lizard-structure Documentation

Release 0.2.dev0

Nelen en Schuurmans

December 24, 2012

CONTENTS

Tagline of this app: **structure of Lizard, defined and documented in a REST interface**.

Lizard is a framework for showing water-related information in a web interface. We build most of what's now Lizard in the Python web framework Django. We're now separating the various bits and pieces more formally with a REST web API.

A REST API means you can tie in easier into Lizard with your own software instead of buying into Lizard's whole Python and Django stack.

Lizard-structure provides the documentation on the API. It also provides base view classes for Django to make it very easy to support the API with all the existing Django Lizard apps. The **main goal** however is documentation.

Here is the table of contents:

CHAPTER

ONE

INTRODUCING THE FOUR CORE LIZARD CONCEPTS

Note: Warning, this is work in progress. Even though there currently are four levels, there's a big chance that something is missing. It is very likely that a "FilterFolder" or "FilterLayer" or something like that will be added.

There are four core concepts in Lizard's structure:

Data source

Lizard can connect to many kinds of data. A *Data source* provides such a connection.

Layer tree

Within an *Data source*, there will be one or more basic groups of data. Every group of data is what we call a *Layer tree*.

Layer

Every *Layer tree* has multiple layers in some sort of structure. A *Layer* is most often a map layer, but it doesn't have to be.

Feature

A *Layer* consists of features. A map layer might show water level measurement points: every one of those is a *Feature*.

Note: There are four levels. No more. That's Lizard's structure! You could call it **Lizard's world view**. Most of what we encountered in the Lizard websites of the last couple of years fits this structure. And if you need something extra special, you can just create a regular Lizard Django application and you'll have all the freedom to do weird things that you can wish for.

1.1 Data source

A main Lizard characteristic is that it can show data from many different sources. (With "show" we can mean quite elaborate web interfaces, btw.) For every data source, there is a separate Lizard Django application (currently). One to read FEWS data from a database. Another to read it from a JDBC coupling. One to link to geoserver WMS layers. Another to show river dike calculations.

So in the end, if a Lizard website connects to you via the lizard-structure API, Lizard connects with you as a data source.

You, as a data surce, are the starting point for Lizard to talk to you. You'll give lizard a list of layer trees which it can display in its interface, for instance.

1.2 Layer tree

A layer tree is a large-scale grouping of the data available in an *Data source*. Do not have too many of these. As an example: if your data source provides water level measurements, a good layer tree level might be the water board or municipality or whatever you have as top-level customer. So every municipality becomes a *Layer tree*.

The goal you need to keep in mind here is that a *Layer tree* often translates into a separate page in the Lizard web interface. If that is what you want: fine. If not: you need to re-think what you're calling a layer tree.

1.3 Layer

A layer is best understood as simply a map layer. One of the map layers you place over a google or openstreetmap base map. It doesn't really matter whether it is a WMS layer or geojson or even a simple non-map list of items: for the concept you simply need to think "map layer" and you've got the correct mental picture.

1.4 Feature

If a *Layer* is basically a map layer, a *Feature* is an item on that map layer. A river, a dike segment, a water level measurement. A feature is the lowest useful level of information.

The best way to think about a feature is of something that you can click on on a map. You click it and you get a graph of the data. Or a table with more information. Or a PDF.

And in case the *Layer* wasn't a map layer but just a list of features, it still holds true that a feature is something with a table, graph or PDF. In this case it simply is one of the items in that list.

CHAPTER

TWO

DJANGO BASE VIEWS FOR IMPLEMENTING YOUR API

`lizard_structure.views` provides base views for each of the core concepts defined in *Introducing the four core Lizard concepts*. The basic premise is that lizard-structure only *shows* a data source's structure. There are no edit actions, so no POST/PUT/DELETE: only GET.

For the Django API we use Django REST framework.

Every view has a doctring that can mostly be used as-is by the subclasses that implement the actual functionality. The docstring is rendered by Django REST framework in the html API interface, so the view's docstring is the most important information your API user is going to see. The base views' docstring must be really clear and concise!

You normally do not have to implement any `.get()` method on a view, that is all taken care of. Every view tells you which methods you have to fill in to get the base view working with your data.

2.1 Data source view

Base view for a *Data source*.

class `lizard_structure.views.`**`DataSourceView`**`(`***kwargs*`)`

Information about the data source itself and its list of layer trees.

Use this to discover the layer trees you can show in your user interface. The result is a dictionary with the following items:

about_ourselves Metadata about ourselves, like the software version that generated it. Do not depend on the actual items in here, just display them when desired as background information.

layer_trees The list of available layer trees. A layer tree is a collection of layers you can show on a map or in an overview.

There is one method you always have to implement:

`layer_trees``()`

Return list of layer trees.

Overwrite this method in your subclass and return a list of `lizard_structure.items.LayerTreeItem` instances you create from whatever constitutes a layer tree in your own models. To give you an idea, here are some example layer trees:

- Categories in lizard-wms/lizard-maptree.
- FEWS connections in lizard-fewsjdbc.

If you want to return more information about ourselves than the default:

about_ourselves()
Return metadata about ourselves.

The result should be a flat dictionary, so only key/value pairs. By default "generator" is returned with our package name and version as returned by `our_name_and_version()`.

Normally, you should not have to modify or implement the other methods.

get(*request*, *format=None*)
Return about_ourselves and layer trees as REST response.

our_name_and_version()
Return automatically detected name and version number of our package.

The default should be OK in most cases.

2.2 TODO: Layer tree view

Base view for a *Layer tree*.

class lizard_structure.views.**LayerTreeView**(***kwargs*)
Information about the layer tree and its list of layers.

2.3 TODO: Layer view

Base view for a *Layer*.

class lizard_structure.views.**LayerView**(***kwargs*)
Information about the layer and its list of features.

2.4 TODO: Feature view

Base view for a *Feature*.

class lizard_structure.views.**FeatureView**(***kwargs*)
Information about the feature and most importantly its representations.

CHAPTER

THREE

ITEM DEFINITIONS: WHAT KEY/VALUE PAIRS TO EXPECT

`lizard_structure.items` provides item definitions. An item definition is a formal specification of what kinds of key/value pairs you can expect in a JSON object (or Python dictionary) for such diverse items such as a layer, a menu item, a feature, a data source.

Technically, an item definition is nothing but a Python class that returns a dictionary. It is implemented as a class for the following reasons:

- To make sure you comply to the specification. No undefined keys, no missing mandatory keys.
- To allow for default values.
- To make sure we can generate always-correct always-up-to-date documentation.

3.1 Layer tree

class `lizard_structure.items.`**`LayerTreeItem`**`(`***kwargs*`)`
Item definition for layertrees

Available values

url Optional. (**Default value**: None).

name Name of the item (**Default value**: None).

description

Description of the item, perhaps shown when hovering above it. HTML tags are allowed so you can add links or definition links.

(**Default value**: None).

3.2 Heading in a menu

class `lizard_structure.items.`**`HeadingItem`**`(`***kwargs*`)`
Wrapper/interface for heading objects in a LayerTree/menu.

Fixed values:

menu_type (**Fixed value**: u'heading')

Available values

klass Optional. (**Default value**: None).

extra_data Optional. (**Default value**: None).

heading_level Optional. (**Default value**: 1).

name Name of the item (**Default value**: None).

description

Description of the item, perhaps shown when hovering above it. HTML tags are allowed so you can add links or definition links.

(**Default value**: None).

3.3 Layer (workspace acceptable) in a menu

class lizard_structure.items.**LayerItem**(***kwargs*)

Wrapper/interface for layer/acceptable objects in a LayerTree/menu.

Fixed values:

menu_type (**Fixed value**: u'workspace_acceptable')

Available values

wms_params Optional. (**Default value**: None).

wms_options Optional. (**Default value**: None).

name Name of the item (**Default value**: None).

wms_url Optional. (**Default value**: None).

description

Description of the item, perhaps shown when hovering above it. HTML tags are allowed so you can add links or definition links.

(**Default value**: None).

CHAPTER

FOUR

HELPER FUNCTIONS AND BASE CLASSES

Both `lizard_structure.items` and `lizard_structure.views` have helper functions and base classes. We document them here to keep the view and the item definitions documentation clean.

4.1 Helper base view

class `lizard_structure.views.`**`BaseAPIView`**(***kwargs*)

Base view that provides custom docstring rendering.

You should not have to subclass from `BaseAPIView` yourself, it is only used as a base for the other ones. The custom docstring handling happens by overwriting the `get_description()` expected by Django Rest framework.

`get_description`(*html=False*)

Return the description, optionally as html.

4.2 Base class for building item definitions

class `lizard_structure.items.`**`BaseItem`**(***kwargs*)

Base class for the other items.

Flexible implementation so that we only have to specify the fixed and the default values (as dictionaries).

- Fixed values cannot be set with a keyword argument.
- Default arguments (`None` is fine as value, btw) can be set using keyword arguments, otherwise they get their default values.
- Keys with `None` values are omitted from the resulting dictionary returned by `to_api()`.

`to_api`()

Return our internal dictionary, but strip it of None values first.

4.3 Helper function for generating item definition documentation

lizard_structure.items.**generate_docstring**(*name*, *bases*, *attrs*)

Generate a docstring based on the class's defaults/fixed attributes.

Use this function as a metaclass by adding `__metaclass__ = generate_docstring` to every individual subclass of `BaseItem`.

CHAPTER

FIVE

PROJECT DOCUMENTATION

Lizard-structure project documentation

Tagline of this app: **structure of Lizard, defined and documented in a REST interface**.

Most important project links

- Code is on github: https://github.com/lizardsystem/lizard-structure
- Documentation on readthedocs.org: https://lizard-structure.readthedocs.org/
- Tested on travis: https://travis-ci.org/lizardsystem/lizard-structure .
- Installable via pypi: http://pypi.python.org/pypi/lizard-structure .

Changelog of lizard-structure

0.2 (unreleased)

- Renamed "project" to "layer tree" because it is clearer.
- Added "item definitions" to properly document and specify items such as menu headers and projects. Their end result is a dictionary that will be returned as json by the API.
- Added lots of documentation, including documentation generated from the docstrings. The docstring documentation is carefully managed so that the documentation as a whole remains clear and logical to read.
- Renamed "application" to "data source" as "application" looks too much like "Django application". Inside Lizard, the icons in lizard-ui's interface are also called "application icons", so we don't use this overloaded term here.
- Using version from `setup.py` in the sphinx documentation.

0.1 (2012-12-05)

- Documented the four core Lizard concepts. [reinout]
- Set up documentation generation at https://lizard-structure.readthedocs.org/ . [reinout]

- Set up testing on travis: https://travis-ci.org/lizardsystem/lizard-structure . [reinout]
- Removed lizard-ui dependency. [reinout]
- Initial project structure created with nensskel 1.30.dev0. [reinout]

Credits

- Reinout van Rees started this library.

PYTHON MODULE INDEX

l