Datasette Documentation

Simon Willison

Jul 11, 2019

Contents

A tool for exploring and publishing data

Datasette is a tool for exploring and publishing data. It helps people take data of any shape or size and publish that as an interactive, explorable website and accompanying API.

Datasette is aimed at data journalists, museum curators, archivists, local governments and anyone else who has data that they wish to share with the world. It is part of a *wider ecosystem of tools* dedicated to make working with structured data as productive as possible.

Explore a demo, watch a presentation about the project or *Try Datasette without installing anything using Glitch*.

More examples: https://github.com/simonw/datasette/wiki/Datasettes

CHAPTER 1

Contents

1.1 Getting started

1.1.1 Play with a live demo

The best way to experience Datasette for the first time is with a demo:

- fivethirtyeight.datasettes.com shows Datasette running against over 400 datasets imported from the FiveThirtyEight GitHub repository.
- sf-trees.datasettes.com demonstrates the datasette-cluster-map plugin running against 190,000 trees imported from data.sfgov.org.

1.1.2 Try Datasette without installing anything using Glitch

Glitch is a free online tool for building web apps directly from your web browser. You can use Glitch to try out Datasette without needing to install any software on your own computer.

Here's a demo project on Glitch which you can use as the basis for your own experiments:

glitch.com/~datasette-csvs

Glitch allows you to "remix" any project to create your own copy and start editing it in your browser. You can remix the `datasette-csvs` project by clicking this button:

Find a CSV file and drag it onto the Glitch file explorer panel - `datasette-csvs` will automatically convert it to a SQLite database (using csvs-to-sqlite) and allow you to start exploring it using Datasette.

If your CSV file has a `latitude` and `longitude` column you can visualize it on a map by uncommenting the `datasette-cluster-map` line in the `requirements.txt` file using the Glitch file editor.

Need some data? Try this Public Art Data for the city of Seattle - hit "Export" and select "CSV" to download it as a CSV file.

For more on how this works, see Running Datasette on Glitch.

1.1.3 Using Datasette on your own computer

First, follow the *Installation* instructions. Now you can run Datasette against a SQLite file on your computer using the following command:

```
datasette serve path/to/database.db
```

This will start a web server on port 8001 - visit http://localhost:8001/ to access the web interface.

`serve` is the default subcommand, you can omit it if you like.

Use Chrome on OS X? You can run datasette against your browser history like so:

```
datasette ~/Library/Application\ Support/Google/Chrome/Default/History
```

Now visiting http://localhost:8001/History/downloads will show you a web interface to browse your downloads data:

downloads

576 total rows in this table

This data as .json, .jsono

Link	id	current_path	target_path	start_time	received_bytes
1	1	/Users/simonw/Downloads/DropboxInstaller.dmg	/Users/simonw/Downloads/DropboxInstaller.dmg	13097290269022132	626688
2	2	/Users/simonw/Downloads/1Password-6.0.zip	/Users/simonw/Downloads/1Password-6.0.zip	13097291858423547	45885962
3	3	/Users/simonw/Downloads/Sublime Text Build 3083.dmg	/Users/simonw/Downloads/Sublime Text Build 3083.dmg	13097292198925765	10738996

http://localhost:8001/History/downloads.json will return that data as JSON:

```
{
    "database": "History",
    "columns": [
        "id",
        "current_path",
        "target_path",
        "start_time",
        "received_bytes",
        "total_bytes",
        ...
    ],
    "rows": [
        [
            1,
            "/Users/simonw/Downloads/DropboxInstaller.dmg",
            "/Users/simonw/Downloads/DropboxInstaller.dmg",
            13097290269022132,
            626688,
            0,
            ...
        ]
    ]
}
```

http://localhost:8001/History/downloads.json?_shape=objects will return that data as JSON in a more convenient but less efficient format:

```
{
    ...
    "rows": [
        {
            "start_time": 13097290269022132,
            "interrupt_reason": 0,
            "hash": "",
            "id": 1,
            "site_url": "",
            "referrer": "https://www.dropbox.com/downloading?src=index",
            ...
        }
    ]
}
```

1.1.4 datasette serve options

```
$ datasette serve --help

Usage: datasette serve [OPTIONS] [FILES]...

  Serve up specified SQLite database files with a web UI

Options:
  -i, --immutable PATH      Database files to open in immutable mode
  -h, --host TEXT           host for server, defaults to 127.0.0.1
  -p, --port INTEGER        port for server, defaults to 8001
  --debug                   Enable debug mode - useful for development
  --reload                  Automatically reload if database or code change detected -
                            useful for development
  --cors                    Enable CORS by serving Access-Control-Allow-Origin: *
  --load-extension PATH     Path to a SQLite extension to load
  --inspect-file TEXT       Path to JSON file created using "datasette inspect"
  -m, --metadata FILENAME   Path to JSON file containing license/source metadata
  --template-dir DIRECTORY  Path to directory containing custom templates
  --plugins-dir DIRECTORY   Path to directory containing custom plugins
  --static STATIC MOUNT     mountpoint:path-to-directory for serving static files
  --memory                  Make :memory: database available
  --config CONFIG           Set config option using configname:value
                            datasette.readthedocs.io/en/latest/config.html
  --version-note TEXT       Additional note to show on /-/versions
  --help-config             Show available config options
  --help                    Show this message and exit.
```

1.2 Installation

Note: If you just want to try Datasette out you don't need to install anything: see *Try Datasette without installing anything using Glitch*

There are two main options for installing Datasette. You can install it directly on to your machine, or you can install it using Docker.

Contents

1.2.1 Using Docker

A Docker image containing the latest release of Datasette is published to Docker Hub here: https://hub.docker.com/r/datasetteproject/datasette/

If you have Docker installed (for example with Docker for Mac on OS X) you can download and run this image like so:

```
docker run -p 8001:8001 -v `pwd`:/mnt \
    datasetteproject/datasette \
    datasette -p 8001 -h 0.0.0.0 /mnt/fixtures.db
```

This will start an instance of Datasette running on your machine's port 8001, serving the `fixtures.db` file in your current directory.

Now visit http://127.0.0.1:8001/ to access Datasette.

(You can download a copy of `fixtures.db` from https://latest.datasette.io/fixtures.db)

Loading Spatialite

The `datasetteproject/datasette` image includes a recent version of the *SpatiaLite extension* for SQLite. To load and enable that module, use the following command:

```
docker run -p 8001:8001 -v `pwd`:/mnt \
    datasetteproject/datasette \
    datasette -p 8001 -h 0.0.0.0 /mnt/fixtures.db \
    --load-extension=/usr/local/lib/mod_spatialite.so
```

You can confirm that SpatiaLite is successfully loaded by visiting http://127.0.0.1:8001/-/versions

Installing plugins

If you want to install plugins into your local Datasette Docker image you can do so using the following recipe. This will install the plugins and then save a brand new local image called `datasette-with-plugins`:

```
docker run datasetteproject/datasette \
    pip install datasette-vega

docker commit $(docker ps -lq) datasette-with-plugins
```

You can now run the new custom image like so:

```
docker run -p 8001:8001 -v `pwd`:/mnt \
    datasette-with-plugins \
    datasette -p 8001 -h 0.0.0.0 /mnt/fixtures.db
```

You can confirm that the plugins are installed by visiting http://127.0.0.1:8001/-/plugins

1.2.2 Install using pip

To run Datasette without Docker you will need Python 3.5 or higher.

You can install Datasette and its dependencies using `pip`:

```
pip install datasette
```

If you want to install Datasette in its own virtual environment, use this:

```
python -mvenv datasette-venv
source datasette-venv/bin/activate
pip install datasette
```

You can now run Datasette like so:

```
datasette fixtures.db
```

If you want to start making contributions to the Datasette project by installing a copy that lets you directly modify the code, take a look at our guide to *Setting up a development environment*.

1.3 The Datasette Ecosystem

Datasette sits at the center of a growing ecosystem of open source tools aimed at making it as easy as possible to gather, analyze and publish interesting data.

These tools are divided into two main groups: tools for building SQLite databases (for use with Datasette) and plugins that extend Datasette's functionality.

1.3.1 Tools for creating SQLite databases

csvs-to-sqlite

csvs-to-sqlite lets you take one or more CSV files and load them into a SQLite database. It can also extract repeated columns out into a separate table and configure SQLite full-text search against the contents of specific columns.

sqlite-utils

sqlite-utils is a Python library and CLI tool that provides shortcuts for loading data into SQLite. It can be used programmatically (e.g. in a Jupyter notebook) to load data, and will automatically create SQLite tables with the necessary schema.

The CLI tool can consume JSON streams directly and use them to create tables. It can also be used to query SQLite databases and output the results as CSV or JSON.

See sqlite-utils: a Python library and CLI tool for building SQLite databases for more.

db-to-sqlite

db-to-sqlite is a CLI tool that builds on top of SQLAlchemy and allows you to connect to any database supported by that library (including MySQL, oracle and PostgreSQL), run a SQL query and save the results to a new table in a SQLite database.

You can mirror an entire database (including copying foreign key relationships) with the `--all` option:

```
$ db-to-sqlite --connection="postgresql://simonw@localhost/myblog" --all blog.db
```

dbf-to-sqlite

dbf-to-sqlite works with dBase files such as those produced by Visual FoxPro. It is a command-line tool that can convert one or more `.dbf` file to tables in a SQLite database.

markdown-to-sqlite

markdown-to-sqlite reads Markdown files with embedded YAML metadata (e.g. for Jekyll Front Matter) and creates a SQLite table with a schema matching the metadata. This is useful if you want to keep structured data in text form in a GitHub repository and use that to build a SQLite database.

socrata2sql

socrata2sql is a tool by Andrew Chavez at the Dallas Morning News. It works with Socrata, a widely used platform for local and national government open data portals. It uses the Socrata API to pull down government datasets and store them in a local SQLite database (it can also export data to PostgreSQL, MySQL and other SQLAlchemy-supported databases).

For example, to create a SQLite database of the City of Dallas Payment Register you would run the following command:

```
$ socrata2sql insert www.dallasopendata.com 64pp-jeba
```

1.3.2 Datasette Plugins

Datasette's *plugin system* makes it easy to enhance Datasette with additional functionality.

datasette-cluster-map

datasette-cluster-map is the original Datasette plugin, described in Datasette plugins, and building a clustered map visualization.

The plugin works against any table with latitude and longitude columns. It can load over 100,000 points onto a map to visualize the geographical distribution of the underlying data.

datasette-vega

datasette-vega exposes the powerful Vega charting library, allowing you to construct line, bar and scatter charts against your data and share links to your visualizations.

datasette-auth-github

datasette-auth-github adds an authentication layer to Datasette. Users will have to sign in using their GitHub account before they can view data or interact with Datasette. You can also use it to restrict access to specific GitHub users, or to members of specified GitHub organizations or teams.

datasette-json-html

datasette-json-html renders HTML in Datasette's table view driven by JSON returned from your SQL queries. This provides a way to embed images, links and lists of links directly in Datasette's main interface, defined using custom SQL statements.

datasette-jellyfish

datasette-jellyfish exposes custom SQL functions for a range of common fuzzy string matching functions, including soundex, porter stemming and levenshtein distance. It builds on top of the Jellyfish Python library.

datasette-doublemetaphone

datasette-doublemetaphone by Matthew Somerville adds custom SQL functions for applying the Double Metaphone fuzzy "sounds like" algorithm.

datasette-jq

datasette-jq adds a custom SQL function for filtering and transforming values from JSON columns using the jq expression language.

datasette-render-images

datasette-render-images works with SQLite tables that contain binary image data in BLOB columns. It converts any images it finds into `data-uri` image elements, allowing you to view them directly in the Datasette interface.

datasette-render-binary

datasette-render-binary renders binary data in a slightly more readable fashion: it shows ASCII characters as they are, and shows all other data as monospace octets. Useful as a tool for exploring new unfamiliar databases as it makes it easier to spot if a binary column may contain a decipherable binary format.

datasette-pretty-json

datasette-pretty-json seeks out JSON values in Datasette's table browsing interface and pretty-prints them, making them easier to read.

datasette-sqlite-fts4

datasette-sqlite-fts4 provides search relevance ranking algorithms that can be used with SQLite's FTS4 search module. You can read more about it in Exploring search relevance algorithms with SQLite.

datasette-bplist

datasette-bplist provides tools for working with Apple's binary plist format embedded in SQLite database tables. If you use OS X you already have dozens of SQLite databases hidden away in your `~/Library` folder that include data in this format - this plugin allows you to view the decoded data and run SQL queries against embedded values using a `bplist_to_json(value)` custom SQL function.

datasette-cors

datasette-cors allows you to configure CORS headers for your Datasette instance. You can use this to enable JavaScript running on a whitelisted set of domains to make `fetch()` calls to the JSON API provided by your Datasette instance.

1.4 Pages and API endpoints

The Datasette web application offers a number of different pages that can be accessed to explore the data in question, each of which is accompanied by an equivalent JSON API.

1.4.1 Top-level index

The root page of any Datasette installation is an index page that lists all of the currently attached databases. Some examples:

- fivethirtyeight.datasettes.com
- global-power-plants.datasettes.com
- register-of-members-interests.datasettes.com

Add `/.json` to the end of the URL for the JSON version of the underlying data:

- fivethirtyeight.datasettes.com/.json
- global-power-plants.datasettes.com/.json
- register-of-members-interests.datasettes.com/.json

1.4.2 Database

Each database has a page listing the tables, views and canned queries available for that database. If the *allow_sql* config option is enabled (it's turned on by default) there will also be an interface for executing arbitrary SQL select queries against the data.

Examples:

- fivethirtyeight.datasettes.com/fivethirtyeight-ac35616
- global-power-plants.datasettes.com/global-power-plants-9e55be2

The JSON version of this page provides programmatic access to the underlying data:

- fivethirtyeight.datasettes.com/fivethirtyeight-ac35616.json
- global-power-plants.datasettes.com/global-power-plants-9e55be2.json

Note that these URLs end in a 7 character hash. This hash is derived from the contents of the database, and ensures that each URL is immutable: the data returned from a URL containing the hash will always be the same, since if the contents of the database file changes by even a single byte a new hash will be generated.

If you access one of these URLs with an incorrect hash (say because a new version of the underlying database has been published) Datasette will 302 redirect you to the correct URL. This happens for all URLs below the database page as well.

Thanks to this hashing scheme, Datasette URLs can all be returned with far-future cache expiry headers. This means browsers will cache the data (including data from the JSON APIs) for a long time, and CDNs such as Cloudflare or Fastly can be used to dramatically improve the performance of a Datasette hosted API.

1.4.3 Table

The table page is the heart of Datasette: it allows users to interactively explore the contents of a database table, including sorting, filtering, *Full-text search* and applying *Facets*.

The HTML interface is worth spending some time exploring. As with other pages, you can return the JSON data by appending `.json` to the URL path, before any *?* querystring arguments.

The querystring arguments are described in more detail here: *Table arguments*

You can also use the table page to interactively construct a SQL query - by applying different filters and a sort order for example - and then click the "View and edit SQL" link to see the SQL query that was used for the page and edit and re-submit it.

Some examples:

- ../items lists all of the line-items registered by UK MPs as potential conflicts of interest. It demonstrates Datasette's support for *Full-text search*.
- ../antiquities-act%2Factions_under_antiquities_act is an interface for exploring the "actions under the antiquities act" data table published by FiveThirtyEight.
- ../global-power-plants?country_long=United+Kingdom&primary_fuel=Gas is a filtered table page showing every Gas power plant in the United Kingdom. It includes some default facets (configured using its metadata.json) and uses the datasette-cluster-map plugin to show a map of the results.

1.4.4 Row

Every row in every Datasette table has its own URL. This means individual records can be linked to directly.

Table cells with extremely long text contents are truncated on the table view according to the *truncate_cells_html* setting. If a cell has been truncated the full length version of that cell will be available on the row page.

Rows which are the targets of foreign key references from other tables will show a link to a filtered search for all records that reference that row. Here's an example from the Registers of Members Interests database:

../people/uk.org.publicwhip%2Fperson%2F10001

Note that this URL includes the encoded primary key of the record.

Here's that same page as JSON:

../people/uk.org.publicwhip%2Fperson%2F10001.json

1.5 Publishing data

Datasette includes tools for publishing and deploying your data to the internet. The `datasette publish` command will deploy a new Datasette instance containing your databases directly to a Heroku, Google Cloud or Zeit Now hosting account. You can also use `datasette package` to create a Docker image that bundles your databases together with the datasette application that is used to serve them.

1.5.1 datasette publish

Once you have created a SQLite database (e.g. using csvs-to-sqlite) you can deploy it to a hosting account using a single command.

You will need a hosting account with Heroku or Google Cloud. Once you have created your account you will need to install and configure the `heroku` or `gcloud` command-line tools.

Publishing to Heroku

To publish your data using Heroku, first create an account there and install and configure the Heroku CLI tool.

You can publish a database to Heroku using the following command:

```
datasette publish heroku mydatabase.db
```

This will output some details about the new deployment, including a URL like this one:

```
https://limitless-reef-88278.herokuapp.com/ deployed to Heroku
```

You can specify a custom app name by passing `-n my-app-name` to the publish command. This will also allow you to overwrite an existing app.

```
$ datasette publish heroku --help

Usage: datasette publish heroku [OPTIONS] [FILES]...

Options:
  -m, --metadata FILENAME         Path to JSON file containing metadata to publish
  --extra-options TEXT            Extra options to pass to datasette serve
  --branch TEXT                   Install datasette from a GitHub branch e.g. master
  --template-dir DIRECTORY        Path to directory containing custom templates
  --plugins-dir DIRECTORY         Path to directory containing custom plugins
  --static STATIC MOUNT           mountpoint:path-to-directory for serving static␣
↪files
  --install TEXT                  Additional packages (e.g. plugins) to install
  --plugin-secret <TEXT TEXT TEXT>...
                                  Secrets to pass to plugins, e.g. --plugin-secret
                                  datasette-auth-github client_id xxx
  --version-note TEXT             Additional note to show on /-/versions
  --title TEXT                    Title for metadata
  --license TEXT                  License label for metadata
  --license_url TEXT              License URL for metadata
  --source TEXT                   Source label for metadata
  --source_url TEXT               Source URL for metadata
  --about TEXT                    About label for metadata
  --about_url TEXT                About URL for metadata
```

(continues on next page)

(continued from previous page)

```
  -n, --name TEXT                 Application name to use when deploying
  --help                          Show this message and exit.
```

Publishing to Google Cloud Run

Google Cloud Run launched as a beta in in April 2019. It allows you to publish data in a scale-to-zero environment, so your application will start running when the first request is received and will shut down again when traffic ceases. This means you only pay for time spent serving traffic.

You will first need to install and configure the Google Cloud CLI tools by following these instructions.

You can then publish a database to Google Cloud Run using the following command:

```
datasette publish cloudrun mydatabase.db
```

You may need to interact with prompts from the tool. Once it has finished it will output a URL like this one:

```
Service [datasette] revision [datasette-00001] has been deployed
and is serving traffic at https://datasette-j7hipcg4aq-uc.a.run.app
```

During the deployment the tool will prompt you for the name of your service. You can reuse an existing name to replace your previous deployment with your new version, or pick a new name to deploy to a new URL.

```
$ datasette publish cloudrun --help

Usage: datasette publish cloudrun [OPTIONS] [FILES]...

Options:
  -m, --metadata FILENAME         Path to JSON file containing metadata to publish
  --extra-options TEXT            Extra options to pass to datasette serve
  --branch TEXT                   Install datasette from a GitHub branch e.g. master
  --template-dir DIRECTORY        Path to directory containing custom templates
  --plugins-dir DIRECTORY         Path to directory containing custom plugins
  --static STATIC MOUNT           mountpoint:path-to-directory for serving static␣
↪files
  --install TEXT                  Additional packages (e.g. plugins) to install
  --plugin-secret <TEXT TEXT TEXT>...
                                  Secrets to pass to plugins, e.g. --plugin-secret
                                  datasette-auth-github client_id xxx
  --version-note TEXT             Additional note to show on /-/versions
  --title TEXT                    Title for metadata
  --license TEXT                  License label for metadata
  --license_url TEXT              License URL for metadata
  --source TEXT                   Source label for metadata
  --source_url TEXT               Source URL for metadata
  --about TEXT                    About label for metadata
  --about_url TEXT                About URL for metadata
  -n, --name TEXT                 Application name to use when building
  --service TEXT                  Cloud Run service to deploy (or over-write)
  --spatialite                    Enable SpatialLite extension
  --show-files                    Output the generated Dockerfile and metadata.json
  --help                          Show this message and exit.
```

Publishing to Zeit Now v1

Datasette can be deployed to Zeit Now's older v1 hosting platform. They no longer accept new signups for this service, so this option is currently only available if you created an account before January 2019.

To publish your database(s) to a new instance hosted by Zeit Now v1, install the now cli tool and then run the following command:

```
datasette publish nowv1 mydatabase.db
```

This will upload your database to Zeit Now, assign you a new URL and install and start a new instance of Datasette to serve your database.

The command will output a URL that looks something like this:

```
https://datasette-elkksjmyfj.now.sh
```

You can navigate to this URL to see live logs of the deployment process. Your new Datasette instance will be available at that URL.

Once the deployment has completed, you can assign a custom URL to your instance using the `now alias` command:

```
now alias https://datasette-elkksjmyfj.now.sh datasette-publish-demo.now.sh
```

You can use `anything-you-like.now.sh`, provided no one else has already registered that alias.

You can also use custom domains, if you first register them with Zeit Now.

```
$ datasette publish nowv1 --help

Usage: datasette publish nowv1 [OPTIONS] [FILES]...

Options:
  -m, --metadata FILENAME         Path to JSON file containing metadata to publish
  --extra-options TEXT            Extra options to pass to datasette serve
  --branch TEXT                   Install datasette from a GitHub branch e.g. master
  --template-dir DIRECTORY        Path to directory containing custom templates
  --plugins-dir DIRECTORY         Path to directory containing custom plugins
  --static STATIC MOUNT           mountpoint:path-to-directory for serving static␣
↪files
  --install TEXT                  Additional packages (e.g. plugins) to install
  --plugin-secret <TEXT TEXT TEXT>...
                                  Secrets to pass to plugins, e.g. --plugin-secret
                                  datasette-auth-github client_id xxx
  --version-note TEXT             Additional note to show on /-/versions
  --title TEXT                    Title for metadata
  --license TEXT                  License label for metadata
  --license_url TEXT              License URL for metadata
  --source TEXT                   Source label for metadata
  --source_url TEXT               Source URL for metadata
  --about TEXT                    About label for metadata
  --about_url TEXT                About URL for metadata
  -n, --name TEXT                 Application name to use when deploying
  --force                         Pass --force option to now
  --token TEXT                    Auth token to use for deploy
  --alias TEXT                    Desired alias e.g. yoursite.now.sh
  --spatialite                    Enable SpatialLite extension
  --show-files                    Output the generated Dockerfile and metadata.json
  --help                          Show this message and exit.
```

Custom metadata and plugins

`datasette publish` accepts a number of additional options which can be used to further customize your Datasette instance.

You can define your own *Metadata* and deploy that with your instance like so:

```
datasette publish nowv1 mydatabase.db -m metadata.json
```

If you just want to set the title, license or source information you can do that directly using extra options to `datasette publish`:

```
datasette publish nowv1 mydatabase.db \
    --title="Title of my database" \
    --source="Where the data originated" \
    --source_url="http://www.example.com/"
```

You can also specify plugins you would like to install. For example, if you want to include the datasette-vega visualization plugin you can use the following:

```
datasette publish nowv1 mydatabase.db --install=datasette-vega
```

If a plugin has any *Secret configuration values* you can use the `--plugin-secret` option to set those secrets at publish time. For example, using Heroku with datasette-auth-github you might run the following command:

```
$ datasette publish heroku my_database.db \
    --name my-heroku-app-demo \
    --install=datasette-auth-github \
    --plugin-secret datasette-auth-github client_id your_client_id \
    --plugin-secret datasette-auth-github client_secret your_client_secret
```

1.5.2 datasette package

If you have docker installed (e.g. using Docker for Mac) you can use the `datasette package` command to create a new Docker image in your local repository containing the datasette app bundled together with your selected SQLite databases:

```
datasette package mydatabase.db
```

Here's example output for the package command:

```
$ datasette package parlgov.db --extra-options="--config sql_time_limit_ms:2500"
Sending build context to Docker daemon  4.459MB
Step 1/7 : FROM python:3
 ---> 79e1dc9af1c1
Step 2/7 : COPY . /app
 ---> Using cache
 ---> cd4ec67de656
Step 3/7 : WORKDIR /app
 ---> Using cache
 ---> 139699e91621
Step 4/7 : RUN pip install datasette
 ---> Using cache
 ---> 340efa82bfd7
Step 5/7 : RUN datasette inspect parlgov.db --inspect-file inspect-data.json
 ---> Using cache
```

(continues on next page)

(continued from previous page)

```
 ---> 5fddbe990314
Step 6/7 : EXPOSE 8001
 ---> Using cache
 ---> 8e83844b0fed
Step 7/7 : CMD datasette serve parlgov.db --port 8001 --inspect-file inspect-data.
↪json --config sql_time_limit_ms:2500
 ---> Using cache
 ---> 1bd380ea8af3
Successfully built 1bd380ea8af3
```

You can now run the resulting container like so:

```
docker run -p 8081:8001 1bd380ea8af3
```

This exposes port 8001 inside the container as port 8081 on your host machine, so you can access the application at `http://localhost:8081/`

A full list of options can be seen by running `datasette package --help`:

```
$ datasette package --help

Usage: datasette package [OPTIONS] FILES...

  Package specified SQLite files into a new datasette Docker container

Options:
  -t, --tag TEXT            Name for the resulting Docker container, can optionally␣
↪use
                            name:tag format
  -m, --metadata FILENAME   Path to JSON file containing metadata to publish
  --extra-options TEXT      Extra options to pass to datasette serve
  --branch TEXT             Install datasette from a GitHub branch e.g. master
  --template-dir DIRECTORY  Path to directory containing custom templates
  --plugins-dir DIRECTORY   Path to directory containing custom plugins
  --static STATIC MOUNT     mountpoint:path-to-directory for serving static files
  --install TEXT            Additional packages (e.g. plugins) to install
  --spatialite              Enable SpatialLite extension
  --version-note TEXT       Additional note to show on /-/versions
  --title TEXT              Title for metadata
  --license TEXT            License label for metadata
  --license_url TEXT        License URL for metadata
  --source TEXT             Source label for metadata
  --source_url TEXT         Source URL for metadata
  --about TEXT              About label for metadata
  --about_url TEXT          About URL for metadata
  --help                    Show this message and exit.
```

1.6 JSON API

Datasette provides a JSON API for your SQLite databases. Anything you can do through the Datasette user interface can also be accessed as JSON via the API.

To access the API for a page, either click on the `.json` link on that page or edit the URL and add a `.json` extension to it.

If you started Datasette with the `--cors` option, each JSON endpoint will be served with the following additional HTTP header:

```
Access-Control-Allow-Origin: *
```

This means JavaScript running on any domain will be able to make cross-origin requests to fetch the data.

If you start Datasette without the `--cors` option only JavaScript running on the same domain as Datasette will be able to access the API.

1.6.1 Different shapes

The default JSON representation of data from a SQLite table or custom query looks like this:

```
{
    "database": "sf-trees",
    "table": "qSpecies",
    "columns": [
        "id",
        "value"
    ],
    "rows": [
        [
            1,
            "Myoporum laetum :: Myoporum"
        ],
        [
            2,
            "Metrosideros excelsa :: New Zealand Xmas Tree"
        ],
        [
            3,
            "Pinus radiata :: Monterey Pine"
        ]
    ],
    "truncated": false,
    "next": "100",
    "next_url": "http://127.0.0.1:8001/sf-trees-02c8ef1/qSpecies.json?_next=100",
    "query_ms": 1.9571781158447266
}
```

The `columns` key lists the columns that are being returned, and the `rows` key then returns a list of lists, each one representing a row. The order of the values in each row corresponds to the columns.

The `_shape` parameter can be used to access alternative formats for the `rows` key which may be more convenient for your application. There are three options:

- `?_shape=arrays` - `"rows"` is the default option, shown above
- `?_shape=objects` - `"rows"` is a list of JSON key/value objects
- `?_shape=array` - an JSON array of objects
- `?_shape=array&_nl=on` - a newline-separated list of JSON objects
- `?_shape=arrayfirst` - a flat JSON array containing just the first value from each row
- `?_shape=object` - a JSON object keyed using the primary keys of the rows

`_shape=objects` looks like this:

```
{
    "database": "sf-trees",
    ...
    "rows": [
        {
            "id": 1,
            "value": "Myoporum laetum :: Myoporum"
        },
        {
            "id": 2,
            "value": "Metrosideros excelsa :: New Zealand Xmas Tree"
        },
        {
            "id": 3,
            "value": "Pinus radiata :: Monterey Pine"
        }
    ]
}
```

`_shape=array` looks like this:

```
[
    {
        "id": 1,
        "value": "Myoporum laetum :: Myoporum"
    },
    {
        "id": 2,
        "value": "Metrosideros excelsa :: New Zealand Xmas Tree"
    },
    {
        "id": 3,
        "value": "Pinus radiata :: Monterey Pine"
    }
]
```

`_shape=array&_nl=on` looks like this:

```
{"id": 1, "value": "Myoporum laetum :: Myoporum"}
{"id": 2, "value": "Metrosideros excelsa :: New Zealand Xmas Tree"}
{"id": 3, "value": "Pinus radiata :: Monterey Pine"}
```

`_shape=arrayfirst` looks like this:

```
[1, 2, 3]
```

`_shape=object` looks like this:

```
{
    "1": {
        "id": 1,
        "value": "Myoporum laetum :: Myoporum"
    },
    "2": {
        "id": 2,
        "value": "Metrosideros excelsa :: New Zealand Xmas Tree"
    },
```

(continues on next page)

(continued from previous page)

```
    "3": {
        "id": 3,
        "value": "Pinus radiata :: Monterey Pine"
    }
]
```

The `object` shape is only available for queries against tables - custom SQL queries and views do not have an obvious primary key so cannot be returned using this format.

The `object` keys are always strings. If your table has a compound primary key, the `object` keys will be a comma-separated string.

1.6.2 Special JSON arguments

Every Datasette endpoint that can return JSON also accepts the following querystring arguments:

`?_shape=SHAPE` The shape of the JSON to return, documented above.

`?_nl=on` When used with `?_shape=array` produces newline-delimited JSON objects.

`?_json=COLUMN1&_json=COLUMN2` If any of your SQLite columns contain JSON values, you can use one or more `_json=` parameters to request that those columns be returned as regular JSON. Without this argument those columns will be returned as JSON objects that have been double-encoded into a JSON string value.

Compare this query without the argument to this query using the argument

`?_json_infinity=on` If your data contains infinity or -infinity values, Datasette will replace them with None when returning them as JSON. If you pass `_json_infinity=1` Datasette will instead return them as `Infinity` or `-Infinity` which is invalid JSON but can be processed by some custom JSON parsers.

`?_timelimit=MS` Sets a custom time limit for the query in ms. You can use this for optimistic queries where you would like Datasette to give up if the query takes too long, for example if you want to implement autocomplete search but only if it can be executed in less than 10ms.

`?_ttl=SECONDS` For how many seconds should this response be cached by HTTP proxies? Use `?_ttl=0` to disable HTTP caching entirely for this request.

1.6.3 Table arguments

The Datasette table view takes a number of special querystring arguments.

Column filter arguments

You can filter the data returned by the table based on column values using a querystring argument.

`?column__exact=value` or `?_column=value` Returns rows where the specified column exactly matches the value.

`?column__not=value` Returns rows where the column does not match the value.

`?column__contains=value` Rows where the string column contains the specified value (`column like "%value%"` in SQL).

`?column__endswith=value` Rows where the string column ends with the specified value (`column like "%value"` in SQL).

?column__startswith=value Rows where the string column starts with the specified value (`column like "value%"` in SQL).

?column__gt=value Rows which are greater than the specified value.

?column__gte=value Rows which are greater than or equal to the specified value.

?column__lt=value Rows which are less than the specified value.

?column__lte=value Rows which are less than or equal to the specified value.

?column__like=value Match rows with a LIKE clause, case insensitive and with `%` as the wildcard character.

?column__glob=value Similar to LIKE but uses Unix wildcard syntax and is case sensitive.

?column__in=value1,value2,value3 Rows where column matches any of the provided values.

You can use a comma separated string, or you can use a JSON array.

The JSON array option is useful if one of your matching values itself contains a comma:

```
?column__in=["value","value,with,commas"]
```

?column__arraycontains=value Works against columns that contain JSON arrays - matches if any of the values in that array match.

This is only available if the `json1` SQLite extension is enabled.

?column__date=value Column is a datestamp occurring on the specified YYYY-MM-DD date, e.g. `2018-01-02`.

?column__isnull=1 Matches rows where the column is null.

?column__notnull=1 Matches rows where the column is not null.

?column__isblank=1 Matches rows where the column is blank, meaning null or the empty string.

?column__notblank=1 Matches rows where the column is not blank.

Special table arguments

?_labels=on/off Expand foreign key references for every possible column. See below.

?_label=COLUMN1&_label=COLUMN2 Expand foreign key references for one or more specified columns.

?_size=1000 or ?_size=max Sets a custom page size. This cannot exceed the `max_returned_rows` limit passed to `datasette serve`. Use `max` to get `max_returned_rows`.

?_sort=COLUMN Sorts the results by the specified column.

?_sort_desc=COLUMN Sorts the results by the specified column in descending order.

?_search=keywords For SQLite tables that have been configured for full-text search executes a search with the provided keywords.

?_search_COLUMN=keywords Like `_search=` but allows you to specify the column to be searched, as opposed to searching all columns that have been indexed by FTS.

?_where=SQL-fragment If the *allow_sql* config option is enabled, this parameter can be used to pass one or more additional SQL fragments to be used in the *WHERE* clause of the SQL used to query the table.

This is particularly useful if you are building a JavaScript application that needs to do something creative but still wants the other conveniences provided by the table view (such as faceting) and hence would like not to have to construct a completely custom SQL query.

Some examples:

- facetable?_where=neighborhood like "%c%"&_where=city_id=3
- facetable?_where=city_id in (select id from facet_cities where name != "Detroit")

?_through={json} This can be used to filter rows via a join against another table.

The JSON parameter must include three keys: `table`, `column` and `value`.

`table` must be a table that the current table is related to via a foreign key relationship.

`column` must be a column in that other table.

`value` is the value that you want to match against.

For example, to filter `roadside_attractions` to just show the attractions that have a characteristic of "museum", you would construct this JSON:

```
{
    "table": "roadside_attraction_characteristics",
    "column": "characteristic_id",
    "value": "1"
}
```

As a URL, that looks like this:

```
?_through={%22table%22:%22roadside_attraction_characteristics%22,
%22column%22:%22characteristic_id%22,%22value%22:%221%22}
```

Here's an example.

?_group_count=COLUMN Executes a SQL query that returns a count of the number of rows matching each unique value in that column, with the most common ordered first.

?_group_count=COLUMN1&_group_count=column2 You can pass multiple `_group_count` columns to return counts against unique combinations of those columns.

?_next=TOKEN Pagination by continuation token - pass the token that was returned in the `"next"` property by the previous page.

?_trace=1 Turns on tracing for this page: SQL queries executed during the request will be gathered and included in the response, either in a new `"_traces"` key for JSON responses or at the bottom of the page if the response is in HTML.

The structure of the data returned here should be considered highly unstable and very likely to change.

1.6.4 Expanding foreign key references

Datasette can detect foreign key relationships and resolve those references into labels. The HTML interface does this by default for every detected foreign key column - you can turn that off using `?_labels=off`.

You can request foreign keys be expanded in JSON using the `_labels=on` or `_label=COLUMN` special querystring parameters. Here's what an expanded row looks like:

```
[
    {
        "rowid": 1,
        "TreeID": 141565,
        "qLegalStatus": {
            "value": 1,
            "label": "Permitted Site"
        },
```

(continues on next page)

(continued from previous page)

```
        "qSpecies": {
            "value": 1,
            "label": "Myoporum laetum :: Myoporum"
        },
        "qAddress": "501X Baker St",
        "SiteOrder": 1
    }
]
```

The column in the foreign key table that is used for the label can be specified in `metadata.json` - see *Specifying the label column for a table*.

1.7 Running SQL queries

Datasette treats SQLite database files as read-only and immutable. This means it is not possible to execute INSERT or UPDATE statements using Datasette, which allows us to expose SELECT statements to the outside world without needing to worry about SQL injection attacks.

The easiest way to execute custom SQL against Datasette is through the web UI. The database index page includes a SQL editor that lets you run any SELECT query you like. You can also construct queries using the filter interface on the tables page, then click "View and edit SQL" to open that query in the custom SQL editor.

Note that this interface is only available if the *allow_sql* option has not been disabled.

Any Datasette SQL query is reflected in the URL of the page, allowing you to bookmark them, share them with others and navigate through previous queries using your browser back button.

You can also retrieve the results of any query as JSON by adding `.json` to the base URL.

1.7.1 Named parameters

Datasette has special support for SQLite named parameters. Consider a SQL query like this:

```
select * from Street_Tree_List
where "PermitNotes" like :notes
and "qSpecies" = :species
```

If you execute this query using the custom query editor, Datasette will extract the two named parameters and use them to construct form fields for you to provide values.

You can also provide values for these fields by constructing a URL:

```
/mydatabase?sql=select...&species=44
```

SQLite string escaping rules will be applied to values passed using named parameters - they will be wrapped in quotes and their content will be correctly escaped.

Datasette disallows custom SQL containing the string PRAGMA, as SQLite pragma statements can be used to change database settings at runtime. If you need to include the string "pragma" in a query you can do so safely using a named parameter.

1.7.2 Views

If you want to bundle some pre-written SQL queries with your Datasette-hosted database you can do so in two ways. The first is to include SQL views in your database - Datasette will then list those views on your database index page.

The easiest way to create views is with the SQLite command-line interface:

```
$ sqlite3 sf-trees.db
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite> CREATE VIEW demo_view AS select qSpecies from Street_Tree_List;
<CTRL+D>
```

1.7.3 Canned queries

As an alternative to adding views to your database, you can define canned queries inside your `metadata.json` file. Here's an example:

```
{
    "databases": {
        "sf-trees": {
            "queries": {
                "just_species": {
                    "sql": "select qSpecies from Street_Tree_List"
                }
            }
        }
    }
}
```

Then run datasette like this:

```
datasette sf-trees.db -m metadata.json
```

Each canned query will be listed on the database index page, and will also get its own URL at:

```
/database-name/canned-query-name
```

For the above example, that URL would be:

```
/sf-trees/just_species
```

You can optionally include `"title"` and `"description"` keys to show a title and description on the canned query page. As with regular table metadata you can alternatively specify `"description_html"` to have your description rendered as HTML (rather than having HTML special characters escaped).

Canned queries support named parameters, so if you include those in the SQL you will then be able to enter them using the form fields on the canned query page or by adding them to the URL. This means canned queries can be used to create custom JSON APIs based on a carefully designed SQL statement.

Here's an example of a canned query with a named parameter:

```
select neighborhood, facet_cities.name, state
from facetable join facet_cities on facetable.city_id = facet_cities.id
where neighborhood like '%' || :text || '%' order by neighborhood;
```

In the canned query JSON it looks like this:

```
{
    "databases": {
        "fixtures": {
            "queries": {
                "neighborhood_search": {
                    "sql": "select neighborhood, facet_cities.name, state\nfrom␣
↪facetable join facet_cities on facetable.city_id = facet_cities.id\nwhere␣
↪neighborhood like '%' || :text || '%' order by neighborhood;",
                    "title": "Search neighborhoods",
                    "description_html": "<b>Demonstrating</b> simple like search"
                }
            }
        }
    }
}
```

You can try this canned query out here: https://latest.datasette.io/fixtures/neighborhood_search?text=town

Note that we are using SQLite string concatenation here - the `||` operator - to add wildcard `%` characters to the string provided by the user.

1.7.4 Pagination

Datasette's default table pagination is designed to be extremely efficient. SQL OFFSET/LIMIT pagination can have a significant performance penalty once you get into multiple thousands of rows, as each page still requires the database to scan through every preceding row to find the correct offset.

When paginating through tables, Datasette instead orders the rows in the table by their primary key and performs a WHERE clause against the last seen primary key for the previous page. For example:

```
select rowid, * from Tree_List where rowid > 200 order by rowid limit 101
```

This represents page three for this particular table, with a page size of 100.

Note that we request 101 items in the limit clause rather than 100. This allows us to detect if we are on the last page of the results: if the query returns less than 101 rows we know we have reached the end of the pagination set. Datasette will only return the first 100 rows - the 101st is used purely to detect if there should be another page.

Since the where clause acts against the index on the primary key, the query is extremely fast even for records that are a long way into the overall pagination set.

1.8 Performance and caching

Datasette runs on top of SQLite, and SQLite has excellent performance. For small databases almost any query should return in just a few milliseconds, and larger databases (100s of MBs or even GBs of data) should perform extremely well provided your queries make sensible use of database indexes.

That said, there are a number of tricks you can use to improve Datasette's performance.

1.8.1 Immutable mode

If you can be certain that a SQLite database file will not be changed by another process you can tell Datasette to open that file in *immutable mode*.

Doing so will disable all locking and change detection, which can result in improved query performance.

This also enables further optimizations relating to HTTP caching, described below.

To open a file in immutable mode pass it to the datasette command using the `-i` option:

```
datasette -i data.db
```

When you open a file in immutable mode like this Datasette will also calculate and cache the row counts for each table in that database when it first starts up, further improving performance.

1.8.2 Using "datasette inspect"

Counting the rows in a table can be a very expensive operation on larger databases. In immutable mode Datasette performs this count only once and caches the results, but this can still cause server startup time to increase by several seconds or more.

If you know that a database is never going to change you can precalculate the table row counts once and store then in a JSON file, then use that file when you later start the server.

To create a JSON file containing the calculated row counts for a database, use the following:

```
datasette inspect data.db --inspect-file=counts.json
```

Then later you can start Datasette against the `counts.json` file and use it to skip the row counting step and speed up server startup:

```
datasette -i data.db --inspect-file=counts.json
```

You need to use the `-i` immutable mode agaist the databse file here or the counts from the JSON file will be ignored.

You will rarely need to use this optimization in every-day use, but several of the `datasette publish` commands described in *Publishing data* use this optimization for better performance when deploying a database file to a hosting provider.

1.8.3 HTTP caching

If your database is immutable and guaranteed not to change, you can gain major performance improvements from Datasette by enabling HTTP caching.

This can work at two different levels. First, it can tell browsers to cache the results of queries and serve future requests from the browser cache.

More significantly, it allows you to run Datasette behind a caching proxy such as Varnish or use a cache provided by a hosted service such as Fastly or Cloudflare. This can provide incredible speed-ups since a query only needs to be executed by Datasette the first time it is accessed - all subsequent hits can then be served by the cache.

Using a caching proxy in this way could enable a Datasette-backed visualization to serve thousands of hits a second while running Datasette itself on extremely inexpensive hosting.

Datasette's integration with HTTP caches can be enabled using a combination of configuration options and querystring arguments.

The *default_cache_ttl* setting sets the default HTTP cache TTL for all Datasette pages. This is 5 seconds unless you change it - you can set it to 0 if you wish to disable HTTP caching entirely.

You can also change the cache timeout on a per-request basis using the `?_ttl=10` querystring parameter. This can be useful when you are working with the Datasette JSON API - you may decide that a specific query can be cached for a longer time, or maybe you need to set `?_ttl=0` for some requests for example if you are running a SQL `order by random()` query.

1.8.4 Hashed URL mode

When you open a database file in immutable mode using the `-i` option, Datasette calculates a SHA-256 hash of the contents of that file on startup. This content hash can then optionally be used to create URLs that are guaranteed to change if the contents of the file changes in the future. This results in URLs that can then be cached indefinitely by both browsers and caching proxies - an enormous potential performance optimization.

You can enable these hashed URLs in two ways: using the *hash_urls* configuration setting (which affects all requests to Datasette) or via the `?_hash=1` querystring parameter (which only applies to the current request).

With hashed URLs enabled, any request to e.g. `/mydatabase/mytable` will 302 redirect to `mydatabase-455fe3a/mytable`. The URL containing the hash will be served with a very long cache expire header - configured using *default_cache_ttl_hashed* which defaults to 365 days.

Since these responses are cached for a long time, you may wish to build API clients against the non-hashed version of these URLs. These 302 redirects are served extremely quickly, so this should still be a performant way to work against the Datasette API.

If you run Datasette behind an HTTP/2 server push aware proxy such as Cloudflare Datasette will serve the 302 redirects in such a way that the redirected page will be efficiently "pushed" to the browser as part of the response, without the browser needing to make a second HTTP request to fetch the redirected resource.

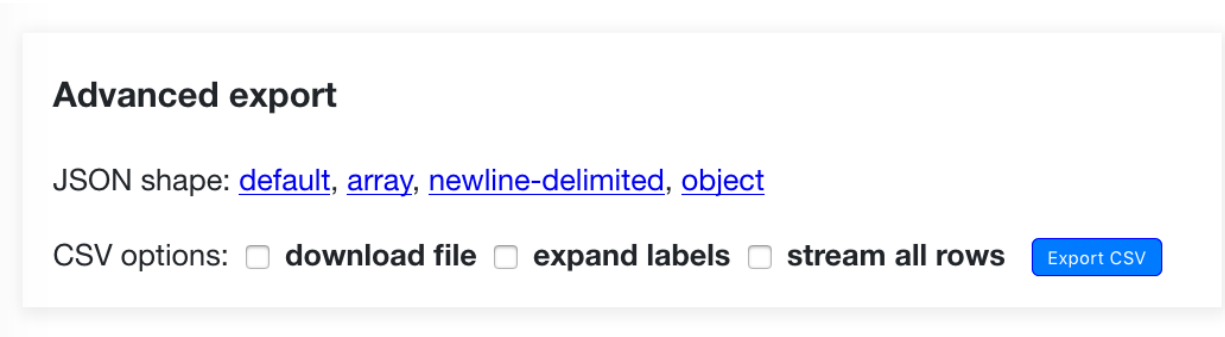
Note: Prior to Datasette 0.28 hashed URL mode was the default behaviour for Datasette, since all database files were assumed to be immutable and unchanging. From 0.28 onwards the default has been to treat database files as mutable unless explicitly configured otherwise.

1.9 CSV Export

Any Datasette table, view or custom SQL query can be exported as CSV.

To obtain the CSV representation of the table you are looking, click the "this data as CSV" link.

You can also use the advanced export form for more control over the resulting file, which looks like this and has the following options:



- **download file** - instead of displaying CSV in your browser, this forces your browser to download the CSV to your downloads directory.
- **expand labels** - if your table has any foreign key references this option will cause the CSV to gain additional `COLUMN_NAME_label` columns with a label for each foreign key derived from the linked table. In this example the `city_id` column is accompanied by a `city_id_label` column.
- **stream all rows** - by default CSV files only contain the first *max_returned_rows* records. This option will cause Datasette to loop through every matching record and return them as a single CSV file.

You can try that out on https://latest.datasette.io/fixtures/facetable?_size=4

1.9.1 Streaming all records

The *stream all rows* option is designed to be as efficient as possible - under the hood it takes advantage of Python 3 asyncio capabilities and Datasette's efficient *pagination* to stream back the full CSV file.

Since databases can get pretty large, by default this option is capped at 100MB - if a table returns more than 100MB of data the last line of the CSV will be a truncation error message.

You can increase or remove this limit using the *max_csv_mb* config setting. You can also disable the CSV export feature entirely using *allow_csv_stream*.

1.9.2 A note on URLs

The default URL for the CSV representation of a table is that table with `.csv` appended to it:

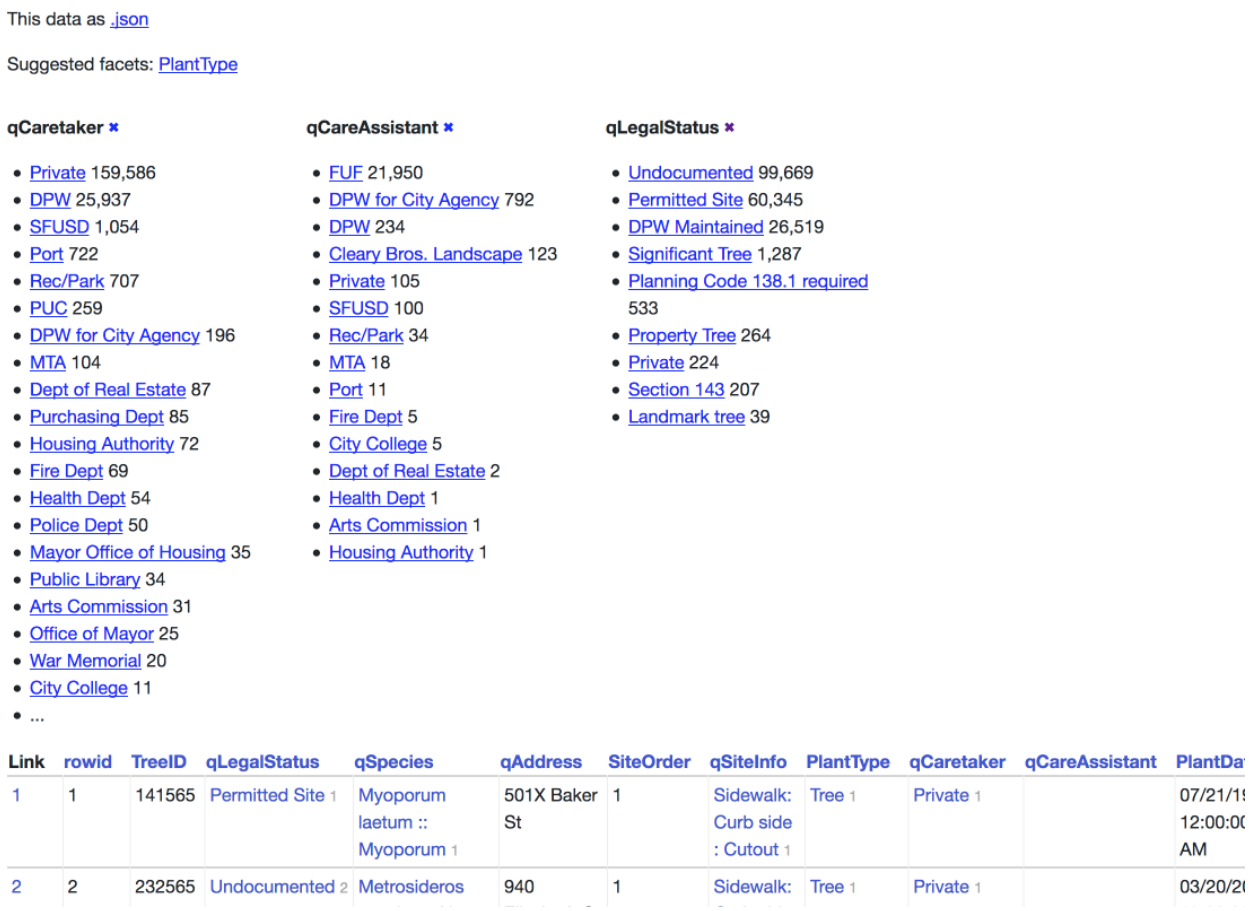
- https://latest.datasette.io/fixtures/facetable - HTML interface
- https://latest.datasette.io/fixtures/facetable.csv - CSV export
- https://latest.datasette.io/fixtures/facetable.json - JSON API

This pattern doesn't work for tables with names that already end in `.csv` or `.json`. For those tables, you can instead use the `_format=` querystring parameter:

- https://latest.datasette.io/fixtures/table%2Fwith%2Fslashes.csv - HTML interface
- https://latest.datasette.io/fixtures/table%2Fwith%2Fslashes.csv?_format=csv - CSV export
- https://latest.datasette.io/fixtures/table%2Fwith%2Fslashes.csv?_format=json - JSON API

1.10 Facets

Datasette facets can be used to add a faceted browse interface to any database table. With facets, tables are displayed along with a summary showing the most common values in specified columns. These values can be selected to further filter the table.



This data as .json

Suggested facets: PlantType

qCaretaker ✖

- Private 159,586
- DPW 25,937
- SFUSD 1,054
- Port 722
- Rec/Park 707
- PUC 259
- DPW for City Agency 196
- MTA 104
- Dept of Real Estate 87
- Purchasing Dept 85
- Housing Authority 72
- Fire Dept 69
- Health Dept 54
- Police Dept 50
- Mayor Office of Housing 35
- Public Library 34
- Arts Commission 31
- Office of Mayor 25
- War Memorial 20
- City College 11
- ...

qCareAssistant ✖

- FUF 21,950
- DPW for City Agency 792
- DPW 234
- Cleary Bros. Landscape 123
- Private 105
- SFUSD 100
- Rec/Park 34
- MTA 18
- Port 11
- Fire Dept 5
- City College 5
- Dept of Real Estate 2
- Health Dept 1
- Arts Commission 1
- Housing Authority 1

qLegalStatus ✖

- Undocumented 99,669
- Permitted Site 60,345
- DPW Maintained 26,519
- Significant Tree 1,287
- Planning Code 138.1 required 533
- Property Tree 264
- Private 224
- Section 143 207
- Landmark tree 39

Link	rowid	TreeID	qLegalStatus	qSpecies	qAddress	SiteOrder	qSiteInfo	PlantType	qCaretaker	qCareAssistant	PlantDa
1	1	141565	Permitted Site 1	Myoporum laetum :: Myoporum 1	501X Baker St	1	Sidewalk: Curb side : Cutout 1	Tree 1	Private 1		07/21/1 12:00:0 AM
2	2	232565	Undocumented 2	Metrosideros	940	1	Sidewalk:	Tree 1	Private 1		03/20/2

Facets can be specified in two ways: using querystring parameters, or in `metadata.json` configuration for the table.

1.10.1 Facets in querystrings

To turn on faceting for specific columns on a Datasette table view, add one or more `_facet=COLUMN` parameters to the URL. For example, if you want to turn on facets for the `city_id` and `state` columns, construct a URL that looks like this:

```
/dbname/tablename?_facet=state&_facet=city_id
```

This works for both the HTML interface and the `.json` view. When enabled, facets will cause a `facet_results` block to be added to the JSON output, looking something like this:

```
{
  "state": {
    "name": "state",
    "results": [
      {
        "value": "CA",
        "label": "CA",
        "count": 10,
        "toggle_url": "http://...?_facet=city_id&_facet=state&state=CA",
        "selected": false
      },
```

(continues on next page)

(continued from previous page)


```
      {
        "value": "MI",
        "label": "MI",
        "count": 4,
        "toggle_url": "http://...?_facet=city_id&_facet=state&state=MI",
        "selected": false
      },
      {
        "value": "MC",
        "label": "MC",
        "count": 1,
        "toggle_url": "http://...?_facet=city_id&_facet=state&state=MC",
        "selected": false
      }
    ],
    "truncated": false
  }
  "city_id": {
    "name": "city_id",
    "results": [
      {
        "value": 1,
        "label": "San Francisco",
        "count": 6,
        "toggle_url": "http://...?_facet=city_id&_facet=state&city_id=1",
        "selected": false
      },
      {
        "value": 2,
        "label": "Los Angeles",
        "count": 4,
        "toggle_url": "http://...?_facet=city_id&_facet=state&city_id=2",
        "selected": false
      },
      {
        "value": 3,
        "label": "Detroit",
        "count": 4,
        "toggle_url": "http://...?_facet=city_id&_facet=state&city_id=3",
        "selected": false
      },
      {
        "value": 4,
        "label": "Memnonia",
        "count": 1,
        "toggle_url": "http://...?_facet=city_id&_facet=state&city_id=4",
        "selected": false
      }
    ],
    "truncated": false
  }
}
```


If Datasette detects that a column is a foreign key, the `"label"` property will be automatically derived from the detected label column on the referenced table.

1.10.2 Facets in metadata.json

You can turn facets on by default for specific tables by adding them to a `"facets"` key in a Datasette *Metadata* file.

Here's an example that turns on faceting by default for the `qLegalStatus` column in the `Street_Tree_List` table in the `sf-trees` database:

```
{
  "databases": {
    "sf-trees": {
      "tables": {
        "Street_Tree_List": {
          "facets": ["qLegalStatus"]
        }
      }
    }
  }
}
```

Facets defined in this way will always be shown in the interface and returned in the API, regardless of the `_facet` arguments passed to the view.

1.10.3 Suggested facets

Datasette's table UI will suggest facets for the user to apply, based on the following criteria:

For the currently filtered data are there any columns which, if applied as a facet...

- Will return 30 or less unique options
- Will return more than one unique option
- Will return less unique options than the total number of filtered rows
- And the query used to evaluate this criteria can be completed in under 50ms

That last point is particularly important: Datasette runs a query for every column that is displayed on a page, which could get expensive - so to avoid slow load times it sets a time limit of just 50ms for each of those queries. This means suggested facets are unlikely to appear for tables with millions of records in them.

1.10.4 Speeding up facets with indexes

The performance of facets can be greatly improved by adding indexes on the columns you wish to facet by. Adding indexes can be performed using the `sqlite3` command-line utility. Here's how to add an index on the `state` column in a table called `Food_Trucks`:

```
$ sqlite3 mydatabase.db
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite> CREATE INDEX Food_Trucks_state ON Food_Trucks("state");
```

1.10.5 Facet by JSON array

If your SQLite installation provides the `json1` extension (you can check using */-/versions*) Datasette will automatically detect columns that contain JSON arrays of values and offer a faceting interface against those columns.

This is useful for modelling things like tags without needing to break them out into a new table.

Example here: latest.datasette.io/fixtures/facetable?_facet_array=tags

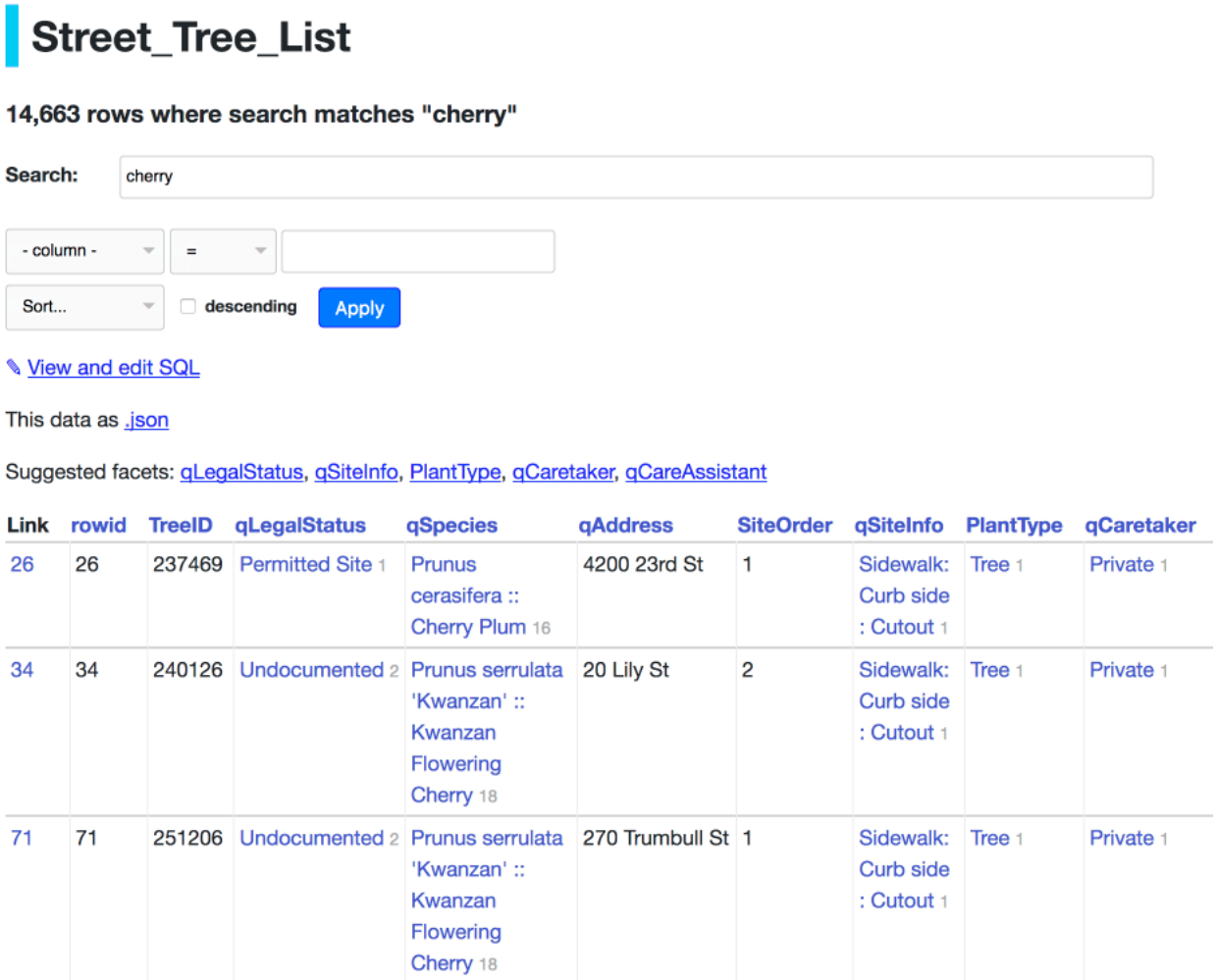
1.10.6 Facet by date

If Datasette finds any columns that contain dates in the first 100 values, it will offer a faceting interface against the dates of those values. This works especially well against timestamp values such as `2019-03-01 12:44:00`.

Example here: latest.datasette.io/fixtures/facetable?_facet_date=created

1.11 Full-text search

SQLite includes a powerful mechanism for enabling full-text search against SQLite records. Datasette can detect if a table has had full-text search configured for it in the underlying database and display a search interface for filtering that table.



Datasette automatically detects which tables have been configured for full-text search.

1.11.1 FTS versions

There are three different versions of the SQLite FTS module: FTS3, FTS4 and FTS5. You can tell which versions are supported by your instance of Datasette by checking the `/-/versions` page.

FTS5 is the most advanced module but may not be available in the SQLite version that is bundled with your Python installation. Most importantly, FTS5 is the only version that has the ability to order by search relevance without needing extra code.

If you can't be sure that FTS5 will be available, you should use FTS4.

1.11.2 Adding full-text search to a SQLite table

Datasette takes advantage of the external content mechanism in SQLite, which allows a full-text search virtual table to be associated with the contents of another SQLite table.

To set up full-text search for a table, you need to do two things:

- Create a new FTS virtual table associated with your table
- Populate that FTS table with the data that you would like to be able to run searches against

Configuring FTS using sqlite-utils

sqlite-utils is a CLI utility and Python library for manipulating SQLite databases. You can use it from Python code to configure FTS search, or you can achieve the same goal using the accompanying command-line tool.

Here's how to use `sqlite-utils` to enable full-text search for an `items` table across the `name` and `description` columns:

```
$ sqlite-utils enable-fts mydatabase.db items name description
```

Configuring FTS using csvs-to-sqlite

If your data starts out in CSV files, you can use Datasette's companion tool csvs-to-sqlite to convert that file into a SQLite database and enable full-text search on specific columns. For a file called `items.csv` where you want full-text search to operate against the `name` and `description` columns you would run the following:

```
$ csvs-to-sqlite items.csv items.db -f name -f description
```

Configuring FTS by hand

We recommend using sqlite-utils, but if you want to hand-roll a SQLite full-text search table you can do so using the following SQL.

To enable full-text search for a table called `items` that works against the `name` and `description` columns, you would run this SQL to create a new `items_fts` FTS virtual table:

```
CREATE VIRTUAL TABLE "items_fts" USING FTS4 (
    name,
    description,
    content="items"
);
```

This creates a set of tables to power full-text search against `items`. The new `items_fts` table will be detected by Datasette as the `fts_table` for the `items` table.

Creating the table is not enough: you also need to populate it with a copy of the data that you wish to make searchable. You can do that using the following SQL:

```
INSERT INTO "items_fts" (rowid, name, description)
    SELECT rowid, name, description FROM items;
```

If your table has columns that are foreign key references to other tables you can include that data in your full-text search index using a join. Imagine the `items` table has a foreign key column called `category_id` which refers to a `categories` table - you could create a full-text search table like this:

```
CREATE VIRTUAL TABLE "items_fts" USING FTS4 (
    name,
    description,
    category_name,
    content="items"
);
```

And then populate it like this:

```
INSERT INTO "items_fts" (rowid, name, description, category_name)
    SELECT items.rowid,
    items.name,
    items.description,
    categories.name
    FROM items JOIN categories ON items.category_id=categories.id;
```

You can use this technique to populate the full-text search index from any combination of tables and joins that makes sense for your project.

1.11.3 Configuring full-text search for a table or view

If a table has a corresponding FTS table set up using the `content=` argument to `CREATE VIRTUAL TABLE` shown above, Datasette will detect it automatically and add a search interface to the table page for that table.

You can also manually configure which table should be used for full-text search using querystring parameters or *Metadata*. You can set the associated FTS table for a specific table and you can also set one for a view - if you do that, the page for that SQL view will offer a search option.

Use `?_fts_table=x` to over-ride the FTS table for a specific page. If the primary key was something other than `rowid` you can use `?_fts_pk=col` to set that as well. This is particularly useful for views, for example:

https://latest.datasette.io/fixtures/searchable_view?_fts_table=searchable_fts&_fts_pk=pk

The `fts_table` metadata property can be used to specify an associated FTS table. If the primary key column in your table which was used to populate the FTS table is something other than `rowid`, you can specify the column to use with the `fts_pk` property.

Here is an example which enables full-text search for a `display_ads` view which is defined against the `ads` table and hence needs to run FTS against the `ads_fts` table, using the `id` as the primary key:

```
{
  "databases": {
    "russian-ads": {
      "tables": {
        "display_ads": {
```

(continues on next page)

(continued from previous page)

```
            "fts_table": "ads_fts",
            "fts_pk": "id"
          }
        }
      }
    }
}
```

1.11.4 The table view API

Table views that support full-text search can be queried using the `?_search=TERMS` querystring parameter. This will run the search against content from all of the columns that have been included in the index.

SQLite full-text search supports wildcards. This means you can easily implement prefix auto-complete by including an asterisk at the end of the search term - for example:

```
/dbname/tablename/?_search=rob*
```

This will return all records containing at least one word that starts with the letters `rob`.

You can also run searches against just the content of a specific named column by using `_search_COLNAME=TERMS` - for example, this would search for just rows where the `name` column in the FTS index mentions `Sarah`:

```
/dbname/tablename/?_search_name=Sarah
```

1.11.5 Searches using custom SQL

You can include full-text search results in custom SQL queries. The general pattern with SQLite search is to run the search as a sub-select that returns rowid values, then include those rowids in another part of the query.

You can see the syntax for a basic search by running that search on a table page and then clicking "View and edit SQL" to see the underlying SQL. For example, consider this search for cherry trees in San Francisco:

```
/sf-trees/Street_Tree_List?_search=cherry
```

If you click View and edit SQL you'll see that the underlying SQL looks like this:

```
select rowid, * from Street_Tree_List
where rowid in (
    select rowid from [Street_Tree_List_fts]
    where [Street_Tree_List_fts] match "cherry"
) order by rowid limit 101
```

1.12 SpatiaLite

The SpatiaLite module for SQLite adds features for handling geographic and spatial data. For an example of what you can do with it, see the tutorial Building a location to time zone API with SpatiaLite, OpenStreetMap and Datasette.

To use it with Datasette, you need to install the `mod_spatialite` dynamic library. This can then be loaded into Datasette using the `--load-extension` command-line option.

1.12.1 Installation

Installing SpatiaLite on OS X

The easiest way to install SpatiaLite on OS X is to use Homebrew.

```
brew update
brew install spatialite-tools
```

This will install the `spatialite` command-line tool and the `mod_spatialite` dynamic library.

You can now run Datasette like so:

```
datasette --load-extension=/usr/local/lib/mod_spatialite.dylib
```

Installing SpatiaLite on Linux

SpatiaLite is packaged for most Linux distributions.

```
apt install spatialite-bin libsqlite3-mod-spatialite
```

Depending on your distribution, you should be able to run Datasette something like this:

```
datasette --load-extension=/usr/lib/x86_64-linux-gnu/mod_spatialite.so
```

If you are unsure of the location of the module, try running `locate mod_spatialite` and see what comes back.

Building SpatiaLite from source

The packaged versions of SpatiaLite usually provide SpatiaLite 4.3.0a. For an example of how to build the most recent unstable version, 4.4.0-RC0 (which includes the powerful VirtualKNN module), take a look at the Datasette Dockerfile.

1.12.2 Spatial indexing latitude/longitude columns

Here's a recipe for taking a table with existing latitude and longitude columns, adding a SpatiaLite POINT geometry column to that table, populating the new column and then populating a spatial index:

```
import sqlite3
conn = sqlite3.connect('museums.db')
# Lead the spatialite extension:
conn.enable_load_extension(True)
conn.load_extension('/usr/local/lib/mod_spatialite.dylib')
# Initialize spatial metadata for this database:
conn.execute('select InitSpatialMetadata(1)')
# Add a geometry column called point_geom to our museums table:
conn.execute("SELECT AddGeometryColumn('museums', 'point_geom', 4326, 'POINT', 2);")
# Now update that geometry column with the lat/lon points
conn.execute('''
    UPDATE events SET
    point_geom = GeomFromText('POINT('||"longitude"||' '||"latitude"||')',4326);
''')
# Now add a spatial index to that column
```

(continues on next page)

(continued from previous page)

```
conn.execute('select CreateSpatialIndex("museums", "point_geom");')
# If you don't commit your changes will not be persisted:
conn.commit()
conn.close()
```

1.12.3 Making use of a spatial index

SpatiaLite spatial indexes are R*Trees. They allow you to run efficient bounding box queries using a sub-select, with a similar pattern to that used for *Searches using custom SQL*.

In the above example, the resulting index will be called `idx_museums_point_geom`. This takes the form of a SQLite virtual table. You can inspect its contents using the following query:

```
select * from idx_museums_point_geom limit 10;
```

Here's a live example: timezones-api.now.sh/timezones/idx_timezones_Geometry

pkid	xmin	xmax	ymin	ymax
1	-8.601725578308105	-2.4930307865142822	4.162120819091797	10.74019718170166
2	-3.2607860565185547	1.27329421043396	4.539252281188965	11.174856185913086
3	32.997581481933594	47.98238754272461	3.3974475860595703	14.894054412841797
4	-8.66890811920166	11.997337341308594	18.9681453704834	37.296207427978516
5	36.43336486816406	43.300174713134766	12.354820251464844	18.070993423461914

You can now construct efficient bounding box queries that will make use of the index like this:

```
select * from museums where museums.rowid in (
    SELECT pkid FROM idx_museums_point_geom
    -- left-hand-edge of point > left-hand-edge of bbox (minx)
    where xmin > :bbox_minx
    -- right-hand-edge of point < right-hand-edge of bbox (maxx)
    and xmax < :bbox_maxx
    -- bottom-edge of point > bottom-edge of bbox (miny)
    and ymin > :bbox_miny
    -- top-edge of point < top-edge of bbox (maxy)
    and ymax < :bbox_maxy
);
```

Spatial indexes can be created against polygon columns as well as point columns, in which case they will represent the minimum bounding rectangle of that polygon. This is useful for accelerating `within` queries, as seen in the Timezones API example.

1.12.4 Importing shapefiles into SpatiaLite

The shapefile format is a common format for distributing geospatial data. You can use the `spatialite` command-line tool to create a new database table from a shapefile.

Try it now with the North America shapefile available from the University of North Carolina Global River Database project. Download the file and unzip it (this will create files called `narivs.dbf`, `narivs.prj`, `narivs.shp` and `narivs.shx` in the current directory), then run the following:

```
$ spatialite rivers-database.db
SpatiaLite version ..: 4.3.0a       Supported Extensions:
...
spatialite> .loadshp narivs rivers CP1252 23032
========
Loading shapefile at 'narivs' into SQLite table 'rivers'
...
Inserted 467973 rows into 'rivers' from SHAPEFILE
```

This will load the data from the `narivs` shapefile into a new database table called `rivers`.

Exit out of `spatialite` (using `Ctrl+D`) and run Datasette against your new database like this:

```
datasette rivers-database.db \
    --load-extension=/usr/local/lib/mod_spatialite.dylib
```

If you browse to `http://localhost:8001/rivers-database/rivers` you will see the new table... but the `Geometry` column will contain unreadable binary data (SpatiaLite uses a custom format based on WKB).

The easiest way to turn this into semi-readable data is to use the SpatiaLite `AsGeoJSON` function. Try the following using the SQL query interface at `http://localhost:8001/rivers-database`:

```
select *, AsGeoJSON(Geometry) from rivers limit 10;
```

This will give you back an additional column of GeoJSON. You can copy and paste GeoJSON from this column into the debugging tool at geojson.io to visualize it on a map.

To see a more interesting example, try ordering the records with the longest geometry first. Since there are 467,000 rows in the table you will first need to increase the SQL time limit imposed by Datasette:

```
datasette rivers-database.db \
    --load-extension=/usr/local/lib/mod_spatialite.dylib \
    --config sql_time_limit_ms:10000
```

Now try the following query:

```
select *, AsGeoJSON(Geometry) from rivers
order by length(Geometry) desc limit 10;
```

1.12.5 Importing GeoJSON polygons using Shapely

Another common form of polygon data is the GeoJSON format. This can be imported into SpatiaLite directly, or by using the Shapely Python library.

Who's On First is an excellent source of openly licensed GeoJSON polygons. Let's import the geographical polygon for Wales. First, we can use the Who's On First Spelunker tool to find the record for Wales:

spelunker.whosonfirst.org/id/404227475

That page includes a link to the GeoJSON record, which can be accessed here:

data.whosonfirst.org/404/227/475/404227475.geojson

Here's Python code to create a SQLite database, enable SpatiaLite, create a places table and then add a record for Wales:

```
import sqlite3
conn = sqlite3.connect('places.db')
# Enable SpatialLite extension
conn.enable_load_extension(True)
conn.load_extension('/usr/local/lib/mod_spatialite.dylib')
# Create the masic countries table
conn.execute('select InitSpatialMetadata(1)')
conn.execute('create table places (id integer primary key, name text);')
# Add a MULTIPOLYGON Geometry column
conn.execute("SELECT AddGeometryColumn('places', 'geom', 4326, 'MULTIPOLYGON', 2);")
# Add a spatial index against the new column
conn.execute("SELECT CreateSpatialIndex('places', 'geom');")
# Now populate the table
from shapely.geometry.multipolygon import MultiPolygon
from shapely.geometry import shape
import requests
geojson = requests.get('https://data.whosonfirst.org/404/227/475/404227475.geojson').
↪json()
# Convert to "Well Known Text" format
wkt = shape(geojson['geometry']).wkt
# Insert and commit the record
conn.execute("INSERT INTO places (id, name, geom) VALUES(null, ?, GeomFromText(?,␣
↪4326))", (
   "Wales", wkt
))
conn.commit()
```

1.12.6 Querying polygons using within()

The `within()` SQL function can be used to check if a point is within a geometry:

```
select
    name
from
    places
where
   within(GeomFromText('POINT(-3.1724366 51.4704448)'), places.geom);
```

The `GeomFromText()` function takes a string of well-known text. Note that the order used here is `longitude` then `latitude`.

To run that same `within()` query in a way that benefits from the spatial index, use the following:

```
select
    name
from
    places
where
    within(GeomFromText('POINT(-3.1724366 51.4704448)'), places.geom)
    and rowid in (
        SELECT pkid FROM idx_places_geom
        where xmin < -3.1724366
        and xmax > -3.1724366
        and ymin < 51.4704448
        and ymax > 51.4704448
    );
```

1.13 Metadata

Data loves metadata. Any time you run Datasette you can optionally include a JSON file with metadata about your databases and tables. Datasette will then display that information in the web UI.

Run Datasette like this:

```
datasette database1.db database2.db --metadata metadata.json
```

Your `metadata.json` file can look something like this:

```
{
    "title": "Custom title for your index page",
    "description": "Some description text can go here",
    "license": "ODbL",
    "license_url": "https://opendatacommons.org/licenses/odbl/",
    "source": "Original Data Source",
    "source_url": "http://example.com/"
}
```

The above metadata will be displayed on the index page of your Datasette-powered site. The source and license information will also be included in the footer of every page served by Datasette.

Any special HTML characters in `description` will be escaped. If you want to include HTML in your description, you can use a `description_html` property instead.

1.13.1 Per-database and per-table metadata

Metadata at the top level of the JSON will be shown on the index page and in the footer on every page of the site. The license and source is expected to apply to all of your data.

You can also provide metadata at the per-database or per-table level, like this:

```
{
    "databases": {
        "database1": {
            "source": "Alternative source",
            "source_url": "http://example.com/",
            "tables": {
                "example_table": {
                    "description_html": "Custom <em>table</em> description",
                    "license": "CC BY 3.0 US",
                    "license_url": "https://creativecommons.org/licenses/by/3.0/us/"
                }
            }
        }
    }
}
```

Each of the top-level metadata fields can be used at the database and table level.

1.13.2 Source, license and about

The three visible metadata fields you can apply to everything, specific databases or specific tables are source, license and about. All three are optionaly.

source and **source_url** should be used to indicate where the underlying data came from.

license and **license_url** should be used to indicate the license under which the data can be used.

about and **about_url** can be used to link to further information about the project - an accompanying blog entry for example.

For each of these you can provide just the `*_url` field and Datasette will treat that as the default link label text and display the URL directly on the page.

1.13.3 Specifying units for a column

Datasette supports attaching units to a column, which will be used when displaying values from that column. SI prefixes will be used where appropriate.

Column units are configured in the metadata like so:

```
{
    "databases": {
        "database1": {
            "tables": {
                "example_table": {
                    "units": {
                        "column1": "metres",
                        "column2": "Hz"
                    }
                }
            }
        }
    }
}
```

Units are interpreted using Pint, and you can see the full list of available units in Pint's unit registry. You can also add custom units to the metadata, which will be registered with Pint:

```
{
    "custom_units": [
        "decibel = [] = dB"
    ]
}
```

1.13.4 Setting which columns can be used for sorting

Datasette allows any column to be used for sorting by default. If you need to control which columns are available for sorting you can do so using the optional `sortable_columns` key:

```
{
    "databases": {
        "database1": {
            "tables": {
                "example_table": {
                    "sortable_columns": [
                        "height",
                        "weight"
                    ]
                }
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }
}
```

This will restrict sorting of `example_table` to just the `height` and `weight` columns.

You can also disable sorting entirely by setting `"sortable_columns": []`

By default, database views in Datasette do not support sorting. You can use `sortable_columns` to enable specific sort orders for a view called `name_of_view` in the database `my_database` like so:

```
{
    "databases": {
        "my_database": {
            "tables": {
                "name_of_view": {
                    "sortable_columns": [
                        "clicks",
                        "impressions"
                    ]
                }
            }
        }
    }
}
```

1.13.5 Specifying the label column for a table

Datasette's HTML interface attempts to display foreign key references as labelled hyperlinks. By default, it looks for referenced tables that only have two columns: a primary key column and one other. It assumes that the second column should be used as the link label.

If your table has more than two columns you can specify which column should be used for the link label with the `label_column` property:

```
{
    "databases": {
        "database1": {
            "tables": {
                "example_table": {
                    "label_column": "title"
                }
            }
        }
    }
}
```

1.13.6 Hiding tables

You can hide tables from the database listing view (in the same way that FTS and Spatialite tables are automatically hidden) using `"hidden": true`:

```
{
    "databases": {
        "database1": {
            "tables": {
                "example_table": {
                    "hidden": true
                }
            }
        }
    }
}
```

1.14 Configuration

Datasette provides a number of configuration options. These can be set using the `--config name:value` option to `datasette serve`.

You can set multiple configuration options at once like this:

```
datasette mydatabase.db --config default_page_size:50 \
    --config sql_time_limit_ms:3500 \
    --config max_returned_rows:2000
```

To prevent rogue, long-running queries from making a Datasette instance inaccessible to other users, Datasette imposes some limits on the SQL that you can execute. These are exposed as config options which you can over-ride.

1.14.1 default_page_size

The default number of rows returned by the table page. You can over-ride this on a per-page basis using the `?_size=80` querystring parameter, provided you do not specify a value higher than the `max_returned_rows` setting. You can set this default using `--config` like so:

```
datasette mydatabase.db --config default_page_size:50
```

1.14.2 sql_time_limit_ms

By default, queries have a time limit of one second. If a query takes longer than this to run Datasette will terminate the query and return an error.

If this time limit is too short for you, you can customize it using the `sql_time_limit_ms` limit - for example, to increase it to 3.5 seconds:

```
datasette mydatabase.db --config sql_time_limit_ms:3500
```

You can optionally set a lower time limit for an individual query using the `?_timelimit=100` querystring argument:

```
/my-database/my-table?qSpecies=44&_timelimit=100
```

This would set the time limit to 100ms for that specific query. This feature is useful if you are working with databases of unknown size and complexity - a query that might make perfect sense for a smaller table could take too long to execute on a table with millions of rows. By setting custom time limits you can execute queries "optimistically" - e.g. give me an exact count of rows matching this query but only if it takes less than 100ms to calculate.

1.14.3 max_returned_rows

Datasette returns a maximum of 1,000 rows of data at a time. If you execute a query that returns more than 1,000 rows, Datasette will return the first 1,000 and include a warning that the result set has been truncated. You can use OFFSET/LIMIT or other methods in your SQL to implement pagination if you need to return more than 1,000 rows.

You can increase or decrease this limit like so:

```
datasette mydatabase.db --config max_returned_rows:2000
```

1.14.4 num_sql_threads

Maximum number of threads in the thread pool Datasette uses to execute SQLite queries. Defaults to 3.

```
datasette mydatabase.db --config num_sql_threads:10
```

1.14.5 allow_facet

Allow users to specify columns they would like to facet on using the `?_facet=COLNAME` URL parameter to the table view.

This is enabled by default. If disabled, facets will still be displayed if they have been specifically enabled in `metadata.json` configuration for the table.

Here's how to disable this feature:

```
datasette mydatabase.db --config allow_facet:off
```

1.14.6 default_facet_size

The default number of unique rows returned by *Facets* is 30. You can customize it like this:

```
datasette mydatabase.db --config default_facet_size:50
```

1.14.7 facet_time_limit_ms

This is the time limit Datasette allows for calculating a facet, which defaults to 200ms:

```
datasette mydatabase.db --config facet_time_limit_ms:1000
```

1.14.8 facet_suggest_time_limit_ms

When Datasette calculates suggested facets it needs to run a SQL query for every column in your table. The default for this time limit is 50ms to account for the fact that it needs to run once for every column. If the time limit is exceeded the column will not be suggested as a facet.

You can increase this time limit like so:

```
datasette mydatabase.db --config facet_suggest_time_limit_ms:500
```

1.14.9 suggest_facets

Should Datasette calculate suggested facets? On by default, turn this off like so:

```
datasette mydatabase.db --config suggest_facets:off
```

1.14.10 allow_download

Should users be able to download the original SQLite database using a link on the database index page? This is turned on by default - to disable database downloads, use the following:

```
datasette mydatabase.db --config allow_download:off
```

1.14.11 allow_sql

Enable/disable the ability for users to run custom SQL directly against a database. To disable this feature, run:

```
datasette mydatabase.db --config allow_sql:off
```

1.14.12 default_cache_ttl

Default HTTP caching max-age header in seconds, used for `Cache-Control: max-age=X`. Can be over-ridden on a per-request basis using the `?_ttl=` querystring parameter. Set this to `0` to disable HTTP caching entirely. Defaults to 5 seconds.

```
datasette mydatabase.db --config default_cache_ttl:60
```

1.14.13 default_cache_ttl_hashed

Default HTTP caching max-age for responses served using using the *hashed-urls mechanism*. Defaults to 365 days (31536000 seconds).

```
datasette mydatabase.db --config default_cache_ttl_hashed:10000
```

1.14.14 cache_size_kb

Sets the amount of memory SQLite uses for its per-connection cache, in KB.

```
datasette mydatabase.db --config cache_size_kb:5000
```

1.14.15 allow_csv_stream

Enables *the CSV export feature* where an entire table (potentially hundreds of thousands of rows) can be exported as a single CSV file. This is turned on by default - you can turn it off like this:

```
datasette mydatabase.db --config allow_csv_stream:off
```

1.14.16 max_csv_mb

The maximum size of CSV that can be exported, in megabytes. Defaults to 100MB. You can disable the limit entirely by settings this to 0:

```
datasette mydatabase.db --config max_csv_mb:0
```

1.14.17 truncate_cells_html

In the HTML table view, truncate any strings that are longer than this value. The full value will still be available in CSV, JSON and on the individual row HTML page. Set this to 0 to disable truncation.

```
datasette mydatabase.db --config truncate_cells_html:0
```

1.14.18 force_https_urls

Forces self-referential URLs in the JSON output to always use the `https://` protocol. This is useful for cases where the application itself is hosted using HTTP but is served to the outside world via a proxy that enables HTTPS.

```
datasette mydatabase.db --config force_https_urls:1
```

1.14.19 hash_urls

When enabled, this setting causes Datasette to append a content hash of the database file to the URL path for every table and query within that database.

When combined with far-future expire headers this ensures that queries can be cached forever, safe in the knowledge that any modifications to the database itself will result in new, uncachcacheed URL paths.

```
datasette mydatabase.db --config hash_urls:1
```

1.15 Introspection

Datasette includes some pages and JSON API endpoints for introspecting the current instance. These can be used to understand some of the internals of Datasette and to see how a particular instance has been configured.

Each of these pages can be viewed in your browser. Add `.json` to the URL to get back the contents as JSON.

1.15.1 /-/metadata

Shows the contents of the `metadata.json` file that was passed to `datasette serve`, if any. Metadata example:

```
{
    "license": "CC Attribution 4.0 License",
    "license_url": "http://creativecommons.org/licenses/by/4.0/",
    "source": "fivethirtyeight/data on GitHub",
    "source_url": "https://github.com/fivethirtyeight/data",
    "title": "Five Thirty Eight",
    "databases": {...}
}
```

1.15.2 /-/versions

Shows the version of Datasette, Python and SQLite. Versions example:


```
{
    "datasette": {
        "version": "0.21"
    },
    "python": {
        "full": "3.6.5 (default, May  5 2018, 03:07:21) \n[GCC 6.3.0 20170516]",
        "version": "3.6.5"
    },
    "sqlite": {
        "extensions": {
            "json1": null
        },
        "fts_versions": [
            "FTS5",
            "FTS4",
            "FTS3"
        ],
        "compile_options": [
            "COMPILER=gcc-6.3.0 20170516",
            "ENABLE_FTS3",
            "ENABLE_FTS4",
            "ENABLE_FTS5",
            "ENABLE_JSON1",
            "ENABLE_RTREE",
            "THREADSAFE=1"
        ],
        "version": "3.16.2"
    }
}
```


1.15.3 /-/plugins

Shows a list of currently installed plugins and their versions. Plugins example:


```
[
    {
        "name": "datasette_cluster_map",
        "static": true,
        "templates": false,
        "version": "0.4"
    }
]
```


1.15.4 /-/config

Shows the *Configuration* options for this instance of Datasette. Config example:


```
{
    "default_facet_size": 30,
    "default_page_size": 100,
```


(continues on next page)

(continued from previous page)

```
    "facet_suggest_time_limit_ms": 50,
    "facet_time_limit_ms": 1000,
    "max_returned_rows": 1000,
    "sql_time_limit_ms": 1000
}
```

1.15.5 /-/databases

Shows currently attached databases. Databases example:

```
[
    {
        "hash": null,
        "is_memory": false,
        "is_mutable": true,
        "name": "fixtures",
        "path": "fixtures.db",
        "size": 225280
    }
]
```

1.16 Customization

Datasette provides a number of ways of customizing the way data is displayed.

1.16.1 Custom CSS and JavaScript

When you launch Datasette, you can specify a custom metadata file like this:

```
datasette mydb.db --metadata metadata.json
```

Your `metadata.json` file can include links that look like this:

```
{
    "extra_css_urls": [
        "https://simonwillison.net/static/css/all.bf8cd891642c.css"
    ],
    "extra_js_urls": [
        "https://code.jquery.com/jquery-3.2.1.slim.min.js"
    ]
}
```

The extra CSS and JavaScript files will be linked in the `<head>` of every page.

You can also specify a SRI (subresource integrity hash) for these assets:

```
{
    "extra_css_urls": [
        {
            "url": "https://simonwillison.net/static/css/all.bf8cd891642c.css",
            "sri": "sha384-9qIZekWUyjCyDIf2YK1FRoKiPJq4PHt6tp/
↪ulnuuyRBvazd0hG7pWbE99zvwSznI"
```

(continues on next page)

(continued from previous page)

```
        }
    ],
    "extra_js_urls": [
        {
            "url": "https://code.jquery.com/jquery-3.2.1.slim.min.js",
            "sri": "sha256-k2WSCIexGzOj3Euiig+TlR8gA0EmPjuc79OEeY5L45g="
        }
    ]
}
```

Modern browsers will only execute the stylesheet or JavaScript if the SRI hash matches the content served. You can generate hashes using www.srihash.org

Every default template includes CSS classes in the body designed to support custom styling.

The index template (the top level page at `/`) gets this:

```
<body class="index">
```

The database template (`/dbname`) gets this:

```
<body class="db db-dbname">
```

The custom SQL template (`/dbname?sql=...`) gets this:

```
<body class="query db-dbname">
```

The table template (`/dbname/tablename`) gets:

```
<body class="table db-dbname table-tablename">
```

The row template (`/dbname/tablename/rowid`) gets:

```
<body class="row db-dbname table-tablename">
```

The `db-x` and `table-x` classes use the database or table names themselves if they are valid CSS identifiers. If they aren't, we strip any invalid characters out and append a 6 character md5 digest of the original name, in order to ensure that multiple tables which resolve to the same stripped character version still have different CSS classes.

Some examples:

```
"simple" => "simple"
"MixedCase" => "MixedCase"
"-no-leading-hyphens" => "no-leading-hyphens-65bea6"
"_no-leading-underscores" => "no-leading-underscores-b921bc"
"no spaces" => "no-spaces-7088d7"
"-" => "336d5e"
"no $ characters" => "no--characters-59e024"
```

`<td>` and `<th>` elements also get custom CSS classes reflecting the database column they are representing, for example:

```
<table>
    <thead>
        <tr>
            <th class="col-id" scope="col">id</th>
            <th class="col-name" scope="col">name</th>
```

(continues on next page)

(continued from previous page)

```
        </tr>
    </thead>
    <tbody>
        <tr>
            <td class="col-id"><a href="...">1</a></td>
            <td class="col-name">SMITH</td>
        </tr>
    </tbody>
</table>
```

1.16.2 Custom templates

By default, Datasette uses default templates that ship with the package.

You can over-ride these templates by specifying a custom `--template-dir` like this:

```
datasette mydb.db --template-dir=mytemplates/
```

Datasette will now first look for templates in that directory, and fall back on the defaults if no matches are found.

It is also possible to over-ride templates on a per-database, per-row or per- table basis.

The lookup rules Datasette uses are as follows:

```
Index page (/):
    index.html

Database page (/mydatabase):
    database-mydatabase.html
    database.html

Custom query page (/mydatabase?sql=...):
    query-mydatabase.html
    query.html

Canned query page (/mydatabase/canned-query):
    query-mydatabase-canned-query.html
    query-mydatabase.html
    query.html

Table page (/mydatabase/mytable):
    table-mydatabase-mytable.html
    table.html

Row page (/mydatabase/mytable/id):
    row-mydatabase-mytable.html
    row.html

Table of rows and columns include on table page:
    _table-table-mydatabase-mytable.html
    _table-mydatabase-mytable.html
    _table.html

Table of rows and columns include on row page:
    _table-row-mydatabase-mytable.html
```

(continues on next page)

(continued from previous page)

```
    _table-mydatabase-mytable.html
    _table.html
```

If a table name has spaces or other unexpected characters in it, the template filename will follow the same rules as our custom `<body>` CSS classes - for example, a table called "Food Trucks" will attempt to load the following templates:

```
table-mydatabase-Food-Trucks-399138.html
table.html
```

You can find out which templates were considered for a specific page by viewing source on that page and looking for an HTML comment at the bottom. The comment will look something like this:

```
<!-- Templates considered: *query-mydb-tz.html, query-mydb.html, query.html -->
```

This example is from the canned query page for a query called "tz" in the database called "mydb". The asterisk shows which template was selected - so in this case, Datasette found a template file called `query-mydb-tz.html` and used that - but if that template had not been found, it would have tried for `query-mydb.html` or the default `query.html`.

It is possible to extend the default templates using Jinja template inheritance. If you want to customize EVERY row template with some additional content you can do so by creating a `row.html` template like this:

```
{% extends "default:row.html" %}

{% block content %}
<h1>EXTRA HTML AT THE TOP OF THE CONTENT BLOCK</h1>
<p>This line renders the original block:</p>
{{ super() }}
{% endblock %}
```

Note the `default:row.html` template name, which ensures Jinja will inherit from the default template.

The `_table.html` template is included by both the row and the table pages, and a list of rows. The default `_table.html` template renders them as an HTML template and can be seen here.

You can provide a custom template that applies to all of your databases and tables, or you can provide custom templates for specific tables using the template naming scheme described above.

If you want to present your data in a format other than an HTML table, you can do so by looping through `display_rows` in your own `_table.html` template. You can use `{{ row["column_name"] }}` to output the raw value of a specific column.

If you want to output the rendered HTML version of a column, including any links to foreign keys, you can use `{{ row.display("column_name") }}`.

Here is an example of a custom `_table.html` template:

```
{% for row in display_rows %}
    <div>
        <h2>{{ row["title"] }}</h2>
        <p>{{ row["description"] }}<lp>
        <p>Category: {{ row.display("category_id") }}</p>
    </div>
{% endfor %}
```

1.17 Plugins

Datasette's plugin system allows additional features to be implemented as Python code (or front-end JavaScript) which can be wrapped up in a separate Python package. The underlying mechanism uses pluggy.

1.17.1 Using plugins

If a plugin has been packaged for distribution using setuptools you can use the plugin by installing it alongside Datasette in the same virtual environment or Docker container.

You can also define one-off per-project plugins by saving them as `plugin_name.py` functions in a `plugins/` folder and then passing that folder to `datasette serve`.

The `datasette publish` and `datasette package` commands both take an optional `--install` argument. You can use this one or more times to tell Datasette to `pip install` specific plugins as part of the process. You can use the name of a package on PyPI or any of the other valid arguments to `pip install` such as a URL to a `.zip` file:

```
datasette publish cloudrun mydb.db \
    --install=datasette-plugin-demos \
    --install=https://url-to-my-package.zip
```

1.17.2 Writing plugins

The easiest way to write a plugin is to create a `my_plugin.py` file and drop it into your `plugins/` directory. Here is an example plugin, which adds a new custom SQL function called `hello_world()` which takes no arguments and returns the string `Hello world!`.

```
from datasette import hookimpl

@hookimpl
def prepare_connection(conn):
    conn.create_function('hello_world', 0, lambda: 'Hello world!')
```

If you save this in `plugins/my_plugin.py` you can then start Datasette like this:

```
datasette serve mydb.db --plugins-dir=plugins/
```

Now you can navigate to http://localhost:8001/mydb and run this SQL:

```
select hello_world();
```

To see the output of your plugin.

1.17.3 Seeing what plugins are installed

You can see a list of installed plugins by navigating to the `/-/plugins` page of your Datasette instance - for example: https://fivethirtyeight.datasettes.com/-/plugins

You can also use the `datasette plugins` command:

```
$ datasette plugins
[
    {
        "name": "datasette_json_html",
        "static": false,
        "templates": false,
        "version": "0.4.0"
    }
]
```

If you run `datasette plugins --all` it will include default plugins that ship as part of Datasette:

```
$ datasette plugins --all
[
    {
        "name": "datasette_json_html",
        "static": false,
        "templates": false,
        "version": "0.4.0"
    },
    {
        "name": "datasette.publish.heroku",
        "static": false,
        "templates": false,
        "version": null
    },
    {
        "name": "datasette.publish.now",
        "static": false,
        "templates": false,
        "version": null
    }
]
```

You can add the `--plugins-dir=` option to include any plugins found in that directory.

1.17.4 Packaging a plugin

Plugins can be packaged using Python setuptools. You can see an example of a packaged plugin at https://github.com/simonw/datasette-plugin-demos

The example consists of two files: a `setup.py` file that defines the plugin:

```
from setuptools import setup

VERSION = '0.1'

setup(
    name='datasette-plugin-demos',
    description='Examples of plugins for Datasette',
    author='Simon Willison',
    url='https://github.com/simonw/datasette-plugin-demos',
    license='Apache License, Version 2.0',
    version=VERSION,
    py_modules=['datasette_plugin_demos'],
    entry_points={
```

(continues on next page)

(continued from previous page)

```
        'datasette': [
            'plugin_demos = datasette_plugin_demos'
        ]
    },
    install_requires=['datasette']
)
```

And a Python module file, `datasette_plugin_demos.py`, that implements the plugin:

```
from datasette import hookimpl
import random


@hookimpl
def prepare_jinja2_environment(env):
    env.filters['uppercase'] = lambda u: u.upper()


@hookimpl
def prepare_connection(conn):
    conn.create_function('random_integer', 2, random.randint)
```

Having built a plugin in this way you can turn it into an installable package using the following command:

```
python3 setup.py sdist
```

This will create a `.tar.gz` file in the `dist/` directory.

You can then install your new plugin into a Datasette virtual environment or Docker container using `pip`:

```
pip install datasette-plugin-demos-0.1.tar.gz
```

To learn how to upload your plugin to PyPI for use by other people, read the PyPA guide to Packaging and distributing projects.

1.17.5 Static assets

If your plugin has a `static/` directory, Datasette will automatically configure itself to serve those static assets from the following path:

```
/-/static-plugins/NAME_OF_PLUGIN_PACKAGE/yourfile.js
```

See the datasette-plugin-demos repository for an example of how to create a package that includes a static folder.

1.17.6 Custom templates

If your plugin has a `templates/` directory, Datasette will attempt to load templates from that directory before it uses its own default templates.

The priority order for template loading is:

- templates from the `--template-dir` argument, if specified
- templates from the `templates/` directory in any installed plugins
- default templates that ship with Datasette

See *Customization* for more details on how to write custom templates, including which filenames to use to customize which parts of the Datasette UI.

1.17.7 Plugin configuration

Plugins can have their own configuration, embedded in a *Metadata* file. Configuration options for plugins live within a `"plugins"` key in that file, which can be included at the root, database or table level.

Here is an example of some plugin configuration for a specific table:

```
{
    "databases: {
        "sf-trees": {
            "tables": {
                "Street_Tree_List": {
                    "plugins": {
                        "datasette-cluster-map": {
                            "latitude_column": "lat",
                            "longitude_column": "lng"
                        }
                    }
                }
            }
        }
    }
}
```

This tells the `datasette-cluster-map` column which latitude and longitude columns should be used for a table called `Street_Tree_List` inside a database file called `sf-trees.db`.

Secret configuration values

Any values embedded in `metadata.json` will be visible to anyone who views the `/-/metadata` page of your Datasette instance. Some plugins may need configuration that should stay secret - API keys for example. There are two ways in which you can store secret configuration values.

As environment variables. If your secret lives in an environment variable that is available to the Datasette process, you can indicate that the configuration value should be read from that environment variable like so:

```
{
    "plugins": {
        "datasette-auth-github": {
            "client_secret": {
                "$env": "GITHUB_CLIENT_SECRET"
            }
        }
    }
}
```

As values in separate files. Your secrets can also live in files on disk. To specify a secret should be read from a file, provide the full file path like this:

```
{
    "plugins": {
        "datasette-auth-github": {
            "client_secret": {
```

(continues on next page)

(continued from previous page)

```
                "$file": "/secrets/client-secret"
            }
        }
    }
}
```

If you are publishing your data using the *datasette publish* family of commands, you can use the `--plugin-secret` option to set these secrets at publish time. For example, using Heroku you might run the following command:

```
$ datasette publish heroku my_database.db \
    --name my-heroku-app-demo \
    --install=datasette-auth-github \
    --plugin-secret datasette-auth-github client_id your_client_id \
    --plugin-secret datasette-auth-github client_secret your_client_secret
```

Writing plugins that accept configuration

When you are writing plugins, you can access plugin configuration like this using the `datasette.plugin_config()` method. If you know you need plugin configuration for a specific table, you can access it like this:

```
plugin_config = datasette.plugin_config(
    "datasette-cluster-map", database="sf-trees", table="Street_Tree_List"
)
```

This will return the `{"latitude_column": "lat", "longitude_column": "lng"}` in the above example.

If it cannot find the requested configuration at the table layer, it will fall back to the database layer and then the root layer. For example, a user may have set the plugin configuration option like so:

```
{
    "databases: {
        "sf-trees": {
            "plugins": {
                "datasette-cluster-map": {
                    "latitude_column": "xlat",
                    "longitude_column": "xlng"
                }
            }
        }
    }
}
```

In this case, the above code would return that configuration for ANY table within the `sf-trees` database.

The plugin configuration could also be set at the top level of `metadata.json`:

```
{
    "title": "This is the top-level title in metadata.json",
    "plugins": {
        "datasette-cluster-map": {
            "latitude_column": "xlat",
            "longitude_column": "xlng"
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Now that `datasette-cluster-map` plugin configuration will apply to every table in every database.

1.17.8 Plugin hooks

When you implement a plugin hook you can accept any or all of the parameters that are documented as being passed to that hook. For example, you can implement a `render_cell` plugin hook like this even though the hook definition defines more parameters than just `value` and `column`:

```
@hookimpl
def render_cell(value, column):
    if column == "stars":
        return "*" * int(value)
```

The full list of available plugin hooks is as follows.

prepare_connection(conn)

conn - sqlite3 connection object The connection that is being opened

This hook is called when a new SQLite database connection is created. You can use it to register custom SQL functions, aggregates and collations. For example:

```
from datasette import hookimpl
import random

@hookimpl
def prepare_connection(conn):
    conn.create_function('random_integer', 2, random.randint)
```

This registers a SQL function called `random_integer` which takes two arguments and can be called like this:

```
select random_integer(1, 10);
```

prepare_jinja2_environment(env)

env - jinja2 Environment The template environment that is being prepared

This hook is called with the Jinja2 environment that is used to evaluate Datasette HTML templates. You can use it to do things like register custom template filters, for example:

```
from datasette import hookimpl

@hookimpl
def prepare_jinja2_environment(env):
    env.filters['uppercase'] = lambda u: u.upper()
```

You can now use this filter in your custom templates like so:

```
Table name: {{ table|uppercase }}
```

extra_css_urls(template, database, table, datasette)

template - string The template that is being rendered, e.g. `database.html`

database - string or None The name of the database

table - string or None The name of the table

datasette - Datasette instance You can use this to access plugin configuration options via `datasette.plugin_config(your_plugin_name)`

Return a list of extra CSS URLs that should be included on the page. These can take advantage of the CSS class hooks described in *Customization*.

This can be a list of URLs:

```
from datasette import hookimpl

@hookimpl
def extra_css_urls():
    return [
        'https://stackpath.bootstrapcdn.com/bootstrap/4.1.0/css/bootstrap.min.css'
    ]
```

Or a list of dictionaries defining both a URL and an SRI hash:

```
from datasette import hookimpl

@hookimpl
def extra_css_urls():
    return [{
        'url': 'https://stackpath.bootstrapcdn.com/bootstrap/4.1.0/css/bootstrap.min.
↪css',
        'sri': 'sha384-
↪9gVQ4dYFwwWSjIDZnLEWnxCjeSWFphJiwGPXr1jddIhOegiu1FwO5qRGvFXOdJZ4',
    }]
```

extra_js_urls(template, database, table, datasette)

Same arguments as `extra_css_urls`.

This works in the same way as `extra_css_urls()` but for JavaScript. You can return either a list of URLs or a list of dictionaries:

```
from datasette import hookimpl

@hookimpl
def extra_js_urls():
    return [{
        'url': 'https://code.jquery.com/jquery-3.3.1.slim.min.js',
        'sri': 'sha384-q8i/
↪X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo',
    }]
```

You can also return URLs to files from your plugin's `static/` directory, if you have one:

```
from datasette import hookimpl
```

(continues on next page)

(continued from previous page)

```
@hookimpl
def extra_js_urls():
    return [
        '/-/static-plugins/your_plugin/app.js'
    ]
```

publish_subcommand(publish)

publish - Click publish command group The Click command group for the `datasette publish` subcommand

This hook allows you to create new providers for the `datasette publish` command. Datasette uses this hook internally to implement the default `now` and `heroku` subcommands, so you can read their source to see examples of this hook in action.

Let's say you want to build a plugin that adds a `datasette publish my_hosting_provider --api_key=xxx mydatabase.db` publish command. Your implementation would start like this:

```
from datasette import hookimpl
from datasette.publish.common import add_common_publish_arguments_and_options
import click


@hookimpl
def publish_subcommand(publish):
    @publish.command()
    @add_common_publish_arguments_and_options
    @click.option(
        "-k",
        "--api_key",
        help="API key for talking to my hosting provider",
    )
    def my_hosting_provider(
        files,
        metadata,
        extra_options,
        branch,
        template_dir,
        plugins_dir,
        static,
        install,
        version_note,
        title,
        license,
        license_url,
        source,
        source_url,
        api_key,
    ):
        # Your implementation goes here
```

render_cell(value, column, table, database, datasette)

Lets you customize the display of values within table cells in the HTML table view.

value - string, integer or None The value that was loaded from the database

column - string The name of the column being rendered

table - string or None The name of the table - or `None` if this is a custom SQL query

database - string The name of the database

datasette - Datasette instance You can use this to access plugin configuration options via `datasette.plugin_config(your_plugin_name)`

If your hook returns `None`, it will be ignored. Use this to indicate that your hook is not able to custom render this particular value.

If the hook returns a string, that string will be rendered in the table cell.

If you want to return HTML markup you can do so by returning a `jinja2.Markup` object.

Datasette will loop through all available `render_cell` hooks and display the value returned by the first one that does not return `None`.

Here is an example of a custom `render_cell()` plugin which looks for values that are a JSON string matching the following format:

```
{"href": "https://www.example.com/", "label": "Name"}
```

If the value matches that pattern, the plugin returns an HTML link element:

```
from datasette import hookimpl
import jinja2
import json


@hookimpl
def render_cell(value):
    # Render {"href": "...", "label": "..."} as link
    if not isinstance(value, str):
        return None
    stripped = value.strip()
    if not stripped.startswith("{") and stripped.endswith("}"):
        return None
    try:
        data = json.loads(value)
    except ValueError:
        return None
    if not isinstance(data, dict):
        return None
    if set(data.keys()) != {"href", "label"}:
        return None
    href = data["href"]
    if not (
        href.startswith("/") or href.startswith("http://")
        or href.startswith("https://")
    ):
        return None
    return jinja2.Markup('<a href="{href}">{label}</a>'.format(
        href=jinja2.escape(data["href"]),
        label=jinja2.escape(data["label"] or "") or " "
    ))
```

extra_body_script(template, database, table, view_name, datasette)

Extra JavaScript to be added to a `<script>` block at the end of the `<body>` element on the page.

`template` - string The template that is being rendered, e.g. `database.html`

`database` - string or None The name of the database, or `None` if the page does not correspond to a database (e.g. the root page)

`table` - string or None The name of the table, or `None` if the page does not correct to a table

`view_name` - string The name of the view being displayed. (*database*, *table*, and *row* are the most important ones.)

`datasette` - Datasette instance You can use this to access plugin configuration options via `datasette.plugin_config(your_plugin_name)`

The `template`, `database` and `table` options can be used to return different code depending on which template is being rendered and which database or table are being processed.

The `datasette` instance is provided primarily so that you can consult any plugin configuration options that may have been set, using the `datasette.plugin_config(plugin_name)` method documented above.

The string that you return from this function will be treated as "safe" for inclusion in a `<script>` block directly in the page, so it is up to you to apply any necessary escaping.

extra_template_vars(template, database, table, view_name, request, datasette)

Extra template variables that should be made available in the rendered template context.

`template` - string The template that is being rendered, e.g. `database.html`

`database` - string or None The name of the database, or `None` if the page does not correspond to a database (e.g. the root page)

`table` - string or None The name of the table, or `None` if the page does not correct to a table

`view_name` - string The name of the view being displayed. (*database*, *table*, and *row* are the most important ones.)

`request` - object The current HTTP request object. `request.scope` provides access to the ASGI scope.

`datasette` - Datasette instance You can use this to access plugin configuration options via `datasette.plugin_config(your_plugin_name)`

This hook can return one of three different types:

Dictionary If you return a dictionary its keys and values will be merged into the template context.

Function that returns a dictionary If you return a function it will be executed. If it returns a dictionary those values will will be merged into the template context.

Function that returns an awaitable function that returns a dictionary You can also return a function which returns an awaitable function which returns a dictionary. This means you can execute additional SQL queries using `datasette.execute()`.

Here's an example plugin that returns an authentication object from the ASGI scope:

```
@hookimpl
def extra_template_vars(request):
    return {
        "auth": request.scope.get("auth")
    }
```

And here's an example which returns the current version of SQLite:

```
@hookimpl
def extra_template_vars(datasette):
    async def inner():
        first_db = list(datasette.databases.keys())[0]
        return {
            "sqlite_version": (
                await datasette.execute(first_db, "select sqlite_version()")
            ).rows[0][0]
        }
    return inner
```

register_output_renderer(datasette)

datasette - Datasette instance You can use this to access plugin configuration options via `datasette.plugin_config(your_plugin_name)`

Allows the plugin to register a new output renderer, to output data in a custom format. The hook function should return a dictionary, or a list of dictionaries, which contain the file extension you want to handle and a callback function:

```
@hookimpl
def register_output_renderer(datasette):
    return {
        'extension': 'test',
        'callback': render_test
    }
```

This will register *render_test* to be called when paths with the extension *.test* (for example */database.test*, */database/table.test*, or */database/table/row.test*) are requested. When a request is received, the callback function is called with three positional arguments:

args - dictionary The GET parameters of the request

data - dictionary The data to be rendered

view_name - string The name of the view where the renderer is being called. (*database*, *table*, and *row* are the most important ones.)

The callback function can return *None*, if it is unable to render the data, or a dictionary with the following keys:

body - string or bytes, optional The response body, default empty

content_type - string, optional The Content-Type header, default *text/plain*

status_code - integer, optional The HTTP status code, default 200

A simple example of an output renderer callback function:

```
def render_test(args, data, view_name):
    return {
        'body': 'Hello World'
    }
```

register_facet_classes()

Return a list of additional Facet subclasses to be registered.

Each Facet subclass implements a new type of facet operation. The class should look like this:

```
class SpecialFacet(Facet):
    # This key must be unique across all facet classes:
    type = "special"

    async def suggest(self):
        # Use self.sql and self.params to suggest some facets
        suggested_facets = []
        suggested_facets.append({
            "name": column, # Or other unique name
            # Construct the URL that will enable this facet:
            "toggle_url": self.ds.absolute_url(
                self.request, path_with_added_args(
                    self.request, {"_facet": column}
                )
            ),
        })
        return suggested_facets

    async def facet_results(self):
        # This should execute the facet operation and return results, again
        # using self.sql and self.params as the starting point
        facet_results = {}
        facets_timed_out = []
        # Do some calculations here...
        for column in columns_selected_for_facet:
            try:
                facet_results_values = []
                # More calculations...
                facet_results_values.append({
                    "value": value,
                    "label": label,
                    "count": count,
                    "toggle_url": self.ds.absolute_url(self.request, toggle_path),
                    "selected": selected,
                })
                facet_results[column] = {
                    "name": column,
                    "results": facet_results_values,
                    "truncated": len(facet_rows_results) > facet_size,
                }
            except QueryInterrupted:
                facets_timed_out.append(column)

        return facet_results, facets_timed_out
```

See datasette/facets.py for examples of how these classes can work.

The plugin hook can then be used to register the new facet class like this:

```
@hookimpl
def register_facet_classes():
    return [SpecialFacet]
```

asgi_wrapper(datasette)

Return an ASGI middleware wrapper function that will be applied to the Datasette ASGI application.

This is a very powerful hook. You can use it to manipulate the entire Datasette response, or even to configure new URL routes that will be handled by your own custom code.

You can write your ASGI code directly against the low-level specification, or you can use the middleware utilites provided by an ASGI framework such as Starlette.

This example plugin adds a `x-databases` HTTP header listing the currently attached databases:

```
from datasette import hookimpl
from functools import wraps


@hookimpl
def asgi_wrapper(datasette):
    def wrap_with_databases_header(app):
        @wraps(app)
        async def add_x_databases_header(scope, recieve, send):
            async def wrapped_send(event):
                if event["type"] == "http.response.start":
                    original_headers = event.get("headers") or []
                    event = {
                        "type": event["type"],
                        "status": event["status"],
                        "headers": original_headers + [
                            [b"x-databases",
                            ", ".join(datasette.databases.keys()).encode("utf-8")]
                        ],
                    }
                await send(event)
            await app(scope, recieve, wrapped_send)
        return add_x_databases_header
    return wrap_with_databases_header
```

1.18 Contributing

Datasette is an open source project. We welcome contributions!

This document describes how to contribute to Datasette core. You can also contribute to the wider Datasette ecosystem by creating new *Plugins*.

1.18.1 General guidelines

- **master should always be releasable**. Incomplete features should live in branches. This ensures that any small bug fixes can be quickly released.
- **The ideal commit** should bundle together the implementation, unit tests and associated documentation updates. The commit message should link to an associated issue.

1.18.2 Setting up a development environment

If you have Python 3.5 or higher installed on your computer (on OS X the easiest way to do this is using homebrew) you can install an editable copy of Datasette using the following steps.

If you want to use GitHub to publish your changes, first create a fork of datasette under your own GitHub account.

Now clone that repository somewhere on your computer:

```
git clone git@github.com:YOURNAME/datasette
```

If you just want to get started without creating your own fork, you can do this instead:

```
git clone git@github.com:simonw/datasette
```

The next step is to create a virtual environment for your project and use it to install Datasette's dependencies:

```
cd datasette
# Create a virtual environment in ./venv
python3 -m venv ./venv
# Now activate the virtual environment, so pip can install into it
source venv/bin/activate
# Install Datasette and its testing dependencies
python3 -m pip install -e .[test]
```

That last line does most of the work: `pip install -e` means "install this package in a way that allows me to edit the source code in place". The `.[test]` option means "use the setup.py in this directory and install the optional testing dependencies as well".

Once you have done this, you can run the Datasette unit tests from inside your `datasette/` directory using pytest like so:

```
pytest
```

To run Datasette itself, just type `datasette`.

You're going to need at least one SQLite database. An easy way to get started is to use the fixtures database that Datasette uses for its own tests.

You can create a copy of that database by running this command:

```
python tests/fixtures.py fixtures.db
```

Now you can run Datasette against the new fixtures database like so:

```
datasette fixtures.db
```

This will start a server at `http://127.0.0.1:8001/`.

Any changes you make in the `datasette/templates` or `datasette/static` folder will be picked up immediately (though you may need to do a force-refresh in your browser to see changes to CSS or JavaScript).

If you want to change Datasette's Python code you can use the `--reload` option to cause Datasette to automatically reload any time the underlying code changes:

```
datasette --reload fixtures.db
```

You can also use the `fixtures.py` script to recreate the testing version of `metadata.json` used by the unit tests. To do that:

```
python tests/fixtures.py fixtures.db fixtures-metadata.json
```

(You may need to delete `fixtures.db` before running this command.)

Then run Datasette like this:

```
datasette fixtures.db -m fixtures-metadata.json
```

1.18.3 Editing and building the documentation

Datasette's documentation lives in the `docs/` directory and is deployed automatically using Read The Docs.

The documentation is written using reStructuredText. You may find this article on The subset of reStructuredText worth committing to memory useful.

You can build it locally by installing `sphinx` and `sphinx_rtd_theme` in your Datasette development environment and then running `make html` directly in the `docs/` directory:

```
# You may first need to activate your virtual environment:
source venv/bin/activate

# Install the dependencies needed to build the docs
pip install -e .[docs]

# Now build the docs
cd docs/
make html
```

This will create the HTML version of the documentation in `docs/_build/html`. You can open it in your browser like so:

```
open _build/html/index.html
```

Any time you make changes to a `.rst` file you can re-run `make html` to update the built documents, then refresh them in your browser.

For added productivity, you can use use sphinx-autobuild to run Sphinx in auto-build mode. This will run a local webserver serving the docs that automatically rebuilds them and refreshes the page any time you hit save in your editor.

`sphinx-autobuild` will have been installed when you ran `pip install -e .[docs]`. In your `docs/` directory you can start the server by running the following:

```
make livehtml
```

Now browse to `http://localhost:8000/` to view the documentation. Any edits you make should be instantly reflected in your browser.

1.18.4 Release process

Datasette releases are performed using tags. When a new version tag is pushed to GitHub, a Travis CI task will perform the following:

- Run the unit tests against all supported Python versions. If the tests pass...
- Set up https://v0-25-1.datasette.io/ (but with the new tag) to point to a live demo of this release
- Build a Docker image of the release and push a tag to https://hub.docker.com/r/datasetteproject/datasette
- Re-point the "latest" tag on Docker Hub to the new image
- Build a wheel bundle of the underlying Python source code
- Push that new wheel up to PyPI: https://pypi.org/project/datasette/

To deploy new releases you will need to have push access to the main Datasette GitHub repository.

Datasette follows Semantic Versioning:

```
major.minor.patch
```

We increment `major` for backwards-incompatible releases. Datasette is currently pre-1.0 so the major version is always `0`.

We increment `minor` for new features.

We increment `patch` for bugfix releass.

To release a new version, first create a commit that updates *the changelog* with highlights of the new version. An example commit can be seen here:

```
# Update changelog
git commit -m "Release 0.25.2" -a
git push
```

For non-bugfix releases you may want to update the news section of `README.md` as part of the same commit.

Wait long enough for Travis to build and deploy the demo version of that commit (otherwise the tag deployment may fail to alias to it properly). Then run the following:

```
git tag 0.25.2
git push --tags
```

Once the release is out, you can manually update https://github.com/simonw/datasette/releases

1.19 Changelog

1.19.1 0.29.1 (2019-07-11)

- Fixed bug with static mounts using relative paths which could lead to traversal exploits (#555) - thanks Abdussamet Kocak!

1.19.2 0.29 (2019-07-07)

ASGI, new plugin hooks, facet by date and much, much more...

ASGI

ASGI is the Asynchronous Server Gateway Interface standard. I've been wanting to convert Datasette into an ASGI application for over a year - Port Datasette to ASGI #272 tracks thirteen months of intermittent development - but with Datasette 0.29 the change is finally released. This also means Datasette now runs on top of Uvicorn and no longer depends on Sanic.

I wrote about the significance of this change in Porting Datasette to ASGI, and Turtles all the way down.

The most exciting consequence of this change is that Datasette plugins can now take advantage of the ASGI standard.

New plugin hook: asgi_wrapper

The *asgi_wrapper(datasette)* plugin hook allows plugins to entirely wrap the Datasette ASGI application in their own ASGI middleware. (#520)

Two new plugins take advantage of this hook:

- datasette-auth-github adds a authentication layer: users will have to sign in using their GitHub account before they can view data or interact with Datasette. You can also use it to restrict access to specific GitHub users, or to members of specified GitHub organizations or teams.
- datasette-cors allows you to configure CORS headers for your Datasette instance. You can use this to enable JavaScript running on a whitelisted set of domains to make `fetch()` calls to the JSON API provided by your Datasette instance.

New plugin hook: extra_template_vars

The plugin_extra_template_vars plugin hook allows plugins to inject their own additional variables into the Datasette template context. This can be used in conjunction with custom templates to customize the Datasette interface. datasette-auth-github uses this hook to add custom HTML to the new top navigation bar (which is designed to be modified by plugins, see #540).

Secret plugin configuration options

Plugins like datasette-auth-github need a safe way to set secret configuration options. Since the default mechanism for configuring plugins exposes those settings in `/-/metadata` a new mechanism was needed. *Secret configuration values* describes how plugins can now specify that their settings should be read from a file or an environment variable:

```
{
    "plugins": {
        "datasette-auth-github": {
            "client_secret": {
                "$env": "GITHUB_CLIENT_SECRET"
            }
        }
    }
}
```

These plugin secrets can be set directly using `datasette publish`. See *Custom metadata and plugins* for details. (#538 and #543)

Facet by date

If a column contains datetime values, Datasette can now facet that column by date. (#481)

Easier custom templates for table rows

If you want to customize the display of individual table rows, you can do so using a `_table.html` template include that looks something like this:

```
{% for row in display_rows %}
    <div>
        <h2>{{ row["title"] }}</h2>
```

(continues on next page)

(continued from previous page)

```
        <p>{{ row["description"] }}<lp>
        <p>Category: {{ row.display("category_id") }}</p>
    </div>
{% endfor %}
```

This is a **backwards incompatible change**. If you previously had a custom template called `_rows_and_columns.html` you need to rename it to `_table.html`.

See *Custom templates* for full details.

?_through= for joins through many-to-many tables

The new `?_through={json}` argument to the Table view allows records to be filtered based on a many-to-many relationship. See *Special table arguments* for full documentation - here's an example. (#355)

This feature was added to help support facet by many-to-many, which isn't quite ready yet but will be coming in the next Datasette release.

Small changes

- Databases published using `datasette publish` now open in *Immutable mode*. (#469)
- `?col__date=` now works for columns containing spaces
- Automatic label detection (for deciding which column to show when linking to a foreign key) has been improved. (#485)
- Fixed bug where pagination broke when combined with an expanded foreign key. (#489)
- Contributors can now run `pip install -e .[docs]` to get all of the dependencies needed to build the documentation, including `cd docs && make livehtml` support.
- Datasette's dependencies are now all specified using the ~= match operator. (#532)
- `white-space: pre-wrap` now used for table creation SQL. (#505)

Full list of commits between 0.28 and 0.29.

1.19.3 0.28 (2019-05-19)

A salmagundi of new features!

Supporting databases that change

From the beginning of the project, Datasette has been designed with read-only databases in mind. If a database is guaranteed not to change it opens up all kinds of interesting opportunities - from taking advantage of SQLite immutable mode and HTTP caching to bundling static copies of the database directly in a Docker container. The interesting ideas in Datasette explores this idea in detail.

As my goals for the project have developed, I realized that read-only databases are no longer the right default. SQLite actually supports concurrent access very well provided only one thread attempts to write to a database at a time, and I keep encountering sensible use-cases for running Datasette on top of a database that is processing inserts and updates.

So, as-of version 0.28 Datasette no longer assumes that a database file will not change. It is now safe to point Datasette at a SQLite database which is being updated by another process.

Making this change was a lot of work - see tracking tickets #418, #419 and #420. It required new thinking around how Datasette should calculate table counts (an expensive operation against a large, changing database) and also meant reconsidering the "content hash" URLs Datasette has used in the past to optimize the performance of HTTP caches.

Datasette can still run against immutable files and gains numerous performance benefits from doing so, but this is no longer the default behaviour. Take a look at the new *Performance and caching* documentation section for details on how to make the most of Datasette against data that you know will be staying read-only and immutable.

Faceting improvements, and faceting plugins

Datasette *Facets* provide an intuitive way to quickly summarize and interact with data. Previously the only supported faceting technique was column faceting, but 0.28 introduces two powerful new capibilities: facet-by-JSON-array and the ability to define further facet types using plugins.

Facet by array (#359) is only available if your SQLite installation provides the `json1` extension. Datasette will automatically detect columns that contain JSON arrays of values and offer a faceting interface against those columns - useful for modelling things like tags without needing to break them out into a new table. See *Facet by JSON array* for more.

The new *register_facet_classes()* plugin hook (#445) can be used to register additional custom facet classes. Each facet class should provide two methods: `suggest()` which suggests facet selections that might be appropriate for a provided SQL query, and `facet_results()` which executes a facet operation and returns results. Datasette's own faceting implementations have been refactored to use the same API as these plugins.

datasette publish cloudrun

Google Cloud Run is a brand new serverless hosting platform from Google, which allows you to build a Docker container which will run only when HTTP traffic is recieved and will shut down (and hence cost you nothing) the rest of the time. It's similar to Zeit's Now v1 Docker hosting platform which sadly is no longer accepting signups from new users.

The new `datasette publish cloudrun` command was contributed by Romain Primet (#434) and publishes selected databases to a new Datasette instance running on Google Cloud Run.

See *Publishing to Google Cloud Run* for full documentation.

register_output_renderer plugins

Russ Garrett implemented a new Datasette plugin hook called *register_output_renderer* (#441) which allows plugins to create additional output renderers in addition to Datasette's default `.json` and `.csv`.

Russ's in-development datasette-geo plugin includes an example of this hook being used to output `.geojson` automatically converted from SpatiaLite.

Medium changes

- Datasette now conforms to the Black coding style (#449) - and has a unit test to enforce this in the future
- **New *Special table arguments*:**
  - `?columnname__in=value1,value2,value3` filter for executing SQL IN queries against a table, see *Table arguments* (#433)
  - `?columnname__date=yyyy-mm-dd` filter which returns rows where the spoecified datetime column falls on the specified date (583b22a)

  - `?tags__arraycontains=tag` filter which acts against a JSON array contained in a column (78e45ea)
  - `?_where=sql-fragment` filter for the table view (#429)
  - `?_fts_table=mytable` and `?_fts_pk=mycolumn` querystring options can be used to specify which FTS table to use for a search query - see *Configuring full-text search for a table or view* (#428)
- You can now pass the same table filter multiple times - for example, `?content__not=world&content__not=hello` will return all rows where the content column is neither `hello` or `world` (#288)
- You can now specify `about` and `about_url` metadata (in addition to `source` and `license`) linking to further information about a project - see *Source, license and about*
- New `?_trace=1` parameter now adds debug information showing every SQL query that was executed while constructing the page (#435)
- `datasette inspect` now just calculates table counts, and does not introspect other database metadata (#462)
- Removed `/-/inspect` page entirely - this will be replaced by something similar in the future, see #465
- Datasette can now run against an in-memory SQLite database. You can do this by starting it without passing any files or by using the new `--memory` option to `datasette serve`. This can be useful for experimenting with SQLite queries that do not access any data, such as `SELECT 1+1` or `SELECT sqlite_version()`.

Small changes

- We now show the size of the database file next to the download link (#172)
- New `/-/databases` introspection page shows currently connected databases (#470)
- Binary data is no longer displayed on the table and row pages (#442 - thanks, Russ Garrett)
- New show/hide SQL links on custom query pages (#415)
- The *extra_body_script* plugin hook now accepts an optional `view_name` argument (#443 - thanks, Russ Garrett)
- Bumped Jinja2 dependency to 2.10.1 (#426)
- All table filters are now documented, and documentation is enforced via unit tests (2c19a27)
- New project guideline: master should stay shippable at all times! (31f36e1)
- Fixed a bug where `sqlite_timelimit()` occasionally failed to clean up after itself (bac4e01)
- We no longer load additional plugins when executing pytest (#438)
- Homepage now links to database views if there are less than five tables in a database (#373)
- The `--cors` option is now respected by error pages (#453)
- `datasette publish heroku` now uses the `--include-vcs-ignore` option, which means it works under Travis CI (#407)
- `datasette publish heroku` now publishes using Python 3.6.8 (666c374)
- Renamed `datasette publish now` to `datasette publish nowv1` (#472)
- `datasette publish nowv1` now accepts multiple `--alias` parameters (09ef305)
- Removed the `datasette skeleton` command (#476)
- The *documentation on how to build the documentation* now recommends `sphinx-autobuild`

1.19.4 0.27.1 (2019-05-09)

- Tiny bugfix release: don't install `tests/` in the wrong place. Thanks, Veit Heller.

1.19.5 0.27 (2019-01-31)

- New command: `datasette plugins` (*documentation*) shows you the currently installed list of plugins.
- Datasette can now output newline-delimited JSON using the new `?_shape=array&_nl=on` querystring option.
- Added documentation on *The Datasette Ecosystem*.
- Now using Python 3.7.2 as the base for the official Datasette Docker image.

1.19.6 0.26.1 (2019-01-10)

- `/-/versions` now includes SQLite `compile_options` (#396)
- datasetteproject/datasette Docker image now uses SQLite 3.26.0 (#397)
- Cleaned up some deprecation warnings under Python 3.7

1.19.7 0.26 (2019-01-02)

- `datasette serve --reload` now restarts Datasette if a database file changes on disk.
- `datasette publish now` now takes an optional `--alias mysite.now.sh` argument. This will attempt to set an alias after the deploy completes.
- Fixed a bug where the advanced CSV export form failed to include the currently selected filters (#393)

1.19.8 0.25.2 (2018-12-16)

- `datasette publish heroku` now uses the `python-3.6.7` runtime
- Added documentation on *how to build the documentation*
- Added documentation covering *our release process*
- Upgraded to pytest 4.0.2

1.19.9 0.25.1 (2018-11-04)

Documentation improvements plus a fix for publishing to Zeit Now.

- `datasette publish now` now uses Zeit's v1 platform, to work around the new 100MB image limit. Thanks, @slygent - closes #366.

1.19.10 0.25 (2018-09-19)

New plugin hooks, improved database view support and an easier way to use more recent versions of SQLite.

- New `publish_subcommand` plugin hook. A plugin can now add additional `datasette publish` publishers in addition to the default `now` and `heroku`, both of which have been refactored into default plugins. *publish_subcommand documentation*. Closes #349
- New `render_cell` plugin hook. Plugins can now customize how values are displayed in the HTML tables produced by Datasette's browseable interface. datasette-json-html and datasette-render-images are two new plugins that use this hook. *render_cell documentation*. Closes #352
- New `extra_body_script` plugin hook, enabling plugins to provide additional JavaScript that should be added to the page footer. *extra_body_script documentation*.
- `extra_css_urls` and `extra_js_urls` hooks now take additional optional parameters, allowing them to be more selective about which pages they apply to. *Documentation*.
- You can now use the *sortable_columns metadata setting* to explicitly enable sort-by-column in the interface for database views, as well as for specific tables.
- The new `fts_table` and `fts_pk` metadata settings can now be used to *explicitly configure full-text search for a table or a view*, even if that table is not directly coupled to the SQLite FTS feature in the database schema itself.
- Datasette will now use pysqlite3 in place of the standard library `sqlite3` module if it has been installed in the current environment. This makes it much easier to run Datasette against a more recent version of SQLite, including the just-released SQLite 3.25.0 which adds window function support. More details on how to use this in #360
- New mechanism that allows *plugin configuration options* to be set using `metadata.json`.

1.19.11 0.24 (2018-07-23)

A number of small new features:

- `datasette publish heroku` now supports `--extra-options`, fixes #334
- Custom error message if SpatiaLite is needed for specified database, closes #331
- New config option: `truncate_cells_html` for *truncating long cell values* in HTML view - closes #330
- Documentation for *datasette publish and datasette package*, closes #337
- Fixed compatibility with Python 3.7
- `datasette publish heroku` now supports app names via the `-n` option, which can also be used to overwrite an existing application [Russ Garrett]
- Title and description metadata can now be set for *canned SQL queries*, closes #342
- New `force_https_on` config option, fixes `https://` API URLs when deploying to Zeit Now - closes #333
- `?_json_infinity=1` querystring argument for handling Infinity/-Infinity values in JSON, closes #332
- URLs displayed in the results of custom SQL queries are now URLified, closes #298

1.19.12 0.23.2 (2018-07-07)

Minor bugfix and documentation release.

- CSV export now respects `--cors`, fixes #326
- *Installation instructions*, including docker image - closes #328
- Fix for row pages for tables with / in, closes #325

1.19.13 0.23.1 (2018-06-21)

Minor bugfix release.

- Correctly display empty strings in HTML table, closes #314
- Allow "." in database filenames, closes #302
- 404s ending in slash redirect to remove that slash, closes #309
- Fixed incorrect display of compound primary keys with foreign key references. Closes #319
- Docs + example of canned SQL query using || concatenation. Closes #321
- Correctly display facets with value of 0 - closes #318
- Default 'expand labels' to checked in CSV advanced export

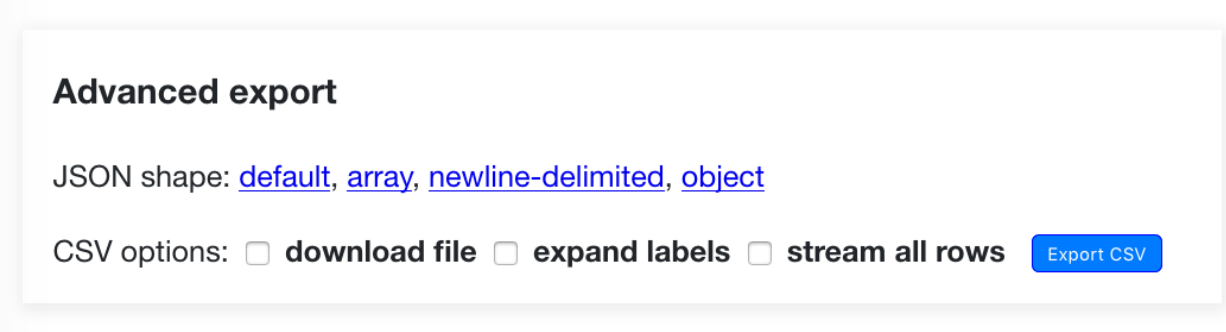
1.19.14 0.23 (2018-06-18)

This release features CSV export, improved options for foreign key expansions, new configuration settings and improved support for SpatiaLite.

See datasette/compare/0.22.1...0.23 for a full list of commits added since the last release.

CSV export

Any Datasette table, view or custom SQL query can now be exported as CSV.



Check out the *CSV export documentation* for more details, or try the feature out on https://fivethirtyeight.datasettes.com/fivethirtyeight/bechdel%2Fmovies

If your table has more than *max_returned_rows* (default 1,000) Datasette provides the option to *stream all rows*. This option takes advantage of async Python and Datasette's efficient *pagination* to iterate through the entire matching result set and stream it back as a downloadable CSV file.

Foreign key expansions

When Datasette detects a foreign key reference it attempts to resolve a label for that reference (automatically or using the *Specifying the label column for a table* metadata option) so it can display a link to the associated row.

This expansion is now also available for JSON and CSV representations of the table, using the new `_labels=on` querystring option. See *Expanding foreign key references* for more details.

New configuration settings

Datasette's *Configuration* now also supports boolean settings. A number of new configuration options have been added:

- `num_sql_threads` - the number of threads used to execute SQLite queries. Defaults to 3.
- `allow_facet` - enable or disable custom *Facets* using the *_facet=* parameter. Defaults to on.
- `suggest_facets` - should Datasette suggest facets? Defaults to on.
- `allow_download` - should users be allowed to download the entire SQLite database? Defaults to on.
- `allow_sql` - should users be allowed to execute custom SQL queries? Defaults to on.
- `default_cache_ttl` - Default HTTP caching max-age header in seconds. Defaults to 365 days - caching can be disabled entirely by settings this to 0.
- `cache_size_kb` - Set the amount of memory SQLite uses for its per-connection cache, in KB.
- `allow_csv_stream` - allow users to stream entire result sets as a single CSV file. Defaults to on.
- `max_csv_mb` - maximum size of a returned CSV file in MB. Defaults to 100MB, set to 0 to disable this limit.

Control HTTP caching with ?_ttl=

You can now customize the HTTP max-age header that is sent on a per-URL basis, using the new `?_ttl=` querystring parameter.

You can set this to any value in seconds, or you can set it to 0 to disable HTTP caching entirely.

Consider for example this query which returns a randomly selected member of the Avengers:

```
select * from [avengers/avengers] order by random() limit 1
```

If you hit the following page repeatedly you will get the same result, due to HTTP caching:

/fivethirtyeight?sql=select+*+from+%5Bavengers%2Favengers%5D+order+by+random%28%29+limit+1

By adding *?_ttl=0* to the zero you can ensure the page will not be cached and get back a different super hero every time:

/fivethirtyeight?sql=select+*+from+%5Bavengers%2Favengers%5D+order+by+random%28%29+limit+1&_ttl=0

Improved support for SpatiaLite

The SpatiaLite module for SQLite adds robust geospatial features to the database.

Getting SpatiaLite working can be tricky, especially if you want to use the most recent alpha version (with support for K-nearest neighbor).

Datasette now includes *extensive documentation on SpatiaLite*, and thanks to Ravi Kotecha our GitHub repo includes a Dockerfile that can build the latest SpatiaLite and configure it for use with Datasette.

The `datasette publish` and `datasette package` commands now accept a new `--spatialite` argument which causes them to install and configure SpatiaLite as part of the container they deploy.

latest.datasette.io

Every commit to Datasette master is now automatically deployed by Travis CI to https://latest.datasette.io/ - ensuring there is always a live demo of the latest version of the software.

The demo uses the fixtures from our unit tests, ensuring it demonstrates the same range of functionality that is covered by the tests.

You can see how the deployment mechanism works in our .travis.yml file.

Miscellaneous

- Got JSON data in one of your columns? Use the new `?_json=COLNAME` argument to tell Datasette to return that JSON value directly rather than encoding it as a string.
- If you just want an array of the first value of each row, use the new `?_shape=arrayfirst` option - example.

1.19.15 0.22.1 (2018-05-23)

Bugfix release, plus we now use versioneer for our version numbers.

- Faceting no longer breaks pagination, fixes #282
- Add `__version_info__` derived from *__version__* [Robert Gieseke]

  This might be tuple of more than two values (major and minor version) if commits have been made after a release.
- Add version number support with Versioneer. [Robert Gieseke]

  Versioneer Licence: Public Domain (CC0-1.0)

  Closes #273
- Refactor inspect logic [Russ Garrett]

1.19.16 0.22 (2018-05-20)

The big new feature in this release is *Facets*. Datasette can now apply faceted browse to any column in any table. It will also suggest possible facets. See the Datasette Facets announcement post for more details.

In addition to the work on facets:

- Added docs for introspection endpoints
- New `--config` option, added `--help-config`, closes #274

  Removed the `--page_size=` argument to `datasette serve` in favour of:

```
datasette serve --config default_page_size:50 mydb.db
```

  Added new help section:

```
$ datasette --help-config
Config options:
  default_page_size             Default page size for the table view
                                (default=100)
  max_returned_rows             Maximum rows that can be returned from a table
                                or custom query (default=1000)
  sql_time_limit_ms             Time limit for a SQL query in milliseconds
                                (default=1000)
  default_facet_size            Number of values to return for requested facets
                                (default=30)
  facet_time_limit_ms           Time limit for calculating a requested facet
                                (default=200)
  facet_suggest_time_limit_ms   Time limit for calculating a suggested facet
                                (default=50)
```

- Only apply responsive table styles to `.rows-and-column`

  Otherwise they interfere with tables in the description, e.g. on https://fivethirtyeight.datasettes.com/fivethirtyeight/nba-elo%2Fnbaallelo

- Refactored views into new `views/` modules, refs #256
- Documentation for SQLite full-text search support, closes #253
- `/-/versions` now includes SQLite `fts_versions`, closes #252

1.19.17 0.21 (2018-05-05)

New JSON `_shape=` options, the ability to set table `_size=` and a mechanism for searching within specific columns.

- Default tests to using a longer timelimit

  Every now and then a test will fail in Travis CI on Python 3.5 because it hit the default 20ms SQL time limit.

  Test fixtures now default to a 200ms time limit, and we only use the 20ms time limit for the specific test that tests query interruption. This should make our tests on Python 3.5 in Travis much more stable.

- Support `_search_COLUMN=text` searches, closes #237
- Show version on `/-/plugins` page, closes #248
- `?_size=max` option, closes #249
- Added `/-/versions` and `/-/versions.json`, closes #244

  Sample output:

```
{
  "python": {
    "version": "3.6.3",
    "full": "3.6.3 (default, Oct  4 2017, 06:09:38) \n[GCC 4.2.1 Compatible Apple␣
↪LLVM 9.0.0 (clang-900.0.37)]"
  },
  "datasette": {
    "version": "0.20"
  },
  "sqlite": {
    "version": "3.23.1",
    "extensions": {
      "json1": null,
```

(continues on next page)

(continued from previous page)

```
      "spatialite": "4.3.0a"
    }
  }
}
```

- Renamed `?_sql_time_limit_ms=` to `?_timelimit`, closes #242
- New `?_shape=array` option + tweaks to `_shape`, closes #245
  - Default is now `?_shape=arrays` (renamed from `lists`)
  - New `?_shape=array` returns an array of objects as the root object
  - Changed `?_shape=object` to return the object as the root
  - Updated docs
- FTS tables now detected by `inspect()`, closes #240
- New `?_size=XXX` querystring parameter for table view, closes #229

  Also added documentation for all of the `_special` arguments.

  Plus deleted some duplicate logic implementing `_group_count`.
- If `max_returned_rows==page_size`, increment `max_returned_rows` - fixes #230
- New `hidden: True` option for table metadata, closes #239
- Hide `idx_*` tables if spatialite detected, closes #228
- Added `class=rows-and-columns` to custom query results table
- Added CSS class `rows-and-columns` to main table
- `label_column` option in `metadata.json` - closes #234

1.19.18 0.20 (2018-04-20)

Mostly new work on the *Plugins* mechanism: plugins can now bundle static assets and custom templates, and `datasette publish` has a new `--install=name-of-plugin` option.

- Add col-X classes to HTML table on custom query page
- Fixed out-dated template in documentation
- Plugins can now bundle custom templates, #224
- Added /-/metadata /-/plugins /-/inspect, #225
- Documentation for –install option, refs #223
- Datasette publish/package –install option, #223
- Fix for plugins in Python 3.5, #222
- New plugin hooks: extra_css_urls() and extra_js_urls(), #214
- /-/static-plugins/PLUGIN_NAME/ now serves static/ from plugins
- <th> now gets class=”col-X” - plus added col-X documentation
- Use to_css_class for table cell column classes

  This ensures that columns with spaces in the name will still generate usable CSS class names. Refs #209

- Add column name classes to <td>s, make PK bold [Russ Garrett]
- Don't duplicate simple primary keys in the link column [Russ Garrett]

  When there's a simple (single-column) primary key, it looks weird to duplicate it in the link column.

  This change removes the second PK column and treats the link column as if it were the PK column from a header/sorting perspective.
- Correct escaping for HTML display of row links [Russ Garrett]
- Longer time limit for test_paginate_compound_keys

  It was failing intermittently in Travis - see #209
- Use application/octet-stream for downloadable databses
- Updated PyPI classifiers
- Updated PyPI link to pypi.org

1.19.19 0.19 (2018-04-16)

This is the first preview of the new Datasette plugins mechanism. Only two plugin hooks are available so far - for custom SQL functions and custom template filters. There's plenty more to come - read the documentation and get involved in the tracking ticket if you have feedback on the direction so far.

- Fix for `_sort_desc=sortable_with_nulls` test, refs #216
- Fixed #216 - paginate correctly when sorting by nullable column
- Initial documentation for plugins, closes #213

  https://datasette.readthedocs.io/en/latest/plugins.html
- New `--plugins-dir=plugins/` option (#212)

  New option causing Datasette to load and evaluate all of the Python files in the specified directory and register any plugins that are defined in those files.

  This new option is available for the following commands:

```
datasette serve mydb.db --plugins-dir=plugins/
datasette publish now/heroku mydb.db --plugins-dir=plugins/
datasette package mydb.db --plugins-dir=plugins/
```

- Start of the plugin system, based on pluggy (#210)

  Uses https://pluggy.readthedocs.io/ originally created for the py.test project

  We're starting with two plugin hooks:

  `prepare_connection(conn)`

  This is called when a new SQLite connection is created. It can be used to register custom SQL functions.

  `prepare_jinja2_environment(env)`

  This is called with the Jinja2 environment. It can be used to register custom template tags and filters.

  An example plugin which uses these two hooks can be found at https://github.com/simonw/datasette-plugin-demos or installed using `pip install datasette-plugin-demos`

  Refs #14

- Return HTTP 405 on InvalidUsage rather than 500. [Russ Garrett]

  This also stops it filling up the logs. This happens for HEAD requests at the moment - which perhaps should be handled better, but that's a different issue.

1.19.20 0.18 (2018-04-14)

This release introduces support for units, contributed by Russ Garrett (#203). You can now optionally specify the units for specific columns using `metadata.json`. Once specified, units will be displayed in the HTML view of your table. They also become available for use in filters - if a column is configured with a unit of distance, you can request all rows where that column is less than 50 meters or more than 20 feet for example.

- Link foreign keys which don't have labels. [Russ Garrett]

  This renders unlabeled FKs as simple links.

  Also includes bonus fixes for two minor issues:

  - In foreign key link hrefs the primary key was escaped using HTML escaping rather than URL escaping. This broke some non-integer PKs.
  - Print tracebacks to console when handling 500 errors.

- Fix SQLite error when loading rows with no incoming FKs. [Russ Garrett]

  This fixes `ERROR: conn=<sqlite3.Connection object at 0x10bbb9f10>, sql = 'select ', params = {'id': '1'}` caused by an invalid query when loading incoming FKs.

  The error was ignored due to async but it still got printed to the console.

- Allow custom units to be registered with Pint. [Russ Garrett]
- Support units in filters. [Russ Garrett]
- Tidy up units support. [Russ Garrett]
  - Add units to exported JSON
  - Units key in metadata skeleton
  - Docs
- Initial units support. [Russ Garrett]

  Add support for specifying units for a column in `metadata.json` and rendering them on display using pint

1.19.21 0.17 (2018-04-13)

- Release 0.17 to fix issues with PyPI

1.19.22 0.16 (2018-04-13)

- Better mechanism for handling errors; 404s for missing table/database

  New error mechanism closes #193

  404s for missing tables/databases closes #184

- long_description in markdown for the new PyPI
- Hide Spatialite system tables. [Russ Garrett]

- Allow `explain select` / `explain query plan select` #201
- Datasette inspect now finds primary_keys #195
- Ability to sort using form fields (for mobile portrait mode) #199

  We now display sort options as a select box plus a descending checkbox, which means you can apply sort orders even in portrait mode on a mobile phone where the column headers are hidden.

1.19.23 0.15 (2018-04-09)

The biggest new feature in this release is the ability to sort by column. On the table page the column headers can now be clicked to apply sort (or descending sort), or you can specify `?_sort=column` or `?_sort_desc=column` directly in the URL.

- `table_rows` => `table_rows_count`, `filtered_table_rows` => `filtered_table_rows_count`

  Renamed properties. Closes #194

- New `sortable_columns` option in `metadata.json` to control sort options.

  You can now explicitly set which columns in a table can be used for sorting using the `_sort` and `_sort_desc` arguments using `metadata.json`:

```
{
    "databases": {
        "database1": {
            "tables": {
                "example_table": {
                    "sortable_columns": [
                        "height",
                        "weight"
                    ]
                }
            }
        }
    }
}
```

  Refs #189

- Column headers now link to sort/desc sort - refs #189
- `_sort` and `_sort_desc` parameters for table views

  Allows for paginated sorted results based on a specified column.

  Refs #189

- Total row count now correct even if `_next` applied
- Use .custom_sql() for _group_count implementation (refs #150)
- Make HTML title more readable in query template (#180) [Ryan Pitts]
- New `?_shape=objects/object/lists` param for JSON API (#192)

  New `_shape=` parameter replacing old `.jsono` extension

  Now instead of this:

```
/database/table.jsono
```

We use the `_shape` parameter like this:

```
/database/table.json?_shape=objects
```

Also introduced a new `_shape` called `object` which looks like this:

```
/database/table.json?_shape=object
```

Returning an object for the rows key:

```
...
"rows": {
    "pk1": {
        ...
    },
    "pk2": {
        ...
    }
}
```

Refs #122

- Utility for writing test database fixtures to a .db file

  `python tests/fixtures.py /tmp/hello.db`

  This is useful for making a SQLite database of the test fixtures for interactive exploration.

- Compound primary key `_next=` now plays well with extra filters

  Closes #190

- Fixed bug with keyset pagination over compound primary keys

  Refs #190

- Database/Table views inherit `source/license/source_url/license_url` metadata

  If you set the `source_url/license_url/source/license` fields in your root metadata those values will now be inherited all the way down to the database and table templates.

  The `title/description` are NOT inherited.

  Also added unit tests for the HTML generated by the metadata.

  Refs #185

- Add metadata, if it exists, to heroku temp dir (#178) [Tony Hirst]
- Initial documentation for pagination
- Broke up test_app into test_api and test_html
- Fixed bug with .json path regular expression

  I had a table called `geojson` and it caused an exception because the regex was matching `.json` and not `\.json`

- Deploy to Heroku with Python 3.6.3

1.19.24 0.14 (2017-12-09)

The theme of this release is customization: Datasette now allows every aspect of its presentation to be customized either using additional CSS or by providing entirely new templates.

Datasette's metadata.json format has also been expanded, to allow per-database and per-table metadata. A new `datasette skeleton` command can be used to generate a skeleton JSON file ready to be filled in with per-database and per-table details.

The `metadata.json` file can also be used to define canned queries, as a more powerful alternative to SQL views.

- `extra_css_urls`/`extra_js_urls` in metadata

  A mechanism in the `metadata.json` format for adding custom CSS and JS urls.

  Create a `metadata.json` file that looks like this:

```
{
    "extra_css_urls": [
        "https://simonwillison.net/static/css/all.bf8cd891642c.css"
    ],
    "extra_js_urls": [
        "https://code.jquery.com/jquery-3.2.1.slim.min.js"
    ]
}
```

  Then start datasette like this:

```
datasette mydb.db --metadata=metadata.json
```

  The CSS and JavaScript files will be linked in the `<head>` of every page.

  You can also specify a SRI (subresource integrity hash) for these assets:

```
{
    "extra_css_urls": [
        {
            "url": "https://simonwillison.net/static/css/all.bf8cd891642c.css",
            "sri": "sha384-9qIZekWUyjCyDIf2YK1FRoKiPJq4PHt6tp/
↪ulnuuyRBvazd0hG7pWbE99zvwSznI"
        }
    ],
    "extra_js_urls": [
        {
            "url": "https://code.jquery.com/jquery-3.2.1.slim.min.js",
            "sri": "sha256-k2WSCIexGzOj3Euiig+TlR8gA0EmPjuc79OEeY5L45g="
        }
    ]
}
```

  Modern browsers will only execute the stylesheet or JavaScript if the SRI hash matches the content served. You can generate hashes using https://www.srihash.org/

- Auto-link column values that look like URLs (#153)
- CSS styling hooks as classes on the body (#153)

  Every template now gets CSS classes in the body designed to support custom styling.

  The index template (the top level page at `/`) gets this:

```
<body class="index">
```

The database template (`/dbname/`) gets this:

```
<body class="db db-dbname">
```

The table template (`/dbname/tablename`) gets:

```
<body class="table db-dbname table-tablename">
```

The row template (`/dbname/tablename/rowid`) gets:

```
<body class="row db-dbname table-tablename">
```

The `db-x` and `table-x` classes use the database or table names themselves IF they are valid CSS identifiers. If they aren't, we strip any invalid characters out and append a 6 character md5 digest of the original name, in order to ensure that multiple tables which resolve to the same stripped character version still have different CSS classes.

Some examples (extracted from the unit tests):

```
"simple" => "simple"
"MixedCase" => "MixedCase"
"-no-leading-hyphens" => "no-leading-hyphens-65bea6"
"_no-leading-underscores" => "no-leading-underscores-b921bc"
"no spaces" => "no-spaces-7088d7"
"-" => "336d5e"
"no $ characters" => "no--characters-59e024"
```

- `datasette --template-dir=mytemplates/` argument

  You can now pass an additional argument specifying a directory to look for custom templates in.

  Datasette will fall back on the default templates if a template is not found in that directory.

- Ability to over-ride templates for individual tables/databases.

  It is now possible to over-ride templates on a per-database / per-row or per- table basis.

  When you access e.g. `/mydatabase/mytable` Datasette will look for the following:

```
- table-mydatabase-mytable.html
- table.html
```

  If you provided a `--template-dir` argument to datasette serve it will look in that directory first.

  The lookup rules are as follows:

```
Index page (/):
    index.html

Database page (/mydatabase):
    database-mydatabase.html
    database.html

Table page (/mydatabase/mytable):
    table-mydatabase-mytable.html
    table.html
```

(continues on next page)

(continued from previous page)

```
Row page (/mydatabase/mytable/id):
    row-mydatabase-mytable.html
    row.html
```

If a table name has spaces or other unexpected characters in it, the template filename will follow the same rules as our custom <body> CSS classes - for example, a table called "Food Trucks" will attempt to load the following templates:

```
table-mydatabase-Food-Trucks-399138.html
table.html
```

It is possible to extend the default templates using Jinja template inheritance. If you want to customize EVERY row template with some additional content you can do so by creating a row.html template like this:

```
{% extends "default:row.html" %}

{% block content %}
<h1>EXTRA HTML AT THE TOP OF THE CONTENT BLOCK</h1>
<p>This line renders the original block:</p>
{{ super() }}
{% endblock %}
```

- `--static` option for datasette serve (#160)

  You can now tell Datasette to serve static files from a specific location at a specific mountpoint.

  For example:

```
datasette serve mydb.db --static extra-css:/tmp/static/css
```

  Now if you visit this URL:

```
http://localhost:8001/extra-css/blah.css
```

  The following file will be served:

```
/tmp/static/css/blah.css
```

- Canned query support.

  Named canned queries can now be defined in `metadata.json` like this:

```
{
    "databases": {
        "timezones": {
            "queries": {
                "timezone_for_point": "select tzid from timezones ..."
            }
        }
    }
}
```

  These will be shown in a new "Queries" section beneath "Views" on the database page.

- New `datasette skeleton` command for generating `metadata.json` (#164)
- `metadata.json` support for per-table/per-database metadata (#165)

  Also added support for descriptions and HTML descriptions.

Here's an example metadata.json file illustrating custom per-database and per- table metadata:

```
{
    "title": "Overall datasette title",
    "description_html": "This is a <em>description with HTML</em>.",
    "databases": {
        "db1": {
            "title": "First database",
            "description": "This is a string description & has no HTML",
            "license_url": "http://example.com/",
        "license": "The example license",
            "queries": {
              "canned_query": "select * from table1 limit 3;"
            },
            "tables": {
                "table1": {
                    "title": "Custom title for table1",
                    "description": "Tables can have descriptions too",
                    "source": "This has a custom source",
                    "source_url": "http://example.com/"
                }
            }
        }
    }
}
```

- Renamed `datasette build` command to `datasette inspect` (#130)
- Upgrade to Sanic 0.7.0 (#168)

  https://github.com/channelcat/sanic/releases/tag/0.7.0

- Package and publish commands now accept `--static` and `--template-dir`

  Example usage:

```
datasette package --static css:extra-css/ --static js:extra-js/ \
  sf-trees.db --template-dir templates/ --tag sf-trees --branch master
```

  This creates a local Docker image that includes copies of the templates/, extra-css/ and extra-js/ directories. You can then run it like this:

```
docker run -p 8001:8001 sf-trees
```

  For publishing to Zeit now:

```
datasette publish now --static css:extra-css/ --static js:extra-js/ \
  sf-trees.db --template-dir templates/ --name sf-trees --branch master
```

- HTML comment showing which templates were considered for a page (#171)

1.19.25 0.13 (2017-11-24)

- Search now applies to current filters.

  Combined search into the same form as filters.

  Closes #133

- Much tidier design for table view header.

  Closes #147

- Added `?column__not=blah` filter.

  Closes #148

- Row page now resolves foreign keys.

  Closes #132

- Further tweaks to select/input filter styling.

  Refs #86 - thanks for the help, @natbat!

- Show linked foreign key in table cells.

- Added UI for editing table filters.

  Refs #86

- Hide FTS-created tables on index pages.

  Closes #129

- Add publish to heroku support [Jacob Kaplan-Moss]

  ```
  datasette publish heroku mydb.db
  ```

  Pull request #104

- Initial implementation of `?_group_count=column`.

  URL shortcut for counting rows grouped by one or more columns.

  `?_group_count=column1&_group_count=column2` works as well.

  SQL generated looks like this:

  ```
  select "qSpecies", count(*) as "count"
  from Street_Tree_List
  group by "qSpecies"
  order by "count" desc limit 100
  ```

  Or for two columns like this:

  ```
  select "qSpecies", "qSiteInfo", count(*) as "count"
  from Street_Tree_List
  group by "qSpecies", "qSiteInfo"
  order by "count" desc limit 100
  ```

  Refs #44

- Added `--build=master` option to datasette publish and package.

  The `datasette publish` and `datasette package` commands both now accept an optional `--build` argument. If provided, this can be used to specify a branch published to GitHub that should be built into the container.

  This makes it easier to test code that has not yet been officially released to PyPI, e.g.:

  ```
  datasette publish now mydb.db --branch=master
  ```

- Implemented `?_search=XXX` + UI if a FTS table is detected.

  Closes #131

- Added `datasette --version` support.
- Table views now show expanded foreign key references, if possible.

  If a table has foreign key columns, and those foreign key tables have `label_columns`, the TableView will now query those other tables for the corresponding values and display those values as links in the corresponding table cells.

  label_columns are currently detected by the `inspect()` function, which looks for any table that has just two columns - an ID column and one other - and sets the `label_column` to be that second non-ID column.
- Don't prevent tabbing to "Run SQL" button (#117) [Robert Gieseke]

  See comment in #115
- Add keyboard shortcut to execute SQL query (#115) [Robert Gieseke]
- Allow `--load-extension` to be set via environment variable.
- Add support for `?field__isnull=1` (#107) [Ray N]
- Add spatialite, switch to debian and local build (#114) [Ariel Núñez]
- Added `--load-extension` argument to datasette serve.

  Allows loading of SQLite extensions. Refs #110.

1.19.26 0.12 (2017-11-16)

- Added `__version__`, now displayed as tooltip in page footer (#108).
- Added initial docs, including a changelog (#99).
- Turned on auto-escaping in Jinja.
- Added a UI for editing named parameters (#96).

  You can now construct a custom SQL statement using SQLite named parameters (e.g. `:name`) and datasette will display form fields for editing those parameters. Here's an example which lets you see the most popular names for dogs of different species registered through various dog registration schemes in Australia.
- Pin to specific Jinja version. (#100).
- Default to 127.0.0.1 not 0.0.0.0. (#98).
- Added extra metadata options to publish and package commands. (#92).

  You can now run these commands like so:

```
datasette now publish mydb.db \
    --title="My Title" \
    --source="Source" \
    --source_url="http://www.example.com/" \
    --license="CC0" \
    --license_url="https://creativecommons.org/publicdomain/zero/1.0/"
```

  This will write those values into the metadata.json that is packaged with the app. If you also pass `--metadata=metadata.json` that file will be updated with the extra values before being written into the Docker image.
- Added simple production-ready Dockerfile (#94) [Andrew Cutler]
- New `?_sql_time_limit_ms=10` argument to database and table page (#95)
- SQL syntax highlighting with Codemirror (#89) [Tom Dyson]

1.19.27 0.11 (2017-11-14)

- Added `datasette publish now --force` option.

  This calls `now` with `--force` - useful as it means you get a fresh copy of datasette even if Now has already cached that docker layer.

- Enable `--cors` by default when running in a container.

1.19.28 0.10 (2017-11-14)

- Fixed #83 - 500 error on individual row pages.
- Stop using sqlite WITH RECURSIVE in our tests.

  The version of Python 3 running in Travis CI doesn't support this.

1.19.29 0.9 (2017-11-13)

- Added `--sql_time_limit_ms` and `--extra-options`.

  The serve command now accepts `--sql_time_limit_ms` for customizing the SQL time limit.

  The publish and package commands now accept `--extra-options` which can be used to specify additional options to be passed to the datasite serve command when it executes inside the resulting Docker containers.

1.19.30 0.8 (2017-11-13)

- V0.8 - added PyPI metadata, ready to ship.
- Implemented offset/limit pagination for views (#70).
- Improved pagination. (#78)
- Limit on max rows returned, controlled by `--max_returned_rows` option. (#69)

  If someone executes 'select * from table' against a table with a million rows in it, we could run into problems: just serializing that much data as JSON is likely to lock up the server.

  Solution: we now have a hard limit on the maximum number of rows that can be returned by a query. If that limit is exceeded, the server will return a `"truncated": true` field in the JSON.

  This limit can be optionally controlled by the new `--max_returned_rows` option. Setting that option to 0 disables the limit entirely.