Yota Documentation

Release 0.1

Isaac Cook

September 26, 2013

CONTENTS

Yota is a Python form generation library with the following unique features:

- Easy integration of realtime validation. Trigger a server side form validation with any JavaScript event on your input fields. (Client side in planning)
- Dynamic form structures allow for complex forms with on the fly changes. Inject different input fields or validation methods into a specific instance of your Form where needed.
- Default themed with Bootstrap, allowing you to quickly throw together useful forms that look nice.

In addition to these features, Yota also includes most of the features that you would see with other form libraries.

- Simple declarative syntax for defining form validation and layout
- Customizable template driven schemas
- Ability to operate with almost any framework and use any rendering engine. (Default is jinja2)

Philosophically Yota aims to have a ton of flexibility, since designing powerful webforms is infrequently a cookie cutter operation. This was the main problem the designers had with other libraries is that they ended up getting in the way if they wanted to do anything abnormal. At the same time however it is important that sensible default be easy to use and implement, making the creation of common forms trivial and lowering the inital learning curve.

CHAPTER

ONE

OVERALL ARCHITECTURE

Yota allows you to create Forms quickly by declaring a class that is made up of Nodes and Checks. Nodes drive the rendering of your form while Checks drive validation of user input. Yotas power is derived from its integration of server side and client side components, and a growing set of quality default Nodes and Validators.

Form

The primary method of interaction with Yota, the Form class acts as a structure to contain all of the information about your Forms structure and configuration. Forms are usually just a collection of Nodes and Checks with some configuration data. Most method calls will be made on Form objects.

Nodes

Nodes are the actual bits that make up your forms output. Nodes link together rendering templates and neccessary context information. Nodes are very abstract, and could be used to render anything, although most render form elements. The Forms attempts to make a minimum of assumptions about the Nodes attributes.

Validators and Checks

Checks form the bridge between your Nodes and your validators. Validators are supplied with the names of Nodes that are used in the actual Validation callable. At validation time these names are resolved to the actual Node reference.

Renderers

Renderers provide a pluggable interface through which you can render your form. This allows interchange of different templating engines, etc.

CHAPTER

TWO

CONTENTS

2.1 Using Forms

2.1.1 A Simple Form

This is the core of Yota's functionality. To create a Form with Yota you must inherit from the From superclass like in the following example.

```
from yota import Form
from yota.nodes import *

class PersonalForm(Form):

    first = EntryNode()
    last = EntryNode()
    address = EntryNode()
    submit = SubmitNode(title="Submit")
```

Forms are simply a collection of Nodes and Checks. The Checks drive validation of the Form and will be talked about next, while the Nodes drive rendering. Conceptually Nodes can be thought of as a single input section in your form, but it can actually be anything that is destined to generate some HTML or Javascript in your Form. For example you may wish to place a header at the beginning to the Form even though it isn't used for any data entry. Most keyword arguments passed to a Node are passed directly to their rendering context, and thus their use is completely up to user choice. More information on Nodes can be found in the *Nodes* documentation section. Your new Form class inherits lots of functionality for common tasks such as rendering and validation.

To render our Form we can call the `Form.render()` function on an instance of our Form object:

```
>>> personal = PersonalForm()
>>> personal.render()
'<form method="post">
...
</form>'
```

As talked about in the Node documentation, each Node by default has an associated template that is used to render it. The render function essentially passes the list of Nodes in the Form onto the *Renderer*. Most renderers will render each each Node's template and append them all together. In addition to the Nodes that you have defined in your subclass, a Node for the beginning and end of your Form will automatically be injected. The is a convenience that can be disabled by setting the `Form.auto_start_close` to False. We can see this functionality in action in the below example:

```
>>> form = PersonalForm()
>>> for node in form._node_list:
...     print node._attr_name
```

```
...
start
first
last
address
submit
close
```

Even though 'first' was our first element in the Form and 'submit' was our last, the Nodes 'start' and 'close' have been prepended and appended respectively. By default these Nodes load from templates 'form_open.html' and 'form_close.html', however these values can be easily overridden, as can the entire start and close Nodes. For more information see the `Form.auto_start_close`, `Form.start_template`, `Form.close_template`. Passing in a 'start' or 'close' attribute, either through keywords or subclass attributes, will override the default generated Nodes, but it will still place them at the beginning and end.

2.1.2 Validation Intro

To add some validation to our Form we need to create a `Check`. Checks are just containers for Validators and hold information about how the Validator should be executed. The below code will add a Check for the 'first' `Node` to ensure a minimum length of 5 characters.

```
from yota import Form
from yota.nodes import *
from yota.validation import

class PersonalForm(Form):

    first = EntryNode()
    _first_valid = Check(MinLengthValidator(5), 'first')
    last = EntryNode()
    address = EntryNode()
    submit = SubmitNode(title="Submit")
```

The constructor prototype may help provide some reference for the explaination:

When you define a Check object you are essentially specifying a Validator that needs to be run when the Form data is validated, and the information that needs to be passed to said Validator. Attr_args and attr_kwargs should be strings that define what data will get passed into the Validator at validation time. For instance in the above example that data that was entered for the 'first' Node will get passed to the validator. More information on Checks and Validators can be found on the *Validators and Checks* page.

2.1.3 Dynamic Forms

One of the key features of Yota is the ability to make changes to the Form schema at runtime with little effort. For example, say you wanted to make a Form that allowed the user to enter a list of names, and the form included a button that added another field with JavaScript. Or perhaps you would like to create a Form that is slightly different depending on session data. With a dynamic Form schema managing these situations can be much easier.

Since the Form object that is used to run validation after a submission needs to match the Form object that was used to originally render the Form there are some considerations that need to be made. There are of course many ways to try and solve this synchronization problem, but here is a straightforward solution that should apply to most situations.

This section currently needs expansion, however a thoroughly commented example can be found in the yota_examples github repository.

2.1.4 Form API

class yota.**Form**(***kwargs*)

This is the base class that all user defined forms should inherit from, and as such it is the main way to access functionality in Yota. It provides the core functionality involved with setting up and rendering the form.

Parameters

- **context** – This is a context specifically for the special form open and form close nodes, canonically called start and close.
- **g_context** – This is a global context that will be passed to all nodes in rendering thorugh their rendering context as 'g' variable.
- **start_template** – The template used when automatically injecting a start Node. See yota.Form.auto_start_close for more information.
- **close_template** – The template used when automatically injecting a close Node. See yota.Form.auto_start_close for more information.
- **auto_start_close** – Dictates whether or not start and close Nodes will be automatically appended/prepended to your form. Note that this must be set via __init__ or your class definition since it must be set before __init__ for the Form is run.
- **hidden** – A dictionary of hidden key/value pairs to be injected into the form. This is frequently used to pass dynamic form parameters into the validator.

_event_lists = {}

_gen_validate(*data*, *piecewise=False*)

This is an internal utility function that does the grunt work of running validation logic for a Form. It is called by the other primary validation methods.

_node_list = []

_parse_shorthand_validator(*node*)

Loops thorugh all the Nodes and checks for shorthand validators. After inserting their checks into the form obj they are removed from the node. This is because a validation may be called multiple times on a single form instance.

_process_errors()

_processor

This is a class that performs post processing on whatever is passed in as data during validation. The intended purpose of this was to write processors that translated submitted form data from the format of the web framework being used to a format that Yota expects. It also allows things like filtering stripping characters or encoding all data that enters a validator.

alias of FlaskPostProcessor

_renderer

This is a class object that is used to perform the actual rendering steps, allowing different rendering engines to be swapped out. More about this in the section Renderer

alias of JinjaRenderer

_reserved_attr_names = ('context', 'hidden', 'g_context', 'start_template', 'close_template', 'auto_start_close', '_re

_setup_node(*node*)

An internal function performs some safety checks, sets attribute, and set_identifiers

_validation_list = []

add_listener(*listener*, *type*)
Attaches a `Listener` to an event type. These Listener will be executed when trigger event is called.

auto_start_close = True

close_template = 'form_close'

context = {}

data_by_attr()
Returns a dictionary of currently stored `Node.data` attributes keyed by `Node._attr_name`. Used for returning data after its been processed by validators.

data_by_name()
Returns a dictionary of currently stored `Node.data` attributes keyed by `Node.name`. Used for returning data after its been processed by validators.

error_header_generate(*errors*, *block*)
This function, along with success_header_generate allow you to give form wide information back to the user for both AJAJ validated forms and conventionally validated forms, although the mechanisms are slightly different. Both functions are run at the end of a successful or failed validation call in order to give more information for rendering.

For passing information to AJAJ rendering, simply return a dictionary, or any Python object that can be serialized to JSON. This information gets passed back to the JavaScript callbacks of yota_activate, however each in slightly different ways. success_header_generate's information will get passed to the render_success callback, while error_header_generate will get sent as an error to the render_error callback under the context start.

For passing information into a regular, non AJAJ context simply access the attribute manually similar to below.

```
self.start.add_error(
    {'message': 'Please resolve the errors below to continue.'})
```

This will provide a simple error message to your start Node. In practice these functions could also be used to trigger events and other interesting things, although that was not their intended function.

Parameters

- **errors** – This will be a list of all other Nodes that have errors.
- **block** (*boolean*) – Whether or not the form submission will be blocked.

g_context = {}

get_by_attr(*name*)
Safe accessor for looking up a node by `Node._attr_name`

insert(*position*, *new_node_list*)
Inserts a `Node` object or a list of objects at the specified position into the `Form._node_list` of the form. Index -1 is an alias for the end of the list. After insertion the `Node.set_identifiers()` will be called to generate identification for the `Node`. For this to function, `Form._attr_name` must be specified for the node prior to insertion.

insert_after(*prev_attr_name*, *new_node_list*)
Runs through the internal node structure attempting to find a `Node` object whos `Node._attr_name` is prev_attr_name and inserts the passed node after it. If *prev_attr_name* cannot be matched it will be inserted at the end. Internally calls `Form.insert()` and has the same requirements of the `Node`.

Parameters

- **prev_attr_name** (*string*) – The attribute name of the *Node* that you would like to insert after.
- **new_node_list** (*Node or list of Nodes*) – The `Node` or list of Nodes to be inserted.

insert_validator (*new_validators*)
Inserts a validator to the validator list.

Parameters **validator** (*Check*) – The `Check` to be inserted.

json_validate (*data*, *piecewise=False*, *raw=False*)
The same as `Form.validate_render()` except the errors are loaded into a JSON string to be passed back as a query result. This output is designed to be used by the Yota Javascript library.

Parameters

- **piecewise** (*boolean*) – If set to True, the validator will silently ignore validator for which it has insufficient information. This is designed to be used for the AJAJ piecewise validation function, although it does not have to be.
- **raw** (*boolean*) – If set to True then the second return parameter will be a Python dictionary instead of a JSON string

Returns A boolean whether or not the form submission is valid and the json string (or raw dictionary) to pass back to the javascript side. The boolean is an anding of submission (whether the submit button was actually pressed) and the block parameter (whether or not any blocking validators passed)

name = None

render ()
Runs the renderer to parse templates of nodes and generate the form HTML.

Returns A string containing the generated output.

render_error = False

render_success = False

start_template = 'form_open'

success_header_generate ()
Please see the documentation for `Form.error_header_generate()` as it covers this function as well as itself.

title = None

trigger_event (*type*)
Runs all the associated `Listener`'s for a specific event type.

type_class_map = {'info': 'alert alert-info', 'warn': 'alert alert-warn', 'success': 'alert alert-success', 'error': 'alert al
A mapping of error types to their respective class values. Used to render messages to the user from validation. Changing it to render messages differently could be performed as follows:

```
class MyForm(yota.Form):
    first = EntryNode(title='First name', validators=Check(MinLengthValidator(5)))
    last = EntryNode(title='Last name', validators=MinLengthValidator(5)

    # Override the default type_class_map with our own
    type_class_map = {'error': 'alert alert-error my-special-class', # Add an additional cla
                      'info': 'alert alert-info',
                      'success': 'alert alert-success',
                      'warn': 'alert alert-warn'}
```

update_success(*update_dict*, *raw=False*)
This method serves as an easy way to update your success attributes that are passed to the start Node rendering context, or passed back in JSON. It automatically recalls whether the last validation call was to json_validate or validate_render and modifys the correct dictionary accordingly.

Parameters

- **update_dict** – The dictionary of values to update/add.
- **raw** (*bool*) – Whether you would like a pre-compiled JSON string returned, or the raw dictionary.

Returns Return value is either the new JSON string (or raw dict if requested) if json_validate was your last validation call, or a re-render of the form with updated error messages if validate_render was your last call.

validate(*data*)
Runs all the validators associated with the `Form`.

Returns Whether the validators are blocking submission and a list of nodes that have validation messages.

validate_render(*data*)
Runs all the validators on the *data* that is passed in and returns a re-render of the `Form` if there are validation errors, otherwise it returns True representing a successful submission. Since validators are designed to pass error information in through the `Node.errors` attribute then this error information is in turn availible through the rendering context.

Parameters **data** (*dictionary*) – The data to be passed through the *Form._processor*. If the data is in the form of a dictionary where the key is the 'name' of the form field and the data is a string then no post-processing is neccessary.

Returns Whether the validators are blocking submission and a re-render of the form with the validation data passed in.

validator()
This is provided as a convenience method for Validation logic that is one-off, and only intended for a single form. Simply override this function and access any of your Nodes and their data via the self. This method will be called after all other Validators are run.

2.2 Nodes

Nodes drive the actual rendering of your `Form`. Internally a `Form` keeps track of a list of `Node`'s and then passes them off to the *Renderer* when a render of the `Form` is requested. Lets look at a simple example Form as shown in the introduction:

```
from yota import Form
from yota.nodes import *

class PersonalForm(Form):

    first = EntryNode()
    last = EntryNode()
    address = EntryNode()
    submit = SubmitNode(title="Submit")
```

All of the attributes defined in the above class are `Node` instances. Internally there is some trickery that preservers the order of these attributes, but this is not important to understand for using them. Just realize that unlike a regular object in Python, the order of these attributes effects the output of your Form.

Note: Some attribute names are reserved and trying to overwrite them with Node attributes will break things. Ensure that the names you select for your Node attributes do not collide with parameters to `Form` or keyword attributes that you pass to your Form.

The canonical Node is just a reference to some kind of rendering template (by default, Jinaj2 templates) and some associated metadata that will control how the template is rendered.

2.2.1 A Simple Node

Let's examine one of the builtin Nodes availible in Yota, and some of the things we can do with it. Let us look at the `nodes.EntryNode`. It has the following template:

```
{% extends base %}
{% block control %}
<input data-piecewise="{{ piecewise_trigger }}"
       type="text"
       id="{{ id }}"
       value="{{ data }}"
       name="{{ name }}"
       placeholder="{{ placeholder }}">
{% endblock %}
```

Above we see what looks vaugly like HTML. If you're not familiar with Jinja it would be a good idea to give their documentation a cursory glance before proceeding much further. However, the jist is that the sections enclosed in double curly-braces {{ }} will be replaced with variables, while the {% %} enclosed areas represent some sort of control structure. The meat of the above template is the input field. You can see that most of its attributes are replaced by variables.

Now take a look at the extends portion on the first line of our template. This is actually importing another template which is used as the base for many different builtin Nodes in Yota. We can see that template here:

```
<div class="control-group">
    {% block error %}
        {% if errors %}
        <div class="alert alert-error">
            {{ errors[0]['message'] }}
        </div>
        {% endif %}
    {% endblock %}
    {% if label %}
        {% block label %}
            <label class="control-label" for="{{ name }}">{{ title }}</label>
        {% endblock %}
    {% endif %}
    <div class="controls">
        {% block control %}
        {% endblock %}
    </div>
</div>
```

This template is just the default horizontal form layout for Bootstrap. Up top you can see a section reserved for displaying errors and in the middle a section to display a label. At the bottom is where the other template gets injected through the magic of blocks. Again, refer to the Jinja2 documentation for more information on this.

The actual Node definition is basically nothing:

```
class EntryNode(BaseNode):
    template = 'entry'
```

Notice that the template is just entry, not entry.html. This is because the renderer auto-appends the suffix so Nodes can be used across different templating engines.

To understand more of what's going on under the covers, here's some explaination about how the variables used in the above templates are generated.

Identifiers

Some of the values, such as id and name will get automatically generated by `Node.set_identifiers()`, and will be based off of what you name the attribute in you class definition.

Data

The data attribute is automatically populated when validation is run. This is performed by `Node.resolve_data()` and talked about in the Custom Node section below.

Errors

This attriubte is a list of errors generated by validator callables. More about this in validation.

Other

The remained of variables in the above template are just plain old attributes with defaults. Keep in mind that attriubtes/arguments in Yota do not behave quite like they do normally in Python. Learn more about this in Attribute and Argument behaviour in the Form page.

The majority of Node attributes may be overridden either through initialization of the function, like so:

```
my_node = EntryNode(name="Something else", template="custom_entry")
```

Or by setting it as a class attribute in your Node definition like so:

```
class EntryNode(BaseNode):
    template = 'entry'
    _ignores = ['template']
```

However, keep in mind that attributes that are auto-generated, such as name, id, and title should not be set as class attributes since they will get overriden when they are generated. By default, the following attributes are reserved:

- name
- id
- title
- errors
- data
- _attr_name
- _ignores
- _requires

- _create_counter

2.2.2 Custom Nodes

Most Node definitions are quite simple, with the majority simply changing the template being used. More complex Node semantics are availible by overriding some of their built in methods, such as `Node.resolve_data()` or `Node.set_identifiers()`. These are all described in the API documentation, but some examples will be given here of how you might wish to use these methods.

Changing data resolution

The default Node implementation assumes that your Node only contains one input, and as such its data output is assumed to be tied directly to this single input. The `Node.set_identifiers()` method defines a defualt implementation for naming your input field that looks something like this:

```
try:
    self.data = data[self.name]
except KeyError:
    self.data = self._null_val
```

You can see above that the Node's name is used to pick out the data that is associated with this Node. But say your Node includes multiple input fields, perhaps you have a date picker. A simple template may look like this:

```
Month: <input type="text" name="{ name }_month" placeholder="Month" /><br />
Day: <input type="text" name="{ name }_day" placeholder="Day" /><br />
Year: <input type="text" name="{ name }_year" placeholder="Year" /><br />
```

Now of course the `Node.resolve_data()` will fail to find anything associated with "name" since it doesn't exist, and instead an implementation may look something like this.

```
def resolve_data(self, data):
    try:
        day = data[self.name + '_day']
        month = data[self.name + '_month']
        year = data[self.name + '_year']
    except KeyError:
        self.data = self._null_val

    # set data to a tuple of values for validation
    self.data = (year, month, day)
```

Aside from our crappy looking form, and some lack of bounds checking everything is good. Now say we wanted to make this form work with AJAX, and we wanted to make the border of each of the form elements red when there was an error. Well this is a problem, because our JavaScript doesn't implicity know how to find the elements. You could modify your render_error method to manually catch this case, but this wouldn't be a very resiliant option. Instead, we can make our default functions aware of these extra elements. This is done through the json_identifiers method.

Modifying AJAX rendering

`Node.json_identifiers()` is executed by validation methods when sending erros back to the client side via JSON. It is used to give the client side inoformation about where the error data should be placed in the DOM. Essentially your render_error and render_success methods are passed an 'ids' object, and this is a direct serialization of the return from this function. The default render_error and render_success methods expect the following keys:

- 'error_id': This should be an id value of a DOM element that you would like to place your error 'message' in. This is not actually used by default, but is implemented by all builtin Nodes. It corresponds to the DOM element that renders regular errors.
- 'elements': This supplies a list of all ids of form elements in the Node. Error tooltips point to the first element.

set_identifiers

When the Node is added to a Form the set_identifiers method is called to setup some unique names to be used in the template and possibly AJAX. Perhaps you'd like a different semantic for automatically titling your date pickers? Overriding this function may also be wanted if you're writing a Node with multiple form elements in it. This all depends on your preference.

```
def set_identifiers(self, parent_name):
    super(MySuperSpecialNode, self).set_identifiers(parent_name)
    if not hasattr(self, 'title'):
        self.title = self._attr_name.capitalize() + " Very Special"
```

2.2.3 Builtin Nodes

class `yota.nodes.`**`BaseNode`**(***kwargs*)
This base Node supplies the name of the base rendering template that is used for standard form elements. This base template provides error divs and the horizontal form layout for Bootstrap by default through the *horiz.html* base template.

class `yota.nodes.`**`NonDataNode`**(***kwargs*)
A base to inherit from for Nodes that aren't designed to generate output, such as the SubmitNode or the LeaderNode. It must override resolve_data, otherwise the data will be set to `Node._null_val`.

class `yota.nodes.`**`ListNode`**(***kwargs*)

Node for providing a basic drop down list. Requires an attribute that is a list of tuples providing the key and value for the dropdown list items.

Note: The first item of the tuple must be a string in order to match returned data properly and re-select the same list item when a validation error occurs.

Attr items Must be a list of tuples where the first element is the value of the second is the label.

class `yota.nodes.`**`RadioNode`**(***kwargs*)
Node for providing a group of radio buttons. Requires buttons attribute.

Attr buttons Must be a list of tuples where the first element is the value of the second is the label.

class `yota.nodes.`**`CheckGroupNode`**(***kwargs*)
Node for providing a group of checkboxes. Requires boxes attribute. Instead of defining an ID value explicitly the `Node.set_identifiers` defines a prefix value to be prefixed to all id elements for checkboxes in the group. The output data is a list containing the names of the checkboxes that were checked.

Attr boxes Must be a list of tuples where the first element is the name, the second is the label.

class `yota.nodes.`**`ButtonNode`**(***kwargs*)
Creates a button in your form that submits no data.

class `yota.nodes.`**`EntryNode`**(***kwargs*)
Creates an input box for your form.

class yota.nodes.**PasswordNode**(**kwargs)
Creates an input box for your form.

class yota.nodes.**FileNode**(**kwargs)
Creates an input box for your form.

class yota.nodes.**TextareaNode**(**kwargs)
A node with a basic textarea template with defaults provided.

Attr rows The number of rows to make the textarea

Attr columns The number of columns to make the textarea

class yota.nodes.**SubmitNode**(**kwargs)
Displays a submit button on the right side to align with Form elements

class yota.nodes.**LeaderNode**(**kwargs)
A Node that does few special things to setup and close the form. Intended for use in the start and end Nodes.

2.2.4 Node API

class yota.**Node**(**kwargs)
Nodes are holders of context for rendering and displaying validating for a portion of your Form. This default base Node is designed to provide a template along with specific context information to a templating engine such as Jinja2. For validation a Node acts as an information source or an error sink. Essentially Nodes can be used to source data for use in a Check, and they can then be delivered some sort of validation error via a the internal errors attribute.

Note: By default all keyword attributes passed to a Node's init function are passed onto the rendering context. To override this, use the Node._ignores attribute.

Parameters

- **_attr_name** (*string*) – This is how the Node is identified in the Form. If populated automatically if the Node is defined in an a Form class definition, however if the Node is added dynamically it will need to be defined before adding it to the Form.
- **_ignores** (*list*) – A List of attribute names to explicity not include in the rendering context. Mostly a niceity for keeping the rendering context clutter free.
- **_requires** (*list*) – A List of attributes that will be required at render time. An exception will be thrown if these attributes are not present. Useful for things like lists that require certain data to render properly.
- **template** (*string*) – String name of the template to be parsed upon rendering. This is passed into the *Form._renderer* so it needs to be whatever that is designed to accept. Jinja2 is looking for a filename like 'node' that occurs in it's search path.
- **validators** – An optional attribute that specifies a Check object, or list of Check objects to be associated with the Node. This is automatically at render time.
- **_null_val** – When form submission data is passed in for validation and the Node.resolve_data() method cannot identify anything, the data attribute will be set to this value. Defaults to "".

The default Node init method accepts any keyword arguments and adds them to the Node's rendering context. In addition any class attributes may be added to custom Nodes and these attributes will be copied at instantiation time and passed into the rendering context.

_attr_name = None

_create_counter = 0
Allows tracking the order of Node creation

_ignores = ['template', 'validator']

_null_val = ''

_requires = []

add_error(*error*)
This method serves mostly as a wrapper alowing for different error ordering semantics, or possibly error post-processing. Errors from validation methods should be added in this way allowing them to be caught. More information about what gets passed in in the *Validators and Checks* section.

data = ''

errors = []

get_context(*g_context*)
Builds our rendering context for the Node at render time. By default all attributes of the Node are added to the global namespace and the global rendering context is passed in under the variable 'g'. This function is designed to be overridden for customization. :param g_context: The global rendering context passed in from the rendering method.

Parameters g_context – This is the global context passed in from the parent Form object. By default it's included under the 'g' key, similar to Flask's globals.

get_list_names()
As the title suggests this needs to return an iterable of names. These should be names corresponding to form elements that the Node will generate. This list is uesed by piecewise validation to determine if a Node has been visisted based on a list of names that have been visited, bridging Nodes to elements.

json_identifiers()
Allows passing arbitrary identification information to your JSON error rendering callback. For instance, a common use case is the display an error message in a pre-defined div with a specific id. Well you may perhaps pass in an 'error_div_id' attribute to the JSON callback to use when trying to render this error. The default for Yota builtin nodes is to pass 'error_id' indicating the id of the error container in addition to a list containing all input elements in the Node's ids.

label = True

piecewise_trigger = 'blur'

resolve_data(*data*)
This method links data from form submission back to Nodes. HTML form data is represented by a dictionary that is keyed by the 'name' attribute of the form element. Since most Nodes only render a single form element, and the default set_identifiers generates a single 'name' attribute for the Node then this function attempts to find data by linking the two together. However, if you were to change that semantic this would need to change. Look at the CheckGroupNode for a reference impplementation of this behaviour, or the Docs under "Custom Nodes". This method should operate by setting its own data attribute, as this is how Validators conventionally look for data.

... note:: This method will throw an exception at validation time if the data dictionary contains no key name, so it important to override this function to a NoOp if your Node generates no data. NonDataNode was created for this exact purpose.

Parameters data – The dictionary of data that is passed to your validation method call.

set_identifiers (*parent_name*)
This function gets called by the parent *Form* when it is initialized or inserted. It is designed to set various unique identifiers. By default it generates an id for the Node that is {parent_name}_{_attr_id}, a title for the Node that is the _attr_name capitalized, and a name for the element that is just the _attr_name. All of these attributes are then passed onto the rendering context of the Node by default. By default all of these attributes will yield to attributes passed into the __init__ method.

Parameters **parent_name** (*string*) – The name of the parent form. Useful in ensuring unique identifiers on your element names.

template = None

validators = []

2.3 Validators and Checks

Validators allow you to provide users feedback on their input through structured, reusable callables. Validators can be supplied an arbitrary number of inputs as well as dispatch information (errors, warnings, etc) to an arbitrary number of output Nodes.

2.3.1 Using Validators In Your Form

Validators are generally added into your Form schema in a way similar to adding Nodes; that is, by declaring attributes in your Form definition. There is a long syntax that is more explicit as well as a shorthand that can add convenience for simple validators. The explicit declaration can be seen below through the definition of a Check.

```
class MyForm(yota.Form):
    # This syntax shortens up the above explicit syntax for simple
    # validators
    first = EntryNode(title='First name')
    _first_valid = Check(MinLengthValidator(5), 'first')
```

The syntax above defines a single `yota.nodes.EntryNode` and an associated validator that ensures the entered value is at least 5 characters long. This is done through the declaration of a `yota.Check` object. The Check accepts the actual validator as its first argument, followed by the names of Nodes that you will be validating. The above example binds our `yota.validators.MinLengthValidator` to a Node with the attribute name 'first'. Later when we try to validate the Form the string 'first' will be used to lookup our Node and supply the appropriate information to the validator method. Nodes in Yota are identified by their attribute name as given in the class declaration. However, If you later add a *Node* dynamically it will need to specify the _attr_name attribute upon declaration explicitly. More on this in *Dynamic Forms*.

The above syntax gives us some nice power. We can supply that validation method with as many Nodes as we would like in a clear way. But what if we want to write a bunch of validators that only validate a single Node? Then the above is quite verbose, and below shows an implicit declaration that is a nice option for simple validators, and is just syntactic sugar for the above syntax.

```
class MyForm(yota.Form):
    # This syntax shortens up the above explicit syntax for simple
    # validators. An arg of 'first' will automatically be added to the
    # Check object for you.
    first = EntryNode(title='First name',
                      validators=Check(MinLengthValidator(5)))

    # This even more brief syntax will automatically build the Check
    # object for you since it's just boilerplate at this point
```

```
    last = EntryNode(title='Last name', validator=MinLengthValidator(5)

    # This syntax however is just like above. Be aware that your
    # attribute name will not be automatically added since your
    # explicitly defining args
    address = EntryNode(validators=
                Check(MinLengthValidator(9), 'address'))

    # In addition, you can specify a list of validators, or a tuple
    addr = EntryNode(title='Address', validators=[MinLengthValidator(5),
                                                  MaxLengthValidator(25)])
```

Note: If neither kwargs or args are specified and cannot be implicitly determined an exception will be thrown.

2.3.2 Validator Execution

With the regular form validation method `Form.validate_render()` the error values after validation are maintained in *errors* and passed into the rendering context. In your `Node` template, the error can then be used for anything related to rendering and will contain exactly what was returned by your validator.

With either the piecewise JSON validation method or the regular JSON validation method the data will get translated into JSON. This JSON string is designed to be passed back via an AJAX request and fed into Yota's JacaScript jQuery plugin, although it could be used in other ways. Details about this functionality are in the AJAX documentation section.

To continue our example series above, we may now try and execute a validation action on our Form. For this example we will use a Flask view, although the concepts should be fairly obvious and transfer to most frameworks easily.

```
class MyForm(yota.Form):
    # Our same form definition as above but stripped of the now un-needed
    # comments
    first = EntryNode(title='First name',
                      validators=Check(MinLengthValidator(5)))
    last = EntryNode(title='Last name', validators=MinLengthValidator(5)
    address = EntryNode(validators=
                Check(MinLengthValidator(9), 'address'))

# In Flask routes are declared with annotations. Basically mapping a URL to
# this method
@app.route("/ourform", methods=['GET', 'POST'])
def basic():
    # Create an instance of our Form class
    form = MyForm()

    # When the form is submitted to this URL (by default forms submit to
    # themselves)
    if request.method == 'POST':
        # Run our convenience method designed for regular forms
        # 'success' if validation passed, 'out' is the re-rendering of the form
        success, out = form.validate_render(request.form)

        # if validation passed, we should be doing something
        if success:
            # Load up our validated data
            data = form.get_by_attribute()
```

```
            # Create a pretend SQLAlchemy object. Basically, we want to try
            # and save the data somehow...
            res = User(first=data['first'],
                       last=data['last'],
                       address=data['address'])

            # Attempt to save our changes
            try:
                DBSession.add(new_user)
                DBSession.commit()
            except (sqlalchemy.exc, sqlalchemy.orm.exc) as e:
                # An error with our query has occurred, change the message
                # and update our rendered output
                out = form.update_success(
                    {'message': ('An error with our database occurred!')})
    else:
        # By default we just render an empty form
        out = form.render()

    def success_header_generate(self):
        return {'message': 'Thanks for your submission!'}

    return render_template('basic.html',
                           form=out)
```

2.3.3 Making Custom Validators

A validator should be a Python callable. The callable will be accessed through a `Check` object that provides context on how you would like your validator to be executed *in this given instance*. Checks are what provide your validation callable with the data it is going to validate. Essentially they are context resolvers, which is part of what allows Yota to be so dynamic.

When the validation callable is run it is supplied with a reference to a `Node`. The submitted data that is associated with that `Node` will be loaded into the data attribute automatically. At this point, perhaps an example will help clarify.

```
import yota

def MyValidator(node_in):
    if len(node_in.data) > 5:
        node_in.add_error({'message': "You're text is too long!"})

class MyForm(yota.Form):
    test_node = yota.nodes.EntryNode()
    _test_check = yota.validators.Check(MyValidator, 'test_node')
```

In the above exmaple we made a simple validator that throws an error if your input value is longer than 5 characters. You can see the creation of the Check instance in the Form declaration supplies the string 'test_node'. This is indicating the name of the Node that you would like to supply to the Validator as input.

Note: In Yota, all Nodes are uniquely identified by an attribute _attr_name. This gets automatically set to the value of the attribute you assigned the Node to in your Form declaration.

Later when the validator is to be called the string is replaced by a refernce to a `Node` with the specified `Node._attr_name`. The method behind this maddness is that it allows for dynamically adding Nodes at and up until vaildation time, as well as dynamic injection of validation rules themselves. In addition your validation methods can now request as much data as you'd like, and subsequently can disperse errors to any Nodes they are supplied with.

2.3.4 Return Semantics

Validators need not return anything explicitly, but instead provide output by appending error information to one of their supplied Node's errors list attribute via the method `Node.add_error()`. This method is simply a wrapper around appending to a list so that different ordering or filtering semantics may be used if desired. The data can be put into this list is fairly flexible, although a dictionary is recommended. If you are running a JSON based validation method the data must by serializable, otherwise it may be anything since it is merely passed into the rendering context of your templates.

That said, the builtin templtes are setup to recieve specific things. The default templates are setup to look two keys: a dictionary with a single key 'message' which will be printed, and 'type' to denote the alert style. If a type is ommitted then type will default to an error for rendering purposes. Looking at a builtin validator should provide additional clarity.

```
class IntegerValidator(object):
    """ Checks if the value is an integer and converts it to one if it is

    :param message: (optional) The message to present to the user upon failure.
    :type message: string
    """
    # A minor optimization that is borderline silly
    __slots__ = ["message"]

    def __init__(self, message=None):
        self.message = message if message else "Value must only contain numbers"
        super(IntegerValidator, self).__init__()

    def __call__(self, target):
        # This provides a conversion as well as a validation
        try:
            target.data = int(target.data)
        except ValueError:
            # Type can be safely ommitted because this is an error
            target.add_error({'message': self.message})
```

For rendering errors you may notice the _type_class key being looked for in the error.html template. This is generated internally from what you enter as 'type' in your return dictionary. This is resolved by the `Form.type_class_map`, which maps types in the key to classes to be applied in the value. An example usage might be that you'd like to add your own class to the error display.

Note: If you wish to make use of Special Key Values you will be required to use dictionaries to return errors.

2.3.5 Special Key Values

Block
If set to False the validation message will not prevent the form from submitting. As might be expected, a single blocking validator will cause the block flag to return true. This is useful for things like notification of password strength, etc. Errors returned are assumed to be blocking unless specified otherwise.

2.3.6 Builtin Validators

The default pattern for builtin Validators in Yota is to return a dictionary with a key 'message' containing the error. This is also the pattern that the builtin `Node`'s except when rendering errors, and therefore is the recommended format

when building your own validators.

class yota.validators.**MinLengthValidator**(*length*, *message=None*)
Checks to see if data is at least length long.

Parameters

- **length** (*integer*) – The minimum length of the data.
- **message** (*string*) – The message to present to the user upon failure.

class yota.validators.**MaxLengthValidator**(*length*, *message=None*)
Checks to see if data is at most length long.

Parameters

- **length** (*integer*) – The maximum length of the data.
- **message** (*string*) – The message to present to the user upon failure.

class yota.validators.**MinMaxValidator**(*min*, *max*, *minmsg=None*, *maxmsg=None*)
Checks if the value is between the min and max values given

Parameters

- **message** (*string*) – (optional) The message to present to the user upon failure.
- **min** – The minimum length of the data.
- **max** – The maximum length of the data.

class yota.validators.**RequiredValidator**(*message=None*)
Checks to make sure the user entered something.

Parameters **message** (*string*) – (optional) The message to present to the user upon failure.

class yota.validators.**RegexValidator**(*regex=None*, *message=None*)
Quick and easy check to see if the input matches the given regex.

Parameters

- **regex** (*string*) – (optional) The regex to run against the input.
- **message** (*string*) – (optional) The message to present to the user upon failure.

class yota.validators.**EmailValidator**(*message=None*)
A direct port of the Django Email validator. Checks to see if an email is valid using regular expressions.

class yota.validators.**UsernameValidator**(*message=None*)
Quick and easy check to see if a field matches a stamdard username regex. This regex matches a string from 3-20 characters long and composed only of numbers, letters, hyphens, and underscores.

Parameters **message** (*string*) – (optional) The message to present to the user upon failure.

class yota.validators.**PasswordStrengthValidator**(*regex=None*, *message=None*)
A validator to check the password strength.

Parameters

- **regex** (*list*) – (optional) The regex to run against the input.
- **message** (*string*) – (optional) The message to present to the user upon failure.

class yota.validators.**MatchingValidator**(*message=None*)
Checks if two nodes values match eachother. The error is delivered to the first node.

Parameters **message** (*string*) – (optional) The message to present to the user upon failure.

class yota.validators.**IntegerValidator**(*message=None*)
Checks if the value is an integer and converts it to one if it is

Parameters **message** (*string*) – (optional) The message to present to the user upon failure.

2.3.7 Check API

class yota.**Check**(*callable*, **args*, ***kwargs*)
This object wraps a validator callable and is intended to be used in your *Form* subclass definition.

Parameters

- **validator** (*callable*) – This is required to be a callable object that will carry out the actual validation. Many generic validators exist, or you can roll your own.
- **args** (*list*) – A list of strings, or a single string, representing that _attr_name of the *Node* you would like passed into the validator. Once a validator is called this string will get resolved into the Node object
- **kwargs** (*dict*) – Same as args above except it allows passing in node information as keyword arguments to the validator callable.

Check objects are designed to be declared in your form subclass.

node_visited(*visited*)
Used by piecewise validation to determine if all the Nodes involved in the validator have been "visited" and thus are ready for the validator to be run

2.4 Renderers

2.4.1 Custom Templates

Most people end up needing to design templates different from the ones built in at some point. Because of this Yota is setup for specifying a search path for custom templates. By default Yota will only look at its own template directory. It is typical to add a search path that points somewhere within your project. Yota will take the first template it finds that matches, so a simple way to ensure your custom templates are prioritized is to insert your path first in the list. A typical example might look something like this:

```
import os
from yota.renderers import JinjaRenderer

JinjaRenderer.search_path.insert(0, os.path.dirname(os.path.realpath(__file__)) +
                                 "/assets/yota/templates/")
```

2.4.2 Switching Template Sets

Yota provides potential for multiple default template sets. The default template set is designed for use with Bootstrap 2.3.2, but a Bootstrap 3.0 implementation can used by modifying the JinjaRenderer attribute templ_type to 'bs3'. More can be read at renderers.JinjaRenderer.templ_type.

2.4.3 Rendering Engines

By default Jinja2 is the renderer for Yota, however support for other renderes is possible by setting the `Form._renderer` to a different class that implements the proper interface. Currently the default and only option is `renderers.JinjaRenderer`, however other implementations should be easy to write. The default Nodes `Node.template` property lacks a file extension and expects the renderer to auto-append this before calling the template, thus allowing the Node to work accross different renderers.

Renderers are invoked when a render method of a `Form` is executed. currently these include `Form.render()` and `Form.validate_render()`. renderers were designed mainly to allow the interchange of template engines and context gathering semantics.

Renderer Interface

As of now only one method must be implemented by a Renderer: the render method. It accepts two parameters, a list of Nodes to be rendererd in order and a dictionary that contains the global context to include in every template context. Looking at the source for JinjaRenderer will provide some guidence on how you might write your own Renderer.

Switching Renderers

A standard pattern would be to set the Form class object `Form._renderer` attribute allowing the attribute change to be effectively global. This would normally be done in whatever setup function your web framework provides.

2.4.4 JinjaRenderer API

class `yota.renderers.`**JinjaRenderer**

env
Simple lazy loader for the Jinja2 enviroment

render(*nodes*, *g_context*)
Loop over each Node passed in by nodes and render it into a big blob of a string. Passes g_context to each nodes `Node.get_context()`.

search_path = []
The list of paths that Jinja will look for templates in. It scans sequentially, so inserting custom template paths at the beginning is an easy way to override default templates without touching Yota. The default path is appended to the end of this list the first time render is called.

suffix = '.html'
The default template suffix

templ_type = 'bs2'
Allows you to switch the default template set being used. Default templates for JinjaRenderer are stored in /templates/jinja/{templ_type}/. Below code being run before your render method will change templates to Bootstrap 3.0. This is usually run when setting up web framework configs to take effect globally. See the flask example for more information.

```
import os
from yota.renderers import JinjaRenderer

JinjaRenderer.templ_type = "bs3"
```

Note: In order to display errors correctly the `Form.type_class_map` must be overriden so alert-error can be changed to alert-danger for Bootstrap 3.

2.5 AJAX Validation

Yota provides a JavaScript architecture in addition to specializatied validation methods to enable easy construction of AJAX based Form validation. Yota supports two main modes of AJAX based validation: piecewise and on-submit. Piecewise validation attempts to validate a portion of the Form (only the portions that the user has visited) on some kind of trigger event, such as the user moving to the next form element, or keydown on a specific element. On-submit is just what it sounds like: when the form is submitted an asynchronous server call is made and the validation results are rendererd or a success action is executed. These features allow you to give your user faster feedback on what mistakes they made to ease the form filling process.

Note: The AJAX validator examples below rely on using the *id* attribute for a Node. This value is set by the default in `Node.set_identifiers()`.

Server side implementation for AJAX validation is designed to be used with `Form.json_validate()`. When a submission is detected (usually by detecting a POST request), the method can be run to return the json encoded validation results as the response. This response is then in turn parsed by Yota's JavaScript library which can then execute callback functions that you can design. To a user of the library, the implementation differeneces of on-submit and piecewise are minor.

Note: Yota's JavaScript library is desinged as a jQuery plugin, and as such jQuery is also required to use these features.

As is classic for jQuery plugins configuration information is passed to Yota's library through an options object.

`options.`**`render_error`**

This attribute should be a function. It is called whenever new information is recieved about a Node. The status attribute dictates what action should be performed.

param string status This dictates the type of new information that was recieved. The first state is "error", and this means the Node is recieving an error for the first time. Common actions would be to un-hide an error div, or something similar. The second state is "update". This means that an error is currently registered with a Node, however we've recieved another batch of error for that Node. The errors are not necessarily different than the current errors. Finally, "no_error" indicates that there is no longer an error at this Node, and error messages should be removed. This will only be called if there is currently an error registered at the Node.

param object ids This is the return information from the `Node.json_identifiers()` function for the Node with which the error is being registered. It was intented to connect the rendering context that generates the DOM to your JavaScript that will be injecting into the DOM.

param object data This is the json encoded `Node.errors` that should be populated by your validators. More about this can be found in the Node documentation, or the Validation documentation.

`options.`**`render_success`**

This attribute should be a function. It is called when the form submission succeeds, or rather it doesn't block. More information on blocking can be found in the Validators section.

param object data This is information directly generated from your `Form.success_header_generate()` function. It is setup to display some sort of message that applies to the entire form such as an error working with the database, or proper submission of the data. In addition, some special key values can be set to trigger the execution of builtin convenience methods, such as redirection of the browser. More on this in Success Actions.

param object ids This is the return information from the `Node.json_identifiers()` function **for the start Node**. It was intented to connect the rendering context that generates the DOM to your JavaScript that will be injecting into the DOM.

`options.`**`piecewise`**

Whether or not this form should be processed in a piecewise fashion. The default Node teamplte form_open will automatically populate this option when you put 'piecewise' in your global context.

2.5.1 Success Actions

As was touched on in the JavaScript render_success function above, the method `Form.success_header_generate()` can be overriden to perform common post submission actions, or to pass information that you may want to use in your render_success function. The default render_success method will look for a 'message' key in the return value and display this in a Bootstrap success alert, or do nothing if this key is not present. All actions are performed upon successful submission, but prior to the `render_success` method being called. A simple example is shown below.

```
class MyForm(yota.Form):
    first = EntryNode(title='First name',
                      validator=Check(MinLengthValidator(5)))

    def success_header_generate(self):
        return {'message': 'Thanks for your submission!'}
```

Or if we wanted to redirect the user after submitting the form:

```
class MyForm(yota.Form):
    first = EntryNode(title='First name',
                      validator=Check(MinLengthValidator(5)))

    def success_header_generate(self):
        return {'redirect': 'http://google.com/'}
```

Information on the avilible special post-submission actions are below.

Redirection

Include the key 'redirect' in your return dictionary and the browser will be sent to the url specified via method `window.location.replace`.

Google Analytics Logging

Under the key 'ga_run' return a list or tuple of four values, matching the four values used in Google Analytics API function `ga`. More information can be found at the URL below.

https://developers.google.com/analytics/devguides/collection/analyticsjs/events

Clear Form Elements

Pass the key 'clear_element' equal to True and upon submission all input fields in the form will be reset.

Custom Action

Include the key 'custom_success' as a string of valid JavaScript and it will be evaled for you.

2.5.2 On-Submit Validation

A simple on submit validation should be very simple if you're sticking with the default Nodes. These Nodes are already setup to pass the required error div ids and element ids to the client using the default render_error function in Yota's JavaScript library, so all you really need to do is set the global context key 'ajax' to equal True. This activates the JavaScript library.

2.5.3 Piecewise Validation

On-Submit validation only gives the user feedback when he has submitted the Form, but what if we want to provide more instant feedback? Piecewise validation allows us to fire off a server request to validate the form as we're filling it out based on any JavaScript based trigger.

The server side of this implementation is almost identical to On-Submit validation except that you want to pass the key 'piecewise' to the g_context. Again, this simply triggers the JavaScript library to behave slightly different. All builtin Nodes are designed to work out of the box with the default AJAX callback functions.

Validation Tiggers

A per-Node attribute 'piecewise_trigger' allows you to set when you would like the Form to be submitted for incremental validation. This can be any JavaScript event type that your input field supports, and defaults to "blur". Common values may be click, change, dblclick, keyup or keydown.

These event triggers are activated when the Yota jQuery plugin is initially called. It scans all input fields in your Form and attaches an AJAX submit action to the input element based on the value of the attribute "data-piecewise". In the default Nodes this is set by the attribute "piecewise_trigger" as can be seen in the code of the entry.html default template.

```
{% extends base %}
{% block control %}
<input data-piecewise="{{ piecewise_trigger }}"
    type="text"
    id="{{ id }}"
    value="{{ data }}"
    name="{{ name }}"
    placeholder="{{ placeholder }}">
{% endblock %}
```

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX