Automated Communication Engine

Release 1.5.0

edX Inc.

Dec 07, 2022

CONTENTS

The Automated Communication Engine, ACE for short, is a Django app for messaging learners on the edX platform. This app can be installed in any Open edX project, but has only been tested with `edx-platform`. Email delivery (via Sailthru or Django email) is the only current delivery channel. In the future we may add support for other delivery channels such as push notifications.

CHAPTER

ONE

GETTING STARTED

If you have not already done so, create/activate a virtualenv. Unless otherwise stated, assume all terminal code below is executed within the virtualenv.

1.1 Install dependencies

Dependencies can be installed via the command below.

```
$ make requirements
```

1.2 Configure delivery channels

Certain delivery channels may require additional configuration before they will function correctly.

1.2.1 `SailthruEmailChannel` Settings

```
ACE_CHANNEL_SAILTHRU_DEBUG = False
ACE_CHANNEL_SAILTHRU_TEMPLATE_NAME = "Some template name"
ACE_CHANNEL_SAILTHRU_API_KEY = "1234567890"
ACE_CHANNEL_SAILTHRU_API_SECRET = "this is secret"
```

1.2.2 `DjangoEmailChannel` Settings

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'localhost'
DEFAULT_FROM_EMAIL = 'hello@example.org'

ACE_CHANNEL_DEFAULT_EMAIL = 'sailthru_email'
ACE_CHANNEL_TRANSACTIONAL_EMAIL = 'django_email'

ACE_ENABLED_CHANNELS = [
    'sailthru_email',
    'django_email',
]
```

1.3 Create a message

Each message sent with ACE is represented by an instance of *Message*. These can be created manually, or can be created by calling *MessageType.personalize()* on a *MessageType* instance. The name and package of the *MessageType* determines what templates will be used when the *Message* is rendered for delivery.

For example, the class

```
# myapp/messages.py

class CustomMessage(edx_ace.message.MessageType):
    pass
```

would use the following templates when rendered for email delivery:

```
myapp/edx_ace/custommessage/email/from_name.txt
myapp/edx_ace/custommessage/email/subject.txt
myapp/edx_ace/custommessage/email/body.html
myapp/edx_ace/custommessage/email/head.html
myapp/edx_ace/custommessage/email/body.txt
```

These all follow the format `{app_label}/edx_ace/{message_name}/{renderer}/{attribute}`, where the `app_label` and `message_name` are defined by the *MessageType* (or the manually created *Message*), and `renderer` and `attribute` come from the renderer being used by the specific delivery channel. The templates will be retrieved using standard Django template resolution mechanisms.

The specific templates needed for existing renderers are listed in *edx_ace.renderers*.

1.4 Transactional messages

Transactional messages such as password reset should be marked as `options.transactional = True`, to ensure that it won't be subject to marketing messages opt-out policies. While not required, transactional messages are recommended to use the `django_email` channel which supports a custom `options.from_address` email. For example:

```
# myapp/messages.py

class PurchaseOrderComplete(edx_ace.message.MessageType):
    def __init__(self, *args, **kwargs):
        super(PurchaseOrderComplete, self).__init__(*args, **kwargs)

        self.options['transactional'] = True
        self.options['from_address'] = settings.ECOMMERCE_FROM_EMAIL
```

1.5 Send a message

The simplest way to send a message using ACE is to just create it, and call *edx_ace.ace.send()*.

```
from edx_ace import ace
from edx_ace.messages import Message

msg = Message(
    name="test_message",
    app_label="my_app",
    recipient=Recipient(lms_user_id='123456', email='a_user@example.com'),
    language='en',
    context={
        'stuff': 'to personalize the message',
    }
)
ace.send(msg)
```

The `name` and `app_label` attributes are required in order for ACE to look up the correct templates in the Django environment.

For messages being sent from multiple places in the code, it can be simpler to define a *MessageType* first, and then *MessageType.personalize()* it.

```
from edx_ace import ace
from edx_ace.messages import Message

class TestMessage(MessageType):
    APP_LABEL = "my_app"  # Optional
    NAME = "test_message"  # Optional

msg_type = TestMessage(
    context={
        'generic_stuff': 'that is applicable to all recipients'
    }
)

for recipient in recipients:
    msg = msg_type.personalize(
        recipient=recipient,
        language='en',
        context={
            'stuff': 'to personalize the message',
        }
    )
    ace.send(msg)
```

CHAPTER

TWO

DESIGN

2.1 Overview

The **Automated Communications Engine (A.C.E.)** is a framework for automatically sending messages to users. It is intended to support the identification of recipients and personalization of messages for each recipient.

The intent is for ACE to provide the application-specific logic that is easiest to manage in an environment that is close to the source-of-truth data and outsource the generic functionality that is complex but not a core competency of an education platform (like sending email). This line, of course, is a bit blurry, so we reserve the right in the future to push more functionality into the third-party provider as our needs evolve. We may even choose to delegate *all* of this functionality to a third-party at some point in the future.

Given the complexity of finding the right people to send the right message to with all of the needed personalization, we try to handle that problem as closely to the source-of-truth as possible instead of trying to manage a complex integration with a third-party system.

The following future requirements might encourage us to shift the line between custom and off-the-shelf in the future:

- Preference management across all channels
- Message analytics (open rates, click through rates etc)
- Frequency management and prioritization of messages across all channels and products
- Cost
- Personalized, intelligent timing of delivery (likely driven by a machine learning model)
- Digests and other summaries

2.2 Goals/Constraints

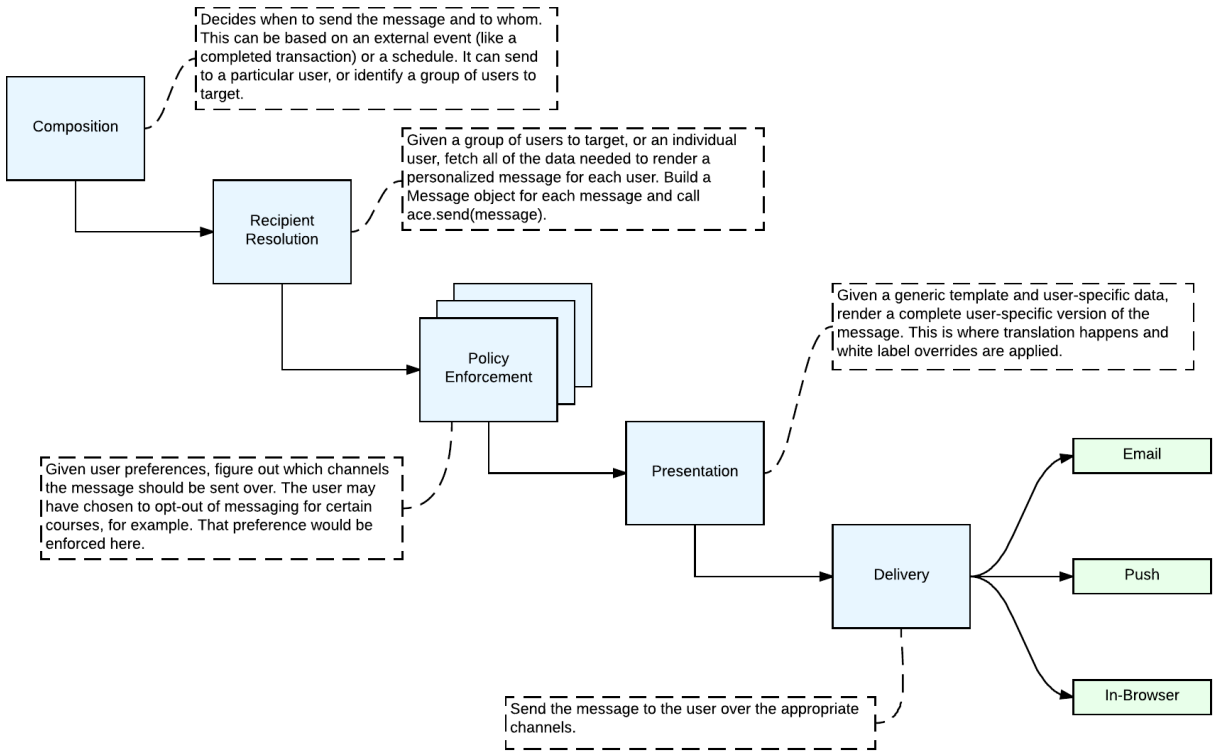
ACE was designed with the following goals in mind:

1. Support edX base requirements (internationalization, accessibility, theming)
2. Allow future extension of adding new delivery mechanisms
3. Allow future extension of message delivery policy

The first goal we supported by choosing to use Django as the templating mechanism, and by making ACE a library that could be used by many Django applications, rather than a separate service that was called by many Django applications.

The second and third goal guided where we added extension points into the ACE architecture.

2.3 Architecture



A **RuntimeEnvironment** is a Django application that has included edx_ace as a djangoapp. ACE is running in the same process as this application.

The **RuntimeEnvironment** provides:

- Translation files (*.po)
- Settings - used to configure ACE
- Template resolution strategies
- Batching and resource management (queueing etc)

A *`MessageType`* represents a type of communication we might want to send to the learner. Our various applications will want to define different *`MessageTypes`*. Examples might include `RecurringReminder`, `OrderConfirmation`, `DeadlineReminder` etc. It is user agnostic, however, it can be used as a factory for *`Message`* objects by "personalizing" the generic message type for a particular user.

A *`Channel`* is a communication channel with the learner. Examples include Email, SMS, Push Notifications, In-Browser Notifications etc.

An **Application** often wants to define several *`MessageTypes`* and knows when to send them to who. It is expected to be implemented as a djangoapp included in the same **RuntimeEnvironment** as ACE.

The **Application** is responsible for defining Django templates for each dynamic field required for each *Channel* for each *MessageType*. For an email, for example, this includes the subject, from_name, body_html etc.

Django templates provide a lot of functionality we want out of the box, including:

- Interpolation of variables
- Control structures (conditionals, loops etc)
- Translations
- White label site-specific overrides
- Inheritance - this is very useful for emails since they often share a lot of HTML
- Static asset management - managing images and CDNs

The **Application** can define *MessageType* and use them to create *Messages*, or it can simply create the *Messages* directly.

A *Recipient* defines the contact information for the person who is intended to receive the message. It must contain all of the needed information for each enabled *Channel*. For example, this might include the user's email address and/or notification key (for mobile push notifications).

The **Application** executes *ace.send(message)* for each message it wants to send. This triggers the ACE message delivery pipeline.

It calls a series of registered *Policy* objects in sequence to determine if the user should actually receive the message and over which channels. This is where user preferences are enforced. These are dynamically loaded using standard python plugin tools.

The **Presentation** tools render the message templates using the Django template engine.

Each *Channel* has exactly one implementation that is called in sequence to transmit the message to the user over the appropriate channel if and only if the policy has allowed communication over that channel. An example is the *edx_ace.channel.sailthru.SailthruEmailChannel*.

2.4 Decisions

2.4.1 Allow for In-Braze Message Templating

Status

Rejected

Context

With the existing ACE channels, Marketing (edX's marketing team) can't directly manage the content of the emails being sent. In order to enhance their ability to improve edX's marketing content, we (edX Engineering) want to expose the contents and performance of the marketing flows that are powered by ACE to that team.

When exposing those data to Marketing, and allowing them to make changes to improve the content, we would like to make sure that they aren't blocked by the engineering organization. However, we also need to be mindful of existing open-source uses of ACE as well, and that we don't lose the ability to maintain those uses.

While looking at Braze functionality, it was determined that there isn't a good way for us to schedule campaigns with flexible dates based on single events. This is a feature we were looking to use to allow Marketing to handle dynamic pacing email (or schedule-based campaigns). However, because of current restrictions in the Braze platform, this is likely not possible.

Decision

We will use existing ACE emails to trigger events in Braze, and then allow Marketing to send campaigns based on those ACE email events. These events will be sent whenever an email is sent by the Braze channel in ACE, directly using the Braze event API (https://www.braze.com/docs/api/endpoints/user_data/post_user_track/). They will include all the properties currently used to template into the ACE Django templates, and will be named based on the name of the ACE email being sent.

We will also add the ability to disable specific Braze-channel emails from being sent via the API, so that we can control which emails will be sent via event trigger Braze campaigns, and which will be sent via the direct email API.

Rejection

As of March 2021, Braze event properties cannot contain nested data. As a result, we cannot emit events that parallel the current message context attributes into Braze effectively. Instead, we will emit events into Segment for dynamic pacing outside of ACE entirely.

CHAPTER

THREE

TESTING

edx-ace has an assortment of test cases and code quality checks to catch potential problems during development. To run them all in the version of Python you chose for your virtualenv:

```
$ make validate
```

To run just the unit tests:

```
$ make test
```

To run just the unit tests and check diff coverage

```
$ make diff_cover
```

To run just the code quality checks:

```
$ make quality
```

To run the unit tests under every supported Python version and the code quality checks:

```
$ make test-all
```

To generate and open an HTML report of how much of the code is covered by test cases:

```
$ make coverage
```

CHAPTER

FOUR

API DOCUMENTATION

Contents

4.1 Basic Interface

ACE (Automated Communications Engine) is a framework for automatically sending messages to users.

edx_ace exports the typical set of functions and classes needed to use ACE.

class edx_ace.**Channel**
Bases: object

Channels deliver messages to users that have already passed through the presentation and policy steps.

Examples include email messages, push notifications, or in-browser messages. Implementations of this abstract class should not require any parameters be passed into their constructor since they are instantiated.

channel_type must be a *ChannelType*.

channel_type = None

abstract deliver(*message*, *rendered_message*)
Transmit a rendered message to a recipient.

Parameters

- **message** (*Message*) – The message to transmit.
- **rendered_message** (*dict*) – The rendered content of the message that has been personalized for this particular recipient.

classmethod enabled()
Validate settings to determine whether this channel can be enabled.

overrides_delivery_for_message(*message*)
Returns true if this channel specifically wants to handle this message, outside normal channel delivery rules.

For example, say you use a django transactional email channel, but with a default channel of braze. Then if the braze channel is configured with a campaign for a certain transactional message id specifically, it will claim that message via this method and end up delivering it via braze instead of the normal transactional django channel.

class edx_ace.**ChannelType**(*value*)
Bases: enum.Enum

All supported communication channels.

EMAIL = 'email'

PUSH = 'push'

class edx_ace.**Message**(*app_label*, *name*, *recipient*, *expiration_time=None*, *context=NOTHING*, *send_uuid=None*, *options=NOTHING*, *language=None*, *log_level=None*)
Bases: *edx_ace.serialization.MessageAttributeSerializationMixin*

A Message is the core piece of data that is passed into ACE. It captures the message, recipient, and all context needed to render the message for delivery.

Parameters

- **app_label** (*str*) – The name of the Django app that is sending this message. Used to look up the appropriate template during rendering. Required.
- **name** (*str*) – The name of this type of message. Used to look up the appropriate template during rendering. Required.
- **recipient** (*Recipient*) – The intended recipient of the message. Optional.

- **expiration_time** (datetime) – The date and time at which this message expires. After this time, the message should not be delivered. Optional.
- **context** (*dict*) – A dictionary to be supplied to the template at render time as the context.
- **send_uuid** (uuid.UUID) – The uuid.UUID assigned to this bulk-send of many messages.
- **language** (*str*) – The language the message should be rendered in. Optional.

default_context_value()

default_options_value()

generate_uuid()

get_message_specific_logger(*logger*)

Parameters **logger** (logging.Logger) – The logger to be adapted.

Returns: *MessageLoggingAdapter* that is specific to this message.

property log_id
The identity of this message for logging.

report(*key*, *value*)

report_basics()

property unique_name
A unique name for this message, used for logging and reporting.

Returns: str

class edx_ace.**MessageType**(*context=NOTHING*, *expiration_time=None*, *app_label=NOTHING*, *name=NOTHING*, *options=NOTHING*, *log_level=None*)

Bases: *edx_ace.serialization.MessageAttributeSerializationMixin*

A class representing a type of *Message*. An instance of a MessageType is used for each batch send of messages.

Parameters

- **context** (*dict*) – Context to be supplied to all messages sent in this batch of messages.
- **expiration_time** (datetime.datetime) – The time at which these messages expire.
- **app_label** (*str*) – Override the Django app that is used to resolve the template for rendering. Defaults to *APP_LABEL* or to the app that the message type was defined in.
- **name** (*str*) – Override the message name that is used to resolve the template for rendering. Defaults to *NAME* or to the name of the class.

APP_LABEL = None

NAME = None

default_app_label()
Get default app Label.

default_context_value()

default_name()
Return default class name.

default_options_value()

generate_uuid()

personalize(*recipient*, *language*, *user_context*)
Personalize this *MessageType* to a specific recipient, in order to send a specific message.

Parameters

- **recipient** (*Recipient*) – The intended recipient of the message. Optional.
- **language** (*str*) – The language the message should be rendered in. Optional.
- **user_context** (*dict*) – A dictionary containing recipient-specific context to be supplied to the template at render time.

Returns: A new *Message* that has been personalized to a specific recipient.

class edx_ace.**Policy**
Bases: object

A Policy allows an application to specify what *Channel* any specific *Message* shouldn't be sent over. Policies are one of the primary extension mechanisms for ACE, and are registered using the entrypoint openedx.ace.policy.

abstract check(*message*)
Validate the supplied *Message* against a specific delivery policy.

Parameters message (*Message*) – The message to run the policy against.

Returns: ***PolicyResult*** A *PolicyResult* that represents what channels the message should not be delivered over.

classmethod enabled()

class edx_ace.**PolicyResult**(*deny=NOTHING*)
Bases: object

Parameters deny (*set*) – A set of *ChannelType* values that should be excluded when sending a message.

check_set_of_channel_types(*attribute*, *set_value*)

class edx_ace.**Recipient**(*lms_user_id*, *email_address=None*)
Bases: *edx_ace.serialization.MessageAttributeSerializationMixin*

The target for a message.

Parameters

- **lms_user_id** (*int*) – The LMS user ID of the intended recipient.
- **email_address** (*str*) – The email address of the intended recipient. Optional.

class edx_ace.**RecipientResolver**
Bases: object

This class represents a pattern for separating the content of a message (the *MessageType*) from the selection of recipients (the *RecipientResolver*).

abstract send(*msg_type*, **args*, ***kwargs*)
send() a *Message* personalized from msg_type to all recipients selected by this *RecipientResolver*.

Parameters msg_type (*MessageType*) – An instantiated *MessageType* that describes the message batch to send.

edx_ace.**send**(*msg*)

Send a message to a recipient.

Calling this method will result in an attempt being made to deliver the provided message to the recipient. Depending on the configured policies, it may be transmitted to them over one or more channels (email, sms, push etc).

The message must have valid values for all required fields in order for it to be sent. Different channels have different requirements, so care must be taken to ensure that all of the needed information is present in the message before calling `ace.send()`.

Parameters **msg** (Message) – The message to send.

4.2 Sending Messages

The main entry point for sending messages with ACE.

Usage:

```
from edx_ace import ace
from edx_ace.messages import Message

msg = Message(
    name="test_message",
    app_label="my_app",
    recipient=Recipient(lms_user_id='123456', email='a_user@example.com'),
    language='en',
    context={
        'stuff': 'to personalize the message',
    }
)
ace.send(msg)
```

edx_ace.ace.**send**(*msg*)

Send a message to a recipient.

Calling this method will result in an attempt being made to deliver the provided message to the recipient. Depending on the configured policies, it may be transmitted to them over one or more channels (email, sms, push etc).

The message must have valid values for all required fields in order for it to be sent. Different channels have different requirements, so care must be taken to ensure that all of the needed information is present in the message before calling `ace.send()`.

Parameters **msg** (Message) – The message to send.

4.3 Delivery

edx_ace.channel exposes the ACE extension point needed to add new delivery *Channel* instances to an ACE application.

Developers wanting to add a new deliver channel should subclass *Channel*, and then add an entry to the `openedx.ace.channel` entrypoint in their `setup.py`.

class edx_ace.channel.**Channel**
Bases: `object`

Channels deliver messages to users that have already passed through the presentation and policy steps.

Examples include email messages, push notifications, or in-browser messages. Implementations of this abstract class should not require any parameters be passed into their constructor since they are instantiated.

channel_type must be a *ChannelType*.

channel_type = None

abstract deliver(*message*, *rendered_message*)
Transmit a rendered message to a recipient.

Parameters

- **message** (`Message`) – The message to transmit.
- **rendered_message** (*dict*) – The rendered content of the message that has been personalized for this particular recipient.

classmethod enabled()
Validate settings to determine whether this channel can be enabled.

overrides_delivery_for_message(*message*)
Returns true if this channel specifically wants to handle this message, outside normal channel delivery rules.

For example, say you use a django transactional email channel, but with a default channel of braze. Then if the braze channel is configured with a campaign for a certain transactional message id specifically, it will claim that message via this method and end up delivering it via braze instead of the normal transactional django channel.

class edx_ace.channel.**ChannelMap**(*channels_list*)
Bases: `object`

A class that represents a channel map, usually as described in Django settings and *setup.py* files.

get_channel_by_name(*channel_type*, *channel_name*)
Gets a registered a channel by its name and type.

Raises `KeyError` – If either of the channel or its type are not registered.

Returns The channel object.

Return type *Channel*

get_default_channel(*channel_type*)
Returns the first registered channel by type.

Raises *UnsupportedChannelError* – If there's no channel that matched the request.

Parameters **channel_type** (`ChannelType`) – The channel type.

register_channel(*channel*, *channel_name*)
Registers a channel in the channel map.

Parameters

- **channel** (Channel) – The channel to register.
- **channel_name** (*str*) – The channel name, as stated in the *setup.py* file.

class edx_ace.channel.**ChannelType**(*value*)
Bases: enum.Enum

All supported communication channels.

EMAIL = 'email'

PUSH = 'push'

edx_ace.channel.**channels**()
Gathers all available channels.

Note that this function loads all available channels from entry points. It expects the Django setting ACE_ENABLED_CHANNELS to be a list of plugin names that should be enabled. Only one plugin per channel type should appear in that list.

Raises ValueError – If multiple plugins are enabled for the same channel type.

Returns A mapping of channel types to instances of channel objects that can be used to deliver messages.

Return type *ChannelMap*

edx_ace.channel.**get_channel_for_message**(*channel_type*, *message*)
Based on available *channels()* returns a single channels for a message.

Raises *UnsupportedChannelError* – If there's no channel matches the request.

Returns The selected channel object.

Return type *Channel*

4.3.1 edx_ace.channel.sailthru

edx_ace.channel.sailthru implements a SailThru-based email delivery channel for ACE.

class edx_ace.channel.sailthru.**RecoverableErrorCodes**(*value*)
Bases: enum.IntEnum

These error codes are present in responses to requests that can (and should) be retried after waiting for a bit.

INTERNAL_ERROR = 9
Something's gone wrong on Sailthru's end. Your request was probably not saved - try waiting a moment and trying again.

RATE_LIMIT = 43
You have exceeded the limit of requests per minute for the given type (GET or POST) and endpoint. For limit details, see the Rate Limiting section on the API Technical Details page.

Type Too many [type] requests this minute to /[endpoint] API

class edx_ace.channel.sailthru.**ResponseHeaders**(*value*)
Bases: enum.Enum

These are special headers returned in responses from the Sailthru REST API.

RATE_LIMIT_REMAINING = 'X-Rate-Limit-Remaining'

RATE_LIMIT_RESET = 'X-Rate-Limit-Reset'

class edx_ace.channel.sailthru.**SailthruEmailChannel**

Bases: *edx_ace.channel.Channel*

An email channel for delivering messages to users using Sailthru.

This channel makes use of the Sailthru REST API to send messages. It is designed for "at most once" delivery of messages. It will make a reasonable attempt to deliver the message and give up if it can't. It also only confirms that Sailthru has received the request to send the email, it doesn't actually confirm that it made it to the recipient.

The integration with Sailthru requires several Django settings to be defined.

Example

Sample settings:

```
.. settings_start
ACE_CHANNEL_SAILTHRU_DEBUG = False
ACE_CHANNEL_SAILTHRU_TEMPLATE_NAME = "Some template name"
ACE_CHANNEL_SAILTHRU_API_KEY = "1234567890"
ACE_CHANNEL_SAILTHRU_API_SECRET = "this is secret"
.. settings_end
```

The named template in Sailthru should be minimal, most of the rendering happens within ACE. The "From Name" field should be set to `{{ace_template_from_name}}`. The "Subject" field should be set to `{{ace_template_subject}}`. The "Code" for the template should be:

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
    <head>
        {{ace_template_head_html}}
    </head>
    <body>
        {body_html = replace(ace_template_body_html, '{view_url}', view_url)}
        {body_html = replace(body_html, '{optout_confirm_url}', optout_confirm_url)}
        {body_html = replace(body_html, '{forward_url}', forward_url)}
        {body_html = replace(body_html, '{beacon_src}', beacon_src)}
        {body_html}
        <span id="sailthru-message-id" style="display: none;">{message_id()}</span>
        <a href="{optout_confirm_url}" style="display: none;"></a>
    </body>
</html>
```

property action_links

This method is now deprecated in favor of get_action_links, but will continue to work for the time being as it calls get_action_links under the hood.

channel_type = 'email'

deliver(*message*, *rendered_message*)

Transmit a rendered message to a recipient.

Parameters

- **message** (Message) – The message to transmit.
- **rendered_message** (*dict*) – The rendered content of the message that has been personalized for this particular recipient.

classmethod enabled()
Returns: True iff all required settings are not empty and the Sailthru client library is installed.

get_action_links(***kwargs*)
Provides list of action links, called by templates directly. Supported kwargs:

omit_unsubscribe_link (bool): Removes the unsubscribe link from the email. DO NOT send emails with no unsubscribe link unless you are sure it will not violate the CANSPAM act.

property tracker_image_sources
Provides list of trackers, called by templates directly

4.3.2 edx_ace.channel.django_email

edx_ace.channel.django_email implements a Django *send_mail()* email delivery channel for ACE.

class edx_ace.channel.django_email.**DjangoEmailChannel**
Bases: edx_ace.channel.mixins.EmailChannelMixin, *edx_ace.channel.Channel*

A *send_mail()* channel for edX ACE.

This is both useful for providing an alternative to Sailthru and to debug ACE mail by inspecting *django.core.mail.outbox*.

Example

Sample settings:

```
.. settings_start
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'localhost'
DEFAULT_FROM_EMAIL = 'hello@example.org'

ACE_CHANNEL_DEFAULT_EMAIL = 'sailthru_email'
ACE_CHANNEL_TRANSACTIONAL_EMAIL = 'django_email'

ACE_ENABLED_CHANNELS = [
    'sailthru_email',
    'django_email',
]
.. settings_end
```

deliver(*message*, *rendered_message*)
Transmit a rendered message to a recipient.

Parameters

- **message** (Message) – The message to transmit.
- **rendered_message** (*dict*) – The rendered content of the message that has been personalized for this particular recipient.

classmethod enabled()
Returns: True always!

4.4 Exceptions

edx_ace.errors exposes all exceptions that are specific to ACE.

exception edx_ace.errors.**ChannelError**
Bases: Exception

Indicates something went wrong in a delivery channel.

exception edx_ace.errors.**FatalChannelDeliveryError**
Bases: *edx_ace.errors.ChannelError*

A fatal error occurred during channel delivery. Do not retry.

exception edx_ace.errors.**InvalidMessageError**
Bases: Exception

Encountered a message that cannot be sent due to missing or inconsistent information.

exception edx_ace.errors.**RecoverableChannelDeliveryError**(*message*, *next_attempt_time*)
Bases: *edx_ace.errors.ChannelError*

An error occurred during channel delivery that is non-fatal. The caller should re-attempt at a later time.

exception edx_ace.errors.**UnsupportedChannelError**
Bases: *edx_ace.errors.ChannelError*

Raised when an attempt is made to process a message for an unsupported channel.

4.5 Messages

edx_ace.message contains the core *Message* and *MessageType* classes, which allow specification of the content to be delivered by ACE.

class edx_ace.message.**Message**(*app_label*, *name*, *recipient*, *expiration_time=None*, *context=NOTHING*, *send_uuid=None*, *options=NOTHING*, *language=None*, *log_level=None*)
Bases: *edx_ace.serialization.MessageAttributeSerializationMixin*

A Message is the core piece of data that is passed into ACE. It captures the message, recipient, and all context needed to render the message for delivery.

Parameters

- **app_label** (*str*) – The name of the Django app that is sending this message. Used to look up the appropriate template during rendering. Required.
- **name** (*str*) – The name of this type of message. Used to look up the appropriate template during rendering. Required.
- **recipient** (*Recipient*) – The intended recipient of the message. Optional.
- **expiration_time** (datetime) – The date and time at which this message expires. After this time, the message should not be delivered. Optional.
- **context** (*dict*) – A dictionary to be supplied to the template at render time as the context.
- **send_uuid** (uuid.UUID) – The uuid.UUID assigned to this bulk-send of many messages.
- **language** (*str*) – The language the message should be rendered in. Optional.

default_context_value()

default_options_value()

generate_uuid()

get_message_specific_logger(*logger*)

Parameters **logger** (logging.Logger) – The logger to be adapted.

Returns: *MessageLoggingAdapter* that is specific to this message.

property log_id
The identity of this message for logging.

report(*key*, *value*)

report_basics()

property unique_name
A unique name for this message, used for logging and reporting.

Returns: str

class edx_ace.message.**MessageLoggingAdapter**(*logger*, *extra*)
Bases: logging.LoggerAdapter

A logging.LoggingAdapter that prefixes log items with a message *log_id*.ABCMeta

Expects a message key in its extra argument which should contain the *Message* being logged for.

debug(*msg*, **args*, ***kwargs*)
Delegate a debug call to the underlying logger.

process(*msg*, *kwargs*)
Process the logging message and keyword arguments passed in to a logging call to insert contextual information. You can either manipulate the message itself, the keyword args or both. Return the message and kwargs modified (or not) to suit your needs.

Normally, you'll only need to override this one method in a LoggerAdapter subclass for your specific needs.

class edx_ace.message.**MessageType**(*context=NOTHING*, *expiration_time=None*, *app_label=NOTHING*, *name=NOTHING*, *options=NOTHING*, *log_level=None*)
Bases: *edx_ace.serialization.MessageAttributeSerializationMixin*

A class representing a type of *Message*. An instance of a MessageType is used for each batch send of messages.

Parameters

- **context** (*dict*) – Context to be supplied to all messages sent in this batch of messages.
- **expiration_time** (datetime.datetime) – The time at which these messages expire.
- **app_label** (*str*) – Override the Django app that is used to resolve the template for rendering. Defaults to *APP_LABEL* or to the app that the message type was defined in.
- **name** (*str*) – Override the message name that is used to resolve the template for rendering. Defaults to *NAME* or to the name of the class.

APP_LABEL = None

NAME = None

default_app_label()
Get default app Label.

default_context_value()

default_name()
Return default class name.

default_options_value()

generate_uuid()

personalize(*recipient*, *language*, *user_context*)
Personalize this *MessageType* to a specific recipient, in order to send a specific message.

Parameters

- **recipient** (*Recipient*) – The intended recipient of the message. Optional.
- **language** (*str*) – The language the message should be rendered in. Optional.
- **user_context** (*dict*) – A dictionary containing recipient-specific context to be supplied to the template at render time.

Returns: A new *Message* that has been personalized to a specific recipient.

4.6 Monitoring

edx_ace.monitoring exposes functions that are useful for reporting ACE message delivery stats to monitoring services.

edx_ace.monitoring.**report**(*key*, *value*)

edx_ace.monitoring.**report_to_newrelic**(*key*, *value*)

4.7 Delivery Policy

edx_ace.policy contains all classes relating to message policies.

These policies manage which messages should be sent over which channels, and are a point of pluggability in ACE.

class edx_ace.policy.**Policy**
Bases: object

A Policy allows an application to specify what *Channel* any specific *Message* shouldn't be sent over. Policies are one of the primary extension mechanisms for ACE, and are registered using the entrypoint openedx.ace.policy.

abstract check(*message*)
Validate the supplied *Message* against a specific delivery policy.

Parameters message (*Message*) – The message to run the policy against.

Returns: ***PolicyResult*** A *PolicyResult* that represents what channels the message should not be delivered over.

classmethod enabled()

class edx_ace.policy.**PolicyResult**(*deny=NOTHING*)
Bases: object

Parameters deny (*set*) – A set of *ChannelType* values that should be excluded when sending a message.

check_set_of_channel_types(*attribute*, *set_value*)

edx_ace.policy.**channels_for**(*message*)

Parameters **message** (*Message*) – The message apply policies to.

Returns: set A set of *ChannelType* values that are allowed by all policies applied to the message.

edx_ace.policy.**policies**()

4.8 Message Presentation

edx_ace.renderers contains the classes used by ACE to render messages for particular types of delivery channels. Each ChannelType has a distinct subclass of *AbstractRenderer* associated with it, which is used to render messages for all Channel subclasses of that type.

class edx_ace.renderers.**AbstractRenderer**

Bases: object

Base class for message renderers.

A message renderer is responsible for taking one, or more, templates, and context, and outputting a rendered message for a specific message channel (e.g. email, SMS, push notification).

get_template_for_message(*channel*, *message*, *filename*)

Parameters

- **message** (Message) – The message being rendered.
- **filename** (*str*) – The basename of the template file to look up.

Returns The full template path to the template to render.

render(*channel*, *message*)

Renders the given message.

Parameters

- **channel** (Channel) – The channel to render the message for.
- **message** – The message being rendered.

rendered_message_cls = None

class edx_ace.renderers.**EmailRenderer**

Bases: *edx_ace.renderers.AbstractRenderer*

A renderer for *ChannelType.EMAIL* channels.

rendered_message_cls

alias of *edx_ace.renderers.RenderedEmail*

class edx_ace.renderers.**RenderedEmail**(*from_name*, *subject*, *body_html*, *head_html*, *body*)

Bases: object

Encapsulates all values needed to send a *Message* over an *ChannelType.EMAIL*.

An internal module that manages the presentation/rendering step of the ACE pipeline.

edx_ace.presentation.**render**(*channel*, *message*)

Returns the rendered content for the given channel and message.

4.9 Message Recipients

edx_ace.recipient contains *Recipient*, which captures all targeting information needed to deliver a message to some user.

class edx_ace.recipient.**Recipient**(*lms_user_id*, *email_address=None*)
Bases: *edx_ace.serialization.MessageAttributeSerializationMixin*

The target for a message.

Parameters

- **lms_user_id** (*int*) – The LMS user ID of the intended recipient.
- **email_address** (*str*) – The email address of the intended recipient. Optional.

edx_ace.recipient_resolver contains the *RecipientResolver*, which facilitates a design pattern that separates message content from recipient lists.

class edx_ace.recipient_resolver.**RecipientResolver**
Bases: object

This class represents a pattern for separating the content of a message (the *MessageType*) from the selection of recipients (the *RecipientResolver*).

abstract send(*msg_type*, **args*, ***kwargs*)
send() a *Message* personalized from msg_type to all recipients selected by this *RecipientResolver*.

Parameters **msg_type** (*MessageType*) – An instantiated *MessageType* that describes the message batch to send.

4.10 Serialization

edx_ace.serialization contains *MessageAttributeSerializationMixin*, which allows messages to be round-tripped through JSON, and *MessageEncoder*, which actually performs the JSON encoding.

class edx_ace.serialization.**MessageAttributeSerializationMixin**
Bases: object

This mixin allows an object to be serialized to (and deserialized from) a JSON string.

__str__() and *from_string()* function as inverses, and are the primary point of interaction with this mixin by outside clients.

to_json() is used to recursively convert the object to a python dictionary that can then be encoded to a JSON string.

classmethod from_string(*string_value*)
Decode a JSON-encoded string representation of this type.

Parameters **string_value** (*str*) – The JSON string to decode.

Returns An instance of this class.

to_json()

Returns: dict a python dictionary containing all serializable fields of this object, suitable for JSON-encoding.

class edx_ace.serialization.**MessageEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.encoder.JSONEncoder`

Custom Message Encoder.

default(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

4.11 Utils

4.11.1 edx_ace.utils

4.11.2 edx_ace.utils.date

edx_ace.utils.date contains utility functions used for serializing and deserializing dates. It is intended for internal ACE use.

edx_ace.utils.date.**deserialize**(*timestamp_iso8601_str*)

Deserialize a datetime object from an ISO8601 formatted string.

Parameters **timestamp_iso8601_str** (*basestring*) – A timestamp as an ISO8601 formatted string.

Returns A timezone-aware python datetime object.

Return type datetime

edx_ace.utils.date.**get_current_time**()

The current time in the UTC timezone as a timezone-aware datetime object.

edx_ace.utils.date.**serialize**(*timestamp_obj*)

Serialize a datetime object to an ISO8601 formatted string.

Parameters **timestamp_obj** (*datetime*) – The timestamp to serialize.

Returns A string representation of the timestamp in ISO8601 format.

Return type basestring

4.11.3 edx_ace.utils.once

edx_ace.utils.once provides the ability to create a module-level function that caches it's result after the first call (this can be used for lazy-loading expensive computations).

edx_ace.utils.once.**once**(*func*)

Decorates a function that will be called exactly once.

After the function is called once, its result is stored in memory and immediately returned to subsequent callers instead of calling the decorated function again.

Examples

An incrementing value:

```
_counter = 0

@once
def get_counter():
    global _counter
    _counter += 1
    return _counter

def get_counter_updating():
    global _counter
    _counter += 1
    return _counter

print(get_counter())  # This will print "0"
print(get_counter_updating()) # This will print "1"
print(get_counter())  # This will also print "0"
print(get_counter_updating()) # This will print "2"
```

Lazy loading:

```
@once
def load_config():
    with open('config.json', 'r') as cfg_file:
        return json.load(cfg_file)

cfg = load_config()  # This will do the relatively expensive operation to
                     # read the file from disk.
cfg2 = load_config() # This call will not reload the file from disk, it
                     # will use the value returned by the first invocation
                     # of this function.
```

Parameters **func** (*callable*) – The function that should be called exactly once.

Returns The wrapped function.

Return type callable

4.11.4 edx_ace.utils.plugins

edx_ace.utils.plugins contains utility functions used to make working with the ACE plugin system easier. These are intended for internal use by ACE.

edx_ace.utils.plugins.**check_plugin**(*extension*, *namespace*, *names=None*)
Check the extension to see if it's enabled.

Parameters

- **extension** (*stevedore.extension.Extension*) – The extension to check.
- **namespace** (*basestring*) – The namespace that the extension was loaded from.
- **names** (*list*) – A whitelist of extensions that should be checked.

Returns Whether or not this extension is enabled and should be used.

Return type bool

edx_ace.utils.plugins.**get_manager**(*namespace*, *names=None*)
Get the stevedore extension manager for this namespace.

Parameters

- **namespace** (*basestring*) – The entry point namespace to load plugins for.
- **names** (*list*) – A list of names to load. If this is `None` then all extension will be loaded from this namespace.

Returns Extension manager with all extensions instantiated.

Return type stevedore.enabled.EnabledExtensionManager

edx_ace.utils.plugins.**get_plugins**(*namespace*, *names=None*)
Get all extensions for this namespace and list of names.

Parameters

- **namespace** (*basestring*) – The entry point namespace to load plugins for.
- **names** (*list*) – A list of names to load. If this is `None` then all extension will be loaded from this namespace.

Returns A list of extensions.

Return type list

4.12 Testing

4.12.1 edx_ace.test_utils

Test utilities.

Since pytest discourages putting __init__.py into test directory (i.e. making tests a package) one cannot import from anywhere under tests folder. However, some utility classes/methods might be useful in multiple test modules (i.e. factoryboy factories, base test classes). So this package is the place to put them.

class edx_ace.test_utils.**StubPolicy**(*deny_value*)
Bases: *edx_ace.policy.Policy*

Short term policy.

check(*message*)
Validate the supplied *Message* against a specific delivery policy.

Parameters **message** (*Message*) – The message to run the policy against.

Returns: ***PolicyResult*** A *PolicyResult* that represents what channels the message should not be delivered over.

edx_ace.test_utils.**patch_policies**(*test_case*, *policies*)
Set active policies for the duration of a test.

Parameters

- **test_case** (unittest.TestCase) – The test case that is running
- **policies** – The set of active policies to return from *edx_ace.policy.policies()*

4.13 Internal

4.13.1 Delivery

Functions for delivering ACE messages.

This is an internal interface used by *ace.send()*.

edx_ace.delivery.**deliver**(*channel*, *rendered_message*, *message*)
Deliver a message via a particular channel.

Parameters

- **channel** (Channel) – The channel to deliver the message over.
- **rendered_message** (*object*) – Each attribute of this object contains rendered content.
- **message** (Message) – The message that is being sent.

Raises **.UnsupportedChannelError** – If no channel of the requested channel type is available.

CHAPTER

FIVE

CHANGE LOG

5.1 Unreleased

5.2 [1.5.0] - 2022-02-15

- Added support for Django40
- Removed support for Django22, 30 and 31

5.3 [1.4.1] - 2021-12-06

- Adds in the ability to override frequency caps for Braze emails. Can be accessed via Message options using the key `override_frequency_capping`. All emails containing the `transactional` Message option will also override frequency caps.

5.4 [1.4.0] - 2021-11-08

- Deprecate the action_links property
- Add a get_action_links method and template tag to allow passing arguments to action links

5.5 [1.3.1] - 2021-08-17

- Adjust name `handles_delivery_for_message` to `overrides_delivery_for_message`

5.6 [1.3.0] - 2021-08-16

- New channel method `handles_delivery_for_message` for allowing a default channel to claim a message, even if it would normally be delivered to the configured transactional channel.
- Braze: Will handle any message defined in `ACE_CHANNEL_BRAZE_CAMPAIGNS` (using the above new feature) to steal campaign messages from the transactional channel as needed.

5.7 [1.2.0] - 2021-07-16

- Added support for django 3.2

5.8 [1.1.1] - 2021-07-09

- Removed upper constraint from Django

5.9 [1.1.0] - 2021-03-26

- Braze: Add ACE_CHANNEL_BRAZE_FROM_EMAIL setting to override the normal from address
- Sailthru: Remove Braze rollout waffle flag

5.10 [1.0.1] - 2021-03-15

- Braze: Add an unsubscribe action link
- Braze: Don't ask Braze to inline css, as ACE templates already have inline css

5.11 [1.0.0] - 2021-03-11

- BREAKING: Recipient objects now take *lms_user_id* instead of *username*
- New *braze_email* backend, needing the following new configuration:
  - ACE_CHANNEL_BRAZE_API_KEY
  - ACE_CHANNEL_BRAZE_APP_ID
  - ACE_CHANNEL_BRAZE_REST_ENDPOINT (like *rest.iad-01.braze.com*)
  - ACE_CHANNEL_BRAZE_CAMPAIGNS (an optional dictionary of ACE message names to Braze campaign identifiers)

5.12 [0.1.18] - 2020-11-19

- Updated he travis-badge in README.rst to point to travis-ci.com

5.13 [0.1.17] - 2020-10-19

- Use IntEnum to avoid silent failure in value comparisons

5.14 [0.1.16] - 2020-10-17

- Fixed Enum usage for Python 3.8 to avoid TypeError when comparing values

5.15 [0.1.15] - 2020-03-11

- Added support for Python 3.8
- Removed support for Django 2.0 and 2.1

5.16 [0.1.14] - 2020-03-11

- Fix trivial warning from deprecated use of attr library.

5.17 [0.1.13] - 2019-12-06

- Django22 Support.

5.18 [0.1.12] - 2019-10-16

- Reply_to field added in emails.

5.19 [0.1.10] - 2018-11-01

- Django lazy text translations are handled properly.

5.20 [0.1.9] - 2018-07-13

- Updated delivery logging

5.21 [0.1.0] - 2017-08-08

5.21.1 Added

- First release on PyPI.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

F

G

I

L

M

N

O

P

R

S

T

U