Runamic Server

version 1.0

Groep16

May 21, 2017

Contents

Welcome to Runamic Server’s documentation!

server

server package

Subpackages

server.interface package

Submodules

server.interface.closest module

`server.interface.closest.`**`get_id_from_pos`**(`request`)
Respond with the node index closest to the given coordinate.
Query args: lat – coordinate latitude lon – coordinate longitude

`server.interface.closest.`**`get_node_from_pos`**(`request`)
Respond with the node data closest to the given coordinate.
Query args: lat – coordinate latitude lon – coordinate longitude

server.interface.geojson module

class `server.interface.geojson.`**`GeoCoord`**
Bases: **`server.interface.geojson.GEOCOORD`**

static **`from_named`**(`named`)

`server.interface.geojson.`**`respond_path`**(`request_dict`, `path`, `markers`)
Wraps a path into a response object, given the request as sent to the server.
Keyword arguments: request – The request as received by the server path – a set of graph indices. markers – a set of graph indices.
Query arguments: type – the response type (default: “indices”). Possible values are:

> “indices”: return just the list of indices; “coordinates”: return a list of coordinates which are interpretable by the app; “geojson”: return a geojson object. “length” : return length information, not the route. “debug” : useful information for debugging.

server.interface.nodes module

`server.interface.nodes.`**`get_node`**(`request`)
Responds with information about a node.
Query args: index – the index of the node

`server.interface.nodes.`**`in_city`**(`request`)
Responds whether a coordinate is inside the city or not
Query args: lat – coordinate latitude lon – coordinate longitude

server.interface.readme module

`server.interface.readme.`**`readme`**(`request`)

server.interface.routes module

`server.interface.routes.`**`async_exec`**(`graph`, `start`, `end`, `config`)

server.interface.routes.**convert**(*args, **kwargs)
Converts a route into another type
Deprecated and replaced by 'parse'

server.interface.routes.**go_home**(request)
Responds with a route leading the user back to his starting point.
Query args: tag – the route that the user used to run. lon, lat – position of the user. distance – The preferred distance to the starting point.

server.interface.routes.**import_json**(*args, **kwargs)
Processes a json structure that contains route and applies the score in the MongoDB database. The structure should have the following structure: {

'route_id (e.g. 1)' : { 'length': int, 'score': int, 'name': string, 'tags': arrays[string] }
}

The data segment should contain the structure

(can be done through e.g. curl "<domain>/route/import" –data "@filename")

server.interface.routes.**parse**(request)
Convert a tag into another type.
Query args: tag – route to be converted.

server.interface.routes.**rate**(request)
Adds the rating to all edges in a route, and saves it both in the structure and in the database.
Query args: tag – the tag of the route you want to rate rating – a float between 0 and 5

server.interface.routes.**rod**(request)
Responds with a rod starting from an index
This function is mostly useless and should - currently - only be used for testing and/or speed metrics.

server.interface.routes.**route_from_coord**(request)
Responds with a route starting and ending in a certain coordinate
This function uses the lightning rod (Stroobant) algorithm: it generates a rod through the city and then tries different paths going to that rod
Query args: lat – coordinate latitude lon – coordinate longitude type – the response type. See the geojson.respond_path function for more details

server.interface.tests module

class server.interface.tests.**NodeTestCase**(methodName='runTest')
Bases: **django.test.testcases.TestCase**
Test the node interface. Kind of obsolete due to test_urls, but still useful

setUp()

test_get_edge_tuple()

test_get_from()
Fuzzy tests. They test whether the function returns the correct response kind.

test_get_node()
Fuzzy test Test whether indexing yields the correct response type

test_in_city()
Test whether the coordinates are inside/outside the city

server.interface.util module

class server.interface.util.**SerialConn**
Bases: **server.interface.util.SerialConn**

class server.interface.util.**SerialNode**

Bases: **server.interface.util.SerialNode**

server.interface.util.**get_edge_tuple** (_, lat, lon)

server.interface.util.**serialize_node** (graph, index)
Transforms a node index into node information
Keyword args: graph – the current graph index – the node index

Module contents

This module contains all function that are accessible by URL, in other words all functions that form the interface of the REST server

server.logic package

Subpackages

server.logic.city package

Submodules

server.logic.city.city module

class server.logic.city.city.**Coordinate**
Bases: **server.logic.city.city.node**

class server.logic.city.city.**Edge**
Bases: **server.logic.city.city.edge**

class server.logic.city.city.**Mapper**
Bases: **server.logic.city.city.Mapper**
A class that acts as a closure to enhance a graph with XY coordinates

class server.logic.city.city.**Vertex**
Bases: **server.logic.city.city.node**

into_coordinate ()

class server.logic.city.city.**VertexXY**
Bases: **server.logic.city.city.node_xy**

into_coordinate ()

server.logic.city.city.**load** (dirname)
Factory method for creating a graph.
Function args: dirname – name of a directory containing files named "nodes.json" and "ways.json"

server.logic.city.city.**path_length** (graph, nodes, distance_fn=<function <lambda>>)
returns the length of a path in the graph given a node list

server.logic.city.city.**project** (graph, projector)
Creates an enhanced graph with xy coordinates from a graph

Module contents

server.logic.database package

Submodules

server.logic.database.edge_database module

class server.logic.database.edge_database.**EdgeDatabase**
Bases: **object**
Class responsible for saving and loading statistics for edges. Every method in this class is sets up it's own connection to the database. As a result, if you wish to load a lot of data, it is better to use "get_all_edges()" instead of iterating over "get_edge()".

Usage example:

database = EdgeDatabase() database.add_rating(edge_from, edge_to, 6)

add_rating (edge_from, edge_to, rating)
Adds a rating to an edge in the database. If an edge does not exist, it is created. For a bulk version: see 'add_rating_list()' :param edge_from: ID of node on one side. :param edge_to: ID of node on the other side. :param rating: rating to add to this edge total. Type: anything that can be added to a number. :return: Nothing

add_rating_list (edge_list)
Bulk version of 'add_rating()' :param edge_list: List of tuples of the type (edge_from, edge_to, rating) :return: Nothing

get_all_edges ()
Retrieves all edges in the database and returns them in a dict with key (edge_from, edge_to) and values (total_rating, amount_voted, average_rating).

get_amount_voted (edge_from, edge_to)
Looks up an edge in the database and returns the amount of people that voted for it. :param edge_from: ID of node on one side. :param edge_to: ID of node on the other side. :return: average rating for that node. 0 if the edge was not found or has no votes.

get_average_rating (edge_from, edge_to)
Looks up an edge in the database and returns the average rating for it. :param edge_from: ID of node on one side. :param edge_to: ID of node on the other side. :return: average rating for that node. 0 if the edge was not found or has no rating.

get_edge (edge_from, edge_to)
Looks up an edge in the database and returns all the data on it as a tupple: (total_rating, amount_voted, average_rating). :param edge_from: ID of node on one side. :param edge_to: ID of node on the other side. :return: (total_rating, amount_voted, average_rating), None if no edge is found.

get_or_insert_edge (edge_from, edge_to, total_rating=0, amount_voted=0)
Retrieves an edge from the database. If the edge is not present, an new one will be created. :param edge_from: ID of node on one side. :param edge_to: ID of node on the other side. :param total_rating: Total rating to insert of new edge has to be created. :param amount_voted: Amount voted to insert of new edge has to be created. :return: (total_rating, amount_voted, average_rating) of the requested edge.

save_new (edge_from, edge_to, total_rating=0, amount_voted=0)
Saves a new edge into the database. If a node with the same ID's (in the same order) already exists, it will NOT be overwritten. If you wish to initialize a large amount of edges, consider using 'add_rating_list()' with the rating set to 0. For a bulk version: see 'save_new_list()' :param edge_from: ID of node on one side. :param edge_to: ID of node on the other side. :param total_rating: rating of this edge (default 0). :param amount_voted: amount of people that voted for this edge (default 0). :return: Nothing.

save_new_list (edge_list)
Bulk version of 'save_new()'. Note: this bulk version has no default values. :param edge_list: List of tuples of the type (edge_from, edge_to, total_rating, amount_voted) :return: Nothing.

server.logic.database.tests module

class server.logic.database.tests.**DatabaseTestCase** (methodName='runTest')
Bases: **unittest.case.TestCase**

check_edge_in_database (edge_from, edge_to, total_rating, amount_voted)
Utility method for direct checking if an edge is in the database. :return True if an edge with the given parameters is in the database, False otherwise

setUp ()
Set up database class and instruct to use test database.

tearDown ()
Wipe test database.

test_bulk_methods ()
Tests all functionality for save_new_list(), add_rating_list(), get_all_edges().

test_load_methods ()
Tests all functionality for get_average_rating(), get_amount_voted(), get_edge(). Assumes the save methods are working correctly.

test_save_new_and_add_rating ()
Tests all functionality of save_new() and add_rating().

Module contents

server.logic.distance package

Submodules

server.logic.distance.tests module

class server.logic.distance.tests.**ClosestTestCase** (methodName='runTest')
Bases: **django.test.testcases.TestCase**
Class for testing whether the get_closest_edge functions work

test_closest_edge_multi_point ()
This test checks whether the api will localize all street midpoints on the correct egde, so that the closest edge to the point is the actual edge that created the point.

test_closest_edge_single_point ()
This test checks that every node in the graph is closest to an edge that contains the node.

server.logic.distance.util module

Various distance functions.

server.logic.distance.util.**angle** (point_a, mid, point_c)

server.logic.distance.util.**bird_distance** (point_a, point_b)
Simply return the length of a straight line between the two points.

server.logic.distance.util.**distance_to_edge_sqr** (point, edge_start, edge_end)
Retrieves the distance between a segment (edge) and a point.
Function args: point – a Vector of the point edge_start – a Vector of the start point of the edge edge_end – a Vector of the endpoint of the edge Returns – the distance, measured in km

server.logic.distance.util.**dot_distance** (start_a, end_a, start_b, end_b)

server.logic.distance.util.**get_closest_edge** (coord, graph, grid)
Given a coordinate, a list of nodes and a list of edges, return the edge closest to a point.
Function args: coord – object with xy attributes graph – owning graph grid – grid generated from the graph
Returns two nodes, connected according to their owning Graph.

Module contents

server.logic.graph package

Submodules

server.logic.graph.debug module

server.logic.graph.debug.**store_coverage**(grid)
output filled cells to text file "Ghent.txt"

server.logic.graph.debug.**store_graph**(graph)
output city roads to svg file.

server.logic.graph.graph module

class server.logic.graph.graph.**DijkstraIterator**
Bases: **object**
Iterator that yields the edges and nodes of a graph, sorted by distance to the origin.
This class should be instantiated by one of the methods of Graph.

choose(config)
Creates a new iterator that yields a single random node.

filter(rod)
Creates a new iterator that only yields when the destination node is in rod.
Function args: rod – list (!) of node indices.

gen_config(config)
Generate configuration.

next()

root(index)
Yields the shortest path between node index and the starting node.

class server.logic.graph.graph.**Edge**
Bases: **server.logic.graph.graph.Edge**

class server.logic.graph.graph.**Graph**(nodelist, edgelist, from_c=False)
Bases: **object**
Main class for holding information about roads and crossroads. Most of the functionality is contained in the .so file.

add_rating(start_node, end_node, rating)

contains(index)
Checks if a node index is present

generate_dijkstra(start_node, config)

get(index)

get_conn_idval(index)
Returns a list of connections given the index
Connections are returned in (to, data) format for every edge (index, data, to)

get_connids(index)
Returns a list of node indices the node itself is connected to.

get_edges(index)

iter_edges()

Iterates over all edges in a graph, in (id, edge_data, to) format.
Useful for transformation into a new graph

iter_nodes ()
Iterates over all nodes in a graph, in (id, node_data) format.
Useful for transformation into a new graph

list_ids ()
Returns a list of all indices in the graph.

map_graph (vertex_fn**,** edge_fn**)**
Creates a new graph, which has the same structure, but every node and edge data field has the function vertex_fn or edge_fn applied to it.
Function args: vertex_fn – function that maps a node_data to another node_data egde_fn – function that maps an edge_data to another edge_data

class server.logic.graph.graph.**Vertex**
Bases: **server.logic.graph.graph.Node**

server.logic.graph.poison module

class server.logic.graph.poison.**PoisonedGraph** (origin, rod, max_distance, max_value)
Bases: **object**
A class that represents a poisoned graph This class basically overwrites and saves a subset of the original graph, and refers to the original graph itself if the data inside is not found.

generate_dijkstra (start_node**,** config**)**

server.logic.graph.test_example module

class server.logic.graph.test_example.**TestExample** (methodName='runTest')
Bases: **django.test.testcases.TestCase**
Example of how to use this graph library Please read this carefully if you're going to work with it.

example ()

server.logic.graph.tests module

class server.logic.graph.tests.**GraphTestCase** (methodName='runTest')
Bases: **django.test.testcases.TestCase**
Test the size of the GRID

test ()

server.logic.graph.util module

server.logic.graph.util.**distance** (startnode, endnode)
Returns the distance between two nodes in km.

server.logic.graph.util.**haversine** (rad)
Returns the haversine function of an angle in radians.

server.logic.graph.util.**parse** (data)

Module contents

server.logic.grid package

Submodules

server.logic.grid.grid module

class server.logic.grid.grid.**Grid**(context)
Bases: **object**
A class representing a spatial bucket structure.

add_interval(interval)
Adds the owner of an interval to any cell overlapping with the interval.

get(coord)
Returns the content of the cell containing the coordinate

get_xy(coord)
Returns an integer tuple containing the two indices to access a cell.

inside(x, y)
Returns whether the indices are not out of bounds.

class server.logic.grid.grid.**GridBuilder**
Bases: **object**
A class meant for building a grid.

create()
Creates the actual grid.

with_binsize(binsize)
sets the bin size.

with_offset(min_x, min_y)
Sets the grid offset

with_size(width, height)
Sets the grid size

server.logic.grid.interval module

class server.logic.grid.interval.**Interval**
Bases: **server.logic.grid.interval.Interval**
A class representing a twodimensional interval.
Fields: minx, miny, maxx, maxy – lower/upper x/y bound. owner – extra metadata to recognise an interval on retrieval. Usually

contains a tuple of graph nodes (see Graph).

into_grid(bin_size)
Create a grid stretching over the interval, with the given bin size

server.logic.grid.interval.**into_interval**(startnode, endnode, tolerance)
Creates an interval that contains the two nodes with a certain tolerance

server.logic.grid.test_example module

class server.logic.grid.test_example.**GridTestCase**(methodName='runTest')
Bases: **django.test.testcases.TestCase**
Example of how to use a grid.

setUp()

test_grid_addition ()
Test grid addition

test_grid_creation ()
Test grid creation

class server.logic.grid.test_example.**TestNode**
Bases: **server.logic.grid.test_example.Node**

Module contents

server.logic.projection package

Submodules

server.logic.projection.util module

class server.logic.projection.util.**Coordinate**
Bases: **server.logic.projection.util.Coordinate**
A class representing a position projected on a plane.

static **from_named** (node)

into_vector ()

class server.logic.projection.util.**Projection**
Bases: **server.logic.projection.util.Projection**
A class representing a node projected on a plane.

static **from_named** (node)

into_vector ()

class server.logic.projection.util.**Projector** (projection_vector, up_vector)
Bases: **object**
A class that projects nodes on a plane.
It is meant for transforming nodes, with lat/lon coordinates, to Projections, with x/y coordinates, making calculations like distance and angle less accurate, but far easier to implement and cheaper to calculate.

map (node)

class server.logic.projection.util.**Vector**
Bases: **server.logic.projection.util.Vector**
A class containing a 3D vector.

cross (other)

dot (other)

lensqr ()

scale (const)

unit ()

server.logic.projection.util.**get_center_node** (vector_list)
Returns a vector that is of unit length, and about in the middle of a vector cluster.

server.logic.projection.util.**project_city** (node_list)
Returns a list of Projections mirroring a list of Nodes which represent a city.

server.logic.projection.util.**vector_from** (lat, lon)

Transforms lon/lat spherical coordinates into a 3D vector.
for example: (0, 0) -> (1,0,0) (0, 90) -> (0,1,0) (90, *) -> (0,0,1)

Module contents

server.logic.routing package

Submodules

server.logic.routing.compress module

`server.logic.routing.compress.`**`decode`** (`string`)
Decodes the string generated by encode back into an integer

`server.logic.routing.compress.`**`encode`** (`integer`)
Encodes an (very large) integer into a string.

`server.logic.routing.compress.`**`from_string`** (`graph`, `string`)
Creates a rod from a graph and tag.

`server.logic.routing.compress.`**`into_string`** (`graph`, `rod`)
Transforms a list of indices into a tag.

server.logic.routing.config module

class `server.logic.routing.config.`**`RoutingConfig`**
Bases: **`server.logic.routing.config.RoutingConfig`**
The class containing all possible config options for routing.
Keep it up to date with server.config.DEFAULT_ROUTING_CONFIG

`server.logic.routing.config.`**`from_dict`** (`default`, `dictionary`)
Generates a routing configuration from a dictionary

server.logic.routing.directions module

class `server.logic.routing.directions.`**`Direction`**
Bases: **`server.logic.routing.directions.Direction`**
Class for serializing into the direction type

static **`from_index`** (`graph`, `index`, c)

class `server.logic.routing.directions.`**`DirectionDict`**
Bases: **`object`**

`server.logic.routing.directions.`**`get_direction`** (`node_angle`, `threshold`, `left`, `right`)
Return right or left depending on the angle
Function args: node_angle – list of three points forming an angle. threshold – minimum (-maximum) value of the sine of the angle

in order to count
left, right – return values in case of success

`server.logic.routing.directions.`**`into_directions`** (`graph`, `nodelist`, `dirdict`)
Annotate a nodelist with directions
Returns: list of coordinates with direction metadata.

server.logic.routing.poison module

`server.logic.routing.poison.`**`poison_graph`** (`graph`, `venomous_path`, `config`)
Poisons a graph

Function args: graph – the graph venomous_path – the path from which the poison originates config – routing configuration

server.logic.routing.ratings module

`server.logic.routing.ratings.`**`add_rating_list`**(`graph`, `edges`, `rating`)
Adds a rating to an edge in the database. If an edge does not exist, it is created.
Function args: graph – the graph edges – the edges which will get a rating rating – the score which is passed

server.logic.routing.routing module

This file implements the Lightning Rod algorithm for route generation

The algorithm consists of three steps:

1. A "rod" is generated, a shortest path between the starting node and a random node at a set distance.
2. Various routes are generated from the starting node towards the rod. When they collide, the path formed by the lower part of the rod and the generated path is tested on length. If the length satisfies the total length constraint, it is added to the list of possible results.
3. The best path, using other cost metrics, is selected.

`server.logic.routing.routing.`**`annotate_rod`**(`rod`, `graph`, `distance_fn`)
Returns a dictionary with information about distances from the rod to the starting node.

`server.logic.routing.routing.`**`close_rod`**(`graph`, `start_node`, `rod`, `routing_config`, `alt_rod=None`)
Generates a closed route given a rod

`server.logic.routing.routing.`**`generate_rod`**(`graph`, `start_node`, `routing_config`)
Generates a random rod.

server.logic.routing.tests module

class `server.logic.routing.tests.`**`TestEncodeDecode`**(`methodName='runTest'`)
Bases: **`django.test.testcases.TestCase`**
Test whether encoding or decoding actually works

`test_encode_decode`()

class `server.logic.routing.tests.`**`TestPoison`**(`methodName='runTest'`)
Bases: **`django.test.testcases.TestCase`**
Test whether poisoning doesn't change the graph

`poison`()

class `server.logic.routing.tests.`**`TestUtil`**(`methodName='runTest'`)
Bases: **`django.test.testcases.TestCase`**
Test whether the ground and unground functions work

`test_ground_unground`()

server.logic.routing.util module

class `server.logic.routing.util.`**`RandomChooser`**
Bases: **`object`**
Deprecated

`pop`()

`push`(`weight`, `item`)

server.logic.routing.util.**ground** (visited_node_dict, end_node)
'grounds' a rod: it returns a list of nodes given a dict of nodes pointing to their precessor.
For example: {2 : 1, 3 : 2, 5 : 3}, 5 -> [1, 2, 3, 5]

server.logic.routing.util.**unground** (visited_nodes)
Performs the inverse 'grounding' operation

Module contents

Submodules

server.logic.server_util module

server.logic.server_util.**into_json** (struct)
Transforms a named tuple into a json object in a nice way.

server.logic.server_util.**time_fn** (lbd)
Times an expression.

If you need to time the operation:

```
result = heavy_function()
```

Then just write:

```
result = time_fn(lambda : heavy_function())
```

Module contents

This module contains all logic for generating the dynamic data.

server.static package

Submodules

server.static.city module

Module contents

Submodules

server.config module

server.setdebug module

server.settings module

Django settings for server project.

Generated by 'django-admin startproject' using Django 1.10.5.

For more information on this file, see https://docs.djangoproject.com/en/1.10/topics/settings/

For the full list of settings and their values, see https://docs.djangoproject.com/en/1.10/ref/settings/

server.test_urls module

class server.test_urls.**TestUrls** (methodName='runTest')
Bases: **django.test.testcases.TestCase**
Very coarse grained tests. Mainly test whether certain requests cause certain response types. Occasionally also test whether certain requests yield predefined responses.

setUp ()

test_urls ()
Kind of a predefined fuzzy tester: All these url's are valid and should return a valid, formatted response.

server.urls module

server URL Configuration

The *urlpatterns* **list routes URLs to views. For more information please see:**
https://docs.djangoproject.com/en/1.10/topics/http/urls/

Examples: Function views

1. Add an import: from my_app import views
2. Add a URL to urlpatterns: url(r'^$', views.home, name='home')

Class-based views

1. Add an import: from other_app.views import Home
2. Add a URL to urlpatterns: url(r'^$', Home.as_view(), name='home')

Including another URLconf

1. Import the include() function: from django.conf.urls import url, include
2. Add a URL to urlpatterns: url(r'^blog/', include('blog.urls'))

server.wsgi module

WSGI config for server project.

It exposes the WSGI callable as a module-level variable named `application`.

For more information on this file, see https://docs.djangoproject.com/en/1.10/howto/deployment/wsgi/

Module contents

Indices and tables

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

Index

T

U

V

W

Python Module Index

s