
MagicBox Documentation

UNICEF Office of Innovation

Jul 27, 2018

1	Administrative boundaries	3
1.1	Levels	3
1.2	Why we use them	4
1.3	Naming string explained	5
1.3.1	First integer	5
1.3.2	Second and third integer	5
1.3.3	Putting it together	5
2	Access control	7
2.1	Tokens	7
2.2	Rules	8
3	Ingesting data into MagicBox	11
3.1	School data ingestion	11
3.2	School data database	12
3.2.1	Inserted data	12
4	Querying MagicBox data	15
4.1	Cases	15
4.2	Mobility	15
4.3	Mosquitoes	16
4.4	Population	16
4.5	Schools	16
4.5.1	How we do it	16
4.6	Caveats	18
5	Data ingestion rules	19
5.1	Available flags	19
5.2	School data file names	19
5.3	Caveats	20
6	Data validation	21
6.1	How we do it	21
6.1.1	Adding new attributes	22
6.2	Why we do it	22
7	Data visualization	23

7.1	How we do it	23
7.2	Why we do it	23
8	How to deploy an application	25
8.1	Create a feature branch for development	25
8.1.1	Create a tag	25
8.2	Deploying the application	26
8.2.1	Adding Azure remote for repository	26
8.2.2	Pushing changes to the staging slot	26
8.2.3	Swap staging to production (Web)	26
8.2.4	Swap staging to production (CLI)	28
9	GitHub workflow best practices	31
9.1	How to make a new GitHub repository	31
9.1.1	Repository creation	31
9.1.2	Configure repository	32
9.1.3	Set up useful labels	34
9.1.4	Set up continuous integration (CI)	35
9.1.5	Set up code health checks with CodeClimate	35
9.2	How to maintain a GitHub repository	35
9.2.1	Update issue and pull request metadata	35
9.2.2	Adding labels to issues	37
9.2.3	Updating project boards	37
9.2.4	Making pull requests	37
9.2.5	Reviewing pull requests	37
9.3	Communicating about development	38
10	Indices and tables	39

MagicBox is an open-source platform that uses real-time information to inform life-saving humanitarian responses to emergency situations. It's composed of multiple GitHub repositories designed to ingest, aggregate, and serve data.

Administrative boundaries

UNICEF uses **administrative boundaries** (or administrative areas) to define geographic areas in MagicBox and related projects. Administrative boundaries (ABs) extend the concepts of a country, state, or region. Examples of an administrative boundary may be...

- Country
- State
- Province
- County
- Municipality
- And more...

ABs come from [GADM](#) and the [Humanitarian Data Exchange](#) (HumData), databases of global administrative areas. In addition, more ABs may come from other sources like municipal governments. These databases make it easier to work with this data in GIS or related software.

1.1 Levels

Levels are a hierarchy system to explain different concepts of administrative boundaries. For example, a city may belong to a municipality, which belongs to a country. Levels help us explain this hierarchy.

UNICEF uses this system of levels:

- **Level 0:** Countries
- **Level 1:** States
- **Level 2:** Counties
- **Level 3:** Municipalities
- And so on...

1.2 Why we use them

Different definitions of geographic areas by different nations makes worldwide geographic mapping a challenge. Not everyone uses the same names for regions. In MagicBox projects, each region, or **shape**, is assigned an `admin_id`.

An `admin_id` is an interpretation of a political border for a specific time. They point to a specific shape in a shapefile that represents an individual country. Time is a part of the admin ID to ingest historical data. For example, mobility or temperature might require a series from an earlier date where a region was known by a different name.

See the following example of an AB for Afghanistan, with an `admin_id`.

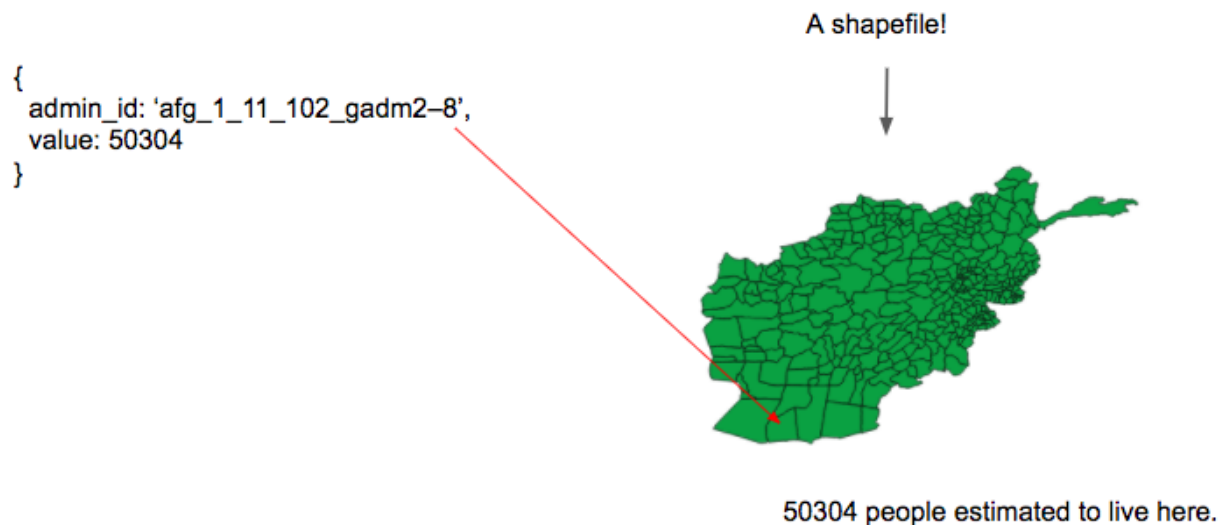


Fig. 1: Example of an administrative boundary for Afghanistan

To understand where `afg_1_11_102-gadm2-8` points to, note that Afghanistan has three **levels** of administrative boundaries. The `admin_id` has three integers, one per admin level:

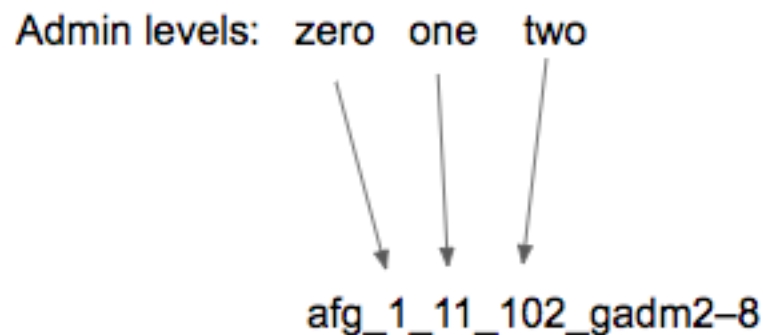


Fig. 2: Afghanistan has three levels of administrative boundaries

1.3 Naming string explained

1.3.1 First integer

The first integer is 1 because Afghanistan is the first country in GADM's database.



Fig. 3: The first few countries in GADM's database

1.3.2 Second and third integer

Afghanistan has 34 shapes at level one and 320 shapes in level two.

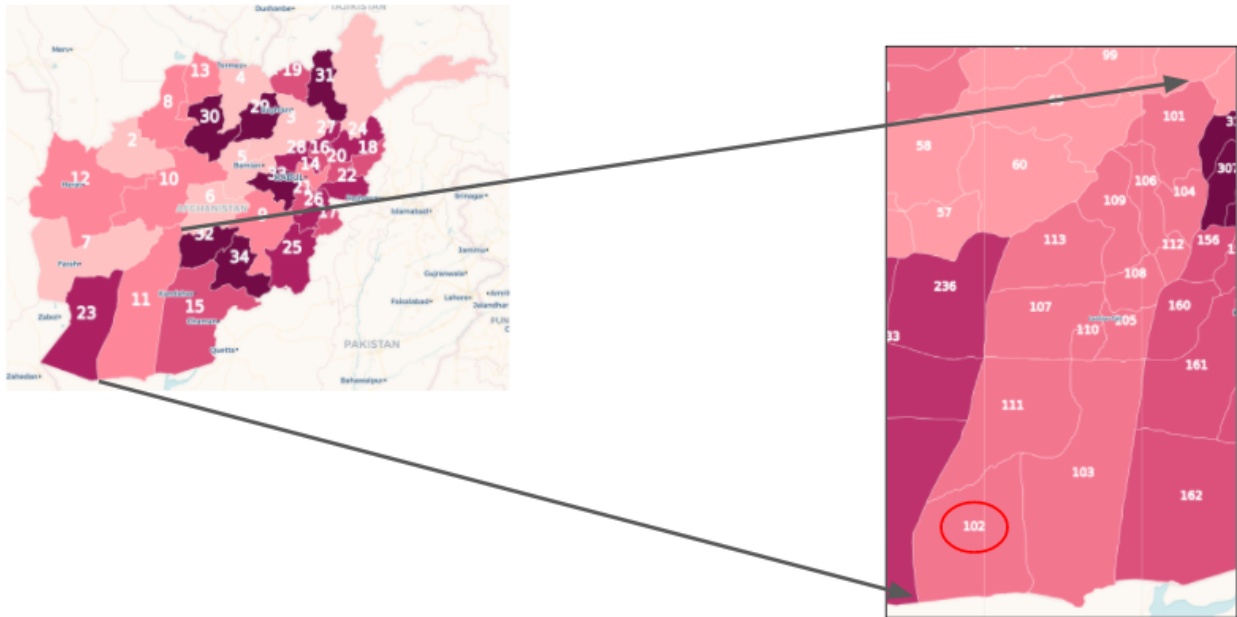


Fig. 4: Afghanistan: Higher level administrative boundaries

1.3.3 Putting it together

Thus, afg_1_11_102-gadm2-8 indicates any population value attached to it is related to:

- First country in the gadm collection
- 11th shape in the admin level 1 shapefile
- 102nd shape in the admin level 2 shapefile

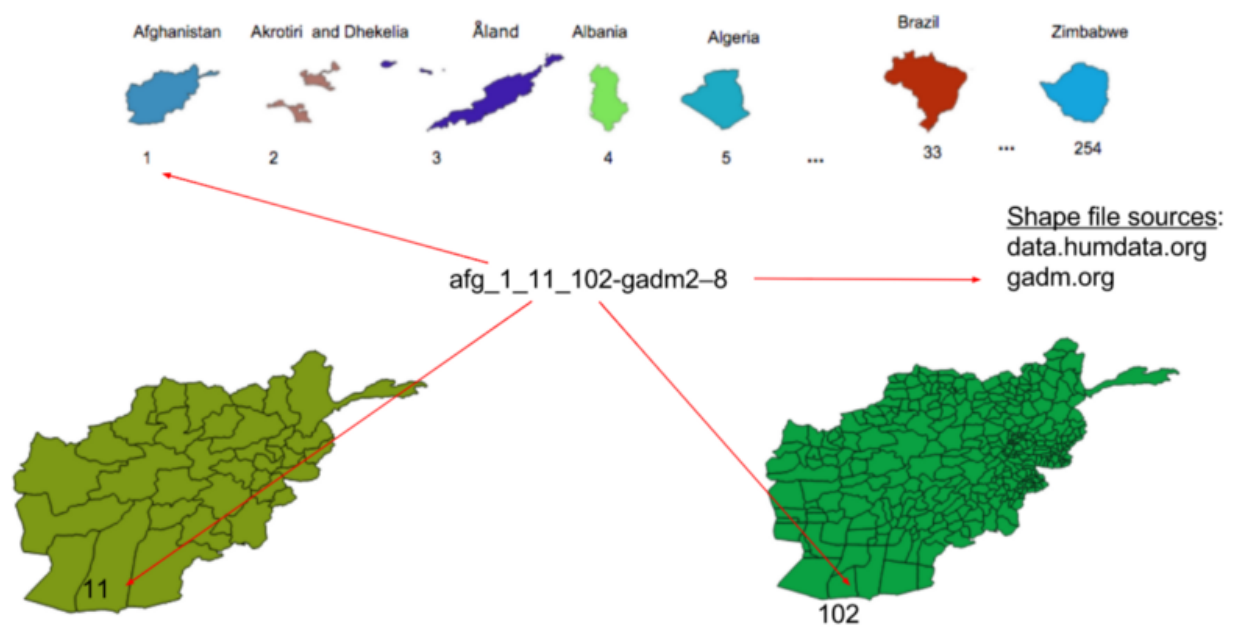


Fig. 5: Putting it together: What the admin ID represents

CHAPTER 2

Access control

Information about schools can be sensitive. Potential bad actors could use school data for malicious purposes and terrorism. How school data is shared outside of the API is important.

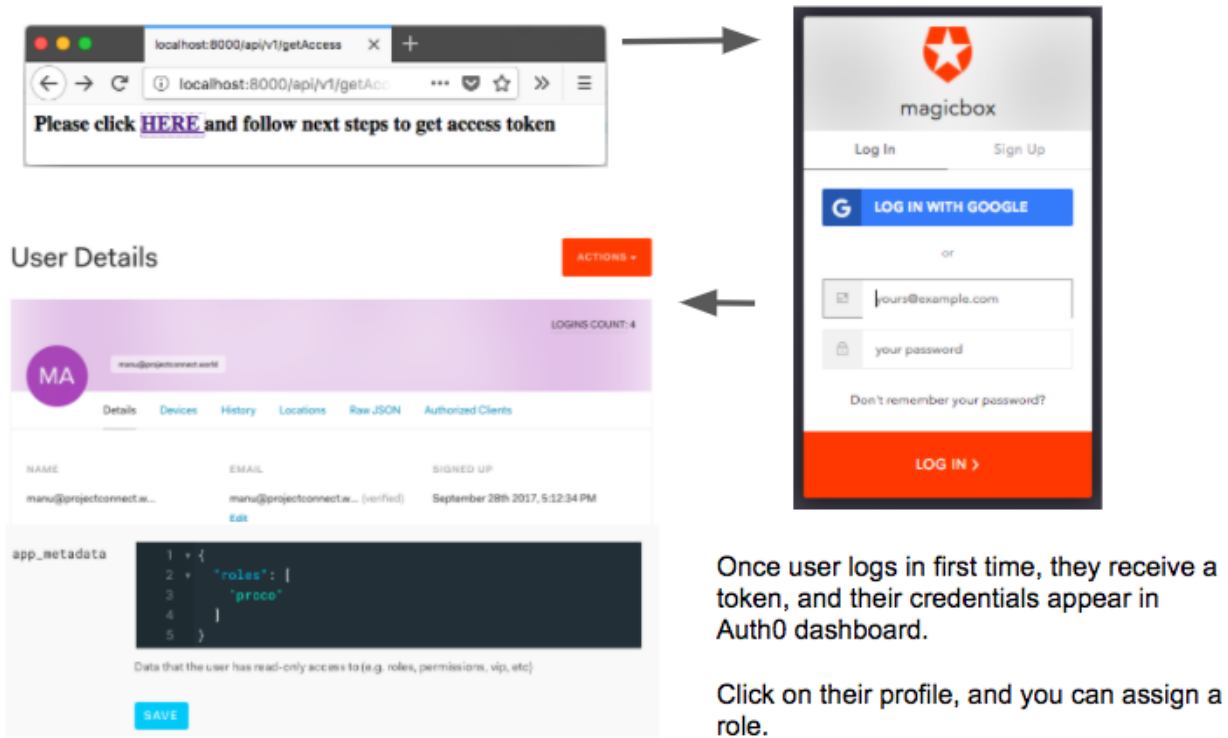
MagicBox uses [Auth0](#) to authenticate users and assign them roles.

2.1 Tokens

Auth0 creates tokens for users authenticating to the API. A user may make a request with their token like this.

```
curl -i localhost:8000/api/v1/schools/countries/GL -H "Token: Bearer_↵  
↵xxxxxxxxxx9gek6Z5Ilnkx"
```

After receiving the token, UNICEF reviews the applicant before privileges are granted. Once our API receives the token, we pass it to Auth0 which returns the user's profile and roles.



2.2 Rules

General rules can be assigned by email domain via Auth0.

See also:

See [Data ingestion rules](#) for more information about data rules.

Edit Rule



Heads up! If you are trying to access a service behind a firewall, make sure to open the right ports and allow inbound connections from these IP addresses: `35.167.74.121,35.166.202.113,35.160.3.103`

Set roles to a user

```
1  function (user, context, callback) {
2    user.app_metadata = user.app_metadata || {};
3    // You can add a Role based on what you want
4    // In this case I check domain
5    var addRolesToUser = function(user, cb) {
6      if (user.email && (user.email.indexOf('@unicef.org') > -1)) {
7        cb(null, ['admin']);
8      } else if (user.email.indexOf('@projectconnect.world') > -1) {
9        cb(null, ['proco']);
10     } else {
11       cb(null, ['user']);
12     }
13   };
14 }
```

Fig. 1: Code that implements checking for the email domain of a user

Ingesting data into MagicBox

Importing new data sets to MagicBox is important for expanding coverage and how we retrieve new data insights. This document explains how we ingest and import new data to MagicBox.

3.1 School data ingestion

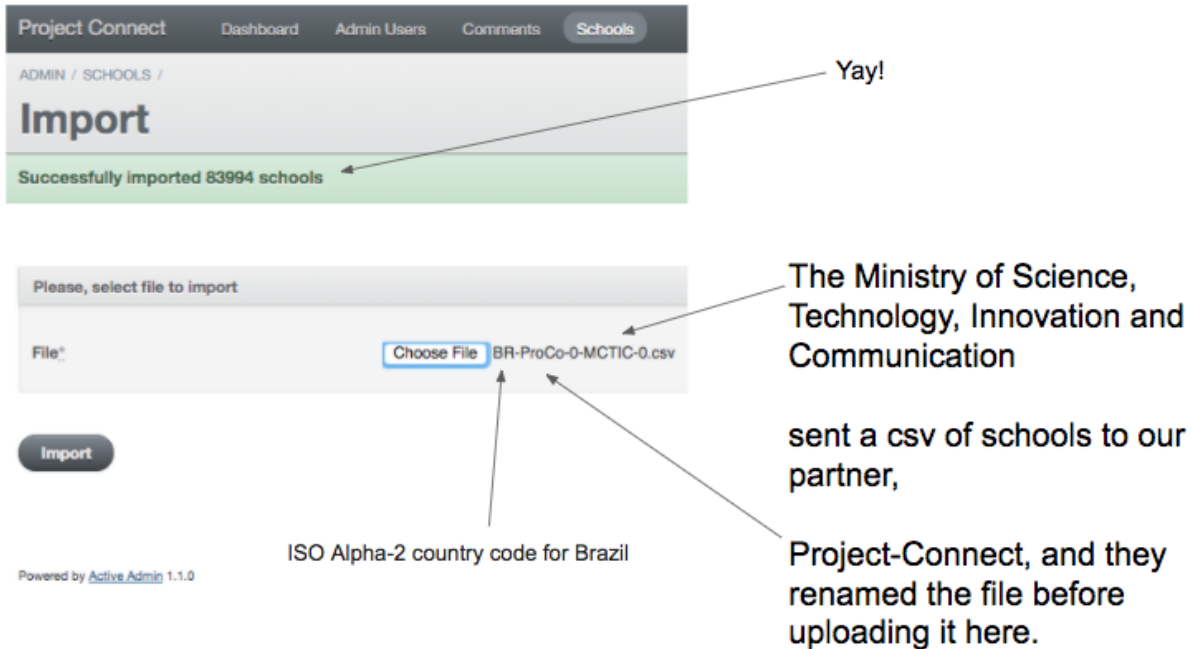
Data for schools comes in CSV files with a specific naming scheme. We developed a Ruby on Rails-based CRUD¹ application, [Project Connect](#) to ingest this data.

There are two admin interfaces for Rails applications:

- Rails Admin
- Active Admin

Active Admin has an [import plugin](#) for uploading CSVs with `before_batch_import` support. We chose Active Admin for this reason.

¹ Create, Read, Update, Delete



3.2 School data database

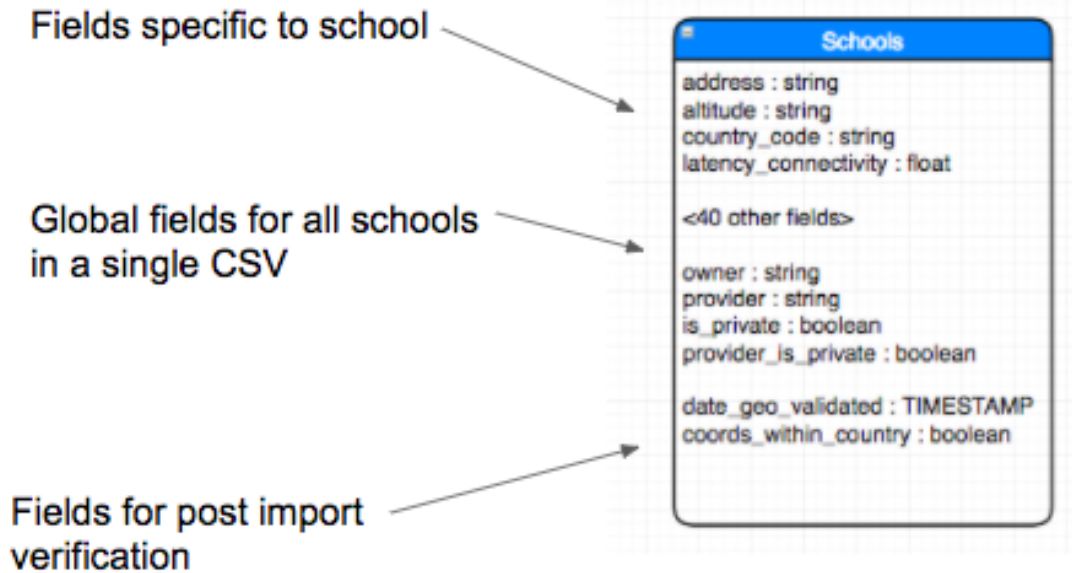
MagicBox needed a database to manage large amounts of school data and process it quickly. To do this, MagicBox uses a relational database, [PostgreSQL](#), to store data. PostgreSQL was chosen because it was used in a tutorial for building a similar use case as ours (thus, a relational database is not a “married” idea).

All school data is stored in a single table named `schools`. This keeps things simple at the expense of some repeated values in the database.

3.2.1 Inserted data

When data is imported, new data is inserted alongside the imported data. Four new types of important data are added:

- Provider / owner of school CSV data
- Organization that received school data
- Identity of uploader
- If uploader chooses to remain private



These values are not included with the data. We use the `before_batch_import` hook in Active Admin to parse file names, insert this data, and assign values.

```
ActiveAdmin.register School do
  before_action only: [:do_import] do
    Thread.current['import.current_admin_user'] = current_admin_user
  end

  before_batch_import: proc { |import|
    # country_lookup = JSON.parse(File.read('./public/country_codes.json'))
    # Get country code from file name
    School.where(datasource: import.file.original_filename).destroy_all
    # Name file will be : 'country_code'-owner-'data_is_private'-provider_name-'provider_wants_to_remain_anonymous'.csv
    file_name_segments = import.file.original_filename.sub(/.csv$/, '').split('-')
    creator_email = Thread.current['import.current_admin_user'].email
    country_code = file_name_segments.first.upcase
    owner = file_name_segments[1]
    is_private = file_name_segments[2].to_i == 1 ? true : false
    provider = file_name_segments[3]
    provider_is_private = file_name_segments[4].to_i == 1 ? true : false
    import.headers["creator_email"] = :creator_email
    import.headers["country_code"] = :country_code
    import.headers["owner"] = :owner
    import.headers["is_private"] = :is_private
    import.headers["provider"] = :provider
    import.headers["provider_is_private"] = :provider_is_private
    import.headers["datasource"] = :datasource
    import.batch_replace(:creator_email, { nil => creator_email })
    import.batch_replace(:country_code, { nil => country_code })
    import.batch_replace(:owner, { nil => owner })
    import.batch_replace(:provider, { nil => provider })
    import.batch_replace(:datasource, { nil => import.file.original_filename })
    import.batch_replace(:is_private, { nil => is_private })
    import.batch_replace(:provider_is_private, { nil => provider_is_private })
  }
end
```

Fig. 1: Implementation of how data is modified before imported. See `schools.rb`

Querying MagicBox data

The MagicBox API ([magicbox-open-api](#)) serves different sets of data through its API. Currently, five datasets are available.

- Cases
- Mobility
- Mosquitoes
- Population
- Schools

The benefit of querying this data is to analyze and cross different data sets together. For example, this is done in [magicbox-maps](#), where population and mosquito prevalence data are crossed together.

This document explains how the API works but does not document how to use it. For more information on using the API, see the [magicbox-open-api](#) repository.

4.1 Cases

Note: This section needs to be written.

4.2 Mobility

Note: This section needs to be written.

4.3 Mosquitoes

Note: This section needs to be written.

4.4 Population

Note: This section needs to be written.

4.5 Schools

The MagicBox API also serves data about schools. The public repositories contain sample data from private data sets. However, in the interest of the privacy and safety of the schools, the full data sets are kept private.

4.5.1 How we do it

There are three endpoints for querying school data. They serve the following data:

1. List of countries with school data
2. Schools in a country
3. Specific school data by ID value

The first endpoint returns a comma-separated list of country codes. There is available data about schools in the countries returned in the list.

The second endpoint returns an array of arrays, where

- First array returns column names (e.g. `lat`, `lon`, `speed_connectivity`, `type_connectivity`)
- Subsequent arrays return values for school records

The third endpoint returns column values from a specific school, as selected by its ID number. Some of the values available are listed below:

- Address
- Network connectivity
- Number of teachers
- Number of students
- If electricity is available
- If running water is available
- And more

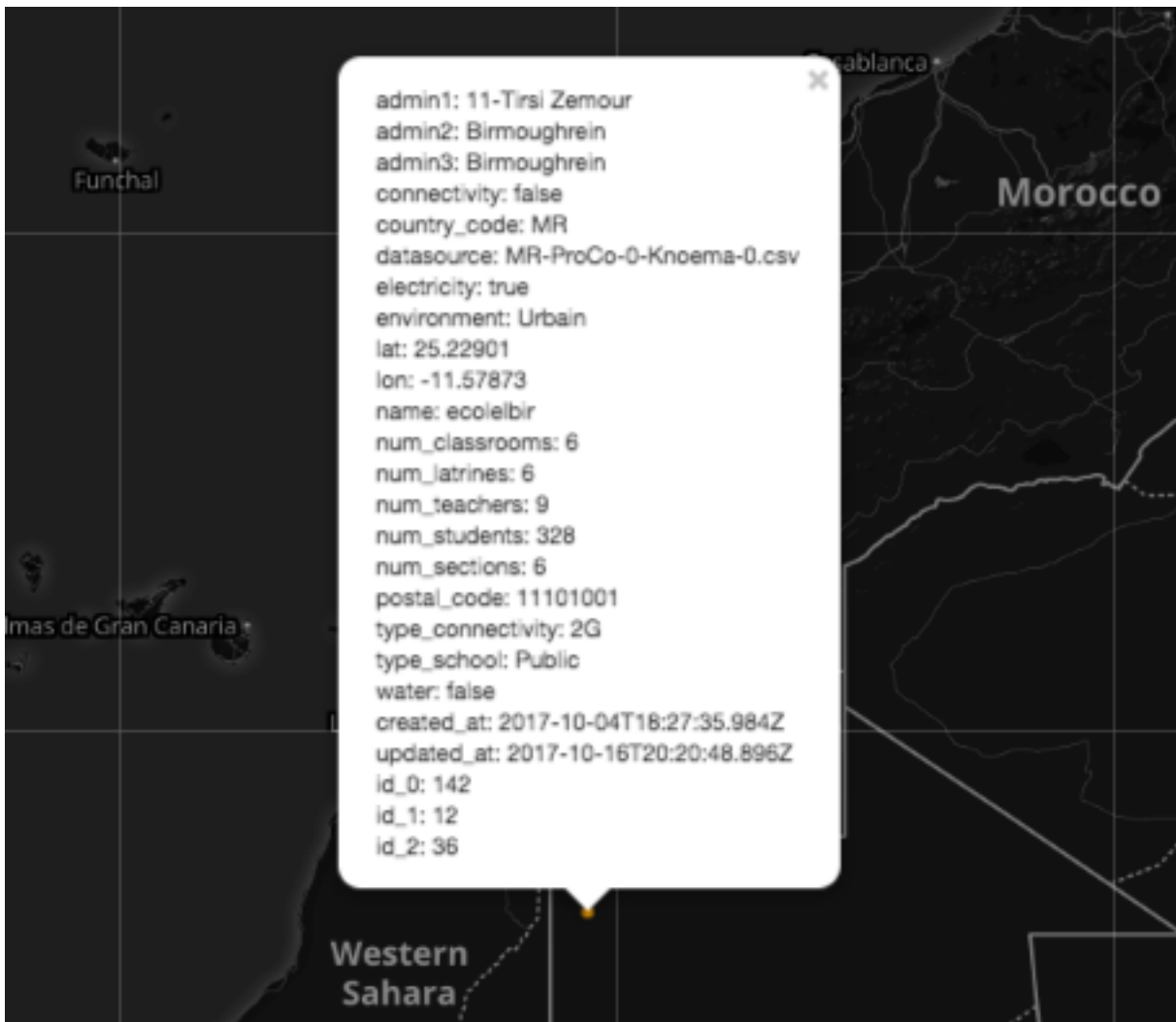


Fig. 1: Example of output from third endpoint inside of a front-end utilizing the API

4.6 Caveats

There are some caveats with querying the MagicBox API.

- Cannot specify or limit columns included in response
- Cannot request values where value is null
- Cannot use operators (e.g. greater than, less than, etc.)

Data ingestion rules

This document explains how the format we receive data in. It also details how certain flags, or rules, are applied to the data when ingested.

5.1 Available flags

There are two flags available. These flags are applied to data when ingested.

- **Private data:** Data is private and is only accessible to UNICEF and uploading organization
- **Anonymity:** Uploading organization chooses to remain anonymous

These flags are not inside of the data sets. The flags must be applied when the data is ingested.

See also:

See [Ingesting data into MagicBox](#) for more information about data ingestion.

5.2 School data file names

School mapping data is provided in a unique file name format. Some information must be gathered from the file name. See this example:

`BR-ProCo-0-MCTIC-0.csv`

There are five parts to the file name.

1. **BR:** Country code (BR is Brazil)
2. **ProCo:** Partner (ProCo is [Project Connect](#))
3. **0:** Sets privacy of data (0 is public, 1 is private)
4. **MCTIC:** Provider (MCTIC is [Ministério da Ciência, Tecnologia, Inovação e Comunicação](#))

5. **0**: Sets privacy of uploader (0 is public, 1 is anonymous)

5.3 Caveats

Code is not yet written to enforce rules. We still need to implement permissions based on user information forwarded from Auth0.

Data validation ensures correct and consistent data is processed and aggregated for use in MagicBox. The validated data is used by implementations of the MagicBox API, such as [magicbox-maps](#).

This article explains how we verify values, identify duplicates, and merge multiple records into one.

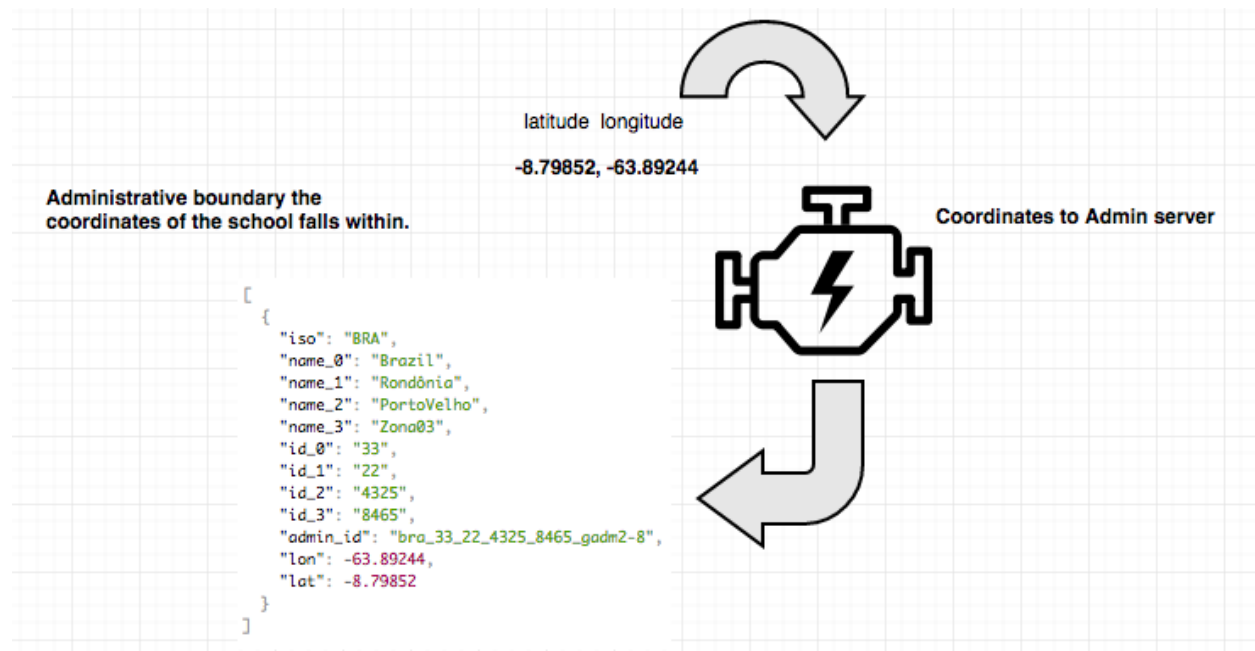
6.1 How we do it

Fig. 1: How [magicbox-latlong-admin-server](#) validates coordinate pairs

Data validation is performed by [magicbox-latlong-admin-server](#). The server verifies a pair of latitude / longitude coordinates are located within the country indicated by the beginning of a file name.

The [data validation program](#) processes every ten minutes as a cron job. It runs a programatic version of this query:

```
// Do a select on all schools that have not been geo validated
"SELECT id, lat, lon, country_code FROM schools WHERE date_geo_validated IS null AND
↪lat IS NOT null AND lon IS NOT null"
```



6.1.1 Adding new attributes

After the data is validated, the `magicbox-latlong-admin-server` updates attributes for some data. The attributes are only added if the engine returns an admin area ID for the target country (see [Administrative boundaries](#)).

1. `date_geo_validated`: To `CURRENT_TIMESTAMP`
2. `coords_within_country`: true or false

6.2 Why we do it

Some CSV files we import and aggregate contain tens of thousands of records. Data validation is an expensive operation. We chose not to verify each school's coordinates in the `before_batch_import` hook for this reason.

Instead, we opted for the `data validation program` to run as a cron job every ten minutes to accomplish this.

MagicBox is a full-stack application. While the back-end API ([magicbox-open-api](#)) serves the data and makes it available for querying, the [magicbox-maps](#) application visualizes the data inside a world map.

7.1 How we do it

[magicbox-maps](#) draws the plot points using OpenStreetMap [Leaflet](#) and [WebGL](#). WebGL allows us to render up to 16 million clickable points at once.

To do this inside of a [React](#) application like [magicbox-maps](#), we created [react-webgl-leaflet](#). This React module enables us to render several plot points simultaneously, like we use for the school mapping data. Every school is represented with a unique, clickable point.

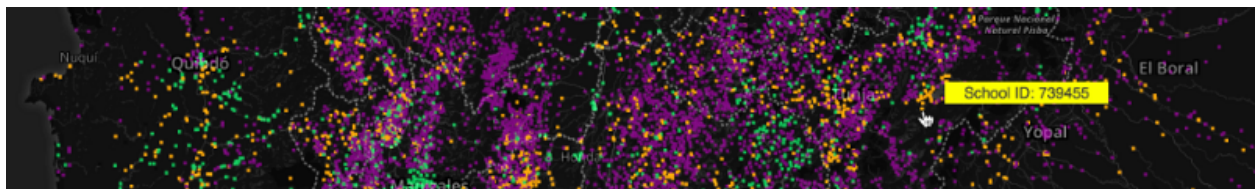


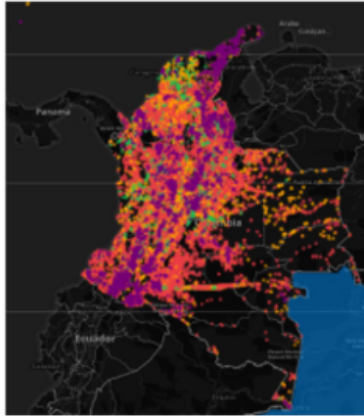
Fig. 1: [react-webgl-leaflet](#) npm module

This is currently used for [magicbox-maps](#).

7.2 Why we do it

Originally, we began using [Leaflet](#) markers. This worked well for Mauritania with 2,936 schools. However, when we moved to Colombia with 49,020 schools, performance took a significant hit. We experimented with a few other options, like heatmaps and clustering.

Leaflet markers *Slow to render. Made app sluggish.*



Heat map *Ugly and useless.*



Clusters *Slow to load, but doesn't affect app.*



In the end, we decided on using [WebGL](#), as explained above.

How to deploy an application

In the UNICEF Office of Innovation team, we use [Azure Web Apps](#) to deploy our applications to the cloud.

This guide will walk you through the deployment process of a WebApp in the Azure environment. At the end of this guide you will be able to develop, tag and deploy the tagged version of the application to the azure servers.

8.1 Create a feature branch for development

While in development, it is useful to make all of your changes in a feature branch named accordingly to the task.

For example, if we want to implement a new feature to show information about the country the user clicked in, we could name the branch accordingly to this task using the command below:

```
git checkout -b show_country_information
```

We can implement our features in this branch and submit a Pull Request to merge it into the master branch of the app.

8.1.1 Create a tag

Once your changes get merged into master, we can create a tag for the current version of the application. Make sure to name the tag accordingly with the project's convention and following the [semantic versioning](#) rules.

For example, suppose you made a simple fix in the application and your changes just got merged from the branch "fix_bug_61" into master. In this project, tags are named using the convention "vX.X.X" and the last tag is "v1.1.1".

Since your fix is a [patch](#), you can create the tag "v1.1.2" with the command

```
git tag v1.1.2
```

Always remember to push the tag to the public repository to make others aware of the current project version. You can push your tags to the public repository with the command.

```
git push origin master --tags
```

8.2 Deploying the application

After our changes get merged into master, we can deploy the application to Azure.

Each UNICEF's application in Azure have two main deployment slots: production and staging.

In the production slot lies the code that you can see in your browser, the application as it is, available for the final user.

The staging slot contains the code preparing to be released to production. Once your code is in this slot you can take a look in a private url to see how your code will respond in production.

If everything looks correct in staging, you can swap the code in production with the code in the staging slot to make your changes available for all users.

8.2.1 Adding Azure remote for repository

The first thing we need to do is add the staging remote to our local git repository.

Within the project directory, you can add the stagin remote for the project issuing the command below.

```
git remote add staging https://<username>@<project-name>-staging.scm.azurewebsites.  
↪net:443/<project-name>.git
```

Make sure to change <project-name> with the project name you are working on and <username> for your username set in the deployment credentials in Azure.

8.2.2 Pushing changes to the staging slot

Push changes to the staging repository issuing the command below

```
git push staging master
```

This command will push the current version in your master branch to staging in azure.

To push a specific tag to Azure, run:

```
git push staging tagname:master
```

Replace “tagname” with the tag you want to push to azure.

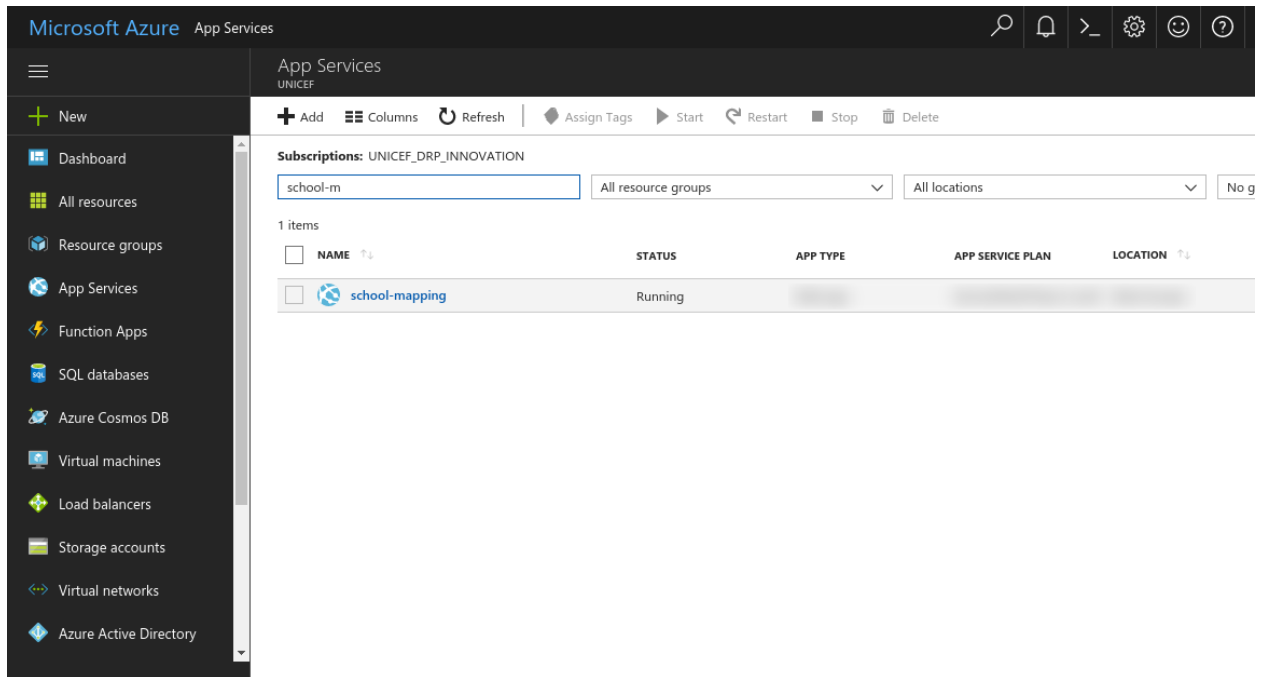
8.2.3 Swap staging to production (Web)

The last step to make your changes available to the public is swap the staging environment with the production environment.

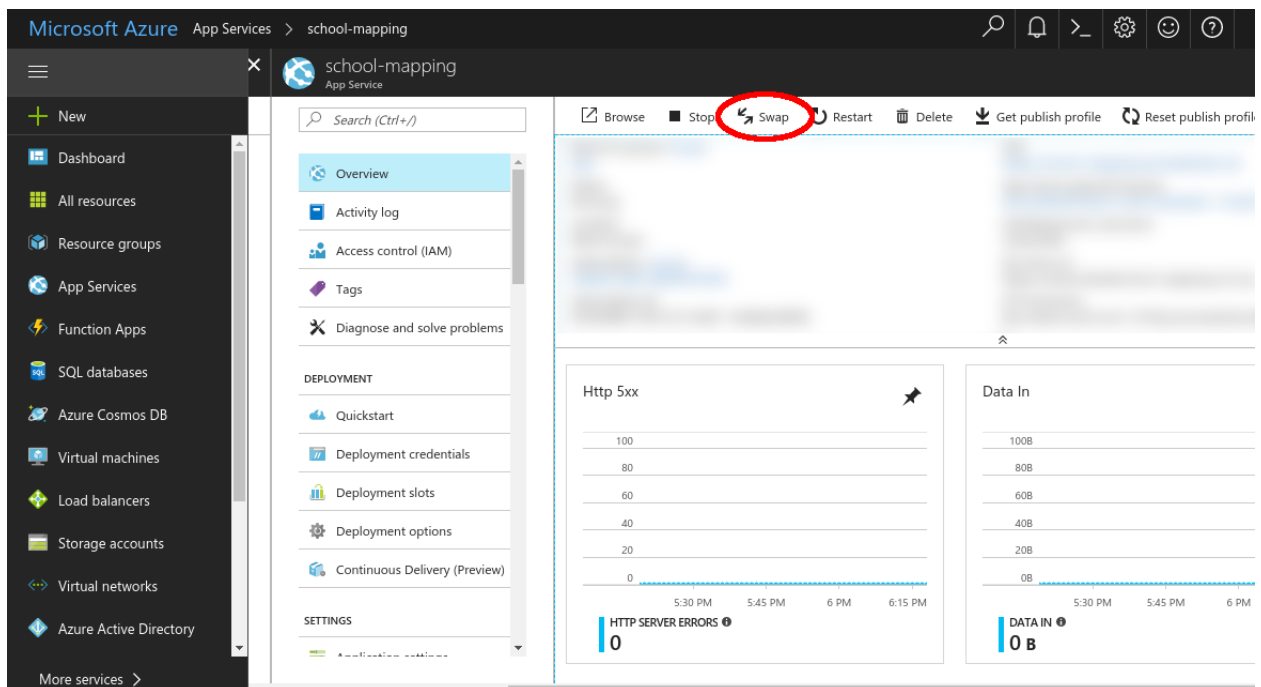
Once your code is working properly in the staging environment, you can swap it with production. Swapping those slots means that the code in the staging environment now answers for the production environment and the code in the production environment now lives in the staging environment.

If a bug was introduced in your changes and it was not caught in the staging environment and is now in the production environment, all you have to do to revert it to the last working version of the application is to swap it again.

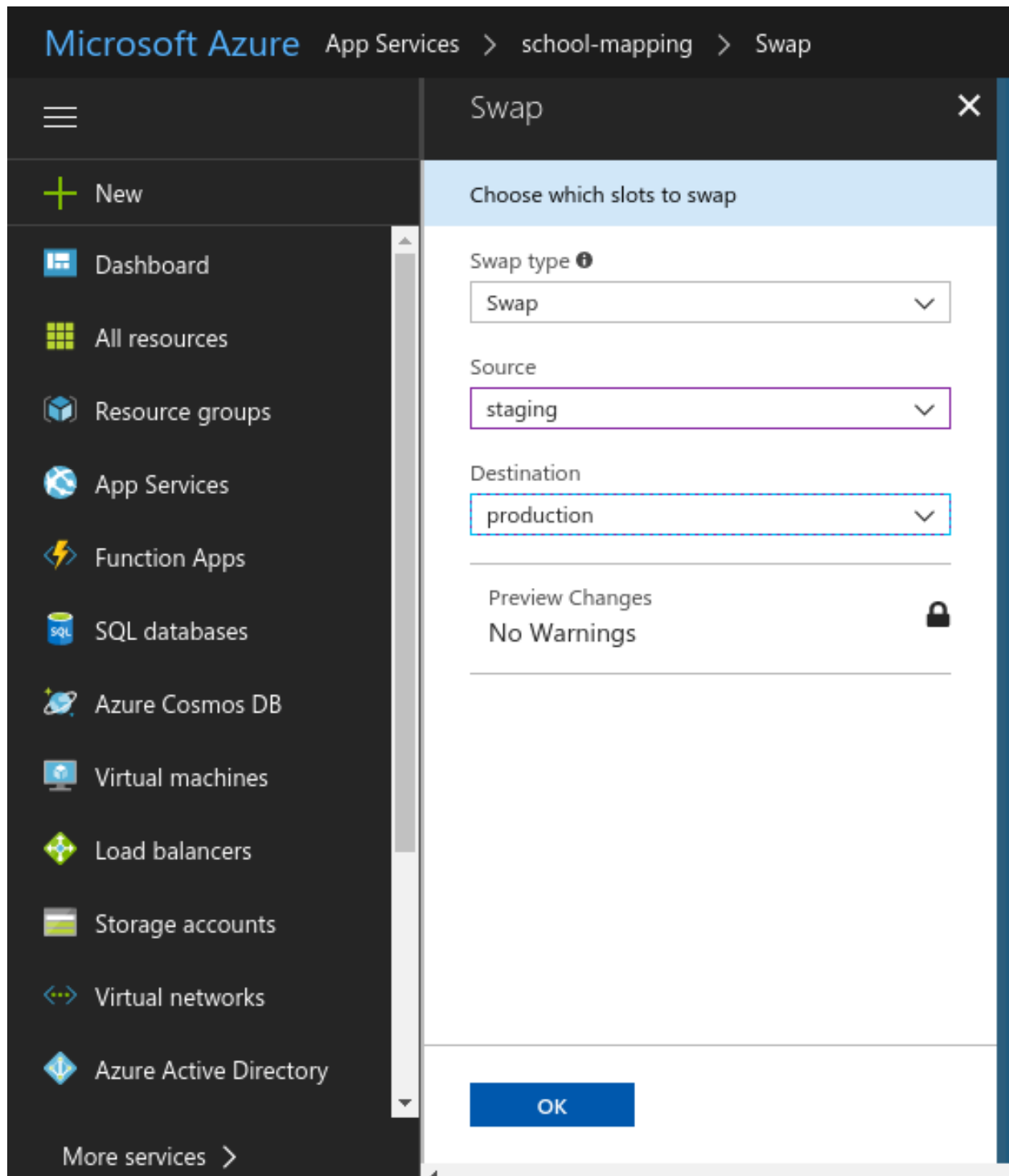
To make the swap, go to the [Azure Panel](#) > App Services, find the project you are working on and select it.



In the overview section you will find the “Swap” button, you can click in it.



Make sure to select in the Swap panel “staging” as source and “production” as destination.



Click OK to make the swap. You will get a notification once it finishes.

To replace your changes for the last working version of the application, just repeat this procedure.

8.2.4 Swap staging to production (CLI)

If you have the right permissions in place you can make the swap using the [Azure CLI](#).

To do this, make sure you are in the ASM mode running

```
azure config mode --help
```

Then you can use

```
azure site swap <appname>
```

GitHub workflow best practices

The UNICEF Innovation development team uses GitHub to host open source projects. This document explains the GitHub workflow maintainers and developers use. It also offers suggestions for best practices on using available tools and integrations.

This document explains...

- How to make a new GitHub repository
- How to maintain a GitHub repository
- How to communicate effectively

9.1 How to make a new GitHub repository

This section explains steps to follow when creating a new GitHub repository. These makes projects more organized, easier to follow from an outsider's perspective, and boosts visibility of development.

9.1.1 Repository creation


These steps focus on the initial repository creation, from github.com/new.

1. **Set a meaningful name:**
 - Try to make purpose obvious in title
 - Use hyphens (–) instead of underscores (_)
2. **Write a description:** Write one or two sentences to quickly describe the project
3. **Initialize with README:** Check to add a README .md file
 - If one is not yet written, initialize one
4. **Add .gitignore for project type:** Find project's programming language and choose its .gitignore file, if available

5. **Add BSD-3 Clause license:** Standard license used for UNICEF Innovation projects

Create a new repository

A repository contains all the files for your project, including the revision history.


Owner
 unicef ▾


Repository name
my-new-project ✓

Great repository names are short and memorable. Need inspiration? How about **legendary-octo-tribble**.

Description (optional)


Aggregate school data for visualizations in a web UI front-end

☐  **Public**
Anyone can see this repository. You choose who can commit.

☒  **Private**
You choose who can see and commit to this repository.

☒ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **Node** ▾

Add a license: **BSD 3-Clause "New" or "Revised" License** ▾ 

Create repository

Fig. 1: Example of creating a new repository

9.1.2 Configure repository

Now, the repository is created. Make sure these settings are updated:

Disable unneeded tools

Disable any unneeded features or repository tools. If they are needed, they can be turned on again later. Turning off unneeded features makes it easier for someone to find the useful places in the project. It can also indicate if the thing they are looking for (e.g. documentation) is somewhere else.

These features are found under the *Settings* menu for every repository.

Features

☐ **Wikis** ✓
GitHub Wikis is a simple way to let others contribute content. Any GitHub user can create and edit pages to use for documentation, examples, support, or anything you wish.

☒ **Restrict editing to users in teams with push access only**
Public wikis will still be readable by everyone.

☒ **Issues**
Issues integrate lightweight task tracking into your repository. Keep projects on track with issue labels and milestones, and reference them in commit messages.

Get organized with issue templates
Give contributors issue templates that help you cut through the noise and help them push your project forward.[Set up templates](#)

☐ **Projects** ✓
Project boards on GitHub help you organize and prioritize your work. You can create project boards for specific feature work, comprehensive roadmaps, or even release checklists.

Fig. 2: Disable unneeded tools, like a wiki or project boards. There may be documentation or project boards set up elsewhere. This does not disable organization-level project boards.

Set description, URL, topic tags

Make sure every repository has a description and topic tags set. If there is a URL to view a demo of the project or read more about it, include it too.

At the top of every GitHub repository, there are fields of metadata for a description, a URL, and topic tags. A description and URL helps someone understand the project in one or two sentences or see a live demo. Topic tags raise visibility in the GitHub ecosystem and help other people discover new projects.

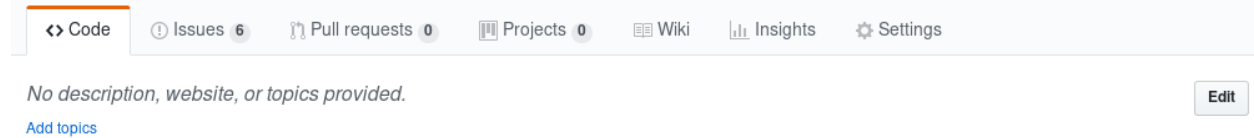


Fig. 3: Example of no description, URL, or topic tags

Do not leave a repository empty like this. Use the *Edit* button on the right to change the description and URL. Two text boxes will appear.

For topic tags, click *Add topics* towards the left. Choose tags related to your project to help identify it. Consider programming languages, frameworks, or other software used. Tags like `humanitarian` or `united-nations` are also examples of tags for types of software.

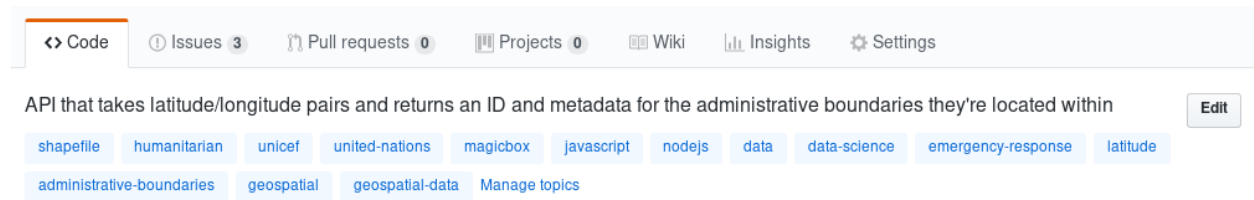


Fig. 4: magicbox-latlon-admin-server with description and topic tags

9.1.3 Set up useful labels

Labels are visual organization tools for your GitHub project. They make issues easier to sort and prioritize tasks. Additionally, they also help new contributors identify areas of interest for your project. They can help improve awareness of different types of contribution methods in your project (e.g. design and documentation tasks).

Configure each repository's labels in a way that makes sense for your project. The labels should mean something to *you* so they are easily applied for sorting later. Every repository's issue and pull request labels are found under the *Issues* tab with the *Labels* button.

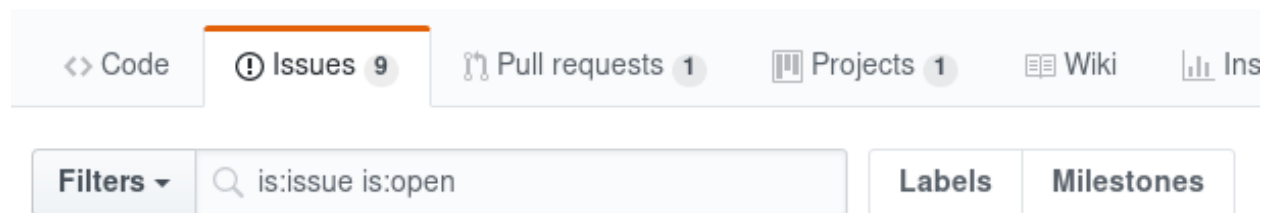


Fig. 5: Click the *Labels* button towards the right of the search bar

A good example of labels is [here](#) on the [unicef/magicbox](#) repository. (To view the color code for a given label, click the “Edit” on its row.)

Not all of these labels will be helpful for a new project. Take ones that make sense, and make new labels specific to the project, if needed.

9.1.4 Set up continuous integration (CI)

Note: To be written.

9.1.5 Set up code health checks with CodeClimate

Note: To be written.

9.2 How to maintain a GitHub repository

This section focuses on “housekeeping” with GitHub projects, including labels and project boards.

Housekeeping is important to maintain a repository. This organizes bugs, feature requests, and the project itself. Organized projects help active contributors stay on track and make realistic deadlines. It also helps new contributors understand what is going on.

Housekeeping has five parts:

1. Issue metadata
2. Adding labels
3. Updating project boards
4. Making pull requests
5. Reviewing pull requests

9.2.1 Update issue and pull request metadata

Every GitHub issue and pull request has four metadata properties:

1. **Assignees:** Who is currently working on this and who is the best point-of-contact for updates
2. **Labels:** Visual cues on task status and importance (see below)
3. **Projects:** Advanced business process management (see below)
4. **Milestone:** Relevant feature or version milestone for an issue or pull request

Assignees and labels should always be used at a minimum. Use projects and milestones when they are available.

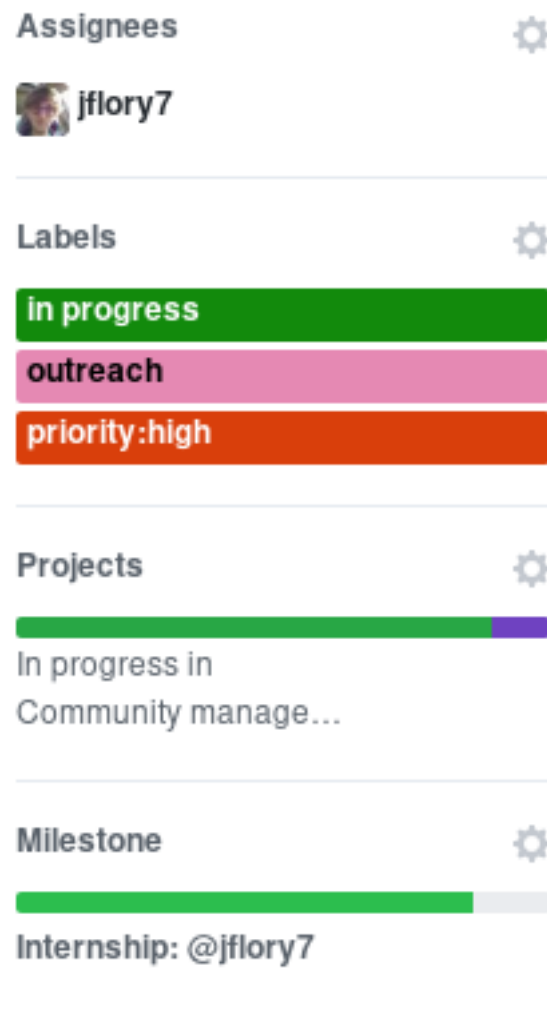


Fig. 6: Set assignees, labels, project boards, and milestones from the side column in every GitHub issue or pull request

9.2.2 Adding labels to issues

Above, labels were mentioned as part of issue and pull request metadata. Maintaining and using labels is a good habit. An issue or pull request might have two to four labels, depending on how the project was set up.

If labels are not yet configured, read [Set up useful labels](#).

Once a week, check issues and pull requests to see if tags are up-to-date. Update or change any labels that are stale (such as priority labels). Add labels from the metadata sub-menu when you open an issue or pull request.

9.2.3 Updating project boards

[GitHub project boards](#) are an organizational tool for the project. They use a [kanban-style](#) approach to organizing GitHub issues and pull requests. Our workflow is explained [on Opensource.com](#).

To update and maintain the project boards. . .

1. Make sure any issues or pull requests not shown are added to the board
2. Ensure important issues are organized by *In progress* or *To Do*
3. Issues not yet ready for consideration go on the backlog
4. All items under *In progress* or *To Do* columns should be GitHub issues, **not** note cards (note cards are okay for the backlog column)

9.2.4 Making pull requests

All major changes to the project should **always be made through a pull request (PR)**. Pull requests are like a registry of changes for a project. It is easy for someone to see what is going in and out of a project. Outside contributors will always have to make pull requests, so it is good practice for core / trusted developers to use pull requests too.

Follow contributing guidelines

The contributing guidelines for all MagicBox projects live [in the unicef/magicbox repository](#).

Always follow these contributing guidelines when working in the project. These are the standards and rules we ask the community to follow when contributing. As project maintainers, it is our responsibility to hold ourselves to the same standards we ask of others. Thus, always make sure current development practices are in-line with what our guidelines.

Write useful commit messages

Writing useful commit messages is a good practice to follow. When looking through project commits, it should be somewhat clear what has changed in the project and how. Short or nondescript commit messages are not helpful to maintainers or new contributors. Commit messages do not need to be paragraphs, but they should clearly indicate what changed or why something changed.

Read [this blog post](#) for more information about keeping git history clean and tidy with `git rebase`.

9.2.5 Reviewing pull requests

Note: To be written.

- triage, triage, triage
- CI
- health checks
- code review comments

9.3 Communicating about development

Communication about development should be kept public as much as possible in [our Gitter chat](#). Whenever you make a new pull request, always share the link in the main Gitter chat room. This lets other developers know you made a change and also gives them an opportunity to review your code. And if you want a code review, be sure to ask for it too.

CHAPTER 10

Indices and tables

- `genindex`
- `modindex`
- `search`