# Cityflows toolbox

v.1

The Cityflows toolbox is aiming to help traffic specialists (engineers) mainly working for the public authorities / cities in the context of mobility enhancement and control. At the current stage the main added value is in the simplicity of making relatively complex statistical analysis based on multi-source data. Our persona should be able to not only get some results from the black box but understand (on a desired level) what is happening inside, so that the outcomes can be used with a confidence.

## General toolbox architecture

The mobility data is usually hardly accessible for general public and thus obtained by our final users (cities) on a case-by-case basis from both public and private actors. With that in mind we have opted to a full separation of toolbox instances between different deployments: each client/case deployment is a new one, without mutualization of resources. From one side it creates an entry barrier: there is no always-on platform already running when a new case is started; from the other side we get a full traceability of both data flows and the resources consumed. In order to simplify the above entry barrier, toolbox is deployed as a collection of Docker containers thus heavily limiting the number of installation steps to follow as well as the compatibility with different platforms.

Taken into account that the toolbox is focusing on a scientific analysis of the data, it was comfortable to make a use of different python libraries available for those purposes. As we were hardly imagining our persona to operate with codes, we were aiming at a user-friendly GUI and to maximize the possible adoption – the web-based application approach was selected. These remarks resulted in the following collection of Docker containers representing the full deployment of the Cityflows toolbox:

* The backend based on Django framework. A well established and supported by the community framework with an extremely versatile collection of libraries, heavily simplifying the development (see below).
* The frontend is based on React JS framework, which is one of the top options for frontend development at the moment.
* After the analysis of the data to be stored / queried for data storage we have opted to PostgreSQL database.

## Frontend container

After logging in to the toolbox, the user can interact with two main parts of the frontend:

* the general map view, which is used for an overview and the selection of the sensors to work with
* the analysis view: from one side here it is possible to further finetune the selection of sensors (the desired modalities and / or directions) and time windows the user is interested, from the other side in different tabs, the user can see the results of the analysis performed on the data he/she has selected.

The map view is built on top of Leaflet JS map framework and is pretty straightforward. It simply receives the collection of available sensors from the API (formal description in the dedicated section). The analysis view is a bit different. There is a collection of pre-defined building blocks, which are previewed (textual information, different chart types, map visualization, …) and depending on the selection made by the user, which are sent to the backend via an API, the latter one returns which building blocks and with which contents should be shown to the user. So there is a strong bias towards the logic control on the backend side, while the frontend is used solely for the interaction with the user.

Main libraries used in the frontend:

* React JS – the global framework
* Leaflet JS – map visualizations
* Chart JS – visualizations of different charts

The frontend container runs without any persistent storage.

The following environment variables should be configured at the launch time:

* REACT\_APP\_BACKEND\_URL: full URL (with protocol), where the backend responds
* REACT\_APP\_OAUTH\_CLIENT\_ID and REACT\_APP\_OAUTH\_CLIENT\_SECRET: a pair of id/secret used for authentication purposes (together with the user login/password). Usually a pair of randomly generated strings
* REACT\_APP\_INITIAL\_MAP\_CONFIG\_JSON: a JSON string, configuring the initial map position in the interface. Example: {"center":[51.003543,3.708080],"zoom":14}
* REACT\_APP\_INITIAL\_TIME\_WINDOWS: a JSON string, configuring the initial time windows pre-populated when the user enters to the toolbox. Example: [{"from":"2021-09-01T00:00:00.000Z","to":"2021-11-01T00:00:00.000Z","Monday":true,"Tuesday":true,"Wednesday":true,"Thursday":true,"Friday":true,"Saturday":true,"Sunday":true,"Holiday":true,"Non-holiday":true,"DBSCAN":true,"MinThreshold":true,"PerformanceThreshold":true}]

The webserver started in the container listens to the port 3000. It should be forwarded to external proxy (taking care of https handling).

## Database

As mentioned above, we have opted for PostgreSQL database, which is completely controlled from the backend side (including the migrations management).

The toolbox was tested with the version tagged postgres:15.1-alpine, available at the Docker hub.

The path “/var/lib/pgsql/data” should be mounted to a persistent storage.

Environment variable POSTGRES\_PASSWORD should be set to a randomly generated string, while the POSTGRES\_USER variable should be set to “cf”.

## Backend container

For obvious reasons in this part most of the logic is contained. The data is managed via a creation of high-level Django models, which are then translated to and managed in the database. Let’s focus first on the structure of models.

### Data Models

The main object of the analysis consists of a collection of measurement (counts) coming from different sensors. As we are working with the traffic data, those counts can be split by transport type (modality) and/or direction. In order to comfortably handle the above-mentioned data, the following model types are created:

* Sensor Type – stores the name of the data provider and used to group together similar sensors. Generated automatically during the data upload (if new)
* Modality – stores the name of the modality and used to classify different types of counts. These are also generated automatically during the data upload (if new). Currently the frontend has pictograms for “Pedestrian”, “Bike”, “Car”, “Background” (total population observed, without modality split). Of course more modalities can be easily added: for example a split to heavy trucks/buses/…
* Sensor – a high level structure directly containing the meta information (unique identifier, location, …) and grouping together the *Sensor Tracks*
* Sensor Track – a single channel of a sensor counting a fixed modality with optional split in direction: if a sensor counts 2 directions separately for a specific modality, there will be 2 different sensor tracks (one with isReverse = True and one with False) for this modality. Sensor tracks group together the hourly and daily counts.
* Hourly Count – a single measurement for a particular sensor track at a fixed date and hour. Contains the total number of observed instances during this hour.
* Daily Count – a sum of hourly counts for a particular date and a particular sensor track. Used to increase the speed of calculations which don’t need an hourly level of temporal precision.

Here we emphasize the fact that the highest temporal resolution stored in the toolbox is equal to 1 hour. That choice was done as from one side the majority of data sources give only this level of temporal resolution and from the other side, the finer resolution (if available) usually has rather noisy behavior. Having the same time resolution makes things directly comparable among sensors. The step of agglomeration of more fine-resolution signal is *a priori* the only preprocessing step required before the data is uploaded to the toolbox.

It is totally fine if not all hourly measurements are present. Those can be absent due to various reasons, like technical issues, privacy reasons or functioning type (for example Telraam uses the camera image processing to generate the counts and thus has limited functionality at night due to the visibility issues). To raise the user awareness of such situations, each sensor track goes through a collection of automated tests to detect which days are comparable to a normal behavior of this particular sensor track and which – not. The results of these tests are stored in the Data Quality Test Result models (on a daily level per sensor track and per Quality Test type). These tests are passed at each data upload. There are 2 important remarks to note here:

1. Data quality is understood here in terms of the coming signal persistence, without focus on the values returned. I.e. if a sensor always gives 8 hourly counts per day, that is considered as a persistent behavior even if on day 1 there were 1000 people observed and on day 2 – 0. The reason of not considering outliers as faulty measurements is that we are potentially interested exactly in those outliers during the analysis (example: an event happening on a particular day).
2. With respect to the above-mentioned tests, the missing measurement has a very different meaning of the measurement equal to 0 count. This means that for sure it is bad to artificially add 0 counts instead of missing measurements. In case it is possible to diversify between a missing count and a 0 count for a specific provider, it is always worth the effort.

The user can easily filter out the sensor data not passing some of the tests in the time-window configuration in the frontend. This potentially permits to focus on the most qualitative parts of the available data.

Together with the above-mentioned application specific models, backend also creates more generic models needed for the Django framework itself (mostly related to the users management) or for some of the used libraries (probably the only relevant to note here is the OAuth2 provider configuration).

### API endpoints for Frontend

The first part of the API for frontend contains everything related to the OAuth2 authentication. Here we completely rely on the Django Oauth Toolkit ( <https://django-oauth-toolkit.readthedocs.io/> ) without any amendments. Thus for further details on this part it is better to consult the original documentation. We limit here to the following remarks:

* All the paths are prefixed by BACKEND\_URL/o/
* Django Oauth Toolkit is rfc-compliant and there is a possibility to use external token providers. As in the current use cases the technical team and the final users are coming from different organizations it was not worth to use the single-sign-on options, that is why we have kept the user’s management on the Django side.

The “content-related” API is served via a unique URL BACKEND\_URL/api/api . It expects a BEARER token in the header and a JSON payload in the request body. The return is in the JSON format as well.

As the URL is common for different functions, the obligatory parameter in the the JSON payload is “endpoint”. Below there is a list of valid endpoints and their specifications.

#### Sensors Collection endpoint

**Body params:**

{

“endpoint”: “getSensorsCollection”

}

**Expected result:**

{

"status":"ok",

"sensorCards": ARRAY\_OF\_AVAILABLE\_SENSORS\_DETAILS

}

For each available sensor the following dictionary is expected:

{

"ref": SENSOR\_UNIQUE\_REFERENCE,

"location": GEOJSON\_LOCATION\_PROPERTIES,

"modalities": ARRAY\_OF\_AVAILABLE\_MODALITIES\_NAMES,

"hasReverse": BOOLEAN\_VALUE\_SPECIFYING\_IF\_TWO\_DIRECTIONS\_ARE\_AVAILABLE,

"sensorType": { “name” : PROVIDER\_NAME },

"extraFields": { “addressString” : “” }

"quality": PERCENTAGE\_OF\_DAYS\_PASSING\_ALL\_TESTS

}

extraFields dictionary is optional (subject to availability) and can have different meta information inside. Other fields are obligatory.

**Example result:**

{

"status": "ok",

"sensorCards": [

{

"ref": "9000001421",

"location": {

"id": "0",

"type": "Feature",

"properties": {},

"geometry": {

"type": "LineString",

"coordinates": [

[3.710991496082532, 50.99672140240988],

[3.7117315960823807, 50.99699980240989]

]

},

"bbox": [3.710991496082532, 50.99672140240988, 3.7117315960823807, 50.99699980240989]

},

"modalities": ["Bike", "Car", "Pedestrian"],

"hasReverse": true,

"sensorType": {

"name": "telraam"

},

"extraFields": {},

"quality": 76.34408602150539

}]

}

#### Sensor Card endpoint

**Body params:**

{

“endpoint”: “getSensorCard”

“ref”: SENSOR\_UNIQUE\_REFERENCE

}

**Expected result:**

{

"status": "ok",

"sensor": SENSOR\_CARD\_AS\_IN\_SENSORS\_COLLECTION

}

**Error result:**

Raised if sensor with specified reference was not found

{

'status': 'error\_occurred',

'error': 'No sensor found'

}

#### Sensor Cards endpoint

**Body params:**

{

“endpoint”: “getSensorCard”

“refs”: ARRAY\_OF\_SENSOR\_UNIQUE\_REFERENCES

}

**Expected result:**

{

"status": "ok",

"sensors": ARRAY\_OF\_SENSOR\_CARDS\_AS\_IN\_SENSORS\_COLLECTION

}

If some references are not found, they are ignored – no error is raised!

#### Main analysis endpoint

**Body params (details below):**

{

“endpoint”: “getMultiSourceTrack”

“viewType”: ONE\_OF\_THE\_AVAILABLE\_ANALYSIS\_TYPES

“timePeriods”: ARRAY\_OF\_TIME\_WINDOWS\_TO\_USE\_FOR\_ANALYSIS

“refModalityReverseCombinations”: CONFIGURATION\_OF\_SENSOR\_TRACKS

“lang”: 2\_LOWERCASE\_LETTERS\_LANGUAGE\_CODE\_TO\_USE\_FOR\_RESPONSE

“basePopulationSensorRef”: WHICH\_SENSOR\_TO\_USE\_AS\_BASE

}

viewType:

Possible analysis types (case insensitive):

* “summary”: general statistics, raw data charts and data persistence analysis
* “sensors split”: this view aims at comparing different sensor tracks via a visualization of relative values of the total population captured
* “daily profiles”: a deeper analysis of per hour data and its evolution over time; includes the statistical comparison between time windows on an hourly level
* “trend analysis”: identification of weekly periodic components and the global trend analysis
* “difference on a daily level”: statistical comparison of the signal on a daily level
* “single track extrapolation”: temporal extrapolation of the signal of single tracks
* ”multiple tracks extrapolation”: temporal extrapolation of the signal of single tracks using the information from other tracks
* “extractRawCSVData”: just filters the data and returns the downloadable CSV file with the selected data

For deeper explanation of different view types we refer to the tool itself and the helpers inside it.

timePeriods:

It is possible to submit different number of “time windows” in an array. Each of them is represented by the following dictionary:

{

"from":UNIX\_TIME\_IN\_MS\_OR\_TEXTUAL\_STRING,

"to": UNIX\_TIME\_IN\_MS\_OR\_TEXTUAL\_STRING,

"Monday": BOOLEAN\_FLAG\_SPECIFYING\_IF\_MONDAYS\_ARE\_USED,

"Tuesday": SIMILAR\_TO\_MONDAY,

"Wednesday": SIMILAR\_TO\_MONDAY,

"Thursday": SIMILAR\_TO\_MONDAY,

"Friday": SIMILAR\_TO\_MONDAY,

"Saturday": SIMILAR\_TO\_MONDAY,

"Sunday": SIMILAR\_TO\_MONDAY,

"Holiday": BOOLEAN\_FLAG\_SPECIFYING\_IF\_PUBLIC\_HOLIDAYS\_ARE\_USED,

"Non-holiday": BOOLEAN\_FLAG\_SPECIFYING\_IF\_NON\_PUBLIC\_HOLIDAYS\_ARE\_USED,

"DBSCAN": BOOLEAN\_FLAG\_SPECIFYING\_IF\_DAYS\_NOT\_PASSING\_THIS\_TEST\_ARE\_USED,

"MinThreshold": SIMILAR\_TO\_DBSCAN,

"PerformanceThreshold": SIMILAR\_TO\_DBSCAN

}

refModalityReverseCombinations:

Configuration of sensor tracks is done via a nested dictionary of the following format:

{

SENSOR\_UNIQUE\_REFERENCE: {

MODALITY\_NAME: {

"false": BOOLEAN\_IF\_NORMAL\_DIRECTION\_IS\_USED,

"true": BOOLEAN\_IF\_REVERSE\_DIRECTION\_IS\_USED

}

}

}

Some remarks here:

* The structure is SENSOR\_REFERENCE -> MODALITY\_NAME -> IS\_REVERSE\_DIRECTION -> BOOLEAN\_VALUE\_TO\_INCLUDE\_OR\_NOT
* IS\_REVERSE\_DIRECTION key is a string to be compliant with JSON specifications
* In this dictionary a Boolean flag set to false is equal to the missing key (sensor track is not used).
* If a sensor doesn’t differentiate 2 directions, then the “false” key should be used

lang:

Currently only “en” and “nl” are supported for the language

basePopulationSensorRef:

The total observed population can vary in different time periods, thus for some applications it might be useful to look not at the absolute values, observed by a sensor, but how they evolve compared to some benchmark. If basePopulationSensorRef set to some sensor unique reference, then in the analysis we will divide the observed counts by a total population seen by that sensor at the corresponding day/hour (if available).

**Body params example:**

{

"endpoint": "getMultiSourceTrack",

"viewType": "Summary",

"timePeriods": [

{

"from": 1630454400000,

"to": 1635724800000,

"Monday": true,

"Tuesday": true,

"Wednesday": true,

"Thursday": true,

"Friday": true,

"Saturday": true,

"Sunday": true,

"Holiday": true,

"Non-holiday": true,

"DBSCAN": true,

"MinThreshold": true,

"PerformanceThreshold": true

}

],

"refModalityReverseCombinations": {

"39472": {

"Car": {

"false": true,

"true": true

},

},

"9000000131": {

"Bike": {

"false": true,

"true": false

},

}

},

"basePopulationSensorRef": null,

"lang": "en"

}

**Expected result:**

{

"status": "ok",

"multiSourceTracks": {

“contents”: ARRAY\_OF\_BASE\_ITEMS\_TO\_SHOW\_IN\_THE\_FRONTEND

}

}

Base items have the following general syntax (other fields are necessary depending on the type) and can be combined together:

{

“type”: “text” / ”lineChart” / “scatterChart” / “table” / “map” / “list” / “selectable”

"title": TITLE\_TO\_SHOW (optional)

“subtitle”: SUBTITLE\_TO\_SHOW (optional)

“isError”: BOOLEAN\_FLAG\_TO\_RAISE\_ATTENTION (optional)

"collapsed": BOOLEAN\_FLAG\_IF\_SHOWN\_COLLAPSED\_BY\_DEFAULT

}

lineChart / scatterChart additional parameters:

The **parameters in bold** are coming from the ChartJS properties and should be checked there in case not self-explanotary ( <https://www.chartjs.org/> ):

{

**“xAxisType”**: “linear” / “time” / …,

**“xAxisStep”**: ,

**“yAxisStep”**:,

**“yAxisMin”**: ,

**“yAxisMax”**:,

“lines”: [

{

“data”: LIST\_OF\_DICTIONARIES\_HAVING\_X\_AND\_Y\_KEYS\_AND\_VALUES,

**“showLine”**: ,

“label”: LABEL\_OF\_THE\_LINE\_FOR\_THE\_LEGEND (optional),

“tooltips”: ARRAY\_OF\_TOOLTIP\_STRINGS\_FOR\_EACH\_POINT\_IN\_DATA (optional),

“pointColors”: ARRAY\_OF\_RGB\_COLORS\_TO\_USE\_FOR\_NODES (optional),

“colorCounter”: THE\_COUNTER\_OF\_THE\_COLOR\_TO\_TAKE\_FROM\_THE\_PALLETTE,

“timePeriodCounter”: WHICH\_TIME\_WINDOW\_IT\_RELATES\_TO (for styling purposes; starts from 0)

**“fill”**: IF\_THE\_SPACE\_TO\_A\_DIFFERENT\_LINE\_SHOULD\_BE\_FILLED\_WITH\_COLOR (optional)

},

…

]

}

table additional parameters:

{

“captions”: ARRAY\_OF\_STRINGS\_FOR\_THE\_TABLE\_HEADER\_LINE,

“lines”: ARRAY\_OF\_ROWS\_EACH\_OF\_THEM\_BEING\_AN\_ARRAY\_OF\_STRINGS\_FOR\_THE\_ROW

}

list additional parameters:

{

“children”: ARRAY\_OF\_BASE\_ITEMS

}

Selectable (shows a tabular structure) additional parameters:

{

“children”: [

{

“label”: LABEL\_TO\_SHOW\_ON\_THE\_CORRESPONDING\_TAB\_BUTTON,

“item”: BASE\_ITEM\_TO\_SHOW\_IN\_TAB

},

…

]

}

map additional parameters:

{

“meta”: {

SHOWING\_SENSOR\_REFERENCE: {

“color”: HTML\_COLOR\_TO\_USE\_FOR\_THIS\_SENSOR

“detailsString”: THE\_TEXT\_TO\_SHOW\_IN\_THE\_HOVER\_POPUP,

“detailsHtmlString”: THE\_HTML\_CODE\_TO\_SHOW\_IN\_THE\_HOVER\_POPUP

},

…

}

}

### Technical views / APIs

BACKEND\_URL/admin/\* and BACKEND\_URL/accounts/\* used for the DJANGO admin panel and authentication routines (see <https://docs.djangoproject.com/en/4.1/> for details).

BACKEND\_URL/upload/csv/ and BACKEND\_URL/pushData are used for data upload (see the corresponding section below).

### External dependencies

The data treatment is articulated around Scikit-Learn ( <https://scikit-learn.org/> ) and Statsmodels ( <https://www.statsmodels.org/> ) libraries.

The preprocessing pipelines are using Pandas ( <https://pandas.pydata.org/> ).

To simplify the classification of days (public holidays identification) Holidays ( <https://github.com/dr-prodigy/python-holidays> ) library is used.

The rest of external dependencies are related to Django framework itself and its optional packages (OAuth2 / CORS / …).

### Container configuration

Django models system automatically manages the translation of the high-level model definition / access to the database tables creation / querying. As those can change in time with the new versions of code developed, the migrations system is put in place, storing the whole history of database scheme modifications and adapting it in case of code changes. These migrations are stored in the raw files on the disk (and not in the database itself). For the external libraries / imported applications, those go together with the library application code and thus coupled with the version control. As the toolbox itself is still in an active phase of development and is responsible for the client data, it is relevant to store the migration files on the persistent storage. Thus **the path “/app/app/app/migrations” of the backend container should be mounted to a latter one.**

The following environment variables must be set for correct functioning of the container:

* internalURLs – the IP address to which the web server proxy is forwarded to. Can be set to '\*' not to include the extra protection.
* apiURL – the full URL (with protocol) at which the backend is accessed
* debugMode – ‘0’or ‘1’ to disable/enable the Django embedded debug mode (not secure for production but permits easy error management).
* CORSWhiteListed – frontend full URL (with protocol). Used for CORS protection exception.
* dbPass – the POSTGRES\_PASSWORD configured for the database container
* dbHost – the name of the database container or the hostname for externally accessible databases

The backend container web-server listens on the port 80. It should be forwarded to external proxy (taking care of https handling).

# First launch installation steps

When the collection of toolbox container is launched it is not yet populated with any (including operational) data. The following steps are needed to make the system functional:

1. Execute the following commands in the backend container:
   1. python3 /app/app/manage.py makemigrations app

(Creation of the migrations files for the toolbox main application)

* 1. python3 /app/app/manage.py migrate

(Execution of the above created and already available external libraries migrations for the database scheme population)

* 1. python3 / app/app/manage.py createsuperuser

(Creation of the super admin user in an interactive shell)

1. Navigate to the BACKEND\_URL/o/applications/, authenticate with the super user credentials and create a new OAUTH2 application with the following parameters:
   1. Client type – confidential
   2. Grant type - resource owner password
   3. Client ID / Client Secret – same as REACT\_APP\_OAUTH\_CLIENT\_ID / REACT\_APP\_OAUTH\_CLIENT\_SECRET environment variables set for the frontend container.
2. (Optional) Navigate to BACKEND\_URL/admin/ -> Users and create additional users who will have access to the toolbox.
3. Upload the sensors data (see below)

Toolbox data upload options

API upload

The first option to upload the data consists in pushing JSON encoded data directly to the backend endpoint. If BACKEND\_URL is the URL, where backend is hosted, then the target endpoint is BACKEND\_URL/pushData.

Currently the data upload is open without any authentication as it is in a write-only mode and thus not involving any privacy issue. Moreover the platforms are not designed for broad audience, thus the risks of random data being pushed are limited. In case needed, the OAUTH2 authentication can be enabled with a single-line decorator in the backend to mimic the protection pipeline used by the frontend.

The JSON should be posted in the body of the request and have the format similar to the following:

{

“sensors”: [

{

“ref”:”SENSOR\_REF”,

"location": {“type”:”point”,”coords”:[LAT, LON] },

“sensorType”:”SENSOR\_TYPE\_STRING”,

“meta”: { ANY\_META\_INFORMATION\_TO\_STORE\_IN\_THE\_BACKEND}, “tracks”:[

{

“modality”: “Car”,

“isReverse”: False,

"counts":[

{

“date”: String in format %Y-%m-%d", “hour”: HOUR\_INTEGER\_VALUE, “count”: OBSERVED\_COUNT\_AT\_THAT\_HOUR

}

]

}

]

]

}

General comments:

* + all the arrays can have multiple values.
  + If the data contradicts the data stored in the database, it will be modified
  + Location can have the following types:
    - point: then coords have the lat,lon pair
    - linestring: then coords have a string similar to “'LINESTRING (4.470560418428409 51.01749874553107, 4.470670605706021 51.01738049739185, … )”
    - geojson: then coords have the geojson string.
  + Sensor type is usually a data provider name
  + Meta information is flexible. The only currently used field (optional) is “addressString” to show the human readable address in the interface
  + Modality can be one of “Car”, “Bike”, “Pedestrian”, “Background”
  + isReverse is used to take into account 2 directions of the street.

## CSV upload

The second option to upload the data consists in the upload of the CSV file dumps to the backend endpoint. If BACKEND\_URL is the URL, where backend is hosted, then the target url is BACKEND\_URL/upload/csv/.

Currently the data upload is open without any authentication as it is in a write-only mode and thus not involving any privacy issue. Moreover the platforms are not designed for broad audience, thus the risks of random data being pushed are limited. In case needed, the password protection can be enabled with a single-line decorator in the backend.

The csv file lines format should follow the following header:

sensor\_type,sensor\_ref,modality,is\_reverse\_channel,date,hour,count,location,meta

The field descriptions are exactly the same as for the API upload, just the structure is in the flat file (thus location and meta are duplicated in each sensor count line in order to avoid multi-stage uploads)