Sim-City-CS Documentation

Release 0.8

Berend Weel

Nov 03, 2017

Contents

Re-GIS (Research Environment for GIS) is an environment for visualizing data on maps. It was developed by the Netherlands eScience Center and TNO.

Contents

1.1 Installation

The installation of Re-GIS consists of six steps. Re-GIS relies on the external programs `docker`, `docker-compose` and `git`. If you already have these programs on your computer, you may skip the three first steps.

1. Install `docker`. The docker documentation explains how to install docker.

2. Install `docker-compose`. The docker-compose documentation outlines how to install docker-compose.

3. Install `git`. The git webiste explains how to install git.

4. Download Re-GIS from the git website:

```
git clone https://github.com/ReGIS-org/regis-stack.git
```

This creates a new directory with the name `regis-stack` with all Re-GIS software

5. Start Re-GIS server via docker-compose:

```
cd regis-stack
docker-compose up -f docker-compose-demo-small.yml --build
```

Note: The first time this will take a long time as it needs to build the docker images.

The server can stopped by pressing `ctrl-c` in the same window

6. Examine Re-GIS in a webbrowser:

```
open http://localhost:8008
```

An alternative is to open the address http://localhost:8008 in a webbrowser.

1.2 Re-GIS Demo

These instructions presume that you have Re-GIS installed and running on your system. See the installation section if you need information about installing and running Re-GIS

When the Re-GIS server has been started with `docker-compose up --build`, you can access Re_GIS via a web browser at the address http://localhost:8008 .

1.2.1 Select a layer

Re-GIS will display a map and offer you the opportunity to put a transparent layer with data on the map. To select a layer:

1. click on the ☰, top left of the page

2. click on the ◆ in the new menu, second from the left

3. select the Fire Paths layer in the new drop down menu

4. move the map to the south of the country of India: left click, hold and drag to move the map left/right/up/down and use the mouse wheel to zoom in and out

5. you will find a blue path in the city of Bangalore

1.2.2 Create a new layer

To create a new layer:

1. click on the ☰, top left of the page

2. click on the ⚗ logo (fourth from left)

3. click on the ✎

4. drag the pin to an area on the map

5. click on Show Simulation Form

6. press Submit in the new window

7. on the left, click on the top blue eye

8. click on the **|three slides of paper|**

9. select the test.geojson item at the bottom

You will see a data layer with the pin logo with a yellow circle around it (.. image:: ../simcity-commonsense/projects/simcity-demo/images/pin.png) at the position selected by you. You can repeat these actions to select more positions with or without surrounding circles of various sizes.

1.3 Creating a New Project

This document describes the steps necesarry to create a new project within the sim-city web interface.

1.3.1 Before you start

In this document we assume that you have the sim-city webinterface and its prerequisites installed and running, if not see *Installation*

1.3.2 Create Project

In the following examples we will call the new project MyProject, replace this with your own project name.

Creating your project consists of 3 steps:

- Creating a project directory
- Add your project to the projects.json file.
- Creating a project.json

Project Location

Projects in the sim-city-cs framework are located in public/data/projects/

Create a project directory

Create your project directory in the public/data/projects directory

From the sim-city-cs directory:

```
cd public/data/projects
mkdir MyProject
```

Add the project to the projects.json file

Open the projects.json file located in public/data/projects

At the bottom of this json file you will find a key called *projects*. This is a list of project objects. Add your project here with the following template:

```
"projects": [
    ...
    {
    "title": "MyProject",
    "url": "data/projects/MyProject/project.json"
    }
    ...
]
```

Create a project.json

Use the project.json at the end of this page as a basis for your new project. Below I go into the several parts of the project.json.

Please note that json does not support comments. If you want to copy and paste parts of the commented sections be sure to remove the comments. Also keep in mind that parts are sometimes ommitted for readability.

Main Structure

Below is the main general structure for a project in the sim-city-cs framework.

```
{
    "title": "MyProject",                       # The project Title. Any string
    "description": "Description for MyProject", # The project Description. Any string
    "url": "https://github.com/MyProject",      # Url of the project, if any
    "isDynamic": true,                          # isDyanamic means the changes to the
                                                # project that are made using the user
                                                # interface are saved to the disk
    "expertMode": 3,                            # Level of expertise of the use.
                                                # For the simcity example project 3
→is used

                                                # Beginner      = 1
                                                # Intermediate = 2
                                                # Expert        = 3
                                                # Admin         = 4
    "userPrivileges": {
        "mca": {                                # Set expert mode for Multi-Criteria
→Analysis
            "expertMode": true
        },
        "heatmap": {                            # Set expert mode for Heatmap
            "expertMode": true
        }
    },
    "baselayers": {},                           # Extra Baselayers to include
                                                # see the section on baselayers
    "dashboards": [],                           # List of dashboards for the project
                                                # see the section on dashboards
    "groups": [],                               # A list of layer groups
                                                # see the section on groups
    "simAdmin": {                               # Extra data for the simulation
→administrator
        "webserviceUrl": "/explore",            # The url for the sim-city-webservice
                                                # Using the docker stack this is /
→explore
                                                # which is short for http://localhost/
→explore
        "simulationName": "MySimulation",       # The name of the default simulation
        "simulationVersion": "latest"           # The name of the default version
    }
}
```

Input Format

Re-GIS works with GeoJSON or JSON input files. Other GIS standards include GML, SHP, KML, CSV. For more information on the formats and convertion, see, e.g. https://www.datavizforall.org/transform/.

Dashboards

The *dashboards* section describes which dashboards a project has, it is a list of dashboard objects. Each dashboard can hold one or more widgets described in the widgets subsection.

The sim-city example project has two dashboards: The Home dashboard and the Job Monitor dashboard.

```
"dashboards": [{
    "id": "home",                         # ID of the dashboard
    "name": "Home",                       # Name of the dashboard
    "editMode": false,                    # Whether it starts in edit mode
    "showMap": true,                      # Whether to show the map
    "showTimeline": false,                # Whether to show the timeline
    "showLeftmenu": true,                 # Whether to show the menu on the left
    "showLegend": true,                   # Whether to show the legend
    "showBackgroundImage": false,         # Whether to show the background layers
    "visiblelayers": [                    # List of layers that are visible by default
        "fireresponse"                    # This is a list of layerids
    ],                                    # These layers are defined further in the document
    "widgets": [                          # List of widgets on the dashboard, explained
→below
        ...
    ],
    "visibleLeftMenuItems": [             # Which of the menu items in the left menu start
        "!lm-layers"                      # as visible. Default is layers
    ]
},
...
]
```

Widgets

The *widgets* section is a list of widgets included in the *dashboard*. In the simcity example project the home dashboard has a *buttonwidget* to allow the user the drag and drop features for the simulation on the map and a *simulation-form* widget for the form to submit a simulation.

```
"widgets": [{
    "id": "1086ec94-4c54-4d84-dc04-9d3673df6d35",   # The id of the widget,
                                                     # must be unique to the
→project
    "directive": "buttonwidget",                     # Which angular directive to
→use
    "enabled": true,                                 # Enabled by default?
    "style": "transparent",                          # The display style
    "left": "435px",                                 # How far from the left to
→display it
    "right": "",                                     # How far from the right to
→display it
    "top": "82px",                                   # How far from the top to
→display it
```

```
      "width": "300px",                          # The width of the widget
      "data": {                                  # Data that will be passed to␣
→the widget
          "layerGroup": "MyProject",
          "buttons": []
      },
      "collapse": false                          # Whether or not to hide the␣
→widget
                                                 # at the start

   },
   ...
   ]
```

Groups

A group is a set of layers grouped together under a common name. These groups are displayed under a collapsable name in the left menu under layers.

```
{
   "id": "MyLayer",                           # The id of the layer group
   "languages": {                            # There is some support for␣
→multiple
                                             # languages
       "en": {
           "title": "My Layer",              # The name of the layer gropu␣
→in english
           "description": "My Awesome Layer" # Description of the layer␣
→group
       }
   },
   "layers": [                               # List of layers to include
                 ...
   ]
},
...
```

Example Layer

```
{
   "id": "MyLayer",                           # id of the layer
   "reference": "mylayer",                    # Reference name
   "languages": {
       "en": {
           "title": "My Layer",              # Layer name
           "description": "My Description"   # Layer description
       }
   },
   "type": "GeoJson",                        # Type of the data in the␣
→layer
                                             # GeoJSON (default), TopoJSON,
→ or WMS

   "url": "resources/myData.json",           # Location of the data. Can␣
→be a url
```

```
                                                   # or a path relative to the␣
→public folder

    "typeUrl": "resources/myTypes.json",          # Location of the resource␣
→type
                                                   # description. For more␣
→information on
                                                   # this see the resource type
                                                   # documentation.

    "enabled": false,                             # Whether the layer is␣
→enabled by default

    "opacity": 50                                 # The opacity (e.g. inverse␣
→transparancy)
                                                   # of the layer
},
```

More about resource type JSON, see section *Resource type json*.

It is also possible to define a layergroup where the layers are taken from an external server. For instance using ows:

```
{
    ...
    "clustering": true,                           # Clustering means the␣
→features of
                                                   # different layers are␣
→combined and
                                                   # stored in one big list

    "layers": [],                                 # Layers can be empty

    "owsurl": "http://my.url.to/an/ows/server",   # Url of the ows server

    "owsgeojson": true                            # Let the system know this is␣
→an OWS
                                                   # layer group
}
```

Baselayers

You can include extra baselayers on top of those defined in the projects.json file. Below is an example for Open-StreetMap (which is already defined in the projects.json file, this is only to illustrate).

```
{
    "title": "OpenStreetMap HOT",
    "subtitle": "Road",
    "url": "http://{s}.tile.openstreetmap.fr/hot/{z}/{x}/{y}.png",
    "isDefault": false,
    "minZoom": 0,
    "maxZoom": 19,
    "cesium_url": "http://c.tile.openstreetmap.fr/hot/",
    "cesium_maptype": "openstreetmap",
    "subdomains": ["a", "b", "c"],
    "attribution": "Tiles courtesy of <a href='http://hot.openstreetmap.org/' target=
→'_blank'>Humanitarian OpenStreetMap Team</a>",
```

```
    "preview": "http://b.tile.openstreetmap.fr/hot/11/1048/675.png"
}
```

Full Project json file

```
{
    "title": "MyProject",
    "description": "Description for MyProject",
    "url": "https://github.com/MyProject",
    "isDynamic": true,
    "expertMode": 3,
    "userPrivileges": {
        "mca": {
            "expertMode": true
        },
        "heatmap": {
            "expertMode": true
        }
    },
    "baselayers": {},
    "dashboards": [{
        "id": "home",
        "name": "Home",
        "editMode": false,
        "showMap": true,
        "showTimeline": false,
        "showLeftmenu": true,
        "showLegend": true,
        "showBackgroundImage": false,
        "visiblelayers": [
            "fireresponse"
        ],
        "widgets": [{
            "id": "1086ec94-4c54-4d84-dc04-9d3673df6d35",
            "directive": "buttonwidget",
            "enabled": true,
            "style": "transparent",
            "left": "435px",
            "right": "",
            "top": "82px",
            "width": "300px",
            "data": {
                "layerGroup": "MyProject",
                "buttons": []
            }
        },
        {
            "id": "simulation-form",
            "directive": "sim-form",
            "enabled": true,
            "style": "transparent",
            "left": "435px",
            "right": "",
            "top": "180px",
            "bottom": "25px",
            "width": "450px",
```

```
            "data": {
                "layerGroup": "MyProject"
            },
            "collapse": true
        }],
        "visibleLeftMenuItems": [
            "!lm-layers"
        ]
    },
    {
        "id": "monitor",
        "name": "Job Monitor",
        "editMode": false,
        "showMap": false,
        "showTimeline": false,
        "showLeftmenu": false,
        "showLegend": false,
        "showBackgroundImage": true,
        "visiblelayers": [
        ],
        "widgets": [{
            "id": "9086ec94-4c54-4d84-dc04-9d3673df6d35",
            "directive": "sim-summary",
            "enabled": true,
            "style": "transparent",
            "left": "50px",
            "right": "",
            "top": "82px",
            "width": "300px"
        },
        {
            "id": "cb86ec94-4c54-4d84-dc04-9d3673df6d35",
            "directive": "sim-job",
            "enabled": true,
            "style": "transparent",
            "left": "375px",
            "right": "",
            "top": "82px"
        }
        ],
        "visibleLeftMenuItems": []
    }
    ],
    "groups": [
    {
        "id": "MyProject",
        "languages": {
            "en": {
                "title": "MyProject",
                "description": "MyProject layers manipulation buttons"
            }
        },
        "layers": []
    }],
    "simAdmin": {
        "webserviceUrl": "/explore",
        "simulationName": "MySim",
        "simulationVersion": "latest"
```

```
    }
}
```

1.4 Adding a New Simulation

Adding a new simulation consists of the following steps:

1. Setting up your simulation.

2. Adding the simulation to the slurm docker.

3. Configuring the webservice.

1.4.1 Setting up your simulation

In order to run your simulation using the sim-city stack it needs to adhere to a few conventions.

One command Your simulation needs to be able to be run with one command. If your simulation requires several commands to be called in succession you should use a bash script to call the commands one after the other.

One json input file The simcity webservice will supply your simulation with one json input file. This input file will be placed in an input directory. This input directory is available as an environment variable *$SIMCITY_IN*, or can be supplied as an argument to your script if set up correctly in the webservice. For more information on the standards of GIS data, see *Input Format*.

One output directory Any output of your simulation should be put in the designated output directory, avaiable as an environment variable: *$SIMCITY_OUT*. As with the input directory the webservice can be set up so this is provided as an input to the simulation command.

Temp directory A temporary directory for the job will be created and is available through the environment variable *$SIMCITY_TMP*. Again this can also be supplied as an argument to the command.

GeoJson output Any output that is geo-spatial information that you would like to display on the map using the frontend should be in geojson format. For more information check out the geojson website.

In general the following environment variables are available for your script:

variable	Description
$SIMCITY_JOBID	Unique identifier for this j
$SIMCITY_IN	Input directory
$SIMCITY_OUT	Output directory
$SIMCITY_TMP	Tmp directory
$SIMCITY_PARAMS	Path of the input.json file

1.4.2 Adding to Slurm docker

Add the code of your simulation to the *simcity-slurm/simulations* directory. Preferably in an seperate subdirectory, for instance *simcity-slurm/simulations/mysim*.

This directory is copied during building the docker image into */home/xenon/simulations/* inside the docker image. E.g. your simulation will be available from the path */home/xenon/simulations/mysim/* when you ssh into the docker container.

If your simulation depends on certain libraries to be installed you need to edit the *simcity-slurm/Dockerfile*. In the Dockerfile you will find the following snippet:

This line installs the java 7 jre. If your simulation does not use java you can remove it here and replace it with other ubuntu packages. If you use python I would suggest to create a virtual environment and use pip to install your requirements, for instance from a requirements.txt.

```
RUN cd /home/xenon/mysim \
    && virtualenv mysim \
    && . mysim/bin/activate \
    && pip install -U pip \
    && pip install -r requirements.txt
```

For more information on running commands in a dockerfile please refer to the dockerfile manual.

1.4.3 Configuring the webservice

To configure the webservice you will need to ad a json file to the webservice docker that describes the input to the simulation. This serves the purpose of letting the front end know which fields to display for input and enables us to validate the input parameters before sending it to the simulation.

The description of the input parameters for your simulation should be in json schema format. We render the input form using Angular schema form. More detail is given in section Simulation json.

Furthermore the description of the simulation points to a *resourceTypeUrl*, this is a file that is served by the webservice as well that describes the geojson output of your simulation so the front-end can display it nicely. This is discussed in section Resource type json.

Simulation json

Similar to adding your code to the slurm docker image you create a json file in the *simcity-webservice/simulations* directory. This json file should have the following layout:

```
{
    "latest": "0.2",      # latest and stable are two standard labels that you should
→include
    "stable": "0.1",      # the default label for the webservice is 'latest'
    "mylabel": "two",     # You can define any label you like however and link it to
→one of the
                          # full definitions below
    "0.1": {              # Each version should be a complete description of the input
→for your
        ...               # simulation as explained below.
    },
    "0.2": {
        ...
    },
    "two":                # These are just strings, so they can be anything.
        ...               # Semantic versioning (cf. http://semver.org/) might not be a
→bad
                          # idea though.
    }
}
```

The description of your simulation should have to the following layout.

```
"0.1": {
    "command": "~/simulations/mysim/run_mysim.sh",      # The command to run the
→simulation
```

```
    "parallelism": "*",                              # The number of cores the␣
↪simulation uses by itself, * means all.
                                                     # this allows sim-city-client␣
↪to run multiple instances of your
                                                     # simulation on the same node␣
↪if the number of cores allows.
    "resourceTypeUrl": "/explore/resource/mysim",    # Url for the resource type␣
↪json file. This can be any url, but
                                                     # the next section explains␣
↪how to add it to the webservice
    "form": [                                         # Optional description of how␣
↪to display the form for submitting
        ...                                          # a simulation of this type.␣
↪Required if using geo coordinates as
    ],                                               # an input.
    "properties": {                                  # json-schema description of␣
↪the input.
        ...
    },
    "required": [                                     # List of required fields.
        ...
    ]
}
```

Properties

The properties describe to the system which parameters your simulation uses and what their type is. The example below shows one such a parameter called *populationSampleFactor* which is a of the number type it has a maximum and a minimum and a default value. These are used by the system to check input before running your simulation as well as to **render the form on the interface**.

Below are two examples of parameters, please refer to the json schema website and this guide.

```
"properties": {
    "populationSampleFactor": {           # Example of a parameter that is a number
        "type": "number",
        "minimum": 0,
        "maximum": 1,
        "default": 0.1,
        "title": "Commute factor",
        "description": "portion of the population (totalling 8.5 million) that␣
↪commutes"
    },
    "fireStations": {                     # Example of an array parameter
        "title": "Fire stations",
        "minItems": 0,
        "type": "array",
        "startEmpty": true,
        "items": {                        # With an array parameter each item must be
            "type":"object",             # described as well
            "properties": {              # Each item in this case has an x and y␣
↪coordinate
                "id": {                  # as well as an id. This is an example of a␣
↪geo-
                    "type":"string"      # coordinate
                },
```

```
                "x": {
                    "type":"number"
                },
                "y": {
                    "type":"number"
                }
            },
            "required": ["x","y"]
        },
        "description": "Please add one or more fire stations to the map",

        # This message is shown when the form does not validate on this field
        "validationMessage": "Please add at least one fire station"
    }
}
```

Form

Form is an array in the description that is used by angular json schema form to render the form. The order of this array determines the order of the fields in the form.

Below is an example. populationSampleFactor does not have any special configuration. fireStations however is special, it has a number of configuration fields, both for the configuration of its display as well as to let the frontend know this is a geo-coordinate input.

Most important here is that its type is "layer", this means the front-end expects this input to be given on a special layer. The name of this layer is given in the "layer" field, this layer is created automatically when this simulation is selected for the front-end.

This in combination with the resourceType description the front-end creates drag-and-drop buttons to add this feature to the input layer.

```
"form": [
    "populationSampleFactor",
            ...
    {
        "key": "fireStations",          # The key used in the form
        "startEmpty": true,             # Do not add a default first item
        "add": null,                    # Do not put an add button in the form
        "remove": null,                 # Do not put a remove button in the form
        "type": "layer",                # Special type to tell the frontend that this
→field
                                        # comes from a geojson layer
        "layer": "test_sim",            # The name of the layer
        "featureId": "FireStation",     # featureId of this type. See Resource type
→json section
        "items": [
            {
                "type": "point2d"       # Special display of this type for each item
→that can be
            }                           # defined in sim-city-cs
        ]
    }
]
```

Resource type json

The resource type description is also a json file. This json file describes the various types of data that are used in your simulation, both in the input and the output.

Only properties that are defined in the *propertyTypeData* and are in the *propertyTypeKeys* of a featureType are available for non-admin users to display and filter on.

Below is an example resource type description which describes three different feature types: FireStations, Fires and Wards. FireStations and Fires are used for the input to the simulation, while wards is the output of the simulation.

The FireStation and Fire feature types describe a unique id and name for these types, as well as some *propertyTypeKeys*, these keys reference the *propertyTypeData* section lower in the file. The style description tells the front end how to display this feature type, which is also used to drag-and-drop these features on the map. In this case it defines a "point" drawing mode using an icon as a display.

The Ward feature type also describes a unique id, a name and a number of *propertyTypeKeys*. In this case the drawing mode is "polygon" which means a shape on the map. The most obvious options for drawing modes are: Point, MultiPoint, Polygon, MultiPolygon, Line and PolyLine.

The *propertyTypeData* section describes the features properties, this is used in the display of the features properties in the right sidebar in the user interface. As said before, describing your features here is crucial to allow non-admin users to display and filter different properties.

```
{
    "id": "matsim",
    "title": "matsim",
    "featureTypes": {
        "FireStation": {
            "id": "SimCity#firestation",
            "name": "FireStation",
            "style": {
                "drawingMode": "Point",
                "iconUri": "images/brandweerposten/Brandweerkazerne.png",
                "cornerRadius": 50,
                "fillColor": "#ffffff",
                "iconWidth": 30,
                "iconHeight": 30,
                "strokeColor": "#ffffff"
            },
            "propertyTypeKeys": "title,notes",
            "u": "bower_components/csweb/dist-bower/images/marker.png"
        },
        "Fire": {
            "id": "SimCity#fire",
            "name": "Fire",
            "style": {
                "drawingMode": "Point",
                "iconUri": "data/images/fire.png",
                "cornerRadius": 50,
                "fillColor": "#ffffff",
                "iconWidth": 30,
                "iconHeight": 30,
                "strokeColor": "#ffffff"
            },
            "propertyTypeKeys": "title,notes",
            "u": "bower_components/csweb/dist-bower/images/marker.png"
        },
        "Ward": {
```

```
            "id": "SimCity#Ward",
            "name": "Ward",
            "style": {
                "nameLabel": "ward_name",
                "drawingMode": "Polygon",
                "cornerRadius": 50,
                "fillColor": "#999999",
                "iconWidth": 30,
                "iconHeight": 30,
                "strokeColor": "#ffffff"
            },
            "propertyTypeKeys": "ward_name;ward_no;cmc_mc_nm;tot_p;first_responder;
↪second_responder",
            "u": "bower_components/csweb/dist-bower/images/marker.png"
        }
    },
    "propertyTypeData": {
        "ward_no": {
            "label": "ward_no",
            "type": "text",
            "title": "Ward Number",
            "visibleInCallOut": true,
            "canEdit": false,
            "isSearchable": true,
            "section": "Metadata"
        },
        "ward_name": {
            "label": "Name",
            "type": "text",
            "title": "Name",
            "visibleInCallOut": true,
            "canEdit": false,
            "isSearchable": true,
            "section": "Metadata"
        },
        "cmc_mc_nm": {
            "label": "cmc_mc_nm",
            "type": "number",
            "title": "City Name",
            "canEdit": false,
            "isSearchable": true,
            "visibleInCallOut": true,
            "section": "Metadata"
        },
        "first_responder": {
            "label": "first_responder",
            "type": "number",
            "title": "First Responder",
            "canEdit": false,
            "isSearchable": true,
            "visibleInCallOut": true
        },
        "second_responder": {
            "label": "second_responder",
            "type": "number",
            "title": "Second Responder",
            "canEdit": false,
            "isSearchable": true,
```

```
        "visibleInCallOut": true
    },
    "tot_p": {
        "label": "tot_p",
        "type": "number",
        "title": "Total Population",
        "canEdit": false,
        "isSearchable": true,
        "visibleInCallOut": true
    }
},
"isDynamic": false
}
```

1.4.4 Troubleshooting

My simulation does not run Please check if your simulation run script is executable from within the docker container. To do this start the regist stack with *docker-compose up –build* then ssh into the docker container using *ssh -p10022 xenon@localhost* using password javagat. Best is to debug your simulation now by running it inside the container in this manner.

If your simulation is running in this manner check whether there is a problem with the paths of where simcity-client is calling your simulation.