Vumi Javascript Sandbox Toolkit Documentation

Release 0.2.18

Praekelt Foundation

June 17, 2016

This is the sandbox toolkit for making writing Javascript applications for Vumi Go's Javascript sandbox.

Interaction Machine

class **ApiError**(*message*)

Thrown when an error occurs when the sandbox api returns a failure response (when `success` is `false`).

Arguments

- **reply** (`object`) – the failure reply given by the api.

class **IMErrorEvent**(*im*)

Emitted when an error occurs during a run of the im.

Arguments

- **im** (`InteractionMachine`) – the interaction machine emitting the event.

- **error** (`InteractionMachine`) – the error that occured.

The event type is `im:error`.

class **IMEvent**()

An event relating to an interaction machine.

Arguments

- **name** (`string`) – the event type's name.

- **im** (`InteractionMachine`) – the interaction machine associated to the event

class **IMShutdownEvent**(*im*)

Occurs when the im is about to shutdown.

Arguments

- **im** (`InteractionMachine`) – the interaction machine emitting the event.

The event type is `im:shutdown`.

class **InboundEventEvent**(*im*, *cmd*)

Emitted when an message status event is received. Typically, this is either an acknowledgement or a delivery report for an outbound message that was sent from the sandbox application.

Arguments

- **im** (`InteractionMachine`) – the interaction machine emitting the event.

- **cmd** (`object`) – the API request cmd containing the inbound user message.

The event type is `inbound_event`.

class **InboundMessageEvent**(*im*, *cmd*)

Emitted when an inbound user message is received by the interaction machine.

> **Arguments**
>
> - **im** (`InteractionMachine`) – the interaction machine firing the event.
>
> - **cmd** (`object`) – the API request cmd containing the inbound user message.

class InteractionMachine (*api*, *app*)

> **Arguments**
>
> - **api** (`SandboxAPI`) – a sandbox API providing access to external resources and inbound messages.
>
> - **app** (`App`) – a collection of states defining an application.

A controller that handles inbound messages and fires events and handles state transitions in response to those messages. In addition, it serves as a bridge between a `App()` (i.e. set of states defining an application) and resources provided by the sandbox API.

static api
A reference to the sandbox API.

static api_request (*cmd_name*, *cmd*)
Raw request to the sandbox API.

> **Arguments**
>
> - **cmd_name** (`string`) – name of the API request to make.
>
> - **cmd** (`object`) – API request data.

Returns a promise fulfilled with the response to the API request, or rejected with a `ApiError()` if a failure response was given.

static app
A reference to the `App()`.

static attach ()
Attaches the im to the given api and app. The sandbox API's event handlers are set to emit the respective events on the interaction machine, then terminate the sandbox once their listeners are done.

static config
A `IMConfig()` instance for the IM's config data. Available when setup is complete (see `InteractionMachine.setup()`).

static contacts
A default `ContactStore()` instance for managing contacts. Available when setup is complete (see `InteractionMachine.setup()`)

static create_and_set_state (*state*)
Creates new state using the given `StateData()` or state name, then sets it as the `InteractionMachine()`'s current state.

> **Arguments**
>
> - **state** (`object, string, or StateData`) – The state to create and set

static create_state (*state*)
Creates a new state using the given `StateData()` or state name.

> **Arguments**
>
> - **state** (`object, string, or StateData`) – The state to create

static done ()
Saves the user, then terminates the sandbox instance.

static **enter_state** (*state*)

> Creates the given state, sets it as the current state, then emits a :class:StateEnterEvent' (on :class:InteractionMachine, then the new state).
>
> > **Arguments**
> >
> > > • **state** (*object, string, or StateData*) – the state to enter

static **exit_state** ()

> Emits a :class:StateExitEvent' (on :class:InteractionMachine, then the state), then resets the interaction machine's state to `null`. If the interaction machine is not on a state, this method is a no-op.

static **fetch_translation** (*lang*)

> Retrieve a [*Translator()*](#) instance corresponding to the translations for the given language. Returns a promise that will be fulfilled with the retrieved translator.
>
> > **Arguments**
> >
> > > • **lang** (*string*) – two letter language code (e.g. `sw`, `en`).
>
> Translations are retrieved from the sandbox configuration resource by looking up keys named `translation.<language-code>`.

static **groups**

> A default [*GroupStore()*](#) instance for managing groups. Available when setup is complete (see [*InteractionMachine.setup()*](#))

static **handle_message** (*msg*)

> Delegates to its subordinate message handlers to handle an inbound message based on the message's session event type. The fallback message handler is defined by `InteractionMachine.handle_message.fallback()`, which by default is an alias for [*InteractionMachine.handle_message.resume()*](#).
>
> If the user is not currently in a session (which happens for new users and users that have reached an [*EndState()*](#) in a previous session), and the message does not have a `session_event` (as is the case for session-less messages such as smses or tweets), we assume the user is starting a new session.
>
> > **Arguments**
> >
> > > • **msg** (*object*) – the received inbound message. *

static **log**

> A [*Logger()*](#) instance for logging message in the sandbox.

static **metrics**

> A default [*MetricStore()*](#) instance for emitting metrics. Available when setup is complete (see [*InteractionMachine.setup()*](#))

static **msg**

> The message command currently being processed. Available when setup is complete (see [*InteractionMachine.setup()*](#)).

static **next_state**

> The next state that the user should move to once the user's input has been processed.

static **on "inbound_message"** (*event*)

> Invoked an inbound user message, triggering state transitions and events as necessary.
>
> > **Arguments**
> >
> > > • **event** ([InboundMessageEvent](#)) – the fired event.
>
> **The steps performed by this method are roughly:**

- Set up the IM (see `InteractionMachine.setup()`)

- If the user is currently in a state (from a previous IM run), switch to this state.

- Otherwise, this is a new user, so switch to the IM's configured start state

- Handle the message based on its session event type (see `InteractionMachine.handle_message()`).

static on "unknown_command" (*event*)

Invoked by a `UnknownCommandEvent()` event when a command without a handler is received (see `UnknownCommandEvent()`). Logs an error.

> **Arguments**
>
> - **event** (`UnknownCommandEvent`) – the fired event.

static outbound

A `OutboundHelper()` for sending out messages. Available when setup is complete (see `InteractionMachine.setup()`)

static reply (*msg*)

Send a response from the current state to the user.

Returns a promise which is fulfilled once the response has been sent.

static resume (*state*)

Creates the given state, sets it as the current state, then emits a :class:StateResumeEvent' (on :class:InteractionMachine, then the new state).

If the created state has a different name to the requested state, a `StateEnterEvent()` is emitted instead. This happens, for example, if the requested state does not exist (see `AppStates.create()`).

> **Arguments**
>
> - **state** (`object, string, or StateData`) – the state to resume

static sandbox_config

A `SandboxConfig()` instance for accessing the sandbox's config data. Available when setup is complete (see `InteractionMachine.setup()`).

static set_state (*state*)

Sets the given `State()` as the `InteractionMachine()`'s current state.

> **Arguments**
>
> - **state** (`State`) – The state set as the current state

static setup (*msg*[, *opts*])

Sets up the interaction machine using the given message.

> **Arguments**
>
> - **msg** (`object`) – the received message to be used to set up the interaction machine.
>
> - **opts.reset** (`boolean`) – whether to reset the user's data, or load them from the kv store

The IM sets up its attributes in the following order:

- sanbox config

- im config

- metric store

- user

- app

Finally, the user's `creation_event` is emitted, then a *SetupEvent()* is emitted for the interaction machine. A promise is returned, which will be fulfilled once all event listeners are done.

static **state**
> The current *State()* object. Updated whenever a new state is entered via a call to *InteractionMachine.switch_state()*,

static **switch_state**(*dest*)
> Switches the IM from its current state to the given destination state. Returns a promise fulfilled once the switch has completed.

> **Arguments**

> - **dest** (*object, string, or StateData*) – the destination state's name or state data

> **The following steps are taken:**

> - The current state is exited (see *InteractionMachine.exit_state()*)

> - The destination state is enter (see *InteractionMachine.enter_state()*)

static **user**
> A *User()* instance for the current user. Available when setup is complete (see *InteractionMachine.setup()*).

InteractionMachine.handle_message.**close**(*msg*)
> Invoked when an inbound message is received with a `close` session event type. Emits a *SessionCloseEvent()* on the interaction machine and waits for its listeners to complete their work.

> **Arguments**

> - **msg** (*object*) – the received inbound message.

InteractionMachine.handle_message.**new**(*msg*)
> Invoked when an inbound message is received with a `new` session event type.

> **Arguments**

> - **msg** (*object*) – the received inbound message.

> Does roughly the following:

> •Emits a *SessionNewEvent()* on the interaction machine and waits for its listeners to complete their work

> •Sends a reply from the current state.

InteractionMachine.handle_message.**resume**(*msg*)
> Invoked when an inbound message is received with a `resume` session event type.

> **Arguments**

> - **msg** (*object*) – the received inbound message.

> Does roughly the following:

> •Emits a *SessionResumeEvent()* on the interaction machine and waits for its listeners to complete their work

- If the message contains usable content, give the content to the state (which fires a *StateInputEvent()*).

- Send a reply from the current state.

class ReplyEvent (*im*)

Emitted after the interaction machine sends a reply to a message sent in by the user.

> **Arguments**
>
> - **im** (*InteractionMachine*) – the interaction machine emitting the event.
>
> - **content** (*string*) – the content of the reply
>
> - **continue_session** (*bool*) – `true` if the reply did not end the session, `false` if the reply ended the session.

The event type is `reply`.

class SessionCloseEvent (*im*, *user_terminated*)

Emitted when a user session ends.

> **Arguments**
>
> - **im** (*InteractionMachine*) – the interaction machine emitting the event.
>
> - **user_terminated** (*boolean*) – true if the session was terminated by the user (including when the user session times out) and false if the session was closed explicitly by the sandbox application.

The event type is `session:close`.

class SessionNewEvent (*im*)

Emitted when a new user session starts.

> **Arguments**
>
> - **im** (*InteractionMachine*) – the interaction machine emitting the event.

The event type is `session:new`.

class SessionResumeEvent (*im*)

Emitted when a new user message arrives for an existing user session.

> **Arguments**
>
> - **im** (*InteractionMachine*) – the interaction machine emitting the event.

The event type is `session:resume`.

class UnknownCommandEvent (*im*, *cmd*)

Emitted when a command without a handler is received.

> **Arguments**
>
> - **im** (*InteractionMachine*) – the interaction machine emitting the event.
>
> - **cmd** (*object*) – the API request that no command handler was found for.

The event type is `unknown_command`.

interact (*api*, *f*)

If `api` is defined, create an *InteractionMachine()* with the *App()* returned by `f`. Otherwise do nothing.

If `f` is an *App()* subclass, `new f()` is used to construct the application instance instead.

> **Arguments**

- **api** (*SandboxAPI*) – a sandbox API providing access to external resources and inbound messages
- **f** (*function*) – a function that returns an *App()* instance or an *App()* class.

Returns the *InteractionMachine()* created or `null` if no *InteractionMachine()* was created.

Usually the return value is ignored since creating an *InteractionMachine()* attachs it to the `api`.

App

class **App** (*start_state_name*[, *opts*])

The main component defining a sandbox application. To be subclassed and given application specific states and logic.

> **Arguments**
>
> - **start_state_name** (*string*) – name of the initial state. New users will enter this state when they first interact with the sandbox application.
>
> - **opts.AppStates** (`AppStates`) – Optional subclass of `AppStates()` to be used for creating and managing states.
>
> - **opts.events** (*object*) – Optional event name-listener mappings to bind. For example:

```
{
    'app:error': function(e) {
        console.log(e);
    },
    'im.user user:new': function(e) {
        console.log(e);
    }
}
```

static **$**

A `LazyTranslator()` instance that can be used throughout the app to for internationalization using gettext. For example, this would send 'Hello, goodbye!' in the user's language:

```
self.states.add('states:start', function(name) {
    return new EndState(name, {text: self.$('Hello, goodbye!')});
});
```

static **exit** ()

Invoked when the interaction machine has emitted an `IMShutdownEvent()`, which occurs after the interaction machine has finished processing the inbound message and has sent out a reply (if relevant). Intended to be overriden and used as a 'teardown' hook. May return a promise.

static **init** ()

Invoked just after setup has completed, and just before 'setup' event is fired to provide subclasses with a setup hook. May return a promise.

static **remove** (*name*)

Removes an added state or state creator.

> **Arguments**

> • **name** (*string*) – name of the state or state creator

class **AppError** (*app*, *message*)
 Thrown when an app-related error occurs.

> **Arguments**
>
>> • **app** (App) – the app related to the error.
>>
>> • **message** (*string*) – the error message.

class **AppErrorEvent** (*app*, *error*)
 Emitted when an error is handled by the app, in case other entities want to know about the handled error.

> **Arguments**
>
>> • **app** (App) – the app emitting the event.
>>
>> • **error** (InteractionMachine) – the error that occured.
>
> The event type is app:error.

class **AppEvent** (*name*, *app*)
 An event relating to an app.

> **Arguments**
>
>> • **name** (*string*) – the name of the event
>>
>> • **app** (App) – the app emitting the event.

class **AppStateError** (*app*, *message*)
 Thrown when an error occurs creating or accessing a state in an app.

> **Arguments**
>
>> • **app** (App) – the app related to the error.
>>
>> • **message** (*string*) – the error message.

class **AppStates** (*app*)
 A set of states for a sandbox application. States may be either statically created via add.state, dynamically loaded via add.creator (or via add for either), or completely dynamically defined by overriding create.

> **Arguments**
>
>> • **app** (App) – the application associated with this set of states.

 static **add** (*state*)
  Adds an already created state by delegating to *AppStates.add.state()*.

> **Arguments**
>
>> • **state** (State) – the state to add

 static **add** (*name*, *creator*)
  Adds a state creator by delegating to *AppStates.add.creator()*.

> **Arguments**
>
>> • **state** (State) – the state to add

 static **create** (*name*, *opts*)
  Creates the given state represented by the given state name by delegating to the associated state creator.

> **Arguments**
>
>> • **name** (*string*) – the name of the state to create.

- **opts** (*object*) – Options for the state creator to use. Optional.

If no creator is found for the requested state name, we create a start state instead.

This function returns a promise.

It may be overridden by *AppStates()* subclasses that wish to provide a completely dynamic set of states.

static **init**()

Invoked just after setup has completed, and just before 'setup' event is fired to provide subclasses with a setup hook. May return a promise.

AppStates.add.**creator**(*name*, *creator*)

Adds a state creator. Invoked by *AppStates.create()*, or throws an error if a creator is already registered under the given state name.

Arguments

- **state_name** (*string*) – name of the state

- **creator** (*function*) –

    A function func(state_name) for creating the state. This function should take the state name should return a state object either directly or via a promise.

    State creators can also delegate to other state creators by using *AppStates.create()*. For example, an app can do something like this:

```
self.states.add('states:start', function() {
    return self.user.metadata.registered
        ? self.states.create('states:main_menu')
        : self.states.create('states:register');
});
```

AppStates.add.**state**(*state*)

Adds an already created state.

Arguments

- **state** (*State*) – the state to add

AppStates.creators.**__error__**(*name*)

Creates the fallback error state.

Arguments

- **name** (*string*) – the name of the state for which an error occurred.

This default implementation creates an EndState with name name and content *"An error occurred. Please try again later"*.

The end state created has the next state set to the start state. If the start state does not exist, we in the error state again..

AppStates.creators.**__start__**(*name*)

Arguments

- **name** (*string*) – the name of the start state.

- **im** (*InteractionMachine*) – the interaction machine the start state is for.

The default implemenation looks up a creator for the state named `name` and calls that. If no such creator exists, it uses `App.creators.__error__()` instead.

States

States are the building blocks of sandbox applications.

3.1 Overview of States

The currently available states are:

- *FreeText*
- *ChoiceState*
- *MenuState*
- *LanguageChoice*
- *PaginatedChoiceState*
- *BookletState*
- *PaginatedState*
- *EndState*

3.1.1 FreeText

A free text state displays a message and allows a person to respond with any text. It may optionally include a function to validate text input and present an error message. It is the swiss army knife of simple question and answer states.

See `FreeText()`.

3.1.2 ChoiceState

A state which displays a list of numbered choices and allows a person to respond by selecting one of the choices. Each choice has a value (what is stored as the person's answer) and a label (the text that is displayed). Choice states may optionally accept choice labels as input (in addition to the number of the choice in the list).

See `ChoiceState()`.

3.1.3 MenuState

An extension of *ChoiceState* for selecting one of a list of states to go to next.

See `MenuState()`.

3.1.4 LanguageChoice

An extension of *ChoiceState* that allows a person to select from a list of languages. The language choice is stored and translations applied to future interactions (if translations are provided).

See `LanguageChoice()`.

3.1.5 PaginatedChoiceState

An extension of *ChoiceState* for displaying long lists of choices by spanning choices across multiple pages. Allows both automatically dividing up the choices displayed on each page and fixing the number of choices displayed on each page, optionally shortening the length of labels to ensure that a specified character limit is not exceeded. Extremely useful for display dynamic sets of options over USSD or SMS.

See `PaginatedChoiceState()`.

3.1.6 BookletState

A state for displaying paginated text, where the text displayed on each page is programatically determined. Useful when presenting medium length pieces of text or pages of related information that need to be split across multiple USSD messages.

See `BookletState()`.

3.1.7 PaginatedState

Similar to *BookletState*, PaginatedState displays paginated text. The difference between the two is that PaginatedState requires the text to be displayed to the user to be given up front. The text is then automatically divided up into pages.

See `PaginatedState()`.

3.1.8 EndState

This displays text and then terminates a session. Vital for ending USSD sessions but also useful to mark the end of a set of interactions with an application.

See `EndState()`.

3.1.9 Writing your own states

You can also write your own states!

Start by extending one of the existing states, or the base `State()` class as needed.

3.2 State reference

A reference guide to all the states available in the toolkit.

class State(*name*, *opts*)

Base class for states in the interaction machine. States can be thought of as a single screen in a set of interactions with the user.

Arguments

- **name** (*string*) – name used to identify and refer to the state

- **opts.metadata** (*object*) – data about the state relevant to the interaction machine's current user. Optional.

- **opts.send_reply** (*boolean*) – whether or not a reply should be sent to the user's message. Default is *true*. May also be a function, which may return its result via a promise.

- **opts.continue_session** (*boolean*) – whether or not this is the last state in a session. Default is `true`. May also be a function, which may return its result via a promise.

- **opts.helper_metadata** (*object*) – additional helper metadata to set on the reply sent to the user. Primarily useful for setting voice metadata for messages destined to be sent as voice calls. Default is *null*. May also be a function, which may return its result via a promise.

- **opts.check** (*function*) – a function `func(input)` for validating a user's response, where `input` is the user's input. If a string or *LazyText()* is returned, the text will be taken as the error response to send back to the user. If a *StateInvalidError()* is returned, its `response` property will be taken as the error response to send back to the user. Any other value returned will be taken as a non-error. The result may be returned via a promise. See `State.validate()`.

- **opts.events** (*object*) – Optional event name-listener mappings to bind. For example:

```
{
    'state:invalid': function(e) {
        console.log(e);
    }
}
```

static display()

The content to be displayed to the user. May return a promise.

static init()

Invoked just after setup has completed, and just before 'setup' event is fired to provide subclasses with a setup hook. May return a promise.

static input()

Accepts input, invokes `State.translate.before_input()`, then emits a `StateInputEvent'()` to allow input to be processed.

static invalidate(*response*)

Invalidates the user's state, sending the given response to the user. Sets the state's `self.error` object to an appropriate error and emits a `StateInvalidEvent()`.

Arguments

- **response** (string or *LazyText()*) – the response to send back to the user

static **invalidate**(*error*)
    Invalidates the user's state using an error. Sets the state's `self.error` object to an appropriate error and
    emits a :class:StateInvalidEvent''.

    Arguments

    - **error** (`StateInvalidError`) – the error to invalidate the user's state with.

static **save_response**(*response*)
    Called by sub-classes to store accepted user responses on the user object.

    Arguments

    - **response** (`string`) – value to store as an answer.

static **setup**(*im*)
    Called before any other methods on the state are called to allow the state to set itself up.

    Arguments

    - **im** (`InteractionMachine`) – interaction machine using the state.

static **show**()
    Translates the state using *`State.translators.before_display()`*, then displays its text.

static **translate**(*i18n*)
    Translate's a state's text using the given translator. May return a promise.

    Arguments

    - **i18n** (`Translator`) – the translation function to be used for translating the text.

State.emit.**input**(*im*)
    Shortcut for emitting an input event for the state (since this is done quite often). See *`StateInputEvent()`*.

State.translators.**before_display**(*i18n*)
    Translate's a state's text using the given translator. Invoked before text is displayed to the user. By default, just
    delegates to *`State.translate()`*. May return a promise.

    Arguments

    - **i18n** (`Translator`) – the translation function to be used for translating the text.

State.translators.**before_input**(*i18n*)
    Translate's a state's text using the given translator. Invoked before user input is processed. By default, just
    delegates to *`State.translate()`*. May return a promise.

    Arguments

    - **i18n** (`Translator`) – the translation function to be used for translating the text.

State:set_next_state(*name*)
    Set the state that the user will visit after this state using the given state name.

    Arguments

    - **name** (`string`) – The name of the next state

State:set_next_state(*fn*$[$, *arg1*$[$, *arg2*$[$, ... $]$$]$$]$)
    Use a function to set the state that the user will visit this state.

    Arguments

    - **fn** (`function`) – a function that returns name of the next state or an options object with
      data about the next state. The value of `this` inside `f` will be the calling state instance. May
      also return its result via a promise.

- **arg1, arg2, ...** (*arguments*) – arguments to pass to `fn`

class **StateEnterEvent**()
> Emitted when the state is entered by a user.
>
> This happens when the state is switched to from another state, or when the state is created if this is the start of a new session).
>
> > **Arguments**
> >
> > - **state** (`State`) – the state being entered.
>
> The event type is `state:enter`.

class **StateEnterEvent**()
> Emitted when the state is exited by the user. This happens immediately before the interaction machine switches to a different state (see *StateEnterEvent()*).
>
> > **Arguments**
> >
> > - **state** (`State`) – the state being exited.
>
> The event type is `state:exit`.

class **StateError**(*state*, *message*)
> Occurs when interacting or manipulating a state causes an error.
>
> > **Arguments**
> >
> > - **state** (`State`) – the state that caused the error.
> >
> > - **message** (`string`) – the error message.

class **StateEvent**(*name*, *state*, *data*)
> An event relating to a state.
>
> > **Arguments**
> >
> > - **name** (`string`) – the event type's name.
> >
> > - **state** (`State`) – the state associated to the event.

class **StateEvent**(*name*, *state*, *error*)
> Emitted when a state becomes invalid.
>
> > **Arguments**
> >
> > - **state** (`State`) – the state associated to the event.
> >
> > - **error** (`StateInvalidError`) – the validation error that occured.

class **StateInputEvent**(*content*)
> Emitted when the user has given input to the state.
>
> > **Arguments**
> >
> > - **state** (`State`) – the state that the input was given to.
> >
> > - **content** (`string`) – text from the user.
>
> The event type is `state:input`.

class **StateInvalidError**(*state*, *response*[, *opts*])
> Occurs when a state receives invalid input. Raised either by a failed validation check or by explicitly calling *State.invalidate()*.
>
> > **Arguments**

- **state** (`State`) – the state that caused the error.

- **response** (*string or LazyText*) – the response to send back to the user.

- **opts.reason** (*string*) – the reason for the error.

- **opts.input** (*string*) – the user input that caused the error, if relevant

static **translate**(*i18n*)

Translate the error response.

Arguments

- **i18n** (`Translator`) – the translation function to be used for translating the text.

class **StateResumeEvent**()

Emitted when the state is resumed.

When the user enters input, the new sandbox run is started, causing the state to be re-created (or resumed) to process the user's input. This means that when this event is emitted, the state has already been entered (see `StateEnterEvent()`) and its content has been shown to the user in a previous sandbox run (provided the session didn't timeout when the send was attempted).

Arguments

- **state** (`State`) – the state being resumed.

The event type is state:resume.

class **StateShowEvent**()

Emitted when a state's is shown to a user, immediately after `State.display()` has completed.

Arguments

- **state** (`State`) – the state being shown.

- **content** (*string*) – the content being shown.

The event type is state:show.

class **Choice**(*value*, *label*)

An individual choice that the user can select inside a `ChoiceState()`.

Arguments

- **value** (*string*) – string used when storing, processing and looking up the choice.

- **label** (*string*) – string displayed to the user.

class **ChoiceState**(*name*, *opts*)

A state which displays a list of numbered choices, then allows the user to respond by selecting one of the choices.

Arguments

- **name** (*string*) – name used to identify and refer to the state

- **opts.question** (*string or LazyText*) – text to display to the user

- **opts.choices** (Array of `Choice()` objects) – ordered list of choices to display

- **opts.error** (*string or LazyText*) – error text to display to the user if bad user input was given. Optional.

- **opts.accept_labels** (*boolean*) – whether choice labels are accepted as the user's responses. For eg, if accept_labels is true, the state will accepts both "1" and "Red" as responses responses if the state's first choice is "Red". Defaults to false.

- **opts.send_reply** (*boolean*) – whether or not a reply should be sent to the user's message. Defaults to *true*.

- **opts.continue_session** (*boolean*) – whether or not this is the last state in a session. Defaults to *true*.

- **opts.next** (*fn_or_str_or_obj*) – state that the user should visit after this state. May either be the name of the next state, an options object representing the next state, or a function of the form f(choice) returning either, where choice is the *Choice()* chosen by the user. If next is null or not defined, the state machine will be left in the current state. See State.set_next_state().

- **opts.events** (*object*) – Optional event name-listener mappings to bind.

static **process_choice**(*choice*)

> Return true if the choice has been handled completely or false if the choice should be propagated to the next state handler.

> This allows sub-classes to provide custom processing for special choices (e.g. forward and back options for navigating through long choice lists).

> **Arguments**

> - **choice** (*Choice*) – choice to be processed.

static **shorten_choices**(*text*, *choices*)

> Hook for replacing choices with shorter ones if needed.

class **LanguageChoice**(*opts*)

> A state for selecting a language.

> **Arguments**

> - **name** (*string*) – name used to identify and refer to the state

> - **opts.next** (*fn_or_str*) – state that the user should visit after this state. Functions should have the form f(choice) and return the name of the next state or a promise that returns the name. The value of this inside f will be the calling *ChoiceState()* instance.

> - **opts.question** (*string or LazyText*) – text to display to the user

> - **opts.error** (*string or LazyText*) – error text to display to the user if we reach this state in error. Optional.

> - **opts.accept_labels** (*boolean*) – whether choice labels are accepted as the user's responses. For eg, if accept_labels is true, the state will accepts both "1" and "Red" as responses responses if the state's first choice is "Red". Defaults to false.

> - **opts.send_reply** (*boolean*) – whether or not a reply should be sent to the user's message. Defaults to *true*.

> - **opts.continue_session** (*boolean*) – whether or not this is the last state in a session. Defaults to *true*.

> - **opts.events** (*object*) – Optional event name-listener mappings to bind.

It functions exactly like *ChoiceState()* except that it also stores the value of the selected choices as the user's language (it is still available as an answer too).

Choice() instances passed to this state should have two-letter language codes as values, e.g.:

```
    new LanguageChoice(
        "select_language",
        {
            next: "next_state",
            question: "What language would you like to use?",
            choices: [ new Choice("sw", "Swahili"), new Choice("en", "English") ]
        }
    );
```

class MenuState(*name*, *opts*)

A *ChoiceState()* whose *Choice()* values are either state names or state options objects. See State.set_next_state() for a description of the options objects.

Supports the same parameters as *ChoiceState()* except that opts.next isn't available.

class PaginatedChoiceState(*name*, *opts*)

A choice state for displaying long lists of choices by spanning the choices across multiple pages.

> **Arguments**
>
> - **name** (*string*) – name used to identify and refer to the state
>
> - **opts.next** (*fn_or_str*) – state that the user should visit after this state. Functions should have the form f(choice) and return the name of the next state or a promise that returns the name. The value of this inside f will be the calling *ChoiceState()* instance.
>
> - **opts.question** (*string*) – text to display to the user
>
> - **opts.error** (*string or LazyText*) – error text to display to the user if we reach this state in error. Optional.
>
> - **opts.accept_labels** (*boolean*) – whether choice labels are accepted as the user's responses. For eg, if accept_labels is true, the state will accepts both "1" and "Red" as responses responses if the state's first choice is "Red". Defaults to false.
>
> - **opts.send_reply** (*boolean*) – whether or not a reply should be sent to the user's message. Defaults to *true*.
>
> - **opts.continue_session** (*boolean*) – whether or not this is the last state in a session. Defaults to *true*.
>
> - **opts.back** (*string*) – the choice label to display to the user for going back a page. Default is *"Back"*.
>
> - **opts.more** (*string*) – the choice label to display to the user for going to the next page Default is *"Next"*.
>
> - **opts.options_per_page** (*int*) – maximum number of choices to display per page. Default is 8. If this option is explicitly given as null, *PaginatedChoiceState()* will automatically divide up the given choices to fit within the character limit given by the 'characters_per_page' option.
>
> - **opts.characters_per_page** (*int*) – maximum number of characters to display per page. Default is *null* (i.e. no maximum), or 160 if the 'characters_per_page' option is explicitly given as null.
>
> - **opts.events** (*object*) – Optional event name-listener mappings to bind.

class BookletState(*name*, *opts*)

A state for displaying paginated text.

> **Arguments**

- **name** (*string*) – name of the state

- **opts.pages** (*integer*) – total number of pages.

- **opts.page_text** (*function*) – function func(n) returning the text of page n. Pages are numbered from 0 to (pages - 1). May return a promise.

- **opts.initial_page** (*integer*) – page number to use when the state is entered. Optional, default is 0.

- **opts.buttons** (*object*) – map of user inputs to amounts to increment the page number by. The special value 'exit' triggers moving to the next state. Optional, default is: {"1": -1, "2": +1, "0": "exit"},

- **opts.footer_text** (*string*) – text to append to every page. Optional, default is: "1 for prev, 2 for next, 0 to end."

- **opts.send_reply** (*boolean*) – whether or not a reply should be sent to the user's message. Defaults to *true*.

- **opts.continue_session** (*boolean*) – whether or not this is the last state in a session. Defaults to *true*.

- **opts.next** (*fn_or_str_or_obj*) – state that the user should visit after this state. May either be the name of the next state, an options object representing the next state, or a function of the form f(content) returning either, where content is the input given by the user. If next is null or not defined, the state machine will be left in the current state. See State.set_next_state().

- **opts.events** (*object*) – Optional event name-listener mappings to bind.

class **PaginatedState**(*name*, *opts*)
 Add state type that divides up the given text into pages.

Arguments

- **name** (*string*) – name used to identify and refer to the state

- **opts.text** (*string or LazyText*) – the content to display to the user.

- **opts.page** – The function to use to determine the text shown to the user.

 The function should return the text to be displayed to the user as a string and take the form fn(i, text, n), where *i*' is the user's 0-indexed current page number, ''text'' is the translated text, n is the maximum number of characters that can fit on the page (after taking into account the nagivation choices) and this is the *PaginatedState()* instance.

 When the function returns a falsy value, page i - 1 is taken as the last page to be displayed to the user. The function may also return a promise fulfilled with the value.

 If this option is not provided, the *PaginatedState()* will use a default function that will display the words that fit on the page based on the values of i and the given 'characters_per_page' option.

- **opts.send_reply** (*boolean*) – whether or not a reply should be sent to the user's message. Defaults to *true*.

- **opts.characters_per_page** (*int*) – maximum number of characters to display per page (including the characters needed for the navigation choices). Default is *160*. 'back', 'more' and 'exit' choices.

- **opts.back** (*string*) – the label to display to the user for going back a page. Defaults to 'Back'.

- **opts.more** (*string*) – the label to display to the user for going to the next page. Defaults to '`More`'.

- **opts.exit** (*string*) – the choice label to display to the user for going to the next state. Defaults to '`Exit`'.

- **opts.continue_session** (*boolean*) – whether or not this is the last state in a session. Defaults to *true*.

- **opts.next** (*function or string*) – state that the user should visit after this state. May either be the name of the next state, an options object representing the next state, or a function of the form `f(content)` returning either, where `content` is the input given by the user when the user chooses to exit the `PaginatedState()`. If `next` is `null` or not defined, the state machine will be left in the current state. See `State.set_next_state()`.

- **opts.events** (*object*) – Optional event name-listener mappings to bind.

class **EndState**(*name*, *opts*)
    A state which displays text and then ends the session.

    Arguments

- **name** (*string*) – name used to identify and refer to the state

- **opts.text** (*string or LazyText*) – text to display to the user

- **opts.next** (*fn_or_str_or_obj*) – state that the user should visit after this state. May either be the name of the next state, an options object representing the next state, or a function of the form `f(content)` returning either, where `content` is the input given by the user. If `next` is `null` or not defined, the state machine will be left in the current state. See `State.set_next_state()`.

- **opts.events** (*object*) – Optional event name-listener mappings to bind.

class **FreeText**(*name*, *opts*)
    A state which displays a text, then allows the user to respond with any text.

    Arguments

- **name** (*string*) – name used to identify and refer to the state

- **opts.question** (*string or LazyText*) – text to display to the user.

- **opts.send_reply** (*boolean*) – whether or not a reply should be sent to the user's message. Defaults to *true*.

- **opts.continue_session** (*boolean*) – whether or not this is the last state in a session. Defaults to *true*.

- **opts.check** (*function*) – a function `func(content)` for validating a user's response, where `content` is the user's input. If a string `LazyText()` is returned, the text will be taken as the error response to send back to the user. If a `StateInvalidError()` is returned, its `response` property will be taken as the error response to send back to the user. Any other value returned will be taken as a non-error. The result may be returned via a promise. See `State.validate()`.

- **opts.next** (*fn_or_str_or_obj*) – state that the user should visit after this state. May either be the name of the next state, an options object representing the next state, or a function of the form `f(content)` returning either, where `content` is the input given by the user. If `next` is `null` or not defined, the state machine will be left in the current state. See `State.set_next_state()`.

- **opts.events** (*object*) – Optional event name-listener mappings to bind.

3.3 What are states?

A state corresponds to a small piece of an application. It might represent a single question in a survey, a menu, a greeting to send or a small booklet of text for someone to page through on their phone.

Each state has a name and a function to construct it, called its creator. The creator takes the name of a state and options and should return an instance of `State()`.

Each state should transfer control to the next state once it is done.

States often have text to be displayed (to a person on their phone) and validation functions to parse input received.

3.4 How are applications built from states?

An application is a set of state creators collected into an `App()`. An `App()` is controlled by an `InteractionMachine()` which manages states and links an application to the low-level sandbox API.

An `InteractionMachine()` receives messages from people (via the sandbox API) and directs those messages to the current state. It also tracks what state a person is interacting with and manages transitions to new states.

Last but not least, an `InteractionMachine()` provides a set of high-level interfaces to the sandbox API's resources. These allow an application to perform actions such as looking up or modifying a contact, logging errors or warnings, making HTTP requests or storing persistent data in a key-value store.

3.5 Delegation and virtual states

Some state creators represent virtual states. Instead of returning a state with the name associated with them, they return a state with a different name. Virtual creators are said to *delegate* to another state.

Delegators usually select between one of a set of other states and help structure applications cleanly and avoid repetition of logic for selecting which state to go to next.

3.6 What kinds of states are available?

An overview of the states available in the toolkit can be found in the Overview of States.

3.7 Reference

A complete reference guide to the available states can be found in the State reference.

Logging

class Logger(*im*)

Provides logging for the app and interaction machine.

Arguments

- **im** (`InteractionMachine`) – the interaction machine associated to the logger.

The initialised logger can also be invoked directly, which delegates to *Logger.info()*:

```
im.log('foo');
```

static critical(*message*)

Logs a message at the `'CRITICAL'` log level

Arguments

- **message** (*string*) – The message to log.

static debug(*message*)

Logs a message at the `'DEBUG'` log level

Arguments

- **message** (*string*) – The message to log.

static error(*message*)

Logs a message at the `'ERROR'` log level

Arguments

- **message** (*string*) – The message to log.

static info(*message*)

Logs a message at the `'INFO'` log level

Arguments

- **message** (*string*) – The message to log.

static warning(*message*)

Logs a message at the `'WARNING'` log level

Arguments

- **message** (*string*) – The message to log.

User

User() is used for short-term information about a user interacting with your application. Most of this information relates to the current interaction session with the user, and includes the user's current state, user's answers to previous states and language preference. While *User()* is used for short-term information about a user, a *Contact()* holds long-term information.

User (*im*)

A structure for managing the current user being interacted with in *InteractionMachine()*.

Arguments

- **im** (*InteractionMachine*) – the interaction machine to which this user is associated

static create (*addr*, *opts*)

Invoked to create a new user. Simply delegates to User.setup(), but sets the user's creation_event to *UserNewEvent()*. Intended to be used to explicitly differentiate newly created users from loaded users with a single action.

static created

Whether this is a new or loaded user.

static default_ttl ()

Returns the default expiry time of saved user state (in seconds).

This may be set using the user_ttl sandbox config key. It defaults to 604800 seconds (seven days). Expiry may be disabled by setting user_ttl to null.

static fetch ()

Fetches the user's current state data from the key-value data store resource. Returns a promised fulfilled with the fetched data.

static get_answer (*state_name*)

Get the user's answer for the state associated with state_name.

Arguments

- **state_name** (*string*) – the name of the state to retrieve an answer for

static is_in_state ([*state_name*])

Determines whether the user is in the state represented by state_name, or whether the user is in any state at all if no arguments are given.

Arguments

- **state_name** (*string*) – the name of the state compare with

static **key** ()
> Returns the key under which to store user state. If `user.store_name` is set, stores the user under `users.<store_name>.<addr>`, or otherwise under `users.<addr>`.

static **load** (*addr*[, *opts*])
> Load a user's current state from the key-value data store resource, then sets the user's `creation_event` to `UserLoadEvent()`. Throws an error if loading fails.
>
> Returns a promise that is fulfilled when the loading and event emitting has completed.
>
> Accepts the same params as `User.setup()`, where the *opts* param contains overrides for the loaded user data.

static **load_or_create** (*addr*[, *opts*])
> Attempts to load a user's current state from the key-value data store resource, creating the user if no existing user was found. Sets the user's `creation_event` to `UserLoadEvent()` if the user was loaded, and `UserNewEvent()` if the user was created.
>
> Returns a promise that is fulfilled when the loading and event emitting has completed.
>
> Accepts the same params as `User.setup()`, where the *opts* param contains overrides for the loaded user data.

static **make_key** (*addr*[, *store_name*])
> Makes the key under which to store a user's state. If *store_name* is set, stores the user under `'users.<store_name>.<addr>`, or otherwise under `<addr>`.
>
> > **Arguments**
> >
> > * **addr** (*string*) – The address used as a key to load and save the user.
> > * **store_name** (*string*) – The namespace path to be used when storing the user.

static **refresh_i18n** ()
> Re-fetches the appropriate language translations. Sets `user.i8n` to a new `Translator()` instance.
>
> Returns a promise that fires once the translations have been refreshed.

static **reset** (*addr*, *opts*)
> Invoked to create a new user. Simply delegates to `User.setup()`, but sets the user's `creation_event` to a `UserResetEvent()`. Intended to be used to explicitly differentiate reset users from both newly created users and loaded users with a single action.

static **save** ()
> Save a user's current state to the key-value data store resource, then emits a `UserSaveEvent()`.
>
> > **Arguments**
> >
> > * **opts.seconds** (*object*) – How long the user's state should be stored for before expiring. See `User.default_ttl()` for how the default is determined.
>
> Returns a promise that is fulfilled once the user data has been saved and events have been emitted.

static **serialize** ()
> Returns an object representing the user. Suitable for JSON stringifying and storage purposes.

static **set_answer** (*state_name*, *answer*)
> Sets the user's answer to the state associated with `state_name`.
>
> > **Arguments**
> >
> > * **state_name** (*string*) – the name of the state to save an answer for
> > * **answer** (*string*) – the user's answer to the state

static **set_lang**(*lang*)
>   Gives the user a new language. If the user's language has changed, their translator is is refreshed (delegates to `User.refresh_i18n()`). Returns a promise that will be fulfilled once the method's work is complete.

>   **Arguments**

>>   • **lang** (`string`) – The two-letter code of the language the user has selected. E.g. *en*, *sw*.

static **setup**(*addr*, *opts*)
>   Sets up the user. Returns a promise that is fulfilled once the setup is complete.

>   **Performs the following steps:**

>>   • Processes the given setup arguments

>>   • Attempts to refresh the translator (involves interaction with the sandbox api).

>>   • Emits a [*SetupEvent()*](#)

>   **Arguments**

>>   • **addr** (`string`) – the address used as a key to load and save the user.

>>   • **opts.lang** (`string`) – the two-letter code of the language the user has selected. E.g. 'en', 'sw'.

>>   • **opts.store_name** (`string`) – an additional namespace path to be used when storing the user. See `User.key()`.

>>   • **opts.state.name** (`string`) – the name of the state most recently visited by the user. Optional.

>>   • **opts.state.metadata** (`string`) – metadata about the state most recently visited by the user. Optional.

>>   • **opts.in_session** (`boolean`) – whether the user is currently in a session. Defaults to `false`.

class **UserEvent**(*user*)
>   An event relating to a user.

>   **Arguments**

>>   • **name** (`string`) – the event type's name.

>>   • **user** ([User](#)) – the user associated to the event.

class **UserLoadEvent**(*user*)
>   Emitted when an existing user is loaded. This typically happens in [*InteractionMachine()*](#) when message arrives from a user for who has already interacted with the system.

>   **Arguments**

>>   • **user** ([User](#)) – the user that was loaded.

>   The event type is `user:load`.

class **UserNewEvent**(*user*)
>   Emitted when a new user is created. This typically happens in [*InteractionMachine()*](#) when message arrives from a user for whom there is no user state (i.e. a new unique user).

>   **Arguments**

>>   • **user** ([User](#)) – the user that was created.

The event type is `user:new`.

class **UserNewEvent** (*user*)

Emitted when a user's data is reset. This typically happens in *InteractionMachine()* when message arrives from a user for whom with its content being "!reset", forcing the user to be reset.

Arguments

- **user** (`User`) – the user that was reset.

The event type is `user:reset`.

class **UserSaveEvent** (*user*)

Emitted when a user is saved. This typically happens in *InteractionMachine()* after an inbound message from the user has been processed as one of the last actions before terminating the sandbox.

Arguments

- **user** (`User`) – the user that was saved.

The event type is `user:save`.

Config

class IMConfig(*im*)

Provides access to an *InteractionMachine()* 's config data.

>   **Arguments**
>
>   - **im** (InteractionMachine) – the interaction machine to which this config is associated

static setup()

Sets up the interaction machine's config by reading the config from its value in the interaction machine's sandbox config (the value of the *config* key in the sandbox config). Emits a Setup() event once setup is complete. returns a promise that is fulfilled after setup is complete and after event listeners have done their work.

class IMConfigError(*message*)

Thrown when an error occurs while validating or accessing something on the interaction machine's config.

>   **Arguments**
>
>   - **config** (IMConfig) – the im's config.
>
>   - **message** (*string*) – the error message.

class SandboxConfig(*im*)

Provides access to the sandbox's config data.

>   **Arguments**
>
>   - **im** (InteractionMachine) – the interaction machine to which this sandbox config is associated

static get(*key*, *opts*)

Retrieve a value from the sandbox application's Vumi Go config. Returns a promise that will be fulfilled with the config value.

>   **Arguments**
>
>   - **key** (*string*) – name of the configuration item to retrieve.
>
>   - **opts.json** (*boolean*) – whether to parse the returned value using JSON.parse. Defaults to false.

Contacts

Contacts hold information about the users interacting with your application. While *User()* is used for short-term information about a user (usually information related to a particular interaction session), a *Contact()* holds long-term information.

class Contact (*attrs*)

Holds long-term information about a user interacting with the application.

> **Arguments**
>
> - **attrs.key** (*string*) – A unique identifier for looking up the contact.
>
> - **attrs.user_account** (*string*) – The name of the vumi go account that the contact is stored under.
>
> - **attrs.msisdn** (*string*) – The contact's msisdn.
>
> - **attrs.gtalk_id** (*string*) – The contact's gtalk address. Optional.
>
> - **attrs.facebook_id** (*string*) – The contact's facebook address. Optional.
>
> - **attrs.twitter_handle** (*string*) – The contact's twitter handle. Optional.
>
> - **attrs.name** (*string*) – The contact's name. Optional.
>
> - **attrs.surname** (*string*) – The contact's surname. Optional.
>
> - **attrs.extra** (*object*) – A data object for additional, app-specific information about a contact. Both the keys and values need to be strings. Optional.
>
> - **attrs.extras-<name>** (*object*) – An alternative way of specifying an extra. Optional.
>
> - **attrs.groups** (*array*) – A list of keys, each belonging to a group that this contact is a member of. Optional.

> **static serialize**()
>
> Returns a deep copy of the contact's attributes.

Contact.do.**reset** (*attrs*)

Resets a contact's attributes to *attrs*. All the contact's current attributes will be lost.

> **Arguments**
>
> - **attrs** (*object*) – the attributes to reset the contact with.

Contact.do.**validate**()

Validates a contact, throwing a ValidationError() if one of its attributes are invalid.

ContactStore(*im*)

> Provides 'ORM-like' access to the sandbox's contacts resource, handling the raw contact resource api requests and allowing people to interact with their contacts as `Contact()` instances.
>
> **Arguments**
>
> > • **im** (`InteractionMachine`) – The interaction machine

static **create**(*attrs*)

> Creates and adds a new contact, returning a corresponding `Contact()` via a promise.
>
> **Arguments**
>
> > • **attrs** (`object`) – The attributes to initialise the new contact with.

```javascript
self.im.contacts.create({
    surname: 'Jones',
    extra: {location: 'CPT'}
}).then(function(contact) {
    console.log(contact instanceof Contact);
});
```

static **for_user**(*opts*)

> Retrieves a contact for the the current user in the `InteractionMachine()`, returning a corresponding `Contact()` via a promise. If no contact exists for the user, a contact is created.
>
> **Arguments**
>
> > • **opts.create** (`boolean`) – Whether to create a contact for the user if it does not yet exist. Defaults to `true`.
> >
> > • **opts.delivery_class** (`string`) – The delivery class corresponding to the current user's address. If not specified, `ContactStore()` uses its fallback, `ContactStore.delivery_class`.

```javascript
self.im.contacts.for_user().then(function(contact) {
    console.log(contact instanceof Contact);
});
```

static **get**(*addr*[, *opts*])

> Retrieves a contact by its address for a particular delivery class, returning a corresponding `Contact()` via a promise.
>
> **Arguments**
>
> > • **addr** (`boolean`) – The address of the contact to be retrieved.
> >
> > • **opts.create** (`boolean`) – Create the contact if it does not yet exist. Defaults to `false`.
> >
> > • **delivery_class** (`string`) – The delivery class corresponding to the given address. If not specified, `ContactStore()` uses its fallback, `ContactStore.delivery_class`.

```javascript
self.im.contacts.get('+27731234567').then(function(contact) {
    console.log(contact instanceof Contact);
});
```

> **The following delivery classes are supported:**
>
> > • `sms`: maps to the contact's `msisdn` attribute
> >
> > • `ussd`: maps to the contact's `msisdn` attribute

- gtalk: maps to the contact's gtalk_id attribute

- twitter: maps to the contact's twitter_handle attribute

static **get** (*key*)

 Retrieves a contact by its key, returning a corresponding *Contact()* via a promise.

  Arguments

   - **key** (*string*) – The contact's key.

```
self.im.contacts.get('1234').then(function(contact) {
    console.log(contact instanceof Contact);
});
```

static **request** (*name*, *cmd*)

 Makes raw requests to the api's contact resource.

  Arguments

   - **name** (*string*) – The name of the contact api method (for eg, 'get')

   - **cmd** (*object*) – The request's command data

static **save** (*contact*)

 Saves the given contact to the store, returning a promise that is fulfilled once the operation completes.

  Arguments

   - **contact** (*Contact*) – The contact to be saved

static **search** (*query*)

 Searches for contacts matching the given Lucene search query, returning an array of the matching *Contact()* instances via a promise. Note that this can be a fairly heavy operation. If only the contact keys are needed, please use ContactStore.search_keys() instead.

  Arguments

   - **query** (*string*) – The Lucene query to perform

```
self.im.contacts.search('name:"Moog"').then(function(contacts) {
    contacts.forEach(function(contact) {
        console.log(contact instanceof Contact);
    });
});
```

static **search_keys** (*query*)

 Searches for contacts matching the given Lucene search query, returning an array of the contacts' keys via a promise.

  Arguments

   - **query** (*string*) – The Lucene query to perform

```
self.im.contacts.search_keys('name:"Moog"').then(function(keys) {
    keys.forEach(function(key) {
        console.log(typeof key == 'string');
    });
});
```

class **Group** (*attrs*)

 Holds information about a group of contacts.

  param string attrs.key a unique identifier for looking up the contact.

> **param string attrs.user_account** the name of the vumi go account that owns this group.
>
> **param string attrs.name** a human-readable name for the group.
>
> **param string attrs.query** the contact search query that determines the contacts in this group. Optional.

> static `serialize`()
>
> > Returns a deep copy of the group's attributes.

Group.do.`reset`(*attrs*)

> Resets a groups's attributes to *attrs*. All the groups's current attributes will be lost.
>
> > **Arguments**
> >
> > - **attrs** (*object*) – the attributes to reset the group with.

Group.do.`validate`()

> Validates a group, throwing a `ValidationError()` if one of its attributes are invalid.

GroupStore(*im*)

> Provides 'ORM-like' access to the sandbox's group resource, allowing people to interact with their groups as `Group()` instances.
>
> > **Arguments**
> >
> > - **im** (`InteractionMachine`) – The interaction machine

> static **get** (*name*[, *opts*])
>
> > Retrieves a group by its name, returning a corresponding `Group()` via a promise.
> >
> > > **Arguments**
> > >
> > > - **name** (*string*) – The name of the group to retrieve.
> > >
> > > - **opts.create** (*boolean*) – Create the group if it does not yet exist. Defaults to `false`.

```javascript
self.im.groups.get('spammers').then(function(group) {
    console.log(group instanceof Group);
});
```

> static **get_by_key**(*key*)
>
> > Retrieves a group by its key, returning a corresponding `Group()` via a promise.
> >
> > > **Arguments**
> > >
> > > - **key** (*string*) – The group's key.

```javascript
self.im.groups.get_by_key('1234').then(function(group) {
    console.log(group instanceof Group);
});
```

> static **list**()
>
> > Returns a promise fulfilled with a list of the `Group()`'s stored under the account associated with the app.

> static **request**(*name*, *cmd*)
>
> > Makes raw requests to the api's group resource.
> >
> > > **Arguments**
> > >
> > > - **name** (*string*) – The name of the group api method (for eg, `'get'`)
> > >
> > > - **cmd** (*object*) – The request's command data

static **save**(*group*)

> Saves the given group to the store, returning a promise that is fulfilled once the operation completes.
>
> > **Arguments**
> >
> > > • **group** ([Group](#)) – The group to be saved

static **search**(*query*)

> Searches for groups matching the given Lucene search query, returning an array of the matching [Group()](#) instances via a promise.
>
> > **Arguments**
> >
> > > • **query** (*string*) – The Lucene query to perform

```javascript
self.im.groups.search('name:"spammers"').then(function(groups) {
    groups.forEach(function(group) {
        console.log(group instanceof Group);
    });
});
```

static **setup**()

> Sets up the store.

static **sizeOf**(*group*)

> Returns a promise fulfilled with the number of contacts that are members of the given group.
>
> > **Arguments**
> >
> > > • **group** ([Group](#)) – The group who's size needs to be determined.

HTTP API

class **HttpApi** (*im*, *opts*)

A helper class for making HTTP requests via the HTTP sandbox resource.

Arguments

- **im** (`InteractionMachine`) – The interaction machine to use when making requests.

- **opts.headers** (`object`) – Default headers to use in HTTP requests.

- **opts.auth** (`object`) – Adds a HTTP Basic authentication to the default headers. Should contain `username` and `password` attributes.

- **opts.verify_options** (`string`) – The default list of options to verify when doing HTTPS requests. Optional.

- **opts.ssl_method** (`string`) – The default ssl method to attempt for HTTPS requests. Optional.

static **decode_response_body** (*body*)

Arguments

- **body** (`string`) – The body to decode.

Sub-classes should override this to decode the response body and throw an exception if the body cannot be parsed. This base implementation returns the body as-is (i.e. decoding is left to the code calling the *HttpApi()*).

static **delete** (*url*, *opts*)

Make an HTTP DELETE request.

Arguments

- **url** (`string`) – The URL to make the request to.

- **opts** (`object`) – Options to pass to *HttpApi.request()*.

Returns a promise which fires with the decoded value of the response body or an object with an error attribute containing the error message.

static **encode_request_data** (*data*)

Arguments

- **data** (`object`) – The data to encode.

Sub-classes should override this to encode the request body and throw an exception if the data cannot be encoded. This base implementation returns the data as-is (i.e. encoding is left to code calling the *HttpApi()*).

static **get** (*url*, *opts*)
    Make an HTTP GET request.

        Arguments

- **url** (*string*) – The URL to make the request to.

- **opts** (*object*) – Options to pass to *HttpApi.request()*.

    Returns a promise which fires with the decoded value of the response body or an object with an error attribute containing the error message.

static **head** (*url*, *opts*)
    Make an HTTP HEAD request.

        Arguments

- **url** (*string*) – The URL to make the request to.

- **opts** (*object*) – Options to pass to *HttpApi.request()*.

    Returns a promise which fires with the decoded value of the response body or an object with an error attribute containing the error message.

static **parse_reply** (*reply*, *request*)
    Check an HTTP reply and throw an *HttpRequestError()* if the sandbox API command was unsuccessful, or otherwise parse the sandbox's reply into a response. If the response status code is not in the 200 range or the reply body cannot be decoded, throw an *HttpResponseError()*.

    Logs an error via the sandbox logging resource in an error is raised.

        Arguments

- **reply** (*object*) – Raw response to the sandbox API command.

- **request** (*HttpRequest*) – The request that initiated the sandbox API command.

    Returns an *HttpResponse()* or throws an *HttpApiError()* (either the *HttpRequestError()* or *HttpResponseError()* derivative, depending on what error occured).

static **post** (*url*, *opts*)
    Make an HTTP POST request.

        Arguments

- **url** (*string*) – The URL to make the request to.

- **opts** (*object*) – Options to pass to *HttpApi.request()*.

    Returns a promise which fires with the decoded value of the response body or an object with an error attribute containing the error message.

static **post** (*url*, *opts*)
    Make an HTTP PATCH request.

        Arguments

- **url** (*string*) – The URL to make the request to.

- **opts** (*object*) – Options to pass to *HttpApi.request()*.

    Returns a promise which fires with the decoded value of the response body or an object with an error attribute containing the error message.

static **put** (*url*, *opts*)
    Make an HTTP PUT request.

> **Arguments**
>
> - **url** (*string*) – The URL to make the request to.
>
> - **opts** (*object*) – Options to pass to *HttpApi.request()*.
>
> Returns a promise which fires with the decoded value of the response body or an object with an error attribute containing the error message.

static **request** (*method*, *url*, *opts*)

> **Arguments**
>
> - **method** (*string*) – The HTTP method to use (e.g. *GET*, *POST*).
>
> - **url** (*string*) – The URL to make the request to. If you pass in query parameters using `opts.params`, don't include any in the URL itself.
>
> - **opts.params** – An object of key-value pairs to append to the URL as query parameters. Can be in any form accepted by node.js's querystring module
>
> - **opts.data** (*object*) – Data to pass as the request body. Will be encoded using *HttpApi.encode_request_data()* before being sent.
>
> - **opts.headers** (*object*) – Additional headers to add to the default headers.
>
> Returns a *HttpResponse()* via a promise. Failures while making and checking the request will be thrown as `HttpApiError`'s, and can be caught with a Q errback. See :meth:`HttpApi.parse_reply()` for more on the response parsing and error throwing.

class **HttpApiError** ()

Thrown when an error occurs while making and checking an HTTP request and the corresponding API reply.

class **HttpRequest** (*request*, *code*, *opts*)

Encapsulates information about an HTTP request made by the *HttpApi()*. Once *HttpRequest.encode()* has been invoked, the request's data is encoded and made available as the request's body.

> **Arguments**
>
> - **method** (*string*) – the HTTP request method.
>
> - **url** (*string*) – the url to send the request to.
>
> - **opts.data** (*string*) – the request's data to be encoded as the request's body. Optional.
>
> - **opts.params** – An object of key-value pairs to append to the URL as query parameters. Can be in any form accepted by node.js's querystring module
>
> - **opts.verify_options** (*string*) – A list of options to verify when doing an HTTPS request. Optional.
>
> - **opts.ssl_method** (*string*) – A request for a specific ssl method to be attempted. Optional.

static **encode** ()

Encodes the request data (if available).

static **to_cmd** ()

Returns a sandbox API command that can be used to initiate this request via the sandbox API.

class **HttpRequestError** (*request*, *reason*)

Thrown when an error occurs while making and checking an HTTP request.

> **Arguments**

- **request** (`HttpRequest`) – the request.

- **reason** (`string`) – the reason for the failure. Optional.

class **HttpResponse** (*request*, *code*, *opts*)

Encapsulates information about an HTTP response given to the `HttpApi()`. Once `HttpResponse.decode()` has been invoked, the response's body is decoded and made available as the response's data.

> **Arguments**
>
> - **request** (`HttpRequest`) – the request that caused the response.
>
> - **code** (`integer`) – the status code for the HTTP response.
>
> - **opts.body** (`string`) – the response's body to be decoded as the response's data. Optional.

> static **decode** ()
>
> Decodes the responses body (if available).

class **HttpResponseError** (*response*, *reason*)

Thrown when an error response is returned by an HTTP request or if the HTTP response body cannot be parsed.

> **Arguments**
>
> - **response** (`HttpResponse`) – the response.
>
> - **reason** (`string`) – the reason for the failure. Optional.

class **JsonApi** (*im*, *opts*)

A helper class for making HTTP requests that send and receive JSON encoded data.

> **Arguments**
>
> - **im** (`InteractionMachine`) – The interaction machine to use when making requests.
>
> - **opts.headers** (`object`) – Default headers to use in HTTP requests. The `Content-Type` header is overridden to be `application/json; charset=utf-8`.
>
> - **opts.auth** (`object`) – Adds a HTTP Basic authentication to the default headers. Should contain `username` and `password` attributes.

> static **decode_response_body** (*body*)
>
> Decode an HTTP response body using `JSON.parse()`.
>
> > **Arguments**
> >
> > - **body** (`string`) – Raw HTTP response body to parse.
>
> Returns the decoded response body.

> static **encode_request_data** (*data*)
>
> Encode an object as JSON using `JSON.stringify()`.
>
> > **Arguments**
> >
> > - **data** (`object`) – Object to encode to JSON.
>
> Returns the serialized object as a string.

Metrics

class MetricStore(*im*)

> Provides metric firing capabilties for the *InteractionMachine()*.
>
> > **Arguments**
> >
> > > • **im** (`InteractionMachine`) – the interaction machine to which this sandbox config is associated
>
> **static fire**(*metric*, *value*, *agg*)
>
> > Fires a metric.
> >
> > > **Arguments**
> > >
> > > > • **metric** (`string`) – the name of the metric
> > > >
> > > > • **value** (`number`) – the value of the metric
> > > >
> > > > • **agg** (`string`) – the aggregation method to use
>
> **static setup**([*opts*])
>
> > Sets up the metric store.
> >
> > > **Arguments**
> > >
> > > > • **opts.store_name** – the store/namespace to use for fired metrics. Defaults to 'default'

MetricStore.fire.**avg**(*metric*, *value*)

> Fires a metric with the `avg` aggregation method.
>
> > **Arguments**
> >
> > > • **metric** (`string`) – the name of the metric
> > >
> > > • **value** (`number`) – the value of the metric

MetricStore.fire.**inc**(*metric*[, *opts*])

> Increments the value for the key `metric` in in the kv store, fires a metric with the new total using the `'last` aggregation method, then returns the total via a promise.
>
> > **Arguments**
> >
> > > • **metric** (`string`) – the name of the metric
> > >
> > > • **opts.amount** (`number`) – the amount to increment by. Defaults to `1`.

MetricStore.fire.**last**(*metric*, *value*)

> Fires a metric with the `last` aggregation method.
>
> > **Arguments**

- **metric** (*string*) – the name of the metric

- **value** (*number*) – the value of the metric

`MetricStore.fire.`**`max`** (*metric*, *value*)
    Fires a metric with the `max` aggregation method.

    Arguments

- **metric** (*string*) – the name of the metric

- **value** (*number*) – the value of the metric

`MetricStore.fire.`**`min`** (*metric*, *value*)
    Fires a metric with the `min` aggregation method.

    Arguments

- **metric** (*string*) – the name of the metric

- **value** (*number*) – the value of the metric

`MetricStore.fire.`**`sum`** (*metric*, *value*)
    Fires a metric with the `sum` aggregation method.

    Arguments

- **metric** (*string*) – the name of the metric

- **value** (*number*) – the value of the metric

Events

class Event()

A structure for events fired in various parts of the toolkit.

> **Arguments**
>
> - **name** (*string*) – the event's name.
>
> - **data** (*string*) – the event's data. Optional.

class Event()

Lightweight wrapper around EventEmitter() for working better with Q promises and the toolkit's *Event()* objects.

Eventable.**emit**(*event*)

Emits the given event and returns a promise that will be fulfilled once each listener is done. This allows listeners to return promises.

> **Arguments**
>
> - **event** (Event) – the event to emit.

static setup()

Shortcut for emitting a setup event for the instance (since this is done quite often). See *SetupEvent()*.

static teardown()

Shortcut for emitting a teardown event for the instance. See *TeardownEvent()*.

Eventable.once.**resolved**(*event_name*)

Returns a promise that will be fulfilled once the event has been emitted. Since a promise can only be fulfilled once, the event listener is removed after the event is emitted. Useful for testing events.

> **Arguments**
>
> - **event_name** (*string*) – the event to listen for.

Eventable.**teardown_listeners**()

Removes all event listeners, with the following exception: listeners for *TeardownEvent()*s get rebound using Eventable.once(), regardless of whether they were orginally bound using Eventable.on() or Eventable.once(). This allows us to remove all event listeners for instances of Eventable(), while still allowing other entities to know when the teardown of the entity has completed.

Not that it is up to the caller to emit the *TeardownEvent()* to clear the listeners.

class SetupEvent(*instance*)

Emitted when an instance of something has been constructed.

> **Arguments**

- **instance** (*object*) – the constructed instance.

class TeardownEvent (*instance*)

Emitted when an instance of something has completed the tasks it needs to complete before it can be safely disposed of.

> **Arguments**
>
> - **instance** (*object*) – the instance.

AppTester

11.1 API

class AppTester(*app*, *opts*)

    Machinery for testing a sandbox application.

    Provides *setup*, *interaction* and *checking* tasks. Whenever a task method is called, its task is scheduled to run next time `AppTester.run()` is called.

    Arguments

- **app** (`App`) – the sandbox app to be tested.

- **opts.api** (`object`) – options to initialise the tester's `DummyApi()` with each reset.

static reset()

    Clears scheduled tasks and data, and uses a new api and interaction machine, clearing things for the next tester run.

static run()

    Runs the tester's scheduled tasks in the order they were scheduled, then resets the tester. Returns a promise which will be fulfilled once the scheduled tasks have run and the tester has reset itself.

11.1.1 Setup Tasks

Setup tasks are used to configure the sandbox app's config and store data before any interaction and checking is done.

`AppTester.`**setup**(*fn*)

    Allows custom setting up of the sandbox application's config and data.

    Arguments

- **fn** (`function`) – function to be used to set up the sandbox application. Takes the form `func(api)`, where `api` is the tester's api instance and `this` is the `AppTester()` instance. May return a promise.

```
tester.setup(function(api) {
    api.config.store.foo = 'bar';
});
```

`AppTester.setup.`**char_limit**(*n*)

    Sets the character limit checked during the checking phase of the tester run. The default character limit is 160.

    Arguments

> • **n** (*object*) – the new character limit to set.

```
tester.setup.char_limit(20);
```

AppTester.setup.**config**(*obj*)

> Updates the sandbox config with the properties given in obj.

> > **Arguments**

> > > • **obj** (*object*) – the properties to update the current app config with.

> > > • **opts.json** (*object*) – whether these config options should be serialized to JSON.

```
tester.setup.config({foo: 'bar'});
```

AppTester.setup.**config**(*obj*)

> Updates the sandbox's app config (the 'config' field in the sandbox config) with the properties given in obj.

> > **Arguments**

> > > • **obj** (*object*) – the properties to update the current app config with.

```
tester.setup.config.app({name: 'some_amazing_app'});
```

AppTester.setup.**endpoint**(*endpoint*, *delivery_class*)

> Updates the sandbox's app config (the 'config' field in the sandbox config) with the given outbound endpoints.

> > **Arguments**

> > > • **str** (*opts.delivery_class*) – the name of the endpoint to configure

> > > • **str** – the name of the delivery class. See ContactStore.get() for a list of the supported delivery classes.

```
tester.setup.config.endpoint('sms_endpoint', {
    delivery_class: 'sms',
});
```

AppTester.setup.**kv**(*obj*)

> Updates the app's kv store with the properties given in obj.

> > **Arguments**

> > > • **obj** (*object*) – the properties to update the current kv store with.

```
tester.setup.kv({foo: 'bar'});
```

AppTester.setup.**user**(*obj*)

> Updates the currently stored data about the user with the properties given in obj.

> > **Arguments**

> > > • **obj** (*object*) – the properties to update the currently stored user data with

```
tester.setup.user({
    addr: '+81',
    lang: 'jp'
});
```

> If any properties other than addr are given, AppTester assumes that this is an existing user. This effects whether a :class:'UserNewEvent() or *UserLoadEvent()* will be fired during the sandbox run.

`AppTester.setup.`**`user`**`(fn)`
> Passes the currently stored user data to the function `fn`, then set the stored user data to the function's result.

> > **Arguments**

> > > • **`fn`** (`function`) – function of the form `func(user)`, where `user` is the currently stored user data object and `this` is the `AppTester()` instance. The stored user data is set with `fn`'s result. May return its result via a promise.

```
tester.setup.user(function(user) {
    user.addr = '+81';
    user.lang = 'jp';
    return user;
})
```

> If any properties other than `addr` are given, `AppTester assumes that this is an existing user.  This effects whether a :class:`UserNewEvent()` or `UserLoadEvent()` will be` fired during the sandbox run.

`AppTester.setup.user.`**`addr`**`(addr)`
> Sets the from address of the user sending a message received by the sandbox app.

> > **Arguments**

> > > • **`addr`** (`string`) – the user's new from address

```
tester.setup.user.addr('+27987654321');
```

`AppTester.setup.user.`**`answer`**`(state_name, answer)`
> Sets the user's answer to a state already encountered.

> > **Arguments**

> > > • **`state_name`** (`string`) – the name of the state to set an answer for.

> > > • **`answer`** (`string`) – the answer given by the user for the state

```
tester.setup.user.answer('initial_state', 'coffee');
```

`AppTester.setup.user.`**`answers`**`(answers)`
> Sets the user's answers to states already encountered by the user.

> > **Arguments**

> > > • **`answers`** (`object`) – (state name, answer) pairs for each state the user has encountered and answered

```
tester.setup.user.answers({
    initial_state: 'coffee',
    coffee_state: 'yes'
});
```

`AppTester.setup.user.`**`lang`**`(lang)`
> Sets the user's language code.

> > **Arguments**

> > > • **`lang`** (`string`) – the user's new language code (eg, 'en' or 'af')

```
tester.setup.user.lang('af');
```

`AppTester.setup.user.`**`metadata`**`(metadata)`
> Updates the user's metadata. Any properties in the current metadata with the same names as properties in the new metadata will overwritten.

Arguments

- **metadata** (`object`) – The new metadata to update the current user metadata with.

```
tester.setup.user.metadata({foo: 'bar'});
```

`AppTester.setup.user.`**`state`**`(state_name[, opts])`
    Sets the state most recently visited by the user using a state name.

> **Arguments**
>
> - **name** (`string`) – The name of the state.
> - **opts.metadata** (`object`) – metadata associated with the state. Optional.
> - **opts.creator_opts** (`object`) – options to be given to the creator associated with the given state name. Optional.

```
tester.setup.user.state('initial_state', {
    metadata: {foo: 'bar'},
    creator_opts: {baz: 'qux'}
});
```

`AppTester.setup.user.`**`state`**`(opts)`
    Sets the state most recently visited by the user using options.

> **Arguments**
>
> - **opts.name** (`string`) – The name of the state.
> - **opts.metadata** (`object`) – Optional state metadata.
> - **opts.creator_opts** (`object`) – options to be given to the creator associated with the given state name. Optional.

```
tester.setup.user.state({
    name: 'initial_state',
    metadata: {foo: 'bar'},
    creator_opts: {baz: 'qux'}
});
```

`AppTester.setup.user.state.`**`creator_opts`**`(opts)`
    Updates the options passed to the state creator of the state most recently visited by the user.

> **Arguments**
>
> - **opts** (`object`) –
>
>   The new options to update the current creator options with. Any properties in the current creator options with the same names as properties in the new options will over-written.
>
>   States are created typically created twice (on the first sandbox run when we switch to the state, and on the next sandbox run when we give the state the user's input). This makes this setup method useful for setting up the options for the second sandbox run.

```
tester.setup.user.state.creator_opts({foo: 'bar'});
```

`AppTester.setup.user.state.`**`metadata`**`(metadata)`
    Updates the metadata of the state most recently visited by the user.

> **Arguments**

- **metadata** (*object*) – The new metadata to update the current state metadata with. Any properties in the current metadata with the same names as properties in the new metadata will overwritten.

```
tester.setup.user.state.metadata({foo: 'bar'});
```

11.1.2 Interaction Tasks

Interaction tasks are used to simulate interaction with the sandbox. *Input* interactions are the most common, where the sandbox receives a message sent in by a user.

AppTester.**input**(*content*)
> Updates the content of the message to be sent from the user into the sandbox. If the content is `null` or `undefined`, defaults the message's session event to `'new'`, or otherwise to ```'resume'`.
>
> > **Arguments**
> >
> > > - **content** (*string or null*) – the new content of the message to be sent

```
tester.input('coffee');
```

AppTester.**input**()
> Updates the content of the message to be sent from the user into the sandbox to be `null` and defaults the message's session event type to 'new'. Typically used to test starting up a session with the user.

```
tester.input();
```

AppTester.**input**(*obj*)
> Updates the message to be sent from the user into the sandbox with the properties given in `obj`.
>
> > **Arguments**
> >
> > > - **obj** (*object*) – the properties to update on the message to be sent

```
tester.input({
    content: 'coffee',
    session_event: 'resume'
});
```

AppTester.**input**(*fn*)
> Passes the current message data to be sent from the user into the sandbox into the function `fn`, then sets it with the function's result.
>
> > **Arguments**
> >
> > > - **fn** (*function*) – function of the form `func(msg)`, where `msg` is the current message data and `this` is the *AppTester()* instance. The current message is updated with `fn`'s result. May return its result via a promise.

```
tester.input(function(msg) {
    msg.content = 'coffee';
    return msg;
})
```

AppTester.input.**content**(*content*)
> Updates the content of the message to be sent from the user into the sandbox.
>
> > **Arguments**
> >
> > > - **content** (*string*) – the new content of the message to be sent

```
     tester.input.content('coffee');
```

AppTester.input.**session_event**(*session_event*)
    Updates the session event of the message to be sent from the user into the sandbox.

    > **Arguments**

    > > • **session_event** (*string*) – the session event of the message to be sent.

    The following session event values are recognised:

    > • `'new'`: used to signal the start of the session, where the session has been initiated by the user. The content
    >   of the message is irrelevant.

    > • `'resume'`: a common message sent in from the user during a session

    > • `'close'`: used to signal the end of the session, where the session has been terminated by the user. The
    >   content of the message is irrelevant.

```
     tester.input.session_event('resume');
```

AppTester.**inputs**(*input1*[, *input2*[, ...]])
    Sets a collection of messages to be sent from the user into the sandbox. Each input corresponds to a new message
    in a new interaction. *AppTester()* setup methods will count for the first interaction, subsequent interactions
    will rely on api state from the previous interaction, and check methods will only happen after the last interaction.

    > **Arguments**

    > > • **input1, input2, ...** (*arguments*) – The messages to be given as input in each
    > >   interaction. If an object is given for an input, the object's properties are used as the actual
    > >   message properties. `null` or string inputs will be taken as the message content for that
    > >   particular message.

```
     tester.inputs(null, 'coffee', '1', {content: '2'});
```

AppTester.**inputs**(*fn*)
    Passes the current messages to be sent from the user into the sandbox into the function `fn`, then sets it with the
    function's result.

    > **Arguments**

    > > • **fn** (*function*) – function of the form `func(msgs)`, where `msgs` is the current mes-
    > >   sages and `this` is the *AppTester()* instance. The current messages are updated with
    > >   `fn`'s result. May return its result via a promise.

```
     tester.inputs(function(msgs) {
         return msgs.concat('coffee');
     })
```

AppTester.**start**()
    Updates the content of the message to be sent from the user into the sandbox to be null and defaults the message's
    session event type to 'new'. Typically used to test starting up a session with the user.

```
     tester.start();
```

11.1.3 Checking Tasks

Checking tasks are used to check the state of the sandbox application and its currently associated user (the user which
sent in a message to the sandbox application). The check tasks are where the test assertions happen.

AppTester.**check**(*fn*)
> Allows custom assertions to be done after a sandbox run.

> > **Arguments**

> > > - **fn** (*function*) – function that will be performing the assertions. Takes the form func(api, im, app), where api is the tester's api instance (by default an instance of *DummyApi()*), im is the tester's *InteractionMachine()* instance, app is the sandbox app being tested and this is the *AppTester()* instance. May return a promise.

```
tester.check(function(api, im, app) {
    assert.notDeepEqual(api.logs, []);
});
```

> static **interaction**(*opts*)
> > Performs the checks typically done after a user has interacted with a sandbox app.

> > > **Arguments**

> > > > - **opts.state** (*string*) – the expected name of user's state at the end of the sandbox run.

> > > > - **opts.reply** (*string*) – the expected content of the reply message sent back to the user after the sandbox run. Optional.

> > > > - **opts.char_limit** (*integer*) – Checks that the content of the reply sent back to the user does not exceed the given character count. Optional.

```
tester.check.interaction({
    state: 'initial_state',
    reply: 'Tea or coffee?'
});
```

AppTester.check.**ends_session**()
> Checks if the reply message sent to the user was set to end the session. This happens, for example, when the user reaches an *EndState()*.

```
tester.check.reply.ends_session();
```

AppTester.check.**reply**(*content*)
> Checks that the content of the reply sent back to the user during the sandbox run equals the expected content. Alias to *AppTester.check.reply.content()*.

> > **Arguments**

> > > - **content** (*string*) – the expected content of the sent out reply.

```
tester.check.reply('Tea or coffee?');
```

AppTester.check.**reply**(*re*)
> Checks that the content of the reply sent back to the user during the sandbox run matches the regex.

> > **Arguments**

> > > - **re** (*RegExp*) – Regular expression to match the content of the sent out reply against.

```
tester.check.reply.content(/Tea or coffee?/);
```

AppTester.check.**reply**(*obj*)
> Checks that the reply sent back to the user during the sandbox run deep equals obj.

> > **Arguments**

- **obj** (*object*) – the properties to check the reply against

```
tester.check.reply({
    content: 'Tea or coffee?'
});
```

AppTester.check.**reply**(*fn*)

> Passes the reply sent back to the user during the sandbox run to the function fn, allowing custom assertions to be done on the reply.

> **Arguments**

> - **fn** (*function*) – function of the form func(reply), where reply is the sent out reply and this is the *AppTester()* instance.

```
tester.check.reply(function(reply) {
    assert.equal(reply.content, 'Tea or coffee?');
})
```

AppTester.check.reply.**char_limit**(*n*)

> Checks that the content of the reply sent back to the user does not exceed the character count given by n.

> **Arguments**

> - **n** (*integer*) – the character count that the sent out reply's content is expected to not exceed.

```
tester.check.reply.char_limit(10);
```

AppTester.check.reply.**content**(*content*)

> Checks that the content of the reply sent back to the user during the sandbox run equals the expected content. Alias to *AppTester.check.reply.content()*.

> **Arguments**

> - **content** (*string*) – the expected content of the sent out reply.

```
tester.check.reply.content('Tea or coffee?');
```

AppTester.check.reply.**content**(*re*)

> Checks that the content of the reply sent back to the user during the sandbox run matches the regex. Alias to *AppTester.check.reply.content()*.

> **Arguments**

> - **re** (*RegExp*) – Regular expression to match the content of the sent out reply against.

```
tester.check.reply.content(/Tea or coffee?/);
```

AppTester.check.reply.**content**(*content*)

> Checks that no reply was sent back to the user.

```
tester.check.no_reply();
```

AppTester.check.reply.**properties**(*obj*)

> Checks that the expected properties given in obj are equal to the corresponding properties of the reply sent back to the user during the sandbox run.

> **Arguments**

> - **obj** (*object*) – the properties to check the reply against

```
    tester.check.reply.properties({
        content: 'Tea or coffee?'
    });
```

AppTester.check.**user**(*obj*)

Checks that once serialized, the user deep equals `obj`.

> **Arguments**
>
> > • **obj** (`object`) – the properties to check the user against

```
    tester.check.user({
        state: {name: 'coffee_state'},
        answers: {initial_state: 'coffee'}
    });
```

AppTester.check.**user**(*fn*)

Passes the current user instance to the function `fn`, allowing custom assertions to be done on the user. May return a promise.

> **Arguments**
>
> > • **fn** (`function`) – function of the form `func(user)`, where `user` is the current user instance and `this` is the *AppTester()* instance.

```
    tester.check.user(function(user) {
        assert.equal(user.state.name, 'coffee_state');
        assert.equal(user.get_answer('initial_state', 'coffee');
    })
```

AppTester.check.user.**answer**(*state_name*, *answer*)

Checks that the user's answer to a state already encountered matches the expected `answer`.

> **Arguments**
>
> > • **state_name** (`string`) – the name of the state to check the answer of.
> >
> > • **answer** (`string`) – the expected answer by the user for the state

```
    tester.check.user.answer('initial_state', 'coffee');
```

AppTester.check.user.**answers**(*answers*)

Checks that the user's answers to states already encountered by the user match the expected `answers`.

> **Arguments**
>
> > • **answers** (`object`) – (`state_name`, `answer`) pairs for each state the user has encountered and answered

```
    tester.check.user.answers({
        initial_state: 'coffee',
        coffee_state: 'yes'
    });
```

AppTester.check.user.**lang**(*lang*)

Checks that the user's language matches the expected language code.

> **Arguments**
>
> > • **lang** (`string`) – the language code (e.g. 'sw', 'en', 'en_ZA') or `null` to check that no language code is set.

```
    tester.check.user.lang('sw');
    tester.check.user.lang(null);
```

AppTester.check.user.**metadata**(*metadata*)

Checks that the user's metadata after a sandbox run deep equals the expected `metadata`.

> **Arguments**
>
> > • **metadata** (*object*) – the expected metadata of the user

```
    tester.check.user.metadata({foo: 'bar'});
```

AppTester.check.user.**properties**(*obj*)

Checks that the expected properties given in `obj` are equal to the corresponding properties of the user after a sandbox run.

> **Arguments**
>
> > • **obj** (*object*) – the properties to check the user against

```
    tester.check.user.properties({
        lang: 'en',
        state: {name: 'coffee_state'},
        answers: {initial_state: 'coffee'}
    });
```

AppTester.check.user.**state**(*name*)

Checks that the name of the user's state after a sandbox run equals the expected `name`.

> **Arguments**
>
> > • **name** (*string*) – the expected name of the current state

```
    tester.check.user.state('coffee_state');
```

AppTester.check.user.**state**(*obj*)

Checks that the user's state after a sandbox run deep equals `obj`.

> **Arguments**
>
> > • **obj.name** (*string*) – the expected name for the state
> >
> > • **obj.metadata** (*object*) – the expected metadata for the state.
> >
> > • **obj.creator_opts** (*object*) – the expected creator options for the state.

```
    tester.check.user.state({
        name: 'coffee_state',
        metadata: {foo: 'bar'},
        creator_opts: {baz: 'qux'}
    });
```

AppTester.check.user.**state**(*fn*)

Passes the user's state data after a sandbox run to the function `fn`, allowing custom assertions to be done on the state.

> **Arguments**
>
> > • **fn** (*function*) – function of the form `func(state)`, where `state` is the current state instance and `this` is the *AppTester()* instance.

```
    tester.check.user.state(function(state) {
        assert.equal(state.name, 'coffee_state');
    })
```

AppTester.check.user.state.**creator_opts**(*creator_opts*)

>   Checks that the creator options of the interaction machine's current state after a sandbox run deep equals the expected options.

>   **Arguments**

>   >   • **creator_opts** (*object*) – the expected creator_opts of the current state

```
tester.check.user.state.creator_opts({foo: 'bar'});
```

AppTester.check.user.state.**metadata**(*metadata*)

>   Checks that the metadata of the interaction machine's current state after a sandbox run deep equals the expected `metadata`.

>   **Arguments**

>   >   • **metadata** (*object*) – the expected metadata of the current state

```
tester.check.user.state.metadata({foo: 'bar'});
```

11.2 Under the Hood

If need be, one can always add custom task types. AppTester's *setup*, *interaction* and *check* tasks all extend the same class, *AppTesterTasks()*.

class **AppTesterTaskSet**()

>   Manages a set of *AppTesterTasks()*. Used by *AppTester()* to control all its task collections (*setup*, *interaction* and *check* tasks) without needing to interact with each collection individually.

>   static **add**(*name*, *tasks*)

>   >   Adds a task collection to this set of task collections.

>   >   **Arguments**

>   >   >   • **name** (*string*) – the name to be used to identify this collection of tasks.

>   >   >   • **tasks** (*AppTesterTasks*) – the collection of tasks to be added.

>   static **attach**()

>   >   Attaches each of the collections' task methods to their tester. See *AppTesterTasks.attach()*.

>   static **get**(*name*)

>   >   Retrieves the task collection associated with the specified name.

>   >   **Arguments**

>   >   >   • **name** (*string*) – the name to be used to look up the collection of tasks.

>   static **invoke**(*method_name*[, *args*])

>   >   Invokes a method on each task collection in the set, returning the results as an array.

>   >   **Arguments**

>   >   >   • **method_name** (*string*) – the name of the method to invoke on each task collection.

>   >   >   • **args** (*array*) – the arguments to invoke the method with.

>   static **length**

>   >   The total number of currently scheduled tasks in this set.

>   static **reset**()

>   >   Resets all of its collections. See *AppTesterTasks.reset()*.

static **run**()
> Runs the set's task collections' tasks in the order the collections were added in.

static **yoink**()
> Attaches the tester's `api`, `im` and `app` to directly to each of its tasks. See `AppTesterTasks.yoink()`.

class **AppTesterTasks**(*tester*)
> A collection of tasks to be run one after the other.
>
> > **Arguments**
> >
> > > • **tester** (`AppTester`) – the tester that this collection of tasks will be scheduled for.

static **after**()
> Hook invoked after all of the scheduled tasks have been run. May return a promise.

static **attach**()
> Attaches the task collection's methods to the collection's associated tester. Any method defined on the testers `self.methods` attribute will be attached as a method on the tester.
>
> The method attached to the tester is constructed to simply schedule the actual task method. For example, if the task collection has a method `self.methods.foo()`, a corresponding method `tester.foo()` will be constructed. When `tester.foo()` is called, a call to `self.methods.foo()` will be scheduled next time this task collection is run.

static **before**()
> Hook invoked before any of the scheduled tasks are run. May return a promise.

static **length**
> The number of currently scheduled tasks in this collection.

static **reset**()
> Attaches the tester's `api`, `im` and `app` to directly this collection of tasks.

static **reset**()
> Clears the task collection's currently scheduled tasks and stored data.

static **run**()
> Runs the collections's scheduled tasks in the order they were scheduled, then performs a reset. Returns a promise which will be fulfilled once the scheduled tasks have run and the collection has reset itself.

static **schedule**(*name*, *fn*, *args*)
> Schedules a task method to be invoked on the next *AppTesterTasks.run()* call.
>
> > **Arguments**
> >
> > > • **name** (*string*) – the name of the task method to be scheduled
> > >
> > > • **fn** (*function*) – the actual task method
> > >
> > > • **args** (*array*) – the args that the task method will be scheduled to invoke.

static **validate**(*name*[, *args*])
> Optional validator invoked each time a task is scheduled.
>
> > **Arguments**
> >
> > > • **name** (*string*) – the name of the task method to be scheduled
> > >
> > > • **args** (*array*) – the args that the task method will be scheduled to invoke.

AppTesterTasks.after.**each**()
> Hook invoked after each scheduled task has been run. May return a promise.

`AppTesterTasks.before.`**`each`**`()`
> Hook invoked before each scheduled task is run. May return a promise.

class **`TaskError`**(*message*)
> Thrown when an error occurs while trying to schedule or run a task.

> > **Arguments**

> > > • **`message`** (*string*) – the error message.

class **`TaskMethodError`**(*message*)
> Thrown when an error occurs while trying to invoke a task method.

> > **Arguments**

> > > • **`method_name`** (*string*) – the name of the task method associated to the error.

> > > • **`message`** (*string*) – the error message.

DummyApi

12.1 API

class DummyApi (*opts*)

A dummy of the sandbox's real api for use tests and demos.

> **Arguments**
>
> - **opts.http** – Options to pass to the api's *DummyHttpResource()*. Optional.
> - **opts.kv** – The data to initialise the kv store with. Options to pass to the api's *DummyHttpResource()*.
> - **opts.config** – Config data given to the api's *DummyConfigResource()* to initialise the sandbox config with.

> **static config**
>
> The api's *DummyConfigResource()*.

> **static contacts**
>
> The api's *DummyContactsResource()*.

> **static groups**
>
> The api's *DummyGroupsResource()*.

> **static http**
>
> The api's *DummyHttpResource()*.

> **static kv**
>
> The api's *DummyHttpResource()*.

> **static log**
>
> The api's *DummyLogResource()*.

> **static metrics**
>
> The api's *DummyMetricsResource()*.

> **static outbound**
>
> The api's *DummyOutboundResource()*.

class DummyLogResource (*name*)

Handles api requests to the log resource from *DummyApi()*.

> **Arguments**
>
> - **name** (*string*) – The name of the resource. Should match the name given in api requests.

static **critical**
> An array of the messages logged at the 'CRITICAL' log level

static **debug**
> An array of the messages logged at the 'DEBUG' log level

static **error**
> An array of the messages logged at the 'CRITICAL' log level

static **info**
> An array of the messages logged at the 'INFO' log level

static **store**
> An object mapping log levels to the messages logged at that level.

static **warning**
> An array of the messages logged at the 'WARNING' log level

class **DummyConfigResource**(*name*)
> Handles api requests to the config resource from *DummyApi()*.

> **Arguments**

> - **name** (*string*) – The name of the resource. Should match the name given in api requests.

static **app**
> A shortcut to DummyConfigResource.store.config (the app's config).

static **json**
> An object specifying which keys in store should be serialized to JSON when being retrieved using 'config.get'. The default for keys not listed is true.

static **store**
> An object containing the sandbox's config data. Properties do not need to be JSON-stringified, this is done when the config is retrieved using a 'config.get' api request.

class **DummyHttpResource**(*name*, *opts*)
> Handles api requests to the http resource from *DummyApi()*.

> **Arguments**

> - **name** (*string*) – The name of the resource. Should match the name given in api requests.

> - **opts.default_encoding** (*string*) – The encoding to use for encoding requests and decoding responses. Possible values are 'json' and 'none'. If a request's Content-Type header is set, the encoding is inferred using that instead.

static **fixtures**
> The resource's fixture set to use to send out responses to requests. See *HttpFixtures()*.

static **requests**
> A list of http requests that have been sent to the resource, where each is of type *HttpRequest()*.

class **HttpFixture**(*opts*)
> Encapsulates an expected http request and the responses that be sent back.

> **Arguments**

> - **opts.request.url** (string or RegExp()) – The request url. If a string is given, the url may include params. If the params are included, these will be decoded and set as the *HttpRequest()*'s params.

> - **opts.request.method** (*string*) – The request method. Defaults to 'GET'.

> - **opts.request.data** (*object*) – The request's un-encoded body data. Optional.

- **opts.request.body** (*object*) – The request's already encoded body data. Optional.
- **opts.request.params** (*object or array*) – An object of key-value pairs to append to the URL as query parameters. Can be in any form accepted by node.js's querystring module
- **opts.request.headers** (*object*) – An object mapping each header name to an array of header values.
- **opts.response** (*object*) – A single response to use for this fixture, for cases where one request is sent out.
- **opts.response.code** (*integer*) – The response's status code
- **opts.response.data** (*object*) – The responses's decoded body data. Optional.
- **opts.response.body** (*object*) – The response's un-decoded body data. Optional.
- **opts.responses** (*array*) – An array of response data objects to use one after the other each time a new request is sent out.
- **opts.repeatable** (*boolean*) – Configures the fixture's response to be reused for every new request. Defaults to *false*.
- **opts.default_encoding** (*string*) – The encoding to use for encoding requests and decoding responses. Possible values are 'json' and 'none'. If the request's 'Content-Type' header is set, the encoding is inferred using that instead.

Either `opts.response` or `opts.responses` can be specified, or neither, but not both. If no responses are given, an 'empty' response with a status code of 200 is used.

static use()
> Returns the fixture's next unused *HttpResponse()*.

class HttpFixtures(*opts*)
> Manages a set of *HttpFixture()* instances.

> **Arguments**

- **opts.match** (*function*) – A function of the form `f(request, fixture)`, where `request` is the request that needs a match, and `fixture` is the current *HttpFixture()* being tested as a match. Should return `true` if the request and fixture match or `false` if they do not match.
- **opts.defaults** (*boolean*) – Defaults to use for each added fixture.
- **opts.default_encoding** (*string*) – The encoding to use for encoding requests and decoding responses. Possible values are 'json' and 'none'. If a request's 'Content-Type' header is set, the encoding is inferred using that instead.

static add(*data*)
> Adds an http fixture to the fixture set from raw data.

> **Arguments**

- **data** (*object*) – The properties of the fixture to be added. See *HttpFixture()*.

> **Returns** The *HttpFixture()* that was created.

static filter(*request*)
> Finds the fixtures that match the given request.

> **Arguments**

- **request** (`HttpRequest`) – The request to find a match for.

class DummyContactsResource(*name*)

Handles api requests to the contacts resource from *DummyApi()*.

> **Arguments**
>
> > • **name** (*string*) – The name of the resource. Should match the name given in api requests.

static add(*contact*)

Adds an already created contact to the resource's store.

> **Arguments**
>
> > • **contact** (*Contact*) – The contact to add.

static add(*attrs*)

Adds an contact to the resource via a data object.

> **Arguments**
>
> > • **attrs** (*object*) – The attributes to initialise a contact with.

static search_results

An object mapping expected search queries to an array of the matching keys.

static store

A list of the resource's currently stored contacts.

class DummyGroupsResource(*name*)

Handles api requests to the groups resource from *DummyApi()*.

> **Arguments**
>
> > • **name** (*string*) – The name of the resource. Should match the name given in api requests.
> >
> > • **contacts** (*DummyContactsResource*) – The contacts resource associated to this groups resource.

static add(*group*)

Adds an already created group to the resource's store.

> **Arguments**
>
> > • **group** (*Group*) – The group to add.

static add(*attrs*)

Adds an group to the resource via a data object.

> **Arguments**
>
> > • **attrs** (*object*) – The attributes to initialise a group with.

static search_results

An object mapping expected search queries to an array of the matching keys.

static store

A list of the resource's currently stored groups.

class DummyKvResource(*name*[, *store*])

Handles api requests to the kv resource from *DummyApi()*.

> **Arguments**
>
> > • **name** (*string*) – The name of the resource. Should match the name given in api requests.
> >
> > • **store** (*object*) – The data to initialise the store with.

static **incr** (*key*[, *amount*])
> Increment the value of an integer key. The current value of the key must be an integer. If the key does not exist, it is set to zero. Returns the result.
>
> > **Arguments**
> >
> > - **key** (*string*) – The key corresponding to the value to increment
> >
> > - **amount** (*integer*) – The amount to increment by. Defaults to 1.

static **set_ttl** (*key*[, *seconds*])
> Set or remove the ttl (expiry time) of a key.
>
> If seconds is `null` or undefined the key is set not to expire (and its ttl is removed).
>
> > **Arguments**
> >
> > - **key** (*string*) – The key to set the ttl for.
> >
> > - **seconds** (*integer*) – The number of seconds to set the ttl to. Defaults to `null`.

static **store**
> An object mapping all the keys in the store to their corresponding values.

static **ttl**
> An object mapping keys set to expire to their lifetime (in seconds).

class **DummyMetricsResource** (*name*)
> Handles api requests to the metrics resource from *DummyApi()*.
>
> > **Arguments**
> >
> > - **name** (*string*) – The name of the resource. Should match the name given in api requests.

static **add** (*metric*)
> Records a fired metric.
>
> > **Arguments**
> >
> > - **data.store** (*string*) – the name of the metric
> >
> > - **data.metric** (*string*) – the name of the metric
> >
> > - **data.agg** (*string*) – the name of the aggregation method
> >
> > - **data.value** (*number*) – the value to store for the metric

static **agg**
> The aggregation method for metrics with the name `metric_name` that have been fired to the store with the name `store_name`.

static **values**
> An array of the metric values for metrics with the name `metric_name` that have been fired to the store with the name `store_name`.

class **DummyOutboundResource** (*name*)
> Handles api requests to the outbound resource from *DummyApi()*.
>
> > **Arguments**
> >
> > - **name** (*string*) – The name of the resource. Should match the name given in api requests.
> >
> > - **config** (*DummyConfigResource*) – A *DummyConfigResource()* to read configured endpoints from.

static **store**
> An array of the sent outbound message objects.

12.2 Under the Hood

class DummyResource(*name*)

> A resource for handling api requests sent to a *DummyApi()*.
>
> > **Arguments**
> >
> > > - **name** (*string*) – The name of the resource. Should match the name given in api requests
> > >   (for eg, name would be 'http' for http.get api request).
>
> **static handle**(*cmd*)
>
> > Handles an api request by delegating to the resource handler that corresponds to cmd.
> >
> > > **Arguments**
> > >
> > > > - **cmd** (*object*) – The api request command to be handled.
>
> **static handlers**
>
> > An object holding the resource's handlers. Each property name should be the name of the resource handler
> > used in api requests (for eg, 'get' for 'http.get'), and each property value should be a function
> > which accepts a command and returns an api result. For eg:
>
> ```
> self.handlers.foo = function(cmd) {
>     return {
>         success: true,
>         bar: 'baz'
>     };
> };
> ```

class DummyResources()

> Controls a *DummyApi()*'s resources and delegates api requests to correspinding resource. *
>
> **static add**(*resource*)
>
> > Adds a resource to the resource collection.
> >
> > > **Arguments**
> > >
> > > > - **resource** (*DummyResource*) – The resource to be added.
>
> **static attach**(*api*)
>
> > Attaches the resource collection's resources directly onto a *DummyApi()*. Simply a convenience to pro-
> > vide users with direct access to the resource.
> >
> > > **Arguments**
> > >
> > > > - **api** (*DummyApi*) – the api to attach to
>
> **static get**(*name*)
>
> > Returns a resource by name
> >
> > > **Arguments**
> > >
> > > > - **name** (*string*) – The name of the resource
>
> **static handle**(*cmd*)
>
> > Handles an api request by delegating to the corresponding resource.
> >
> > > **Arguments**
> > >
> > > > - **cmd** (*object*) – The api request command to be handled.
>
> **static has_resource_for**(*cmd*)
>
> > Determines whether the resource collection has a corresponding resource for cmd.

Arguments

- **cmd** (*object*) – The command to look for a resource for.

Translation

The toolkit supports internationalization using gettext. Apps have an $ attribute available that they can use when they would like to internationalize their text. Here is a simple example:

```
var SomeApp = App.extend(function(self) {
    App.call(self);
    var $ = self.$;

    self.states.add('states:start', function(name) {
        return new FreeText(name, {
            question: $("Hello! Say something!"),
            next: 'states:end'
        });
    });

    self.states.add('states:end', function(name) {
        return new EndState(name, {
            text: $.dgettext('messages', "That's nice, bye!")
        });
    });
});
```

The gettext methods are well documented in the python docs.

13.1 Under the hood

class **LazyText** (*method*, *args*)

Holds information about text to be translated at a later stage.

Arguments

- **method** (*string*) – The gettext method to use for translation

- **args** (*array or arguments*) – The args given to the gettext method to perform the translation

static **apply_translation** (*jed*)

Accepts a Jed() instance and uses it to translate the text.

Arguments

- **jed** (*Jed*) – The jed instance to translate with

static **context** (*ctx*)
> Sets the context to use in translations.

> > **Arguments**

> > > • **ctx** (*object*) – An object containing the context to be used.

```
$('Hello {{ person }}!').context({person: 'Guy'});
```

class **LazyTranslator** ()
> Constructs *LazyText()* instances holding information for translation at a later stage.

> **Supports the following gettext methods:**

> > • gettext = fn(key)

> > • dgettext = fn(domain, key)

> > • ngettext = fn(singular_key, plural_key, value)

> > • dngettext = fn(domain, singular_ley, plural_key, value)

> > • pgettext = fn(context, key)

> > • dpgettext = fn(domain, context, key)

> > • npgettext = fn(context, singular_key, plural_key, value)

> > • dnpgettext = fn(domain, context, singular_key, plural_key, value)

> For information on how these methods should be used, see: http://slexaxton.github.io/Jed/

> static **support** (*method*)
> > Tells the the translator to support calls to method.

> > > **Arguments**

> > > > • **method** (*string*) – The name of the method to support

class **Translator** (*jed*)
> Constructs functions of the form f(text), where text is a string or a *LazyText()*. If a string is provided, the function acts as a no-op. If a lazy translation is given, the function applies the translation using the translator's jed instance.

> > **Arguments**

> > > • **jed** (*Jed*) – A jed instance or options to initialise such a jed instance to translate with.

> static **jed**
> > Direct access to the translator's Jed() instance.

apply_translation (*jed*, *text*)
> Accepts a jed instance and (possibly lazy translation) text and returns the translation result. If a string is provided, the function acts as a no-op. If a lazy translation is given, the function applies the translation using the jed given in the constructor.

> > **Arguments**

> > > • **jed** (*Jed*) – The jed instance to translate with

> > > • **text** (string or *LazyText()*) – Either a string or an object constructed by one of *LazyTranslator()*'s translation methods.

Sending Messages

OutboundHelper(*im*)

> Provides helpers for sending messages.
>
> > **Arguments**
> >
> > - **im** (`InteractionMachine`) – the interaction machine associated to the helper.

static **delivery_class**

> The fallback delivery class to use when sending to a `Contact()`.

static **send**(*opts*)

> Sends a message to an address or contact.
>
> > **Arguments**
> >
> > - **opts.to** (string or `Contact()`.) – The address or contact to send to.
> >
> > - **opts.endpoint** (`string`) – The endpoint to send to over (for e.g. ′sms′). Needs to be one of the endpoints configured in the app's config.
> >
> > - **opts.content** (string or `LazyText()`) – The content to be sent.
> >
> > - **opts.delivery_class** (`string`) – The delivery class to send over for the contact (for e.g. if ′ussd′ is given, the helper will send to the contact's the contact's ′msisdn′ address). If not given, uses the delivery class configured for endpoint in OutboundHelper.endpoints, finally falling back to OutboundHelper.delivery_class. Irrelevant when opts.to is a string. See ContactStore.get() for a list of the supported delivery classes.
> >
> > - **opts.lang** (`string`) – a letter language code (e.g. sw, en) to translate the content. If not given, the content will be translated to the user's current language.

static **send_to_user**(*endpoint*)

> Sends a message to the current user.
>
> > **Arguments**
> >
> > - **opts.endpoint** (`string`) – The endpoint to send to over (for e.g. ′sms′). Needs to be one of the endpoints configured in the app's config.
> >
> > - **opts.content** (`string`) – The content to be sent.
> >
> > - **opts.lang** (`string`) – a letter language code (e.g. sw, en) to translate the content. If not given, the content will be translated to the user's current language.

Utils

class BaseError()

    An extendable error class that inherits from `Error()`.

    Example usage:

```
var MyError = BaseError.extend(function(self, message) {
    self.name = "MyError";
    self.message = message;
});
```

    BaseError() copies `.extend` from *Extendable()* rather than inheriting it because it inherits from `Error()` already.

class DeprecationError()

    Thrown when deprecated functionality is used.

class Extendable()

    A base class for extendable classes.

class Extendable.**extend**(*Child*)

    Create a sub-class.

        Arguments

            • **Child** (*Class*) – The constructor for the child class.

    Example usage:

```
var MyClass = Extendable.extend(function(self, name) {
    self.my_name = name;
});

var OtherClass = MyClass.extend(function(self, other) {
    MyClass.call("custom_name");
    self.other_var = other;
});
```

basic_auth(*username*, *password*)

    Return an HTTP Basic authentication header value for the given username and password.

        Arguments

            • **username** (*string*) – The username to authenticate as.

            • **password** (*string*) – The password to authenticate with.

exists(*v*)
> Return `true` if v is defined and not null, and `false` otherwise.

> > **Arguments**

> > > • **v** (`Object`) – The value to check.

format_addr(*addr*, *type*)
> Format an address as a standardized string.

> This function delegates to the formatter `format_addr[type]` or returns the address unchanged if there is no custom formatter.

> > **Arguments**

> > > • **addr** (`string`) – The address to format.

> > > • **type** (`string`) – The address type for the address.

format_addr.**gtalk_id**(*addr*)
> Canonicalize a Gtalk address by stripping the device-specifier, if any.

> > **Arguments**

> > > • **addr** (`string`) – The Gtalk address to format.

format_addr.**msisdn**(*addr*)
> Canonicalize an MSISDN by adding a + prefix if one is not present.

> > **Arguments**

> > > • **addr** (`string`) – The MSISDN to format.

functor(*obj*)
> Coerce `obj` to a function.

> If `obj` is a function, return the function. Otherwise return a constant function that returns `obj` when called.

> > **Arguments**

> > > • **obj** (`Object`) – The object to coerce.

infer_addr_type(*delivery_class*)
> Return the address type for the given delivery class.

> A `delivery class` is type of system that messages can be sent or received over. Common values are `sms`, `ussd`, `gtalk`, `twitter`, `mxit` and `wechat`.

> An `address type` is a type of address used to identify a user and corresponds to a field on a `Contact()` object. Common values are `msisdn`, `gtalk_id` and `twitter_handler`, `mxit_id` and `wechat_id`.

> If the `delivery_class` isn't know, the `delivery_class` itself is returned as the `address_type`.

> > **Arguments**

> > > • **delivery_class** (`string`) – The delivery class to look up.

```
The mapping of delivery classes to address types is a low-level
implementation detail that is subject to change. Use higher-level
alternatives where possible.
```

inherit(*Parent*, *Child*)
> Inherit the parent's prototype and mark the child as extending the parent.

> > **Arguments**

> > > • **Parent** (`Class`) – The parent class to inherit and extend from.

- **Child** (*Class*) – The child class that inherits and extends.

is_integer(*v*)

   Return `true` if v is of type number and has no fractional part.

   Arguments

   - **v** (*Object*) – The value to check.

maybe_call(*obj*, *that*, *args*)

   Coerce a function to its result.

   If `obj` is a function, call it with the given arguments and return the result. Otherwise return `obj`.

   Arguments

   - **obj** (*Object*) – The function to call or result to return.

   - **that** (*Object*) – The value of `this` to bind to the function.

   - **args** (*Array*) – Arguments to call the function with.

normalize_msisdn(*number*, *country_code*)

   Normalizes an MSISDN number.

   This function will normalize an MSISDN number by removing any invalid characters and adding the country code. It will return null if the given number cannot be normalized.

   This function is based on the MSISDN normalize function found within the vumi utils.

   Arguments

   - **number** (*string*) – The number to normalize.

   - **country_code** (*string*) – (optional) The country code for the number.

starts_with(*haystack*, *needle*)

   Return `true` if haystack starts with needle and `false` otherwise.

   Arguments

   - **haystack** (*string*) – The string to search within.

   - **needle** (*string*) – The string to look for.

   If either parameter is false-like, it is treated as the empty string.

uuid()

   Return a UUID (version 4).

vumi_utc(*date*)

   Format a date in Vumi's date format.

   Arguments

   - **date** (*obj*) – A value `moment` can interpret as a UTC date.

Test Utilities

fail()
> Raises an AssertionError() with "Expected test to fail" as the error message.

make_im(*opts*)
> Constructs an *InteractionMachine()*. Useful for testing things that a *App()* uses, for e.g. an http api helper for a particular app. All options are optional.

> ### Arguments

>> • **opts.app** (App) – The app to be given to the interaction machine. If not given, a new app is created with a start state of 'start'.

>> • **opts.api** (*object or DummyApi*) – If an options object is given, a new *DummyApi()* is created using those options. Sensible defaults are provided for 'config' and 'kv' if those options are not given.

>> • **opts.msg** (*object*) – The message to setup the *InteractionMachine()* with. Uses sensible defaults if not given.

>> • **opts.setup** (*boolean*) – Whether *InteractionMachine.setup()* should be invoked. Defaults to true.

requester(*api*)
> Returns a promise-based function that makes requests to the given api.

> ### Arguments

>> • **api** (DummyApi) – The api to make requests to.

See also Vumi Go's documentation.

Example Applications

To get you started, here are some example applications that may be useful as an example or reference.

17.1 Basic example

A simple app with a *ChoiceState()* and two *EndState()*s. Take a look to find out how to ask a user if they would like tea or coffee.

17.2 Contacts example

Shows the basics for getting and saving contacts, and how to test contacts-based apps.

17.3 Http example

Shows the basics for making http requests and using the responses.

Indices and tables

- genindex
- modindex
- search

A