Datasette Documentation

Simon Willison

Jul 08, 2018

Contents

1 Contents **3**

1.1 Installation . . . 3
1.2 Getting started . . . 4
1.3 JSON API . . . 6
1.4 Running SQL queries . . . 10
1.5 CSV Export . . . 12
1.6 Facets . . . 13
1.7 Full-text search . . . 16
1.8 SpatiaLite . . . 19
1.9 Metadata . . . 23
1.10 Configuration . . . 27
1.11 Introspection . . . 30
1.12 Customization . . . 32
1.13 Plugins . . . 35
1.14 Changelog . . . 39

An instant JSON API for your SQLite databases

Datasette provides an instant, read-only JSON API for any SQLite database. It also provides tools for packaging the database up as a Docker container and deploying that container to hosting providers such as Zeit Now or Heroku.

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

CHAPTER 1

Contents

1.1 Installation

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

1.1.1 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

1.1.2 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
```

1.2 Getting started

1.2.1 Basic usage

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

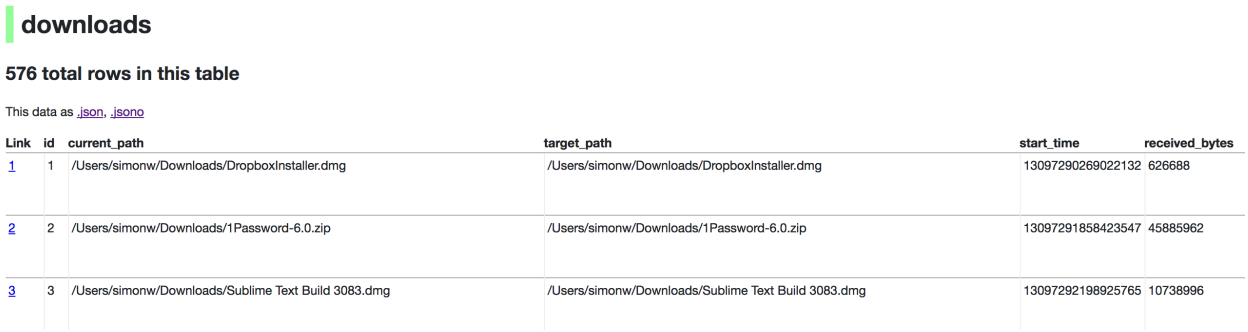
This will start a web server on port 8001 - visit http://localhost:8001/ to access the web interface.

`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",
        ...
    ],
    "table_rows_count": 576,
    "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": "",
```

(continues on next page)

(continued from previous page)

```
            "referrer": "https://www.dropbox.com/downloading?src=index",
            ...
        }
    ]
}
```

1.2.2 datasette serve options

```
$ datasette serve --help
Usage: datasette serve [OPTIONS] [FILES]...

  Serve up specified SQLite database files with a web UI

Options:
  -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 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
  --config CONFIG               Set config option using configname:value
                                datasette.readthedocs.io/en/latest/config.html
  --help-config                 Show available config options
  --help                        Show this message and exit.
```

1.3 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.

1.3.1 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` - the entire response is an array of objects
- `?_shape=arrayfirst` - the entire response is a flat JSON array containing just the first value from each row
- `?_shape=object` - the entire response is a JSON object keyed using the primary keys of the rows

`objects` looks like this:

```
{
    "database": "sf-trees",
    ...
    "rows": [
        {
            "id": 1,
            "value": "Myoporum laetum :: Myoporum"
        },
        {
            "id": 2,
```

(continues on next page)

(continued from previous page)

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

`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"
    }
]
```

`arrayfirst` looks like this:

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

`object` looks like this:

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

1.3.2 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.

?_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

?_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.

1.3.3 Special table arguments

The Datasette table view takes a number of special querystring 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.

?_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.

1.3.4 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*.

1.4 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 cgustom SQL editor.

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.

1.4.1 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.

1.4.2 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>
```

1.4.3 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": "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
```

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": "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;"
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

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.

1.4.4 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.

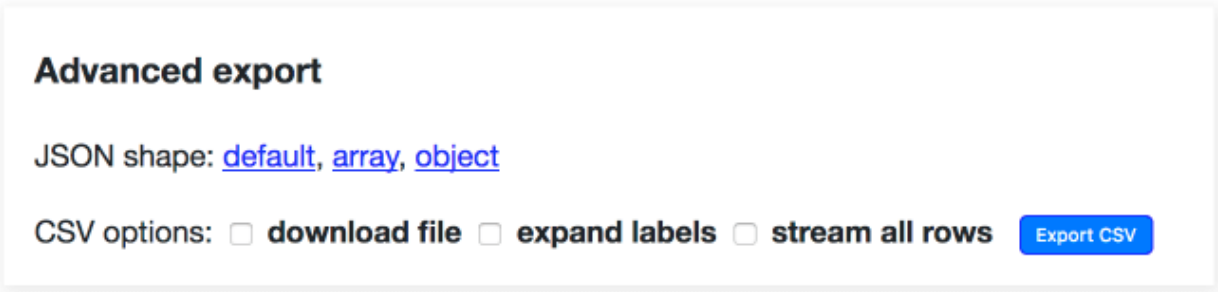
Since the where clause acts against the index on the primary key, the query is extremely fast even for records that are a long way into the overall pagination set.

1.5 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

1.5.1 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*.

1.5.2 A note on URLs

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

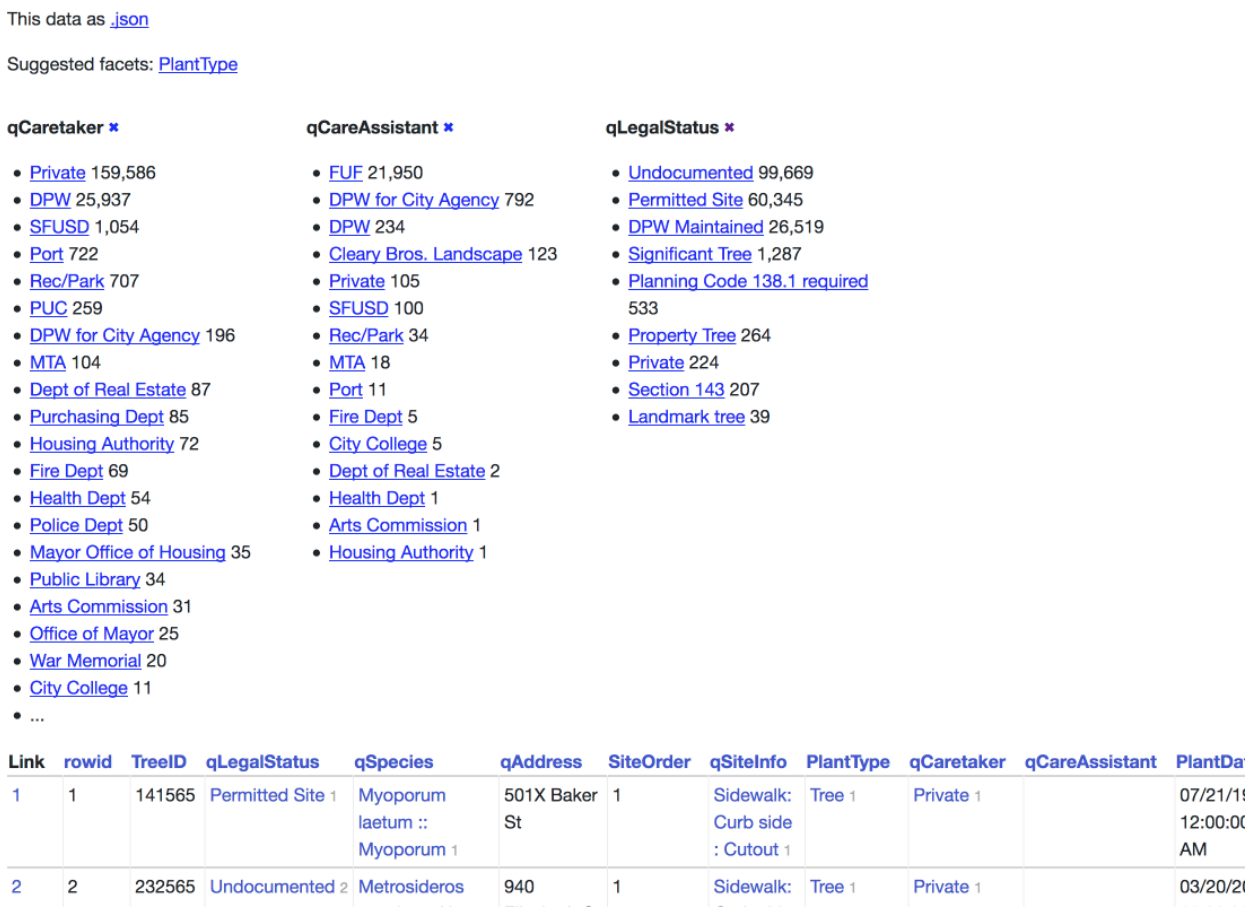
- https://latest.datasette.io/fixtures/facetable - HTML interface
- https://latest.datasette.io/fixtures/facetable.csv - CSV export
- https://latest.datasette.io/fixtures/facetable.json - JSON API

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

1.6 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.

1.6.1 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.

1.6.2 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.

1.6.3 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.

1.6.4 Speeding up facets with indexes

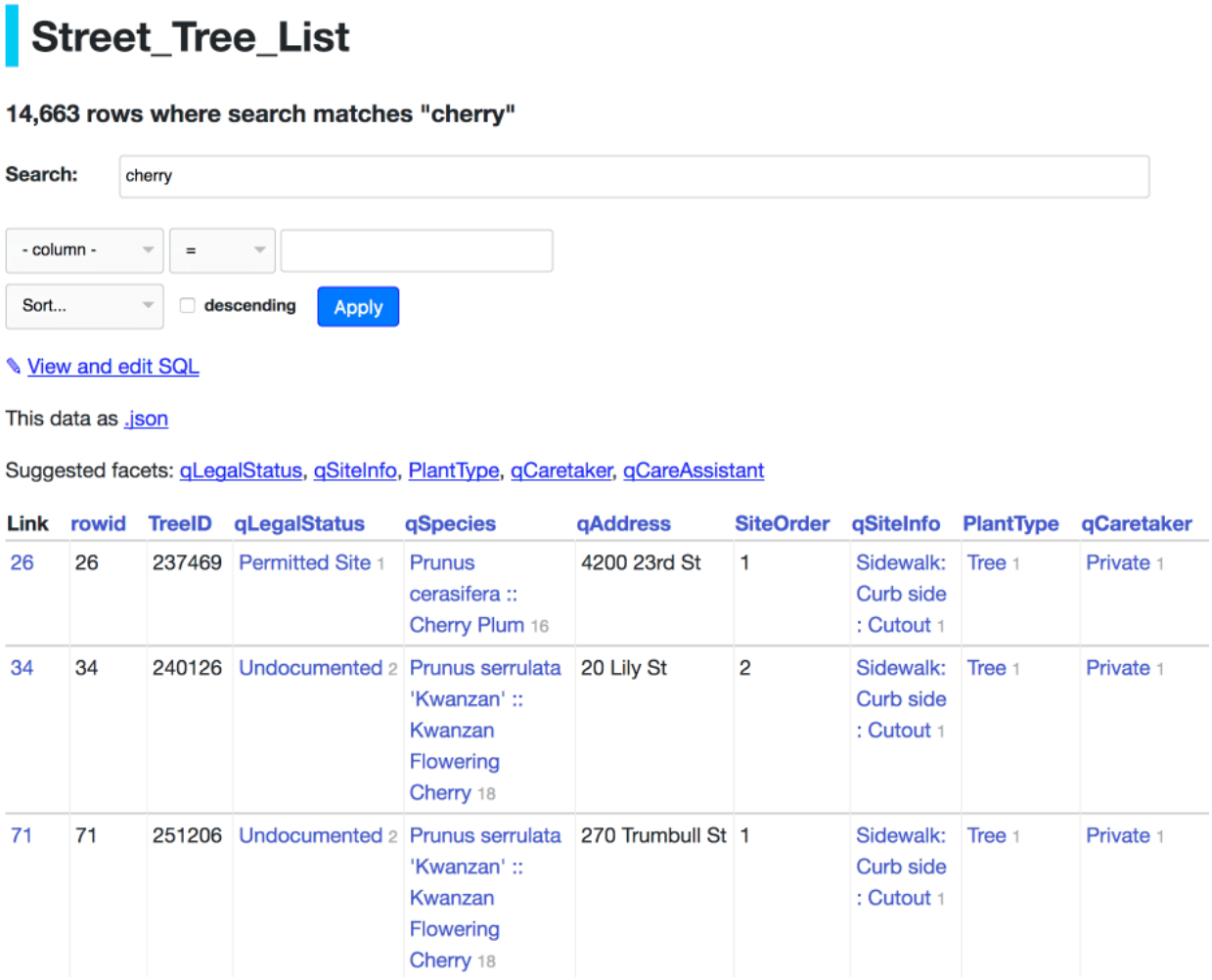
The performance of facets can be greatly improved by adding indexes on the columns you wish to facet by. Adding indexes can be performed using the `sqlite3` command-line utility. Here's how to add an index on the `state` column in a table called `Food_Trucks`:

```
$ 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");
```

1.7 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.



Street_Tree_List

14,663 rows where search matches "cherry"

Search: cherry

- column - = Sort... descending Apply

View and edit SQL

This data as .json

Suggested facets: qLegalStatus, qSiteInfo, PlantType, qCaretaker, qCareAssistant

Link	rowid	TreeID	qLegalStatus	qSpecies	qAddress	SiteOrder	qSiteInfo	PlantType	qCaretaker
26	26	237469	Permitted Site 1	Prunus cerasifera :: Cherry Plum 16	4200 23rd St	1	Sidewalk: Curb side : Cutout 1	Tree 1	Private 1
34	34	240126	Undocumented 2	Prunus serrulata 'Kwanzan' :: Kwanzan Flowering Cherry 18	20 Lily St	2	Sidewalk: Curb side : Cutout 1	Tree 1	Private 1
71	71	251206	Undocumented 2	Prunus serrulata 'Kwanzan' :: Kwanzan Flowering Cherry 18	270 Trumbull St	1	Sidewalk: Curb side : Cutout 1	Tree 1	Private 1

Datasette detects which tables have been configured for full-text search when it first inspects the database on startup (or via the `datasette inspect` command). You can visit the `/-/inspect` page on your Datasette instance to see the results of this inspection. Tables that have been configured for full-text search will have their `fts_table` property set to the name of another table (tables without full-text search will have this property set to `null`).

1.7.1 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 is usually not available is the SQLite version that is bundled with Python. If in doubt, you should use FTS4.

1.7.2 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

To enable full-text search for a table called `items` that works against the `name` and `description` columns, you would run the following 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.

1.7.3 Setting up full-text search 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
```

1.7.4 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
```

1.7.5 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
```

1.8 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.

1.8.1 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.

1.8.2 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);
''')
# If you don't commit your changes will not be persisted:
conn.commit()
conn.close()
```

1.8.3 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.

1.8.4 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;
```

1.8.5 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()
```

(continues on next page)

(continued from previous page)

```
# 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()
```

1.8.6 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
    );
```

1.9 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",
```

(continues on next page)

(continued from previous page)

```
    "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.

1.9.1 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.

1.9.2 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"
                    }
                }
            }
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

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"
    ]
}
```

1.9.3 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"
                    ]
                }
            }
        }
    }
}
```

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

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

1.9.4 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"
                }
            }
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

1.9.5 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
                }
            }
        }
    }
}
```

1.9.6 Generating a metadata skeleton

Tracking down the names of all of your databases and tables and formatting them as JSON can be a little tedious, so Datasette provides a tool to help you generate a "skeleton" JSON file:

```
datasette skeleton database1.db database2.db
```

This will create a `metadata.json` file looking something like this:

```
{
    "title": null,
    "description": null,
    "description_html": null,
    "license": null,
    "license_url": null,
    "source": null,
    "source_url": null,
    "databases": {
        "database1": {
            "title": null,
            "description": null,
            "description_html": null,
            "license": null,
            "license_url": null,
            "source": null,
            "source_url": null,
            "queries": {},
            "tables": {
                "example_table": {
                    "title": null,
                    "description": null,
                    "description_html": null,
```

(continues on next page)

(continued from previous page)

```
                    "license": null,
                    "license_url": null,
                    "source": null,
                    "source_url": null,
                    "units": {}
                }
            }
        },
        "database2": ...
    }
}
```

You can replace any of the `null` values with a JSON string to populate that piece of metadata.

1.10 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.

1.10.1 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
```

1.10.2 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` query string 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.

1.10.3 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
```

1.10.4 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
```

1.10.5 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
```

1.10.6 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
```

1.10.7 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
```

1.10.8 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
```

1.10.9 suggest_facets

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

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

1.10.10 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
```

1.10.11 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
```

1.10.12 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 365 days (31536000 seconds).

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

1.10.13 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
```

1.10.14 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
```

1.10.15 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
```

1.11 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.

1.11.1 /-/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": {...}
}
```

1.11.2 /-/inspect

Shows the result of running `datasette inspect` on the currently loaded databases. This is run automatically when Datasette starts up, or can be run as a separate step and passed to `datasette serve --inspect-file`.

This is an internal implementation detail of Datasette and the format should not be considered stable - it is likely to change in undocumented ways between different releases.

Inspect example:

```
{
    "fivethirtyeight": {
        "file": "fivethirtyeight.db",
        "hash": "5de27e3eceb3f5ba817e0b2e066cea77832592b62d94690b5102a48f385b95fb",
        "tables": {
            "./index": {
                "columns": [
                    "dataset_url",
                    "article_url",
                    "live"
                ],
                "count": 125,
```

(continues on next page)

(continued from previous page)

```
                "foreign_keys": {
                    "incoming": [],
                    "outgoing": []
                },
                "fts_table": null,
                "hidden": false,
                "label_column": null,
                "name": "./index",
                "primary_keys": []
            },
            ...
```

1.11.3 /-/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": [
            "FTS4",
            "FTS3"
        ],
        "version": "3.16.2"
    }
}
```

1.11.4 /-/plugins

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

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

1.11.5 /-/config

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

```
{
    "default_facet_size": 30,
    "default_page_size": 100,
    "facet_suggest_time_limit_ms": 50,
    "facet_time_limit_ms": 1000,
    "max_returned_rows": 1000,
    "sql_time_limit_ms": 1000
}
```

1.12 Customization

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

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

1.12.2 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

Rows and columns include on table page:
    _rows_and_columns-table-mydatabase-mytable.html
    _rows_and_columns-mydatabase-mytable.html
    _rows_and_columns.html

Rows and columns include on row page:
    _rows_and_columns-row-mydatabase-mytable.html
    _rows_and_columns-mydatabase-mytable.html
    _rows_and_columns.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 `_rows_and_columns.html` template is included on both the row and the table page, and displays the content of the row. The default `_rows_and_columns.html` template 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.

Say for example you want to output a certain column as unescaped HTML. You could provide a custom `_rows_and_columns.html` template like this:

```
<table>
    <thead>
        <tr>
            {% for column in display_columns %}
                <th scope="col">{{ column }}</th>
            {% endfor %}
        </tr>
    </thead>
    <tbody>
    {% for row in display_rows %}
        <tr>
            {% for cell in row %}
                <td>
                    {% if cell.column == 'description' %}
                        {{ cell.value|safe }}
                    {% else %}
                        {{ cell.value }}
                    {% endif %}
                </td>
            {% endfor %}
        </tr>
    {% endfor %}
    </tbody>
</table>
```

1.13 Plugins

Datasette's plugin system is currently under active development. It 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.

You can follow the development of plugins in issue #14.

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

1.13.2 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.

1.13.3 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',
```

(continues on next page)

(continued from previous page)

```
    license='Apache License, Version 2.0',
    version=VERSION,
    py_modules=['datasette_plugin_demos'],
    entry_points={
        '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.

1.13.4 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.

1.13.5 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.

1.13.6 Plugin hooks

Datasette will eventually have many more plugin hooks. You can track and contribute to their development in issue #14.

prepare_connection(conn)

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)

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()

Return a list of extra CSS URLs that should be included on every 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()

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

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

1.14 Changelog

1.14.1 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

1.14.2 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

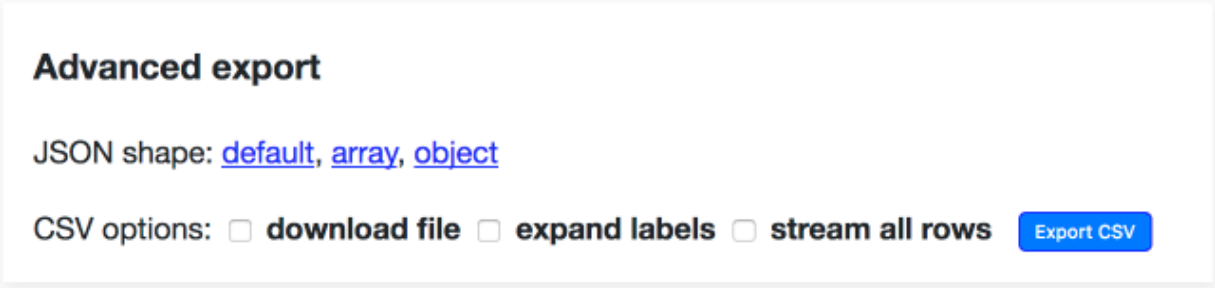
1.14.3 0.23 (2018-06-18)

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.

1.14.4 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]

1.14.5 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)
```

(continues on next page)

(continued from previous page)

```
  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

1.14.6 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,
      "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

1.14.7 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

1.14.8 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.

1.14.9 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

1.14.10 0.17 (2018-04-13)

- Release 0.17 to fix issues with PyPI

1.14.11 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.

1.14.12 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

1.14.13 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

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)

1.14.14 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.

1.14.15 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]

1.14.16 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.

1.14.17 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.

1.14.18 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.

1.14.19 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.