
import {BrowserModule} from "@angular/platform-browser";
import {Component, NgModule} from "@angular/core";
import {FormsModule} from "@angular/forms";
import {HttpClient} from "@angular/common/http";
import {AngularDialogComponent} from "../ui/angulardialog/angular-dialog-component.directive";
import {DialogModule} from "../ui/angulardialog/dialog.module";
import {to2} from "../util/utils";
import {Users} from "../user/userService";
import {IDocument, redirectToItem} from "../item/IItem";
import {TimUtilityModule} from "../ui/tim-utility.module";
import {archivePolicyNames, ArchiveType, ListOptions} from "./listOptionTypes";

@Component({
    selector: "message-list-creation",
    template: `
        <tim-dialog-frame class="form-horizontal">
            <ng-container header>
                Message list creation
            </ng-container>
            <ng-container body>
                <div *ngIf="errorMessage.length > 0" class="alert alert-danger">
                    <ul>
                        <li *ngFor="let error of errorMessage">{{error}}</li>
                    </ul>
                </div>
                <div class="form-group">
                    <label for="list-name" class="list-name text-left control-label col-sm-3">List name: </label>
                    <div class="col-sm-8">
                        <div class="input-group">
                            <input type="text" class="form-control" name="list-name" 
```

```
    id="list-name"
                                    [(ngModel)]="listname"
                                    (keyup)="checkNameRequirementsLocally()"/>
                            <div class="input-group-addon">@</div>
                            <select id="domain-select" class="form-control" name="domain-select" [(ngModel)]="domain">
                                <option [disabled]="domains.length" *ngFor="let domain of domains">{{domain}}</option>
                            </select>
                        </div>
                    </div>
                </div>
                <div class="archive-options">
                    <p class="list-name">List archive policy: </p>
                    <ul class="archive-list">
                        <li *ngFor="let option of archiveOptions">
                            <label class="radio" for="archive-{{option.archiveType}}">
                            <input
                                    name="items-radio"
                                    type="radio"
                                    id="archive-{{option.archiveType}}"
                                    [value]="option.archiveType"
                                    [(ngModel)]="archive"
                            />
                            {{option.policyName}}
                            </label>
                        </li>
                    </ul>
                </div>
            </ng-container>
            <ng-container footer>
                <tim-loading *ngIf="disableCreate"></tim-loading>
                <button [disabled]="disableCreate" class="timButton" type="button" (click)="newList() ">Create</button>
            </ng-container>
        </tim-dialog-frame>
    `,
    styleUrls: ["message-list-create-dialog.component.scss"],
})
export class MessageListCreateDialogComponent extends AngularDialogComponent<
    unknown,
    unknown
> {
    disableCreate: boolean = false;
    protected dialogName = "MessageList";
    listname: string = "";
    errorMessage: string[] = [];

    urlPrefix: string = "/messagelist";

    domains: string[] = [];
    domain: string = "";

    // List has a public archive by default.
    archive: ArchiveType = ArchiveType.PUBLIC;
    archiveOptions = archivePolicyNames;

    constructor(private http: HttpClient) {
        super();
```

```
86      }
87
88      ngOnInit(): void {
89          if (Users.isLoggedIn()) {
90              void this.getDomains();
91          }
92      }
93
94      /**
95       * Fetch possible domains to be used for email lists.
96       * @private
97       */
98      private async getDomains() {
99          const result = await to2(
100             this.http.get<string[]>(`${this.urlPrefix}/domains`).toPromise()
101         );
102         if (result.ok) {
103             this.domains = result.result;
104
105             // Set default domain.
106             this.domain = this.domains[0];
107         } else {
108             this.errorMessage = [
109                 `Failed to load domains, list creation can't continue. The following ↩
    error provides details: ${result.result.error.error}`,
110             ];
111             // Creating a message list isn't possible at this time if domains are not ↩
    given. Therefore disable creation
112             // button.
113             this.disableCreate = true;
114         }
115     }
116
117     /**
118      * Launching the creation of a new list. Verifies the basic name rules in the ↩
    client before involving the server.
119      */
120     async newList() {
121         if (!this.checkNameRequirementsLocally()) {
122             return;
123         }
124
125         this.errorMessage = [];
126         this.disableCreate = true;
127         const result = await this.createList({
128             name: this.listname,
129             domain: this.domain,
130             archive: this.archive,
131         });
132         if (!result.ok) {
133             this.errorMessage = [result.result.error.error];
134             this.disableCreate = false;
135         } else {
136             redirectToItem(result.result);
137         }
138     }
139
140     /**
141      * The call to create new list.
```

```
142      * @param options Required amount of options to create a new message list. Here the necessary arguments are list's
143      * name and archive policy.
144      * @private
145      */
146     private createList(options: ListOptions) {
147         return to2(
148             this.http
149                 .post<IDocument>(`${this.urlPrefix}/createlist`, {options})
150                 .toPromise()
151         );
152     }
153 
154     /**
155      * Check list name requirements locally.
156      *
157      * If you make changes here, make sure to check that the server checks the same things. Otherwise there will
158      * inconsistant name checking and a confused user.
159      *
160      * @returns {boolean} Returns true if name requirements are met. Otherwise returns false.
161      */
162     checkNameRequirementsLocally(): boolean {
163         // Clear old error messages.
164         this.errorMessage = [];
165 
166         // Name length is within length boundaries.
167         if (this.listname.length <= 5 || 36 <= this.listname.length) {
168             this.errorMessage.push("Name not in length boundaries");
169         }
170 
171         // Name starts with a character that is a letter a - z.
172         const regExpStartCharacter: RegExp = /^[a-z]/;
173         if (!regExpStartCharacter.test(this.listname)) {
174             this.errorMessage.push("Name should start with a lowercase letter");
175         }
176 
177         // Name contains at least one digit.
178         const regExpAtLeastOneDigit: RegExp = /\d/;
179         if (!regExpAtLeastOneDigit.test(this.listname)) {
180             this.errorMessage.push("Name should contain at least one digit");
181         }
182 
183         // Name can't contain sequential dots.
184         const regExpMultipleDots: RegExp = /\.\.+/;
185         if (regExpMultipleDots.test(this.listname)) {
186             this.errorMessage.push("Name shouldnÂ´t contain multiple dots");
187         }
188 
189         // Name doesn't end in a dot.
190         // ESLint prefers to not use regex for this. And by "prefer" we mean this won't transpile with a regular
191         // expression.
192         if (this.listname.endsWith(".")) {
193             this.errorMessage.push("Name shouldnÂ´t end in a dot");
194         }
195 
196         // Name contains only acceptable characters, which are:
```

```
        //     letters                  a - z
        //     numbers                  0 - 9
        //     dot                      '.'
        //     underscore               '_'
        //     hyphen (or "minus sign") '-'
        // The following regular expression searches for characters that are *not* one of the above. If those are not
        // found the name is of correct form. Notice that hyphen is in two different roles and one hyphen has
        // to be escaped. The dot does not have to be escaped here.
        const regExpNonAllowedCharacters: RegExp = /[^a-z0-9.\-_]/;
        if (regExpNonAllowedCharacters.test(this.listname)) {
            this.errorMessage.push("Name has forbidden characters");
        }
        return this.errorMessage.length == 0;
    }
}

@NgModule({
    declarations: [MessageListCreateDialogComponent],
    imports: [BrowserModule, DialogModule, FormsModule, TimUtilityModule],
})
export class MessageListModule {}
```

# timApp/static/scripts/tim/messaging/messagingUtils.ts

```
import {HttpClient} from "@angular/common/http";
import {to2} from "../util/utils";

export async function markAsRead(http: HttpClient, message_id: number) {
    const result = await to2(
        http
            .post("/timMessage/mark_as_read", {
                message_id: message_id,
            })
            .toPromise()
    );
    if (!result.ok) {
        console.error(result.result.error.error); // TODO throw error
    }

    return result;
}
```

# timApp/static/scripts/tim/messaging/showMessageListCreatio

```
import {unknown} from "io-ts";
import {angularDialog} from "../ui/angulardialog/dialog.service";

export async function showMessageListCreation() {
    const {MessageListCreateDialogComponent} = await import(
        "./message-list-create-dialog.component"
    );
    return (
        await angularDialog.open(MessageListCreateDialogComponent, unknown, {
            resetSize: true,
        })
    ).result;
}
```

# timApp/static/scripts/tim/messaging/tim-message-send.component.ts

```
1  import {Component, EventEmitter, Input, Output} from "@angular/core";
2  import {HttpClient} from "@angular/common/http";
3  import {$http} from "../util/ngimport";
4  import {to, to2} from "../util/utils";
5  import {Users} from "../user/userService";
6
7  interface TimMessageOptions {
8      archive: boolean;
9      important: boolean;
10     messageChannel: boolean;
11     pageList: string;
12     isPrivate: boolean;
13     reply: boolean;
14     readReceipt: boolean;
15     expires: Date | undefined;
16     sender: string | null;
17     senderEmail: string | null;
18 }
19
20 @Component({
21     selector: "tim-message-send",
22     template: `
23         <div class="csRunDiv tableEmail" style="padding: 1em;" *ngIf="recipientList">
24             <tim-close-button (click)="closeComponent()"></tim-close-button>
25             <p>Recipients:</p>
26             <p><textarea [(ngModel)]="recipientList" rows="4" cols="40" ↩
    (input)="somethingChanged()"></textarea>
27             </p>
28             <p>Subject: <input [(ngModel)]="messageSubject" size="60" ↩
    (input)="somethingChanged()"></p>
29             <p>Message content:</p>
30             <p><textarea [(ngModel)]="messageBody" rows="10" cols="70" ↩
    (input)="somethingChanged()"></textarea></p>
31             <span class="cursor-pointer" (click)="toggleOptions(); ↩
    somethingChanged()"><a><span class="glyphicon"
32                                 [ngClass]="showOptions ? 'glyphicon-minus' : ↩
    'glyphicon-plus'"></span></a>{{showOptions ? "Hide" : "Show"}} message ↩
    options</span>
33             <div *ngIf="showOptions">
34             <label *ngIf="!defaultEmail"><input type="checkbox"
35                                         [(ngModel)]="timMessageOptions.archive" ↩
    disabled>Archive message</label><br/>
36             <label *ngIf="!defaultEmail"><input type="checkbox"
37                                         [(ngModel)]="timMessageOptions.important" ↩
    disabled>Mark message as important (currently only applies to TIM messages)</label>
38             <p class="bold">Send as (choose at least one of the two)</p><!--<label ↩
    *ngIf="!defaultEmail"><input type="checkbox"
39                                         ↩
    [(ngModel)]="timMessageOptions.messageChannel">to recipient's own message ↩
    channels</label><br/>-->
40             <label><input type="checkbox" (change)="notDefault()"
41                                         [(ngModel)]="email">email</label><br/>
42                 <ul *ngIf="email">
43                     <li>
44                     <label><input type="radio"
```

```
45                                      [(ngModel)]="defaultEmail" name="defaultEmail" ↩
    [value]="false">Use TIM to send</label>
46                 </li><li>
47                     <label><input type="radio"
48                                      [(ngModel)]="defaultEmail" name="defaultEmail" ↩
    [value]="true">Use your default email client (recipients will see each others' ↩
    addresses)</label>
49                 </li></ul>
50                 <ul *ngIf="email && !defaultEmail">
51                 <li><label><input type="radio"
52                                      [(ngModel)]="replyAll" name="replyAll" ↩
    [value]="false" checked>Recipient only replies to sender (sees message as ↩
    private)</label></li>
53                 <li><label><input type="radio"
54                                      [(ngModel)]="replyAll" name="replyAll" ↩
    [value]="true">Recipient replies all by default (sees message as a group ↩
    message)</label></li>
55             </ul>
56             <label *ngIf="!defaultEmail"><input type="checkbox" ↩
    (change)="emptyPageList()"
57                                      [(ngModel)]="timMessage">TIM message</label><br/>
58                 <div class="tim-message" *ngIf="timMessage && !defaultEmail"><span ↩
    *ngIf="timMessage && !defaultEmail">Pages to send TIM message to: (enter URL ↩
    addresses)<br/>(URLs will be automatically shortened)</span><br/>
59                 <tim-alert *ngIf="urlError" severity="danger">
60                     {{ urlError }}
61                 </tim-alert>
62             <textarea [(ngModel)]="timMessageOptions.pageList" (input)="checkUrls()" ↩
    rows="4" cols="70"></textarea><br/>
63             <label><input type="checkbox"
64                                      [(ngModel)]="timMessageOptions.isPrivate" ↩
    disabled>Recipient sees TIM message as private</label><br/>
65             <label><input type="checkbox"
66                                      [(ngModel)]="timMessageOptions.reply">TIM ↩
    message can be replied to</label><br/>
67             <label><input type="checkbox"
68                                      [(ngModel)]="timMessageOptions.readReceipt" ↩
    disabled>TIM message can be marked as read</label><br/>
69             <p class="form-group">
70                 <label for="expiration-selector">TIM message will be removed on:</label>
71                 <tim-datetime-picker id="expiration-selector"
72                                      [(time)]="timMessageOptions.expires"
73                                     placeholder="No automatic date">
74                 </tim-datetime-picker>
75             </p><br/></div></div>
76             <p>
77                 <button class="timButton" id="sendButton" ↩
    [disabled]="disableSendButton()"
78                         (click)="sendMessage()">
79                     Send
80                 </button>
81                 <span class="savedtext" *ngIf="messageMsg"> {{messageMsg}}</span>
82             </p>
83         </div>
84     `,
85     styleUrls: ["./tim-message-send.scss"],
86 })
87 export class TimMessageComponent {
88     /**
```

```
89      *  TODO: This component has a minor bug. If the textfield of recipients is ↩
    emptied by hand, the component closes
90      *  and it can't be reopened unless the recipientList variable changes. A ↩
    hypothetical fix would be to use a
91      *  separate flag in the *ngIf, instead of just the recipientList variable. Then ↩
    this flag would only be operated to
92      *  close when the component is closed from the x. It would propably requrie ↩
    change detection for the recipientList
93      *  variable, as its length grows beoynd 0 the the flag is set on?
94      */
95
96      @Input()
97      recipientList: string = "";
98      @Output() recipientListChange = new EventEmitter<string>();
99      messageSubject: string = "";
100     messageBody: string = "";
101     showOptions: boolean = false;
102     emailbcc: boolean = false;
103     emailbccme: boolean = true;
104     email: boolean = true;
105     defaultEmail: boolean = false;
106     replyAll: boolean | undefined = false;
107     messageMsg: string = "";
108     timMessage: boolean = false;
109     urlError: string = "";
110     formChanged: boolean = true;
111     timMessageOptions: TimMessageOptions = {
112         messageChannel: false,
113         archive: false,
114         important: false,
115         isPrivate: false,
116         pageList: "",
117         readReceipt: true,
118         reply: false,
119         expires: undefined,
120         sender: Users.getCurrent().real_name,
121         senderEmail: Users.getCurrent().email,
122     };
123     @Input()
124     docId?: number;
125
126     constructor(private http: HttpClient) {}
127
128     /**
129      * Close the component and propagate the information to parent component.
130      */
131     closeComponent() {
132         this.recipientList = "";
133         this.recipientListChange.emit(this.recipientList);
134     }
135
136     toggleOptions() {
137         this.showOptions = !this.showOptions;
138     }
139
140     notDefault() {
141         this.defaultEmail = false;
142     }
143
```

```
144     emptyPageList() {
145         this.timMessageOptions.pageList = "";
146     }
147
148     somethingChanged() {
149         this.formChanged = true;
150     }
151
152     // Checks if all mandatory fields have values
153     disableSendButton() {
154         return (
155             !this.formChanged ||
156             this.urlError ||
157             (!this.timMessageOptions.messageChannel &&
158                 !this.email &&
159                 !this.timMessage) ||
160             (this.timMessage && !this.timMessageOptions.pageList)
161         );
162     }
163
164     // Checks if the URLs that the user wants to save TIM message to actually exist ↩
    in TIM
165     // and that the user has at least edit access to them
166     // Also shortens existing URLs
167     async checkUrls() {
168         this.urlError = "";
169         const result = await to2(
170             this.http
171                 .post<{shortened_urls: string}>("/timMessage/url_check", {
172                     urls: this.timMessageOptions.pageList,
173                 })
174                 .toPromise()
175         );
176         if (!result.ok) {
177             this.urlError = result.result.error.error;
178             console.error(result.result.error.error);
179         } else {
180             this.timMessageOptions.pageList = result.result.shortened_urls;
181         }
182     }
183
184     // resets form to its initial values
185     resetForm() {
186         this.messageMsg = "Sent!";
187         setTimeout((): void => {
188             this.messageMsg = "";
189         }, 5000);
190         this.showOptions = false;
191         this.emailbcc = false;
192         this.emailbccme = true;
193         this.email = true;
194         this.defaultEmail = false;
195         this.replyAll = false;
196         this.timMessage = false;
197         this.formChanged = false;
198         this.timMessageOptions = {
199             messageChannel: false,
200             archive: false,
201             important: false,
```

```
202             isPrivate: false,
203             pageList: "",
204             readReceipt: true,
205             reply: false,
206             expires: undefined,
207             sender: Users.getCurrent().real_name,
208             senderEmail: Users.getCurrent().email,
209         };
210     }
211
212     public async sendMessage() {
213         // send as TIM message
214         if (this.timMessage) {
215             const result = await this.postTimMessage(this.timMessageOptions);
216             if (!result.ok) {
217                 console.error(result.result.error.error);
218             }
219         }
220         // send as email in TIM
221         if (this.email && !this.defaultEmail) {
222             await this.sendEmailTim();
223             return;
224         }
225         // TODO: iPad do not like ;
226         if (this.email && this.defaultEmail) {
227             let addrs = this.recipientList.replace(/\n/g, ",");
228             let bcc = "";
229             if (this.emailbcc) {
230                 bcc = addrs;
231                 addrs = "";
232             }
233             if (this.emailbccme) {
234                 if (bcc) {
235                     bcc += ",";
236                 }
237                 bcc += Users.getCurrent().email;
238             }
239             window.location.href =
240                 "mailto:" +
241                 addrs +
242                 "?" +
243                 "subject=" +
244                 this.messageSubject +
245                 "&" +
246                 "body=" +
247                 this.messageBody +
248                 "&" +
249                 "bcc=" +
250                 bcc;
251             this.resetForm();
252         }
253         this.resetForm();
254     }
255
256     private postTimMessage(options: TimMessageOptions) {
257         const message = {
258             messageBody: this.messageBody,
259             messageSubject: this.messageSubject,
260             recipients: this.recipientList.split(/\n/g),
```

```
        };
        return to2(
            this.http.post("/timMessage/send", {options, message}).toPromise()
        );
    }

    async sendEmailTim() {
        if (!this.docId) {
            this.messageMsg = "Cannot send email without docId";
            return;
        }
        this.messageMsg = ""; // JSON.stringify(response);
        const url = `/multiSendEmail/${this.docId}`;
        let response;
        // if reply all is chosen
        if (this.replyAll) {
            response = await to(
                $http.post<string[]>(url, {
                    rcpt: this.recipientList.replace(/\n/g, ";"),
                    subject: this.messageSubject,
                    msg: this.messageBody,
                    bccme: this.emailbccme,
                })
            );
            if (!response.ok) {
                this.messageMsg = response.result.data.error;
            } else {
                this.resetForm();
            }
        } else {
            // if only reply to sender is chosen
            const recipients = this.recipientList.split(/\n/g);
            for (const recipient of recipients) {
                response = await to(
                    $http.post<string[]>(url, {
                        rcpt: recipient,
                        subject: this.messageSubject,
                        msg: this.messageBody,
                        bccme: this.emailbccme,
                    })
                );
                if (!response.ok) {
                    this.messageMsg = response.result.data.error;
                }
            }
            this.resetForm();
        }
    }
}
```

# timApp/static/scripts/tim/messaging/tim-message-send.scss

```
tim-datetime-picker {
  display: inline-block;
  width: 50%;
  float: right;
}
```

```
input[type='checkbox'], input[type='radio'] {
  margin-right: 10px;
}

ul {
  list-style-type: none;
}

label {
  font-weight: normal;
}

.bold {
  font-weight: bold;
  margin-top: 10px;
}

.glyphicon {
  margin: 0px 10px 20px 0px;
}

.cursor-pointer {
  font-weight: bold;
}

.tim-message {
  padding-left: 40px;
}

#optionsButton {
  margin-bottom: 10px;
}
```

## timApp/static/scripts/tim/messaging/tim-message-view.component.scss

```
tim-message {
  display: block;
  padding-bottom: 5px;
}
```

## timApp/static/scripts/tim/messaging/tim-message-view.component.ts

```
import {Component, NgModule, OnInit} from "@angular/core";
import {CommonModule} from "@angular/common";
import {itemglobals} from "tim/util/globals";
import {to2} from "tim/util/utils";
import {HttpClient} from "@angular/common/http";
import {TimMessageModule} from "./tim-message.component";

@Component({
    selector: "tim-message-view",
    template: `
        <ng-container *ngFor="let timMessage of timMessages">
            <tim-message [message]=timMessage></tim-message>
```

```
        </ng-container>
    `,
    styleUrls: ["tim-message-view.component.scss"],
})
export class TimMessageViewComponent implements OnInit {
    ngOnInit(): void {
        const current_item = itemglobals().curr_item;
        if (current_item) {
            const itemId = current_item.id;
            void this.loadValues(itemId);
        }
    }

    constructor(private http: HttpClient) {}

    timMessages?: TimMessageData[];

    async loadValues(itemId: number) {
        const messages = await to2(
            // get messages shown on current page
            this.http
                .get<TimMessageData[]>(`/timMessage/get/${itemId}`)
                .toPromise()
        );

        if (messages.ok) {
            this.timMessages = messages.result;
        } else {
            console.error(messages.result.error.error);
        }
    }
}

export interface TimMessageData {
    // Information about the message retrieved from server
    id: number;
    sender: string;
    doc_id: number;
    par_id: string;
    can_mark_as_read: boolean;
    can_reply: boolean;
    display_type: number;
    message_body: string;
    message_subject: string;
    recipients: [string];
}

@NgModule({
    declarations: [TimMessageViewComponent],
    exports: [TimMessageViewComponent],
    imports: [CommonModule, TimMessageModule],
})
export class TimMessageViewModule {}
```

## timApp/static/scripts/tim/messaging/tim-message.component.s

```
.timMessageDisplay {
    border: 1px solid black;
    border-radius: 6px;
```

```
    padding: 6px 8px 8px 8px;
    background-color: var(--basic-color-inverse);

    @media (min-width: 1400px) {
        margin: 0 10% 0 10%;
    }

    .fullMessageContent, .cutMessageContent {
        text-align: justify;
        text-justify: auto;
        margin-bottom: 4px;
    }

    p {
        margin-bottom: 4px;
    }

    .messageHeading {
        font-weight: bold;
    }

    .replyArea {
        padding-top: 5px;

        textarea {
            width: 500px;
            display: block;
        }

        .timButton {
            margin-right: 5px;
        }

        .sent {
            margin-top: 5px;
        }
    }

    .readReceiptArea {
        margin-top: 5px;
        display: grid;
        grid-template-columns: 12% 40% 48%;
        grid-template-areas: "checkbox link button";

        input[type='checkbox'] {
            grid-area: checkbox;
            justify-self: start;
            align-self: start;
        }

        label {
            grid-area: checkbox;
            font-weight: normal;
            justify-self: start;
            align-self: start;
        }

        .readReceiptLink {
            grid-area: link;
```

```
            margin-top: 5px;
            font-size: 80%;
            justify-self: start;
            align-self: start;
        }

        .timButton {
            grid-area: button;
            justify-self: end;
            align-self: start;
        }
    }
}
```

# timApp/static/scripts/tim/messaging/tim-message.component.t

```
import {Component, Input, NgModule, OnInit} from "@angular/core";
import {CommonModule} from "@angular/common";
import {HttpClient} from "@angular/common/http";
import {markAsRead} from "tim/messaging/messagingUtils";
import {FormsModule} from "@angular/forms";
import {to2} from "tim/util/utils";
import {TimUtilityModule} from "tim/ui/tim-utility.module";
import {TimMessageData} from "./tim-message-view.component";

interface ReplyOptions {
    archive: boolean;
    messageChannel: boolean;
    pageList: string;
    recipient: string | null;
    readReceipt: boolean;
    repliesTo?: number;
}

@Component({
    selector: "tim-message",
    template: `
        <ng-container *ngIf="showMessage">
            <div class="timMessageDisplay">
                <tim-close-button class="closeButton" (click)="closeMessage()"></tim-close-button>
                <p class="messageInformation">
                    <span class="from">From: </span>
                    <span class="sender">{{sender}}</span>
                    <!-- TODO Display what group the message is related to
                    <span class="group" *ngIf="messageToGroup">, {{group}}</span>
                    -->
                </p>
                <p class="messageHeading">{{heading}}</p>
                <div class="fullMessageContent" *ngIf="showFullContent">
                    <div class="fullContentText" [innerHTML]="fullContent"></div>
                    <p class="toggleReadMore" *ngIf="messageOverMaxLength">
                        <a (click)="toggleDisplayedContentLength()">Read less</a>
                    </p>
                </div>
                <div class="cutMessageContent" *ngIf="!showFullContent">
                    <div class="shownContentText" [innerHTML]="shownContent"></div>
                    <p class="toggleReadMore"><a (click)="toggleDisplayedContentLength()">Read more</a></p>
```

```
42                 </div>
43                 <div class="buttonArea">
44                     <button class="timButton" *ngIf="canReply" ↩
    (click)="reply()">Reply</button>
45                 </div>
46                 <div class="replyArea" *ngIf="showReply">
47                     <p class="replyTo">To: {{sender}}</p>
48                     <textarea class="replyTextarea" id="reply-message" ↩
    name="reply-message"
49                               [(ngModel)]="replyMessage"></textarea>
50                     <div class="sent">
51                         <button class="timButton" [disabled]="!canSendReply" ↩
    (click)="sendReply()">Send</button>
52                         <span class="replySent" *ngIf="replySent">Sent!</span>
53                     </div>
54                 </div>
55                 <form class="readReceiptArea">
56                     <label class="markAsReadLabel"><input class="markAsReadCheckbox"
57                                                           type="checkbox"
58                                                           name="mark-as-read"
59                                                           id="mark-as-read"
60                                                           [disabled]="!canMarkAsRead ↩
    || markedAsRead"
61                                                           (click)="markAsRead()"/> ↩
    Mark as Read</label>
62                     <span class="readReceiptLink" *ngIf="markedAsRead">
63                         Read receipt can be cancelled in <a ↩
    href="/view/messages/tim-messages">your messages</a>
64                     </span>
65                     <button class="timButton" title="Close Message" ↩
    (click)="closeMessage()">
66                         Close
67                     </button>
68                 </form>
69             </div>
70         </ng-container>
71     `,
72     styleUrls: ["tim-message.component.scss"],
73 })
74 export class TimMessageComponent implements OnInit {
75     @Input()
76     message!: TimMessageData;
77
78     messageMaxLength: number = 210;
79     messageOverMaxLength: boolean = false;
80     showMessage: boolean = true;
81     fullContent?: string;
82     shownContent?: string;
83     showFullContent: boolean = true;
84     showReply: boolean = false;
85     canMarkAsRead: boolean = true;
86     markedAsRead: boolean = false;
87     replyMessage: string = "";
88     replySent: boolean = false;
89     canReply: boolean = true; // show/hide 'Reply' button
90     canSendReply: boolean = true; // enable/disable 'Send' button
91     sender?: string;
92     messageToGroup: boolean = true; // can't get from database
93     group?: string; // can't get from database
```

```
94      heading?: string;
95      replyOptions: ReplyOptions = {
96          archive: true,
97          messageChannel: false,
98          pageList: "messages/tim-messages",
99          recipient: "",
100         readReceipt: true,
101         repliesTo: undefined,
102     };
103
104     constructor(private http: HttpClient) {}
105
106     /**
107      * Toggles between showing the full message content and the shortened version.
108      */
109     toggleDisplayedContentLength(): void {
110         this.showFullContent = !this.showFullContent;
111     }
112
113     /**
114      * Hides the message view; shows an alert about sending a read receipt and an ↩
        option to cancel.
115      */
116     async markAsRead() {
117         const result = await markAsRead(this.http, this.message.id);
118         if (result.ok) {
119             this.markedAsRead = true;
120         }
121     }
122
123     /**
124      * Shows or hides the reply area.
125      */
126     reply(): void {
127         this.showReply = !this.showReply;
128     }
129
130     /**
131      * Allows/disallows marking as read for testing purposes.
132      */
133     readReceipt(): void {
134         this.canMarkAsRead = !this.canMarkAsRead;
135     }
136
137     /**
138      * Sends reply to sender
139      */
140     async sendReply() {
141         this.replySent = true;
142         this.canSendReply = false;
143         if (this.sender) {
144             this.replyOptions.recipient = this.sender;
145             this.replyOptions.repliesTo = this.message.id;
146         } else {
147             console.log("no recipient, can't send");
148         }
149         const result = await to2(
150             this.http
151                 .post("/timMessage/reply", {
```

```
                    options: this.replyOptions,
                    messageBody: {
                        messageBody: this.replyMessage,
                        messageSubject: this.heading + " [Re]",
                        recipients: [this.replyOptions.recipient],
                    },
                })
                .toPromise()
        );
        if (!result.ok) {
            console.error(result.result.error.error);
        }
    }

    /**
     * Hides the entire message. If the message can't be replied to or marked as read
     * by recipient, closing it hides it permanently by marking it as read in database.
     */
    async closeMessage() {
        this.showMessage = false;

        if (!this.canReply && !this.canMarkAsRead) {
            await markAsRead(this.http, this.message.id);
        }
    }

    ngOnInit(): void {
        this.setValues(this.message);
    }

    setValues(timMessage: TimMessageData) {
        this.sender = timMessage.sender;
        // TODO Display what group the message is related to; currently can't retrieve from database
        // this.group = "ohj1k21";
        this.heading = timMessage.message_subject;
        this.fullContent = timMessage.message_body;
        this.canMarkAsRead = timMessage.can_mark_as_read;
        this.canReply = timMessage.can_reply;

        if (this.fullContent.length > this.messageMaxLength) {
            this.messageOverMaxLength = true;
            this.showFullContent = false;
            this.shownContent =
                this.fullContent.substr(0, this.messageMaxLength) + "...";
        }
    }
}

@NgModule({
    declarations: [TimMessageComponent],
    exports: [TimMessageComponent],
    imports: [CommonModule, FormsModule, TimUtilityModule],
})
export class TimMessageModule {}
```

# timApp/static/tim_docs/initial/tim_msg_preamble.md

```
``` {settings=""}
```
```

```
macros:
  subject:
  sendername:
  senderemail:
  recipients:
css: |!!
.messageheader p {
    font-size: small;
}
.messageheader {
    border-bottom-style: solid;
    border-bottom-width: thin;
    margin-bottom: 2em;
}
span.messagetitle {
    font-size: large;
    font-weight: bold;
}
!!
```

```
#- {.messageheader}
[]{.messagetitle} \
Sender: ,  \
Recipients:
```

# timApp/tests/server/test_messagelist.py

```
from timApp.auth.accesshelper import get_doc_or_abort
from timApp.messaging.messagelist.listoptions import ArchiveType
from timApp.messaging.messagelist.messagelist_models import MessageListModel
from timApp.tests.server.timroutetest import TimRouteTest


class MessageListTest(TimRouteTest):
    """Server test for message lists."""

    def creation_test_valid_name(self):
        """Test creating a message list with a valid user and valid list name."""
        # Make testuser1 admin.
        self.login_test1()
        testuser1 = self.current_user
        self.make_admin(testuser1)
        # Create the list. This name corresponds with name requirements. Its long enough, contains only allowed
        # characters and has at least one digit.
        list_name = "valid_list5"
        # The archive type is a mandatory value in list creation, but its exact value doesn't have an impact on this
        # test.
        archive = ArchiveType.PUBLIC
        response_item = self.json_post("/messagelist/createlist",
                                       {"options": {"name": list_name,
                                                    "archive": archive.value}
                                        }, expect_status=200)
        # Expected response is in JSON, and it's a DocEntry of the created admin doc. Get the created admin doc of
        # the message list.
        admin_doc = get_doc_or_abort(response_item["id"])
```

```
        # Verify that the list exists in the db.
        message_list = MessageListModel.query(name=list_name).one()
        # Verify returned admin doc is the same message list's admin doc in db.
        self.assertEqual(message_list.admin_doc, admin_doc)
        # Verify name and archive type are as intented in the db.
        self.assertEqual(message_list.name, list_name)
        self.assertEqual(message_list.archive, archive)
```

# timApp/tests/server/test_tim_message.py

```
from timApp.auth.accesstype import AccessType
from timApp.document.docentry import DocEntry
from timApp.folder.folder import Folder
from timApp.messaging.timMessage.internalmessage_models import InternalMessageDisplay, InternalMessage
from timApp.tests.server.timroutetest import TimRouteTest


class UrlTest(TimRouteTest):
    def test_url_check(self):
        self.login_test2()
        self.create_doc(self.get_personal_item_path("testdoc"))
        self.create_folder(self.get_personal_item_path("testfolder"))

        self.login_test1()
        self.json_post('/timMessage/url_check',
                       {"urls": "http://www.google.com"},
                       expect_status=404)
        self.create_doc(self.get_personal_item_path("testdoc"))
        self.create_folder(self.get_personal_item_path("testfolder"))
        self.json_post('/timMessage/url_check',
                       {"urls": "http://localhost/view/users/test-user-1/testdoc"},
                       expect_status=200)
        self.json_post('/timMessage/url_check',
                       {"urls": "http://localhost/view/users/test-user-1/testdoc"},
                       expect_content={"shortened_urls": "users/test-user-1/testdoc"})
        self.json_post('/timMessage/url_check',
                       {"urls": "http://localhost/view/users/test-user-1/testfolder"},
                       expect_status=200)
        self.json_post('/timMessage/url_check',
                       {"urls": "http://localhost/view/users/test-user-2/testdoc"},
                       expect_status=401)
        self.json_post('/timMessage/url_check',
                       {"urls": "http://localhost/view/users/test-user-2/testfolder"},
                       expect_status=401)
        self.json_post('/timMessage/url_check',
                       {
            "urls": "http://localhost/teacher/users/test-user-1/testdoc\nhttp://localhost/teacher/users/test-user-1
                       expect_status=200)
        self.json_post('/timMessage/url_check',
                       {
            "urls": "http://localhost/teacher/users/test-user-1/testdoc\nhttp://localhost/teacher/users/test-user-1
                       expect_content={"shortened_urls": "users/test-user-1/testdoc\nusers/test-user-1"})
        self.json_post('/timMessage/url_check',
                       {"urls": "  http://localhost/view/users/test-user-1/testdoc"},
                       expect_status=200)
```

```
46          self.json_post('/timMessage/url_check',
47                         {"urls": "http://localhost/view/users/test-user-1/testdoc  "},
48                         expect_status=200)
49          self.json_post('/timMessage/url_check',
50                         {"urls": ↩
    "http://localhost/view/users/test-user-1/testdoc#jjndsg"},
51                         expect_status=200)
52          self.json_post('/timMessage/url_check',
53                         {"urls": "http://localhost/view/users/test-user-1/testfolder/"},
54                         expect_status=200)
55          self.json_post('/timMessage/url_check',
56                         {"urls": ↩
    "http://localhost/view/users/test-user-1/testfolder/testdoc"},
57                         expect_status=404)
58
59
60  class SendMessageTest(TimRouteTest):
61      def test_send_message(self):
62          self.login_test1()
63
64          f = self.create_folder(self.get_personal_item_path('testfolder'))
65          self.test_user_2.grant_access(Folder.get_by_id(f['id']), AccessType.edit)
66
67          self.login_test2()
68          self.json_post('/timMessage/send',
69                         {'options': {
70                             'messageChannel': False,
71                             'archive': False,
72                             'important': False,
73                             'isPrivate': False,
74                             'pageList': "users/test-user-1/testfolder",
75                             'readReceipt': True,
76                             'reply': True,
77                             'sender': self.test_user_2.name,
78                             'senderEmail': self.test_user_2.email},
79                             'message': {
80                                 'messageBody': "test message",
81                                 'messageSubject': "test subject",
82                                 'recipients': [self.test_user_1.email]
83                             }},
84                         expect_status=200)
85
86          self.login_test1()
87          self.get('/view/messages/tim-messages', expect_status=200)  # tim-messages ↩
    folder created successfully
88
89          display = ↩
    InternalMessageDisplay.query.filter_by(usergroup_id=self.get_test_user_1_group_id()).first()
90          msg = InternalMessage.query.filter_by(id=display.message_id).first()
91          msg_doc = DocEntry.query.filter_by(id=msg.doc_id).first()
92          self.get(f'/view/{msg_doc.name}', expect_status=200)  # document for message ↩
    created successfully
```