DataTig

Jan 03, 2023

Contents

DataTig is a set of tools to help when:

- A community of people want to crowd source a data set
- They use a git repository to store the data

In these cases, the tools can be used on the git repository to help people contribute new data or edit existing data, check data quality and transform the data into more useful forms for everyone to re-use.

CHAPTER 1

Tutorial

You and your friends in the city are keen cyclists, and want to start collecting information on local things for cyclists.

You all want to be able to contribute.

DataTig can help!

Work through the following tutorial in order to understand more.

This tutorial assumes that you are comfortable using Git and command line tools.

This tutorial assumes you are using Linux or Mac, and you have Python 3 installed.

This tutorial assumes you want to use certain services, like GitHub. No alternatives will be given in the tutorial, to keep the tutorial simple. In reality, you can use DataTig with many different services. All services suggested in this tutorial have a free option, so you should be able to complete it with no cost.

1.1 Setting up a site on GitHub

1.1.1 What this section covers

- How to start a new repository and site
- How to add your first data
- How to run the tool and see the website that results

1.1.2 Create a repository on GitHub & clone it

The data will be stored in GitHub.

Create a new repository in GitHub and select the *Add a README file* option.

Clone this repository to your local computer.

1.1.3 Set up a DataTig site

In order for DataTig to know that a git repository has data it can process, it needs a configuration file.

In the top level of the repository, create a file *datatig.yaml*. It's contents should be:

```
title: Local Cycling Resources in CityCity
description: Listing all the useful cycling resources in CityCity
```

Commit that new file to Git and push to GitHub.

Now let's add our first resources!

1.1.4 Start with some bike shops

You decide that listing bike shops is a good thing to do.

It's always handy to be able to get a small part quickly.

Create a directory called *shops*.

We are going to store our information in a YAML file, one per shop.

Create a file called *alices-bikes.yaml* in the *shops* directory. For the contents, put:

```
title: Alice's Bikes
url: http://www.alicesbikeshop.co.uk/
```

Create a file called *bobs-bikes.yaml* in the *shops* directory. For the contents, put:

```
title: Bob's Bikes
url: http://www.bobsbikeshop.co.uk/
```

Commit these 2 new files to Git and push to GitHub. We've just added our first data!

1.1.5 Tell DataTig about this new data

Just adding data to the repository isn't enough; we have to tell DataTig about the new data.

To do this, we add information to the *datatig.yml* file we created earlier. Set it's contents to be:

```
title: Local Cycling Resources in CityCity
description: Listing all the useful cycling resources in CityCity
# Every type of data we list should have an entry here
types:
# This defines the shops data
# Every type should have an unique identifier
- id: shops
# For every type, we need to know in which directory we can find the data
  directory: shops
```

Commit this change and push to GitHub.

1.1.6 Install DataTig

This is the minimal information needed to actually run DataTig, so let's do that now!

First install it:

```
pip install datatig
```

1.1.7 Use DataTig to check your data

Now we'll run the tool over your data.

The tool can do several actions. It can check the data to make sure that it is in the correct format. To do so, run:

```
python -m datatig.cli check .
```

This should run with no output, meaning there were no errors.

1.1.8 Use DataTig to build a website about your data

The tool can also build static files that can be served as a website.

This website contains the data in several useful forms.

First, let's build the website.

```
python -m datatig.cli build . --staticsiteoutput _site
```

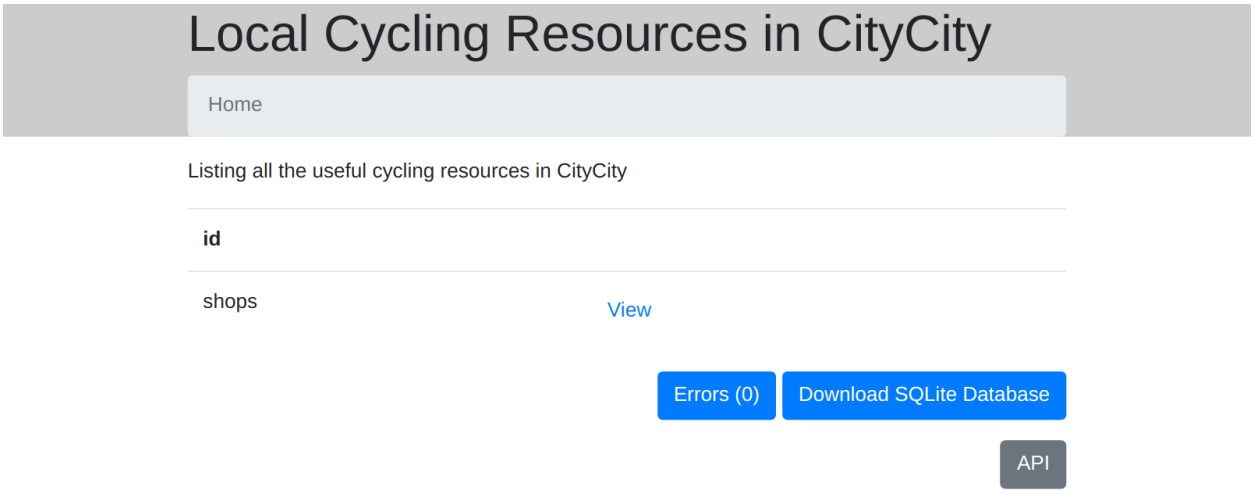
This should build fine (you will not see any output) and in the new *_site* directory that just appeared, you should see some files.

We can serve these so they are easy to open in your web browser.

```
sh -c "cd _site && python3 -m http.server"
```

Now open a web browser and go to http://localhost:8000/

You should see a basic website with a link for shops.



Clicking on "shops" will let you explore the 2 shops we just added. But it only shows us basic information - let's see if we can make that more helpfull.

1.1.9 Next

To continue, visit the next section

1.2 Specifying Structure

1.2.1 Previous

Before doing this, *make sure you have done the previous step*.

1.2.2 What this section covers

- Telling DataTig about the structure of your data so it can be more helpful

1.2.3 Adding Fields Definitions

If we tell DataTig what structure data to expect, it can be much more helpful.

Edit *datatig.yml*. Add in the section about *fields* and *list_fields*:

```
title: Local Cycling Resources in CityCity
description: Listing all the useful cycling resources in CityCity
# Every type of data we list should have an entry here
types:
# This defines the shops data
# Every type should have an unique identifier
- id: shops
  # For every type, we need to know in which directory we can find the data
  directory: shops
  # Define some fields
  fields:
  # Every field should have an unique identifier
  - id: title
    # For each field, we need to tell it where in the data to look for the value.
    key: title
    # For each field, we should give it a friendly title
    title: Title of Shop
  - id: url
    key: url
    title: URL of Website about Shop
  # We need to tell DataTig which fields are important to list when looking at a list␣
↪of all the data.
  list_fields:
  - title
```

In the fields section, we can tell DataTig about as many fields as you want.

Let's also add some data that doesn't meet this structure.

Create a file called *digital-dan.yaml* in the *shops* directory. For the contents, put:

```
title: Digital Dan's Bike Shop
url:
  - http://www.digital-dans-bike-shop.com
  - http://www.facebook.com/digitaldansbikeshop
```

We know 2 URL's, so we are going to put them both in.

Now run the check function again.

```
python -m datatig.cli check .
```

This time, the tool should tell you there is a problem with a bit of data.

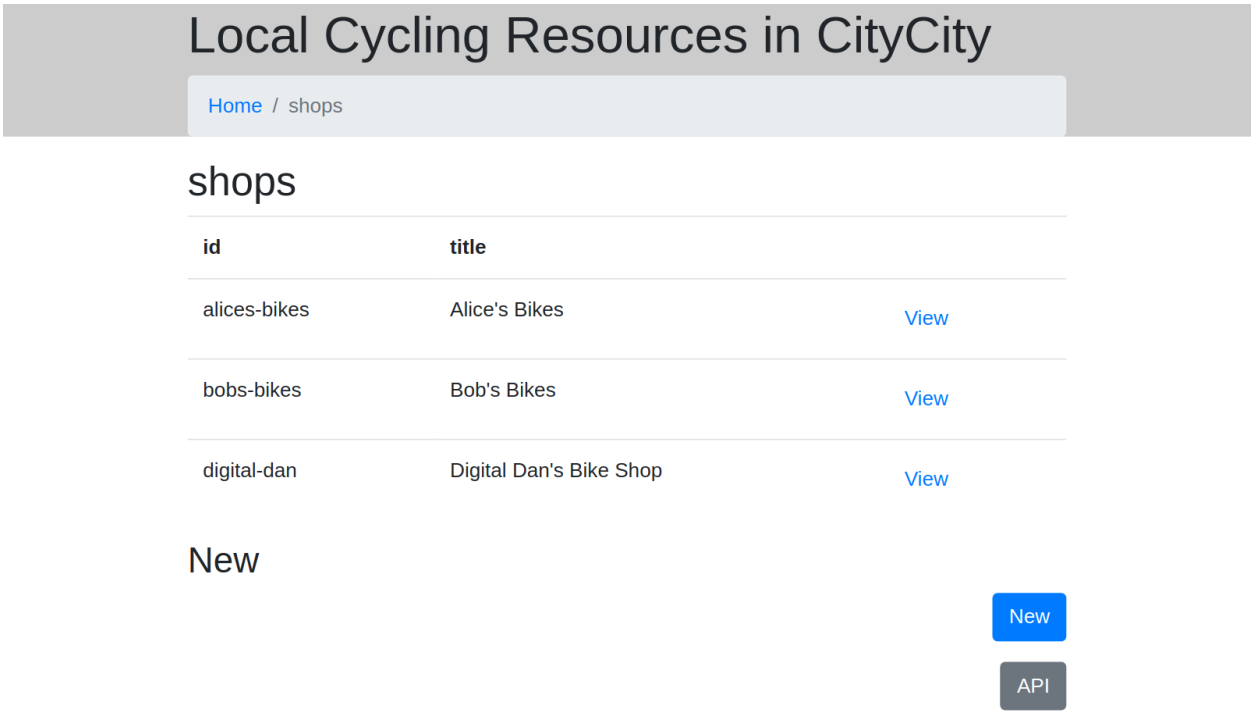
It's telling us there is an error because it expects the URL field to only have one item, not several. Let's ignore that for now.

Instead let's build the website again and look at it. This is the same instructions as the previous step - they are:

```
python -m datatig.cli build . --staticsiteoutput _site
sh -c "cd _site && python3 -m http.server"
```

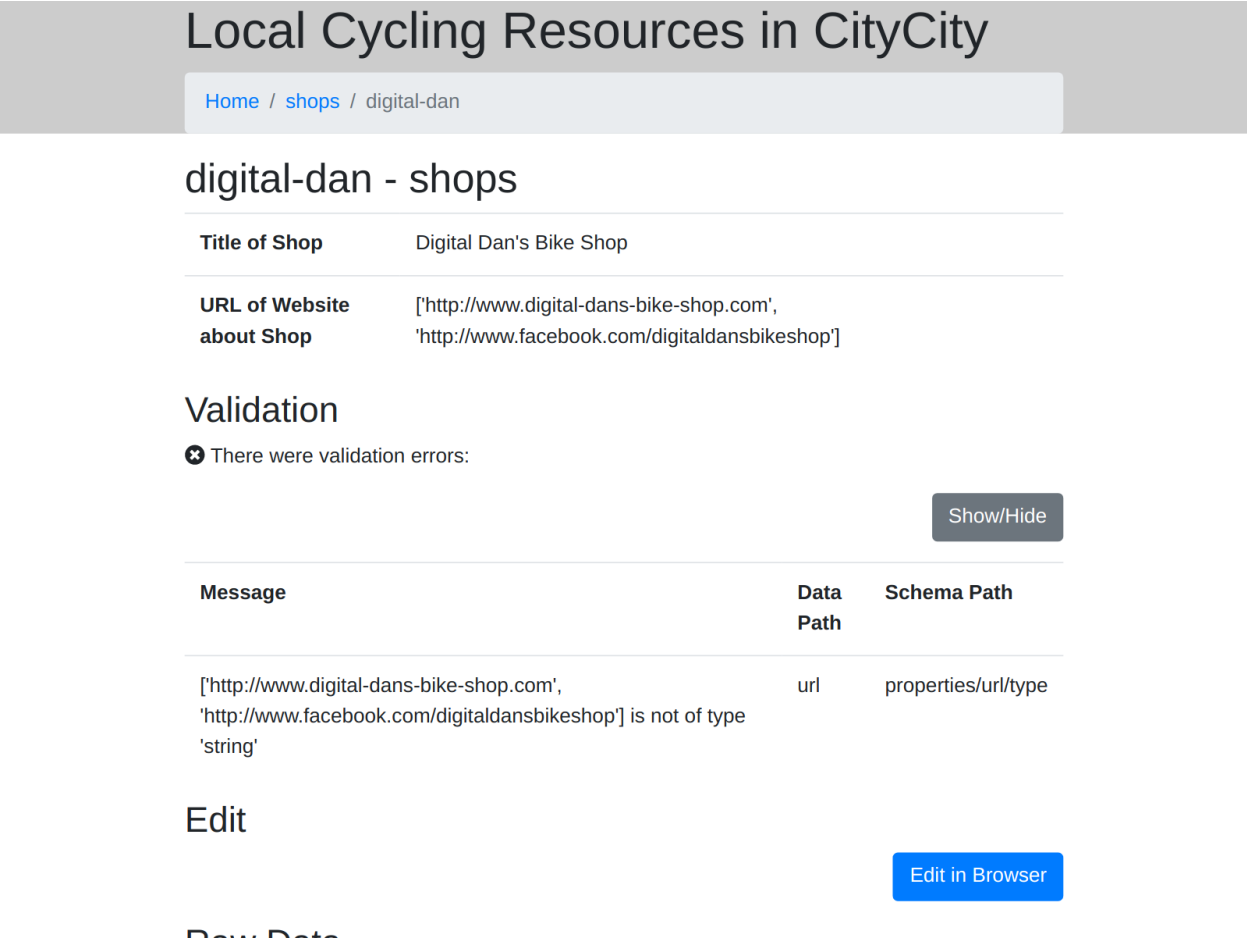
Open a web browser and go to http://localhost:8000/type/shops

You should see more information on the listings page - the titles are included. (The URL's aren't because that was not included in *list_fields*)



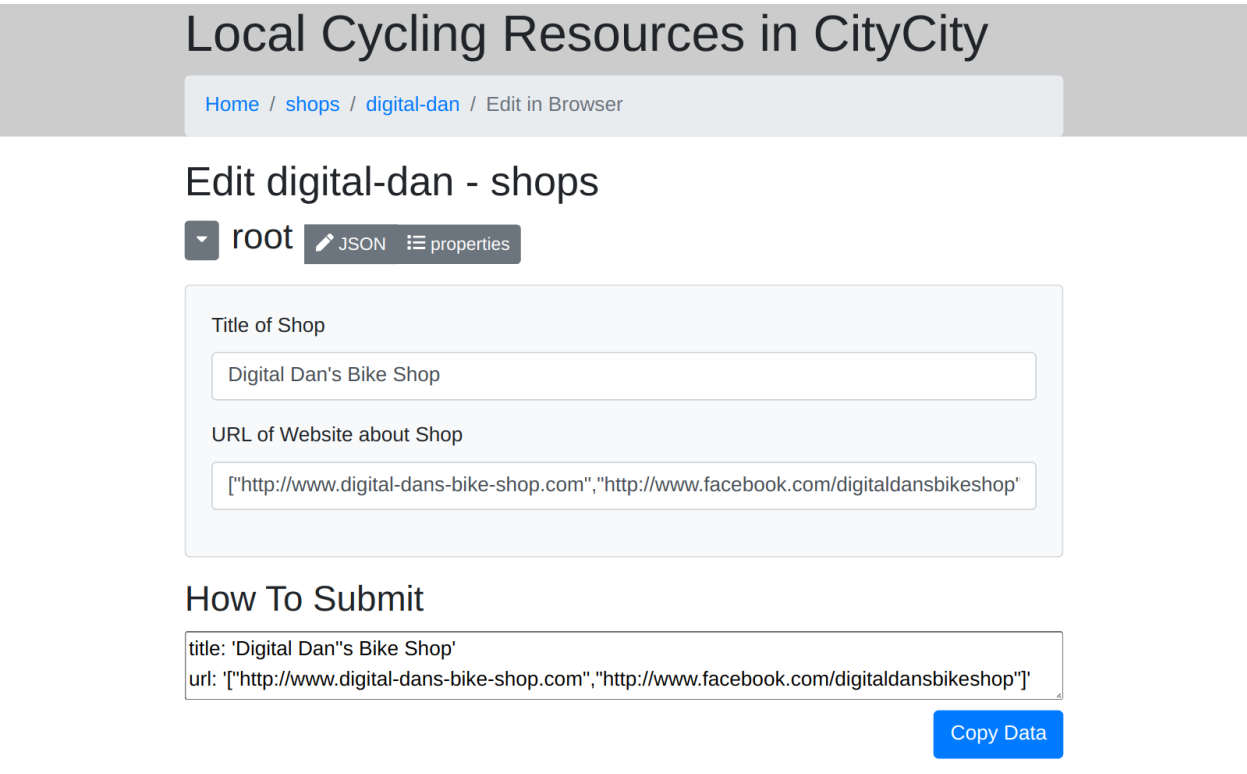
Click on the Digital Dan shop.

You should see more information on the fields and information on the error it found.



Because we have told it information about the fields, there is also an “Edit in Browser” button. Click it.

You should see a form that a person can fill in directly in their browser, and instructions on how to save the data.



1.2.4 Next

To continue, visit the next section

1.3 Checking data automatically

1.3.1 Previous

Before doing this, *make sure you have done the previous step*.

1.3.2 What this section covers

- We have seen how to check data manually; lets have GitHub check any data automatically

1.3.3 Getting GitHub to check data automatically

Previously we have run checks manually by running:

```
python -m datatig.cli check .
```

This is great, but it's not great we have to remember to do this! It would be very easy to forget.

Fortunately, using a free GitHub feature called Actions we can get GitHub to check our data automatically.

Create a directory called "*.github*". Inside that, create a directory called "*workflows*". Inside that, create a file called "*check.yml*", with the contents:

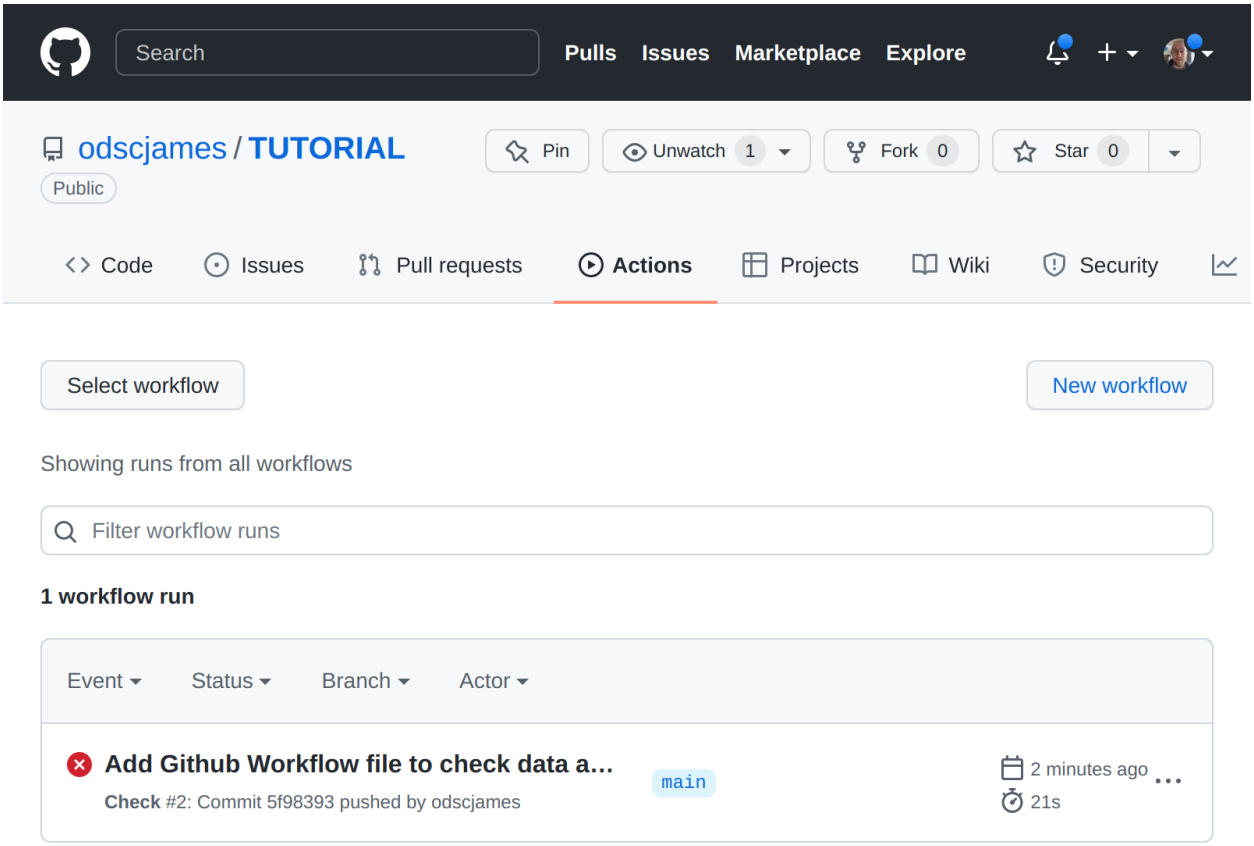
```yaml
name: Check
on: [push, pull_request]

jobs:
  check:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Setup python
      uses: actions/setup-python@v2
      with:
        python-version: 3.9
        architecture: x64

    - run: pip install datatig
    - run: python -m datatig.cli check .
```

The contents tell GitHub Actions when to run, how to install your data and DataTig and finally how to get DataTig to check the data. We won't go into the detail of these in this tutorial.

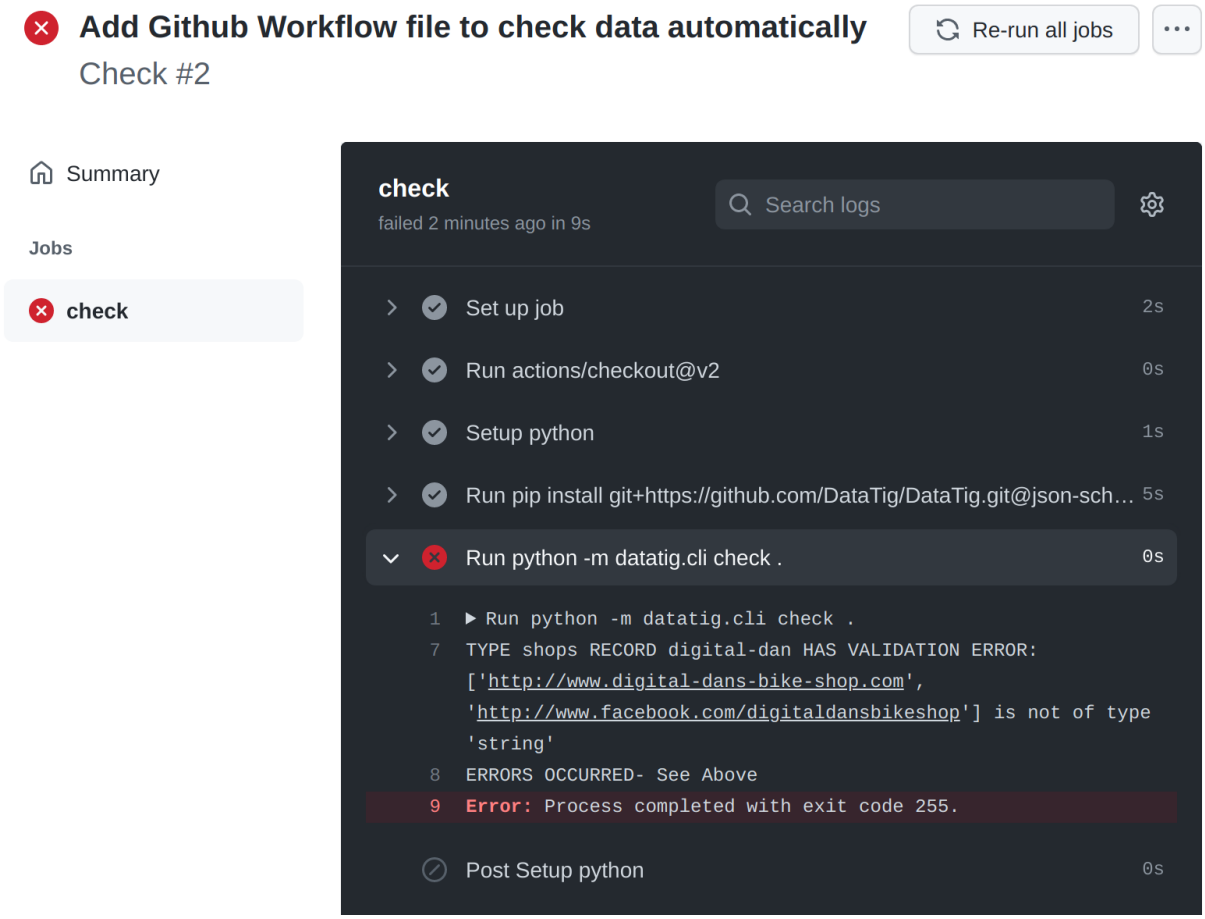
Commit your file *.github/workflows/check.yml* and push it to GitHub.

Go to your repository on GitHub and click the "Actions" tab.

You may have to wait a minute. But after refreshing the tab a few times, you should see GitHub starting to run your action. After a minute it should go Red, indicating that it has failed and indicating that there is a problem with your data.



Click on the run, then check and expand the broken item to see the details of the problem.



Let's try and fix this. We can only list one URL - so let's remove the Facebook one.

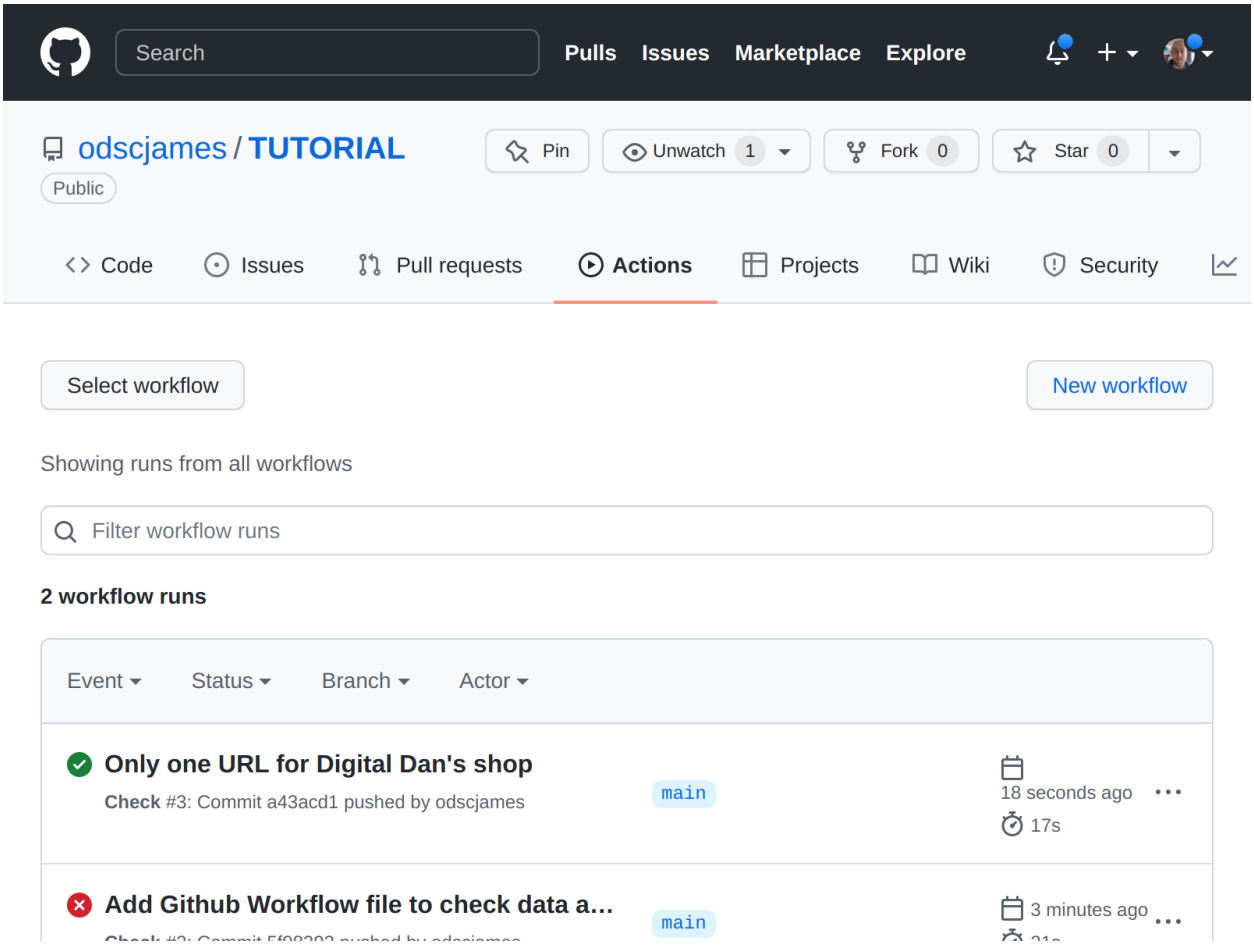
Edit the file called *shops/digital-dan.yaml*. Add the URL:

```
title: Digital Dan's Bike Shop
url: http://www.digital-dans-bike-shop.com
```

Commit your file *shops/digital-dan.yaml* and push it to GitHub.

Go to your repository on GitHub and click the "Actions" tab.

Shortly you should see a new job starting and this time, when it has finished it will turn Green, indicating there are no problems with the data.



GitHub will now automatically check any new data you add, and tell you of any problems.

(Note: *the contents of this section are also available as part of the how-to section*)

1.3.4 Next

To continue, visit the next section

1.4 Deploying the Static Site

1.4.1 Previous

Before doing this, *make sure you have done the previous step*.

1.4.2 What this section covers

- We have seen how to build a website for this manually; lets put this website online automatically

1.4.3 Setting up the website

We have seen how to manually build the website and see it on your computer.

```
python -m datatig.cli build . --staticsiteoutput _site
sh -c "cd _site && python3 -m http.server"
```

This is great, but it's not great we have to remember to update this. Also, this is only available to you. Let's make this website available to everyone.

Fortunately, there are several services that will help with this. We will use one called Netlify.

Go to https://www.netlify.com/ and sign up using your GitHub login.

You will be given the option of creating a new website.

Do so, and pick the repository we have created.

It will now try and build the website and fail, because we haven't told it how to.

In the top level of the repository, create a file *netlify.toml*. It's contents should be:

```
[build]
  publish = "out"
  command = "pip install datatig && export PYTHONPATH=$(pwd) && python -m datatig.cli␣
↪build . --staticsiteoutput out"
```

In the top level of the repository, create a file *runtime.txt*. It's contents should be:

```
3.9
```

This should be a single line, with no extra spaces. It tells Netlify which version of Python to use.

Commit your files *netlify.toml* and *runtime.txt* and push them to GitHub.

In Netlify, you should be able to go to your site and see that a new build is running. This time, it should succeed.

You should be able to click on the address and see this website online.

As you have just seen, any change in GitHub will trigger a build on your website in Netlify, so that this website is kept up to date.

1.4.4 Next

To continue, visit the next section

1.5 Encouraging Contributions

1.5.1 Previous

Before doing this, *make sure you have done the previous step*.

1.5.2 What this section covers

- DataTig can help encourage contributions from people by providing tools and instructions

1.5.3 Tell DataTig where our data lives

We want to encourage people to contribute new data to this site.

DataTig can help us do this, but before it does that, it needs to know where your data lives.

Edit the *datatig.yml* file. We need to add a new section at the bottom:

```
githost:
  type: github
  url: xxxxx/yyyyy
```

Make sure the URL value matches the GitHub repository you created.

Commit this change and push to GitHub.

Now wait a minute until the new version of your static site is rebuilt and deployed.

Go and click on a page for a bike shop. Look in the *Edit* section.

There is a new button, *Edit Raw data directly on GitHub*. This will take you direct to the relevant file on GitHub.

But also, if you click *Edit in Browser* at the bottom of this page you will see a button to take you to GitHub and instructions for people on how to edit.

This encourages people to edit the data and send you pull requests for you to accept or refuse.

1.5.4 Next

To continue, visit the next section

1.6 Using data (API)

1.6.1 Previous

Before doing this, *make sure you have done the previous step*.

1.6.2 What this section covers

- Ways other people can use this data

1.6.3 Others can use your data

We've started our list of bike shops; we've set up a website that encourages other people to contribute to it.

We want others to use this data.

Fortunately, DataTig includes tools to make the data we are collecting very accessible.

Go to the website that Netlify is building for you.

On the home page is a button to "Download SQLite Database". This downloads a database file that includes all the data. It can be opened it a tool like https://sqlitebrowser.org/

On the home page is an "API" button. This is a JSON API that contains data, and links to more API URL's to find more data.

Click on “Shops” then any shop. There is a button to “Download Raw JSON Data”. This will give you the data for that shop in JSON form.

In themselves, these things may not seem very useful. What will someone do with these?

But the point here is that DataTig is making your data available in an API (“Application Programming Interface”) for others to consume and use.

You, or other people, can build other tools such as websites on top of these APIs.

By doing this, others can take your data and re-use it. This means your data has helped more people. It also means that people will be keener to submit new data and corrections to your repository.

This Open Data is something DataTig is designed to encourage.

1.6.4 Next

That’s it! The Tutorial is now complete.

Browse the other sections of the docs for more

CHAPTER 2

Explanation

2.1 Site, Types and Records

2.1.1 Site

Each site is a separate website and data repository.

It should ideally be in one git repository by itself - so one git repository should be thought of as one site.

You can think of sites like a whole database in a database system.

2.1.2 Type

Each site can have multiple types of data in it.

Each type has it's own set of configuration, rules and guidance.

When a site is build, the data for each type is held separately for easy querying.

Each type is identified by it's id, which should be unique.

Each type has a:

- id
- title
- a directory set - this is where in the git repository it's data is stored
- a JSON Schema - if this is included, any data will be validated against it and a web editor will be available
- a default format that should be used when creating new records (eg YAML or JSON)
- a guide spreadsheet - TODO
- a list of field definitions, and which fields to show when listing the data

You can think of types like a table in a database system.

2.1.3 Field

Each type can have a list of field definitions.

These define interesting data to pull out.

Note that there is currently some duplication between the definition of these and the JSON Schema definition.

2.1.4 Record

Each Type can have multiple records of data.

Each record is identified by it's id, which should be unique within it's type.

Each record has a:

- id
- data, which can be edited as a whole and also from which values for each field are extracted

You can think of types like an individual row in a table in a database system.

CHAPTER 3

How To Guide

3.1 Use GitHub actions to check your data

3.1.1 Scenario

You currently have a DataTig site in a GitHub repository.

You want to make sure the data is correct, at all times and when someone makes a pull request.

You can set up GitHub Actions to check this for you.

3.1.2 Steps

Create a YAML file in the GitHub repository.

It must be in the directory: *.github/workflows/*

It must have a YAML extension but it can have any file name you want. We suggest: *check.yml*

The contents should be:

```yaml
name: Check
on: [push, pull_request]

jobs:
  check:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Setup python
      uses: actions/setup-python@v2
      with:
        python-version: 3.9
        architecture: x64
```

(continues on next page)

(continued from previous page)

```
  - run: pip install datatig
  - run: python -m datatig.cli check .
```

Commit this and merge it into your default branch (*main*, or whatever you use).

That's it!

Note that pull requests will only be checked if this file is in the code the pull request is based on. In other words, existing pull requests may not be checked. If you have an existing pull request that you would like to be checked, you must rebase it onto a version of the data that does have the above file in it.

Optionally: Require checks to pass

You can set GitHub to require this check to pass before code is merged.

Before doing this, the check must have run at least once.

To do so:

1. Go to the GitHub repository; eg https://github.com/xxx/yyyy
2. Click *Settings*
3. Click *Branches*
4. In the *Branch protection rules* section either add a new rule for the branch you want, or edit the existing rule
5. Select *Require status checks to pass before merging*
6. Select *check*
7. Save changes

In Tutorial

(Note: *the contents of this section are also available as part of the tutorial*)

CHAPTER 4

Reference

4.1 Data Formats

This tool works with data stored in several different formats in a git repository. In general these are listed starting with the one we recommend the most, so if you aren't sure start at the top.

4.1.1 YAML files, one per record

Each type of data should have a directory of it's own.

Each record is one file.

The id of each record is part of the name of the file. eg:

1. There is a file *cats/bob.yaml*.
2. The type is configured to be the stored in the directory *cats*.
3. The id of the data is *bob*.

We recommend this because:

- One file per record means that many people editing different records at once will not cause merge request conflicts
- Technically aware humans usually find YAML is easier to read or edit by hand

4.1.2 JSON files, one per record

Each type of data should have a directory of it's own.

Each record is one file.

The id of each record is part of the name of the file. eg:

1. There is a file *cats/bob.json*.

2. The type is configured to be the stored in the directory *cats*.
3. The id of the data is *bob*.

We recommend this because:

- One file per record means that many people editing different records at once will not cause merge request conflicts

4.2 CLI

4.2.1 Call

Call via Python:

```
python -m datatig.cli --help
```

4.2.2 Build sub-command

This takes a site and builds some outputs for you.

Call with the directory of the data and at least one of these options:

- *–staticsiteoutput* - A directory in which the files of the static site will be placed. This can already exist.
- *–sqliteoutput* - A location at which a SQL database file will be placed. This should not already exist.
- *–frictionlessoutput* - A location at which a Frictionless Data Zip file will be placed. This should not already exist.

```
python -m datatig.cli build . --staticsiteoutput _site --sqliteoutput database.sqlite
```

Any build errors will be printed to screen. (Data validation errors will not be) If encountered, the process will try to continue ignoring the problem. The exit code of the process will be 0 if a success, or -1 if there were any errors. This means you can use this as part of a C.I./C.D. pipeline and check the response.

If you select static site, you can also pass:

- *–staticsiteurl* - Base URL that resulting website will be hosted at. Should not have a trailing slash. eg 'http://www.example.com/sub-directory'

4.2.3 Check sub-command

This takes a site and checks it for you.

Call with the directory of the data.

Any build errors or data validation errors will be printed to screen.

The exit code of the process will be 0 if a success, or -1 if there were any errors. This means you can use this as part of a C.I./C.D. pipeline and check the response.

```
python -m datatig.cli check .
```

4.2.4 Versioned Build sub-command

This is currently used for internal testing and is not documented

4.3 Python API

There is a Python API that you can use. This means you can use the features of DataTig as part of other Python programs and processes.

Look at the function *go* in the file *datatig/process.py* for details.

4.4 Site Configuration

4.4.1 Location

Each site should have a *datatig.json* or *datatig.yaml* file at it's root. This contains configuration for that site.

The schema of each file is the same.

We recommend using *datatig.yaml* - it is easier to edit, and you can have comments.

4.4.2 General

- *title* - A string. The title of the whole site.
- *description* - A string. A description for the whole site.

A YAML example:

```
title: Test register
description: The data for a test
```

4.4.3 Types

We need to know information about the types of data - think of types like a table in a database.

The *types* key is an array of type information.

Every type has the following options available:

- *id* - every type needs a unique ID.
- *directory* - the directory to look in for data files for this type. Relative to the root of the git repository.
- *fields* - a list of field information. See section below for more.
- *guide_form_xlsx* - A XLSX file to use for generating and importing spreadsheets of a record. See *SpreadSheet forms <https://spreadsheet-forms.readthedocs.io/en/latest/index.html>_*.
- *list_fields* - a list of field id's to show in the static site web interface.
- *json_schema* - a path to a JSON Schema file for this type. Relative to the root of the git repository.
- *pretty_json_indent* - When writing back JSON files, how many spaces to ident by. Defaults to 4.

- *default_format* - When creating new records, what is the default format? 'json', 'md' or 'yaml'. Defaults to 'yaml'.
- *markdown_body_is_field* - When reading or writing markdown files, the body of the MarkDown file is put into a key with this name. Defaults to 'body'.

A YAML example:

```
types:
- id: lists
  directory: lists
  list_fields:
  - code
  - title_en
  json_schema: schema/list-schema.json
  pretty_json_indent: 2
  default_format: json
```

Fields

Every field needs to be defined.

- *id* - every field needs a unique ID within that type.
- *key* - the path in the data to find this value. Note paths are allowed, not just keys.
- *title* - a title for this field.
- *type* - a type for this field. Defaults to *string*.

Allowed types are:

- *string*
- *url*
- *list-strings*
- *date*
- *datetime*
- *boolean*
- *integer*

A YAML example:

```
types:
- id: lists
  fields:
  - id: code
    key: code
    title: Code
  - id: title_en
    key: name/en
    title: Name (EN)
  - id: url
    key: url
    title: URL
    type: url
  - id: description_en
```

(continues on next page)

(continued from previous page)

```
    key: description/en
    title: Description (EN)
```

4.4.4 Git Host

You can specify information about where this git repository is hosted.

Currently the only hosts supported are: * GitHub.com

In a *githost* object, specify the following keys:

- *url* - the URL of the repository. This should not contain the hostname but just the organisation and repository. eg *org-id/register*.
- *primary_branch* - the name of the default or primary branch. Defaults to *main*.

A YAML example:

```
githost:
    url: org-id/register
    primary_branch: main
```