# Starbelly Documentation

***Release 1.1.0***

**Mark E. Haase**

**May 29, 2018**

# Contents

CHAPTER 1

# Introduction

Starbelly is a user-friendly and highly configurable web crawler front end. Compared to other crawling systems, such as Nutch or Scrapy, Starbelly trades off lower scalability for improved usability. Starbelly eschews the arcane configuration files and custom code required for other crawling systems, favoring a GUI for configuration and managment. Starbelly exposes all of its features and data through an efficient API, allowing you to build crawling-based systems on top of it. For example, you might plug in an Elastic Search backend to build a custom search engine, or plug in a scraper to create a data collection pipeline.

CHAPTER 2

# Installation Guide

**Contents**



## 2.1 Production Installation

### 2.1.1 Prerequisites

Starbelly is offered as a collection of Docker images and Docker-compose configurations. If you're not familiar with Docker, it is a system for deploying multiple software components into individual containers and orchestrating the entire system.

First, install Docker using the instructions for your platform.

Next, install Docker Compose using the instructions for your platform.

### 2.1.2 Docker Compose

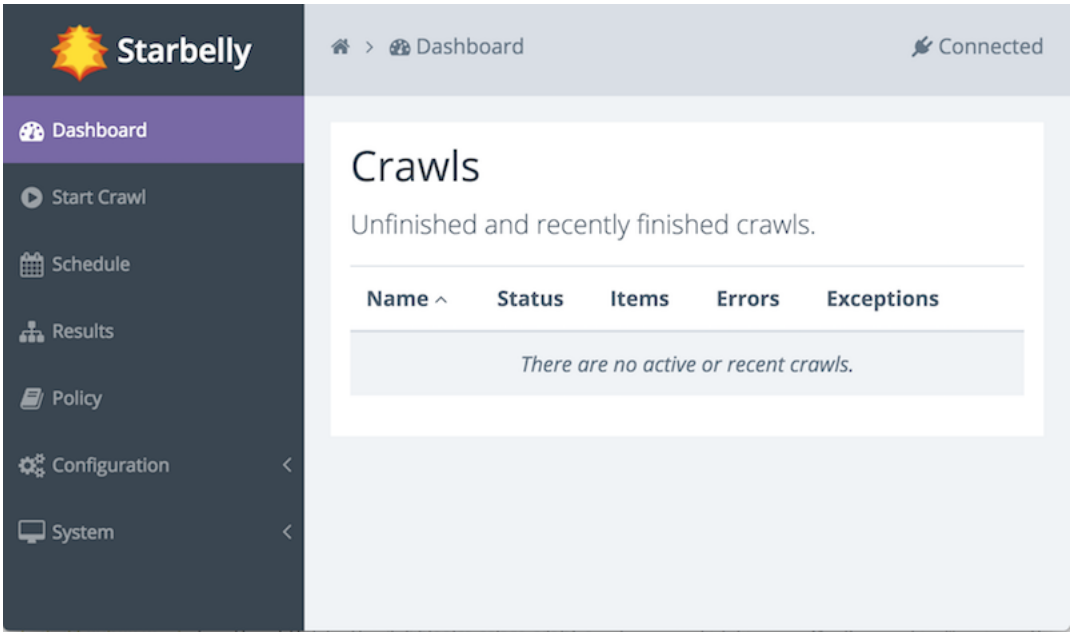
Docker Compose is used to set up and run multiple Docker containers together. You should have installed Docker Compose in the previous step. Now you need a Docker Compose configuration file (usually called `docker-compose.yml`) that specifies what containers need to be created and how they should be configured. A sample docker-compose.yml configuration file is available for Starbelly, but you may need to tailor this file to your unique environment.

Download this zip file and extract it. (If you have Git installed, you can run `git clone git@github.com:HyperionGray/starbelly-docker.git` instead.) From the `starbelly-docker/starbelly` directory, run the following command:

```
$ docker-compose up -d
```

This will download the required Docker images, create the corresponding containers, and then start the entire application on ports 80 and 443. Once the application has started, open up a browser and try navigating to the host where you are running Starbelly. The default username and password is "admin".

You should see the Dashboard:




If you experience any problems, try using the command `docker-compose logs` to view logging output from the Docker containers.

### 2.1.3 Security

If your Starbelly instance is exposed to the internet, then you should immediately do two things to secure it:

1. Change the admin password.
2. Create TLS certificates

The **admin password** is stored in a file called `htpasswd` and it can be created or edited using the `htpasswd` command from the Apache2 utilities package (called `apache2-utils` on Ubuntu distributions). Install that package and then run this command:

```
$ htpasswd -c passwd admin
New password:
Re-type new password:
Adding password for user admin
```

Type in the new password when prompted. You can change passwords or add additional passwords to an existing file by running `htpasswd passwd USER`, where `USER` is the username to change. When you are done, copy the `passwd` file into the Docker container and remove the original.

```
$ docker cp passwd starbelly-web:/etc/nginx/tls/
$ docker exec starbelly-web nginx -s reload
2017/11/02 14:29:37 [notice] 1437#1437: signal process started
$ rm passwd
```

The default **TLS certificate** is automatically generated and self-signed when the container is created. If you have a valid domain name for your Starbelly server, then you should obtain a real certificate for it.

If you have obtained your own certificates, you can install them as follows, where `certificate.pem` is the full certificate chain in PEM format and `privatekey.pem` is the private key in PEM format.

```
$ docker cp certificate.pem starbelly-web:/etc/nginx/tls/server.crt
$ docker cp privatekey.pem starbelly-web:/etc/nginx/tls/server.key
$ docker exec starbelly-web nginx -s reload
2017/11/02 14:29:37 [notice] 1437#1437: signal process started
```

If you do not already have TLS certificates, you may obtain free certificates from Let's Encrypt. First, install the certbot application using the instructions for your platform. Now run certbot to create initial certificates, replacing `YOUR_DOMAIN` with the fully qualified domain name of the server and replacing `/path/to/starbelly-docker/starbelly` with the path where you placed the Starbelly docker configuration files from an earlier step.

```
$ cd /path/to/starbelly-docker/starbelly
$ certbot certonly \
    --webroot -w certbot-webroot \
    -d YOUR_DOMAIN \
    --deploy-hook ./deploy-certbot.py
Plugins selected: Authenticator webroot, Installer None
Enter email address (used for urgent renewal and security notices) (Enter 'c' to
cancel): YOUR EMAIL HERE

-------------------------------------------------------------------------------
Please read the Terms of Service at
https://letsencrypt.org/documents/LE-SA-v1.1.1-August-1-2016.pdf. You must agree
in order to register with the ACME server at
https://acme-staging.api.letsencrypt.org/directory
-------------------------------------------------------------------------------
(A)gree/(C)ancel: a

...snip...
```

This command will attempt to get TLS certificates from the Let's Encrypt server. If you've never run certbot on this server before, it will prompt you to enter a contact e-mail address and agree to the terms of service.

If certificate generation succeeds, it will install those certificates into the Docker container. This certificate is valid for 90 days and will need to be renewed before it expires. Create a daily cron job containing the following command to ensure that the certificate will be renewed appropriately.

```
certbot renew --deploy-hook /path/to/starbelly-docker/starbelly/deploy-certbot.py
```

### 2.1.4 Next Steps

Now that you have Starbelly up and running, take a look at *Your First Crawl*.

## 2.2 Developer Installation

### 2.2.1 Prerequisites

Separate Docker images are provided for developers who wish to contribute code to Starbelly. The development environment contains a few useful changes, such as mounting code from you local machine into the Docker container, automatically restarting the application server when the code is modified, etc.

---

**Important:** You should make sure that you understand the production installation above before attempting the developer installation.

---

In addition to the prerequisites for the production installation, you also need to install git for the developer installation. Checkout the following repositories into the same parent directory. If you do not have a GitHub account, you can check out the repositories using HTTPS:

- git clone https://github.com/hyperiongray/starbelly.git
- git clone https://github.com/hyperiongray/starbelly-docker.git
- git clone https://github.com/hyperiongray/starbelly-protobuf.git
- git clone https://github.com/hyperiongray/starbelly-web-client.git

If you do have a GitHub account, you should check out the repositories using SSH instead:

- git clone git@github.com:HyperionGray/starbelly.git
- git clone git@github.com:HyperionGray/starbelly-docker.git
- git clone git@github.com:HyperionGray/starbelly-protobuf.git
- git clone git@github.com:HyperionGray/starbelly-web-client.git

Finally, you should install Google Dart SDK. Note that the Pub packaging tool included with Dart will install packages to `/var/cache/pub` by default on Linux. This location is mounted into one of the Docker containers so that Pub packages are visible inside the container.

### 2.2.2 Docker Images

Next, you need to build the developer images. These images contain some additional development and debugging tools that are not present in the production images. Run the following commands:

```
$ cd /path/to/starbelly-docker/starbelly-dev
$ docker build -t starbelly-dev-app app
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM hyperiongray/starbelly-app:0.0.1
 ---> 3e201e933024
Step 2/3 : RUN apt-get install -y vim
 ---> Running in e1eca494dd23
Reading package lists...
...snip...
$ docker build -t starbelly-dev-web web
Sending build context to Docker daemon  5.632kB
Step 1/7 : FROM hyperiongray/starbelly-web:0.0.1
 ---> 652fcfaca537
Step 2/7 : RUN cd /tmp &&     openssl req -x509 -newkey rsa:2048                 -
↪keyout server.key -out server.crt                 -days 365 -nodes -subj '/
↪CN=starbelly' &&     mv server.key server.crt /etc/nginx/tls
```

(continues on next page)

(continued from previous page)

```
 ---> Running in dc86f5e609bf
...snip...
```

Now you have built the developer images for Starbelly.

### 2.2.3 Docker Compose

After you have built the developer images for Starbelly, the next step is to use Docker Compose to start up the entire environment. Run the following commands.

```
$ cd /path/to/starbelly-docker/starbelly-dev
$ docker-compose up
Creating network "starbellydev_default" with the default driver
Creating volume "starbellydev_web_tls" with default driver
Creating volume "starbellydev_db_data" with default driver
Creating starbelly-dev-app ...
Creating starbelly-dev-web ...
Creating starbelly-dev-db ...
...snip...
```

You should now be able to open the Starbelly GUI by pointing a web browser at your development server.

---

**Important:** The developer Docker images run the Starbelly server in automatic reload mode. If you edit a source file and save it, the server will automatically restart in order to run your latest code. Note that only changes to Python code and some static assets (e.g. CSS) trigger a reload; changes to configuration files do not trigger a reload.

---

### 2.2.4 TCP Ports

The developer image has some additional TCP ports exposed from the Docker containers to your localhost.

- Port 8000: the Starbelly server (websocket)
- Port 8001: Jupyter notebook (for experiments)
- Port 8002: RethinkDB GUI
- Port 8003: Pub Development Server

### 2.2.5 Dartium

The Starbelly web GUI is written in Dart. If you followed the steps above, you should already have installed the Dart SDK. Standard web browsers do not possess the Dart virtual machine required to run this Dart client code. When you connect with one of these standard browsers, the server automatically compiles the Dart code to JavaScript code and serves it to you.

This process can be slow, and during development you can save a lot of time by using a special browser called Dartium that includes the Dart virtual machine. When you access Starbelly through Dartium, the server will send Dart code to the browser, skipping the slow compilation step. Download Dartium from the Dart downloads page.

You can unzip the Dartium archive anywhere you want. One suitable place is `/opt/dartium`. To run it, execute the following command: `/opt/dartium/chrome`.

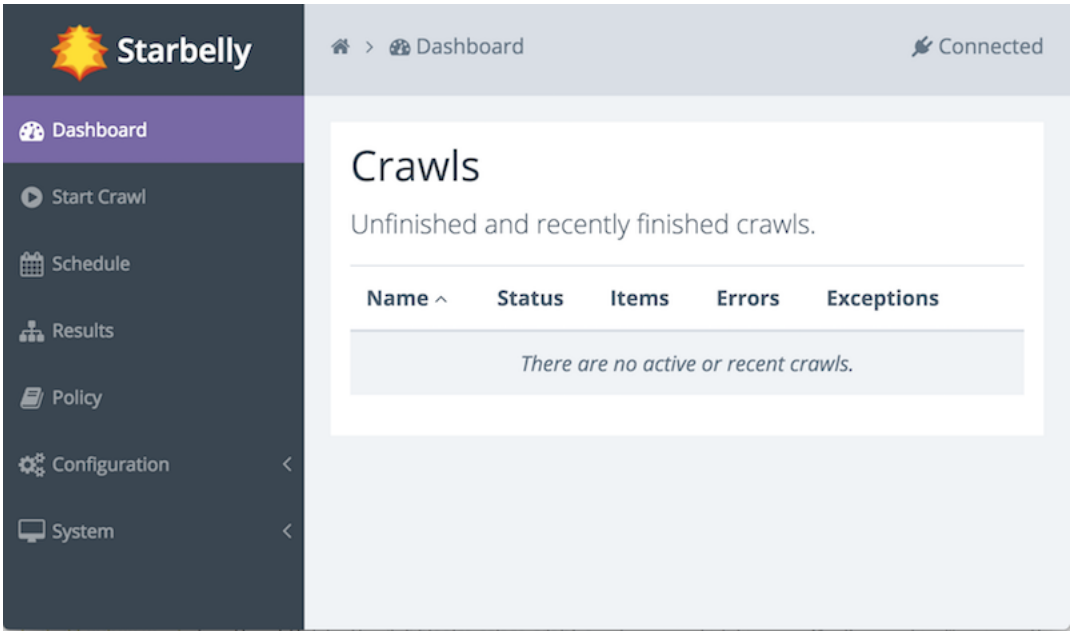
CHAPTER 3

# Your First Crawl

Starbelly offers a lot of ways to fine tune crawling, but for your first crawl, we will just use all of its default settings. The goal of this section is to perform a deep crawl of a news site. Here are a few example sites you may want to try:

- CNN
- Fox News
- New York Times
- Washington Post

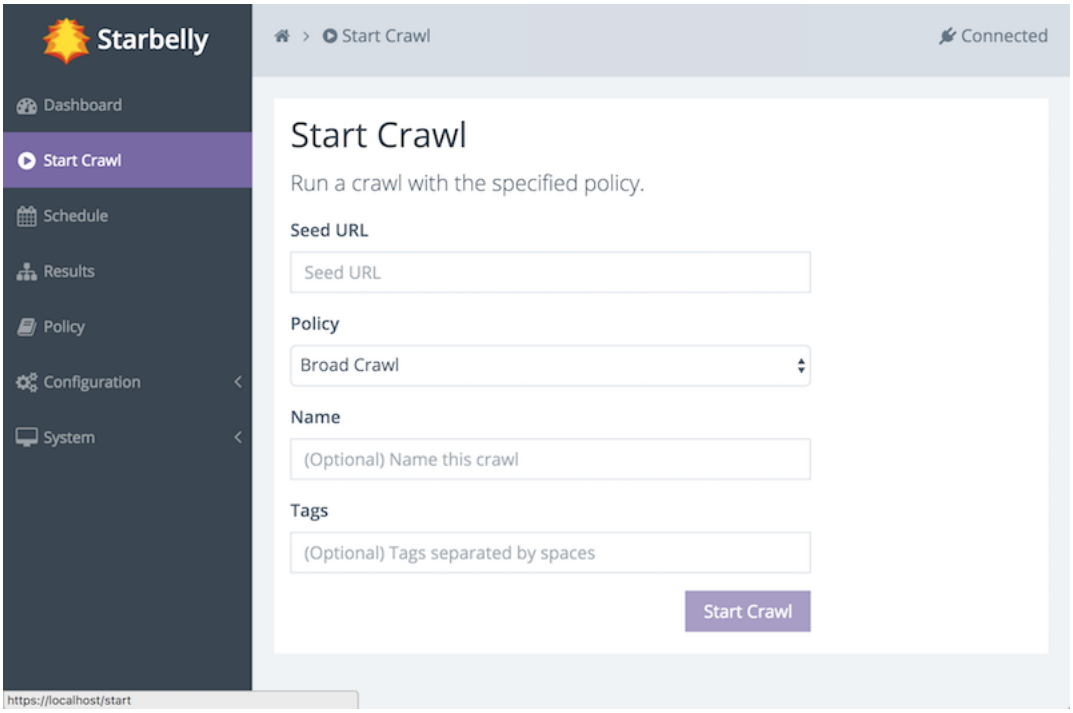
Begin by accessing the Starbelly GUI in your web browser. You should be able to see the dashboard:



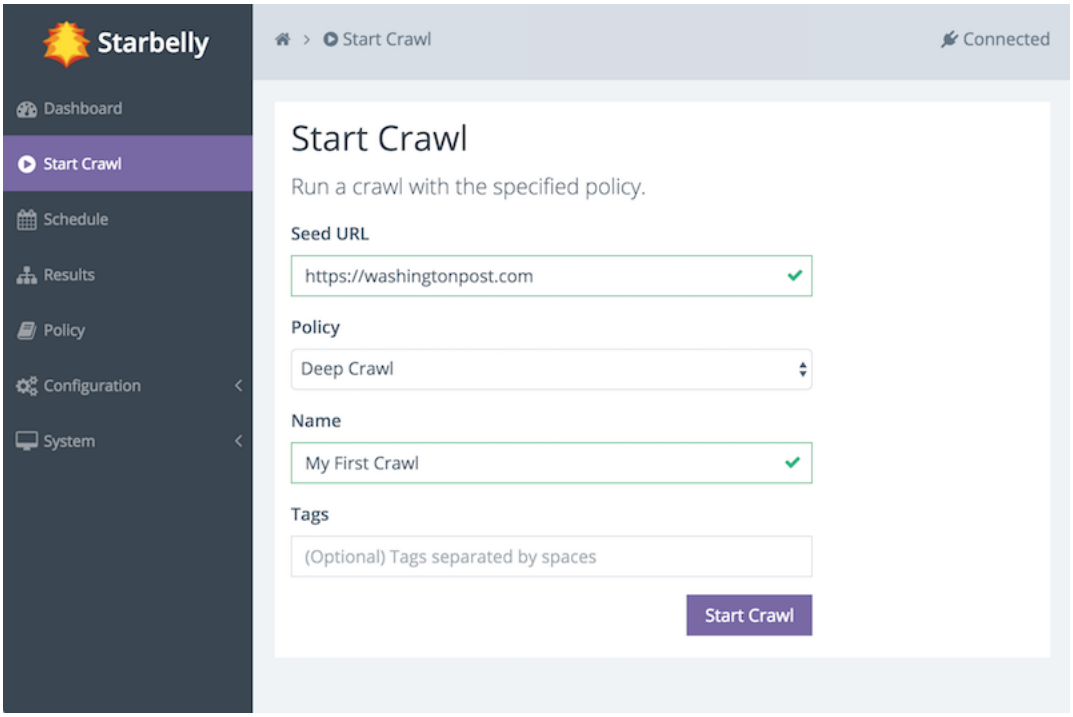

The dashboard is currently empty because you have not started any crawls yet, but let us take a moment to get familiar with the interface. On the left side, under the "Starbelly" logo, is the menu. Click any item in the menu to view it.

There is a gray bar across the top of the screen. The left side of this bar displays breadcrumbs, e.g. the *home* icon and the *dashboard* icon. The breadcrumbs help you keep track of where you are, and also allow you to quickly return to earlier screens. The right side of the bar shows the status "Connected", meaning that it is connected to the Starbelly server.

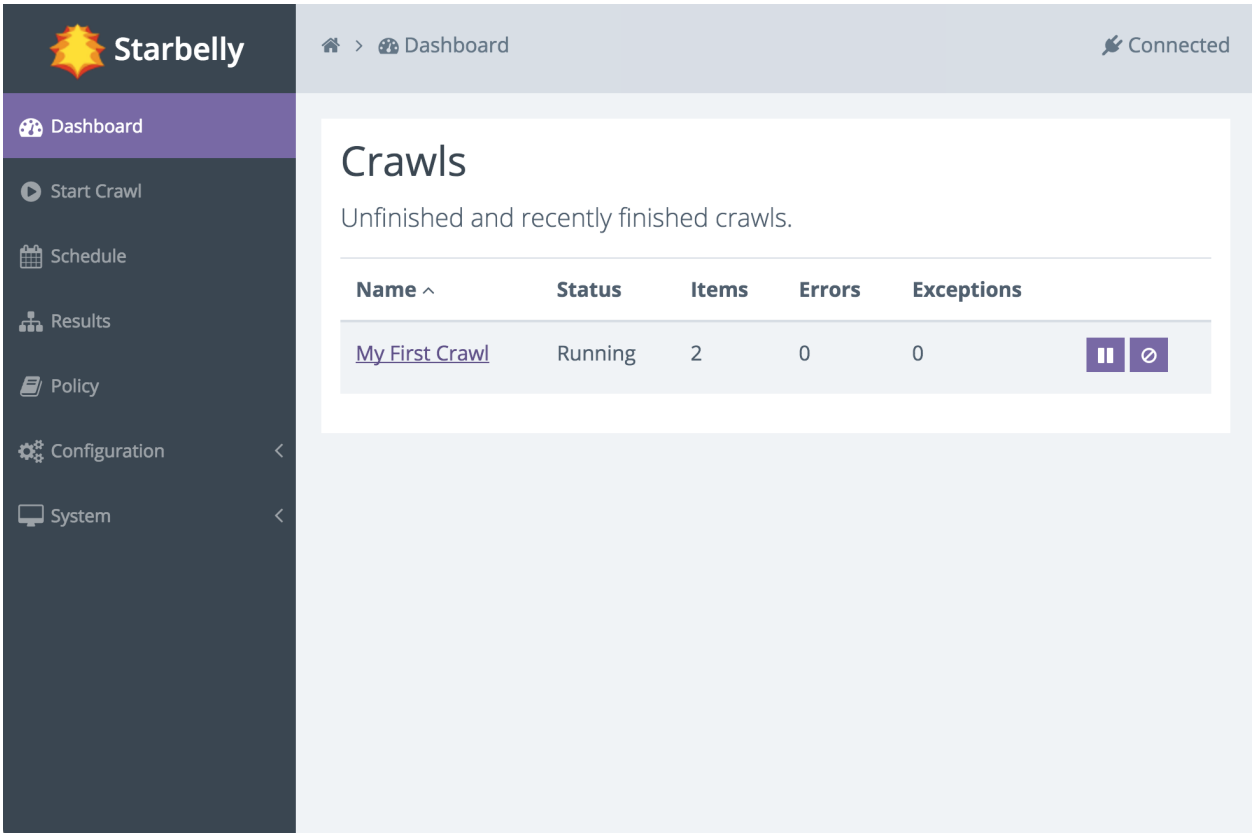
Next, click on *Start Crawl* in the menu.




This screen allows you to start a crawl. First, enter a seed URL. Second, select the *Deep Crawl* policy (more on that in a moment). Optionally, you may assign a name to the crawl. If you do not assign a name, then the crawler will choose a name for you. Finally, you may assign tags. Tags may be used by consumers of crawl data, but they do not have any effect on the crawl itself, so leave it blank for now. (You can edit the tags later if you wish.)

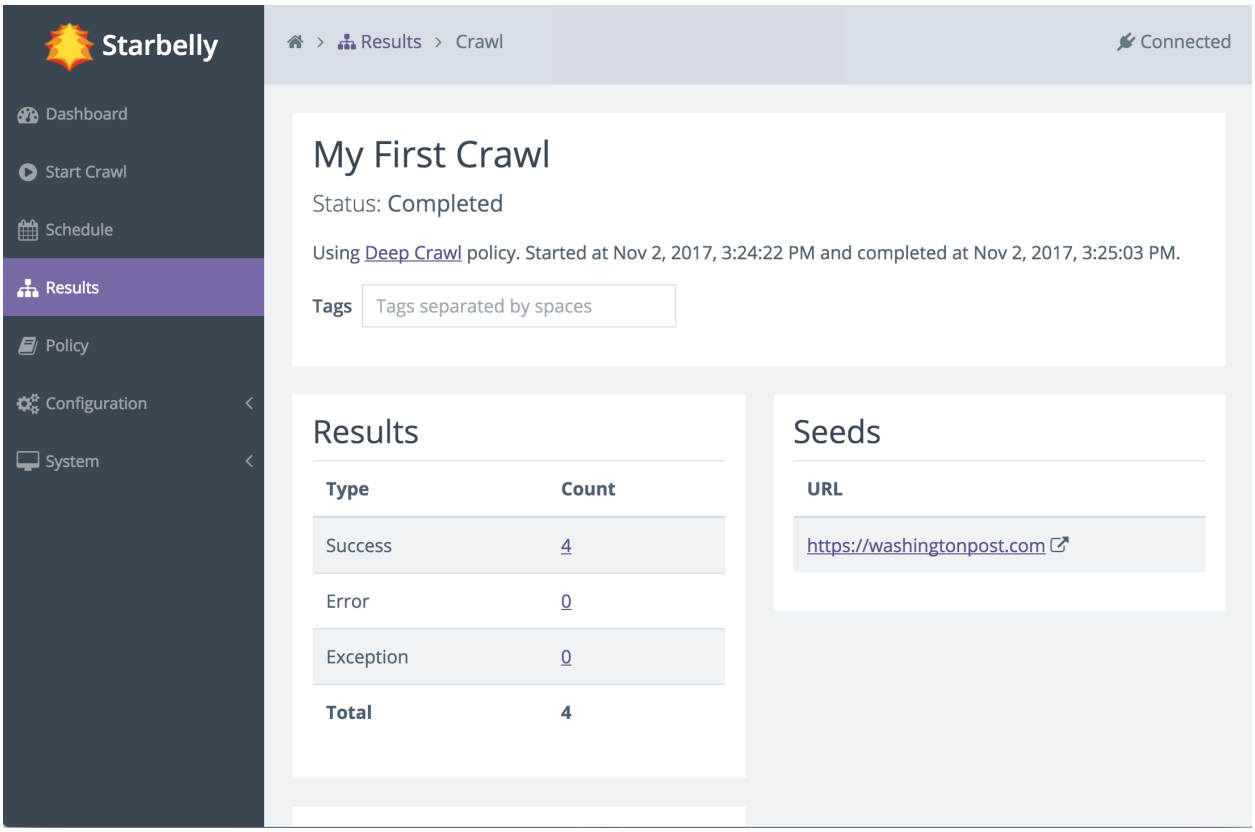



When you are ready, click the *Start Crawl* button. You should see a notification that the crawl is starting. Go back to the Dashboard and you should now be able to see that your crawl is running. The dashboard updates in realtime as the crawler downloads documents.




You can click on the name of the crawl to view details about that crawl job. The details screen also updates in real

time.




The crawl will continue to run, downloading various documents that it finds, until it reaches its end. But how does it decide what documents to download, and how does it decide when the crawl should end? These questions are answered by consulting the *crawl policy*, which guides the crawler's decision making. In this example, we used the default *Deep Crawl* policy, which stays inside the same domain as the seed URL and crawls to depth 10. You may customize this policy or create any number of your own policies to carefully refine crawler behavior.

Where does the crawl data go and what can you do with it? You can view crawl results inside Starbelly, but this feature is intended to help with debugging, not as a practical way to use crawl data. Crawl data is stored inside Starbelly's database until you explicitly delete it (see the *Crawl Results* screen). Starbelly is just a crawling *frontend*, which means that it is designed to be plugged into another application that can read the crawl data and do something useful with it, such as a search engine or a scraping program.

Now that you have a crawl under your belt, you might want to do a deeper dive into *Configuration Guide* and *Policy*.

CHAPTER 4

# Configuration Guide

**Contents**



## 4.1 Overview

Starbelly can be configured entirely through its graphical interface. In fact, this is one of the advantages to using Starbelly: no more arcane configuration files or custom code! The tradeoff, of course, is that Starbelly has fewer configuration options than other crawlers and may not be flexible enough to solve all crawling problems.

The configuration items are all contained in the *Configuration* submenu on the left side of the interface.

## 4.2 CAPTCHA Solvers

Starbelly has the ability to automatically log into a website if it has the appropriate credentials (see *Credentials* below). Some login forms may require a CAPTCHA. In those cases, you may configure a CAPTCHA solving service. Starbelly supports any CAPTCHA service that is compatible with the Antigate API. You may create multiple configurations in order to use multiple backend solvers or just to send different configurations to the same service.

Once you have created a CAPTCHA solver, specify that CAPTCHA solver in a crawl policy in order to send login CAPTCHAs to the solving service during crawls.

## 4.3 Credentials

Starbelly has the ability to automatically log into a website if it has the appropriate credentials. To configure credentials for a site, you only need to specify a login URL. (If the login URL enables single sign-on for multiple subdomains, then you should also specify the domain name that you wish to authenticate on.)

For each domain, you may set up multiple username & password credentials. When the crawler encounters that domain during a crawl, it will randomly pick one of the credentials and attempt to login with it. (The crawler uses machine learning to identify and parse the login form.)

## 4.4 Rate Limits

The crawler observes rate limits between subsequent requests to a single domain. For example, with the default delay of 5 seconds, the crawler will wait 5 seconds after a request completes until it initiates another request to that same domain. Therfore, the crawler will download at most 12 pages per minute from a single domain using the default rate limit. In practice, it will download fewer than 12 pages per minute, since each request itself also takes some non-negligible amount of time.

Furthermore, rate limits apply across all jobs. For example, if you have two different jobs crawling one domain, each job will effectively be limited to 6 pages per minute instead of 12.

On the *Rate Limits* configuration screen, you may change the global limit as well as customize rate limits for specific domains. This allows you to specify lower rate limits for domains that can handle higher traffic. For example, you might crawl web servers on your corporate intranet faster than you crawl a public internet server.

CHAPTER 5

# Policy

**Contents**



## 5.1 Overview

The *crawl policy* is one of the most important and powerful concepts in Starbelly. A policy controls the crawler's behavior and decision making, guiding which links the crawler follows, what kinds of resources it downloads, and how long or how far it runs. When you start a crawl job, you must specify which policy that job should use.

In this part of the documentation, we take a look at the features of the crawl policy. To begin, click *Policy* in the Starbelly menu, then click on an existing policy to view it, or click *New Policy* to create a new policy.

## 5.2 Authentication

The authentication policy determines how a crawler can authenticate itself to a web site. When the crawler sees a domain in a crawl for the first time, it checks to see if it has any credentials for that domain. (See the configuration of Credentials for more information.) If it does, it picks one of the appropriate credentials at random and tries to login with it. Some login forms may require a CAPTCHA. In those cases, you may configure a CAPTCHA solver and specify that solver in the policy.

## 5.3 Robots.txt

Robots.txt is a standard for specifying how crawlers should interact with websites. By default, Starbelly will attempt to download a `robots.txt` from each domain that it visits, and it will obey the directives of any such files that it finds. In some circumstances, however, such as crawling some old sites, it may be useful to ignore or even invert the directives in a site's robots.txt, which you can configure using the policy.

## 5.4 URL Normalization

The crawler attempts to avoid crawling the same URL multiple times. If two links contain exactly identical URLs, then the crawler will only download that resource once. On some sites, especially dynamically generated sites, multiple URLs may refer to the same resource and differ only in the order of URL query parameters or the values of semantically meaningless query parameters like session IDs.

The URL normalization policy allows you to control this behavior. When enabled, the crawler normalizes URLS using a number of techniques, including:

- sorting query parameters alphabetically
- upper case percent encodings
- remove query fragments
- etc.

You may specify URL query parameters that should be discarded during normalization. By default, the crawler discards several common session ID parameters. Alternatively, you can disable URL normalization completely, although this may result in lots of duplicated downloads.

## 5.5 URL Rules

The URL rules policy controls how a crawler selects links to follow. For each page that is downloaded, the crawler extracts candidate links. For each candidate link, the crawler checks the rules one-by-one until a rule matches, then the crawler applies the matching rule.

For example, the default *Deep Crawl* policy contains two URL rules:

1. If the URL *matches* the regex `^https?://({SEED_DOMAINS})/` then *add* `1.0`.
2. Else *multiply by* `0.0`.

Let's say the URL is seeded with `http://foo.com/bar`. It downloads this document and assigns it a cost of 1.0. Cost is roughly similar to the concept of *crawl depth* in other crawlers, but it is a bit more sophisticated. Each link is assigned a cost based on the cost of the document where it was found and the URL rule that it matches. If a link cost evaluates to zero, then the link is thrown away. If the link is greater than zero but less than the "Max Cost" specified

in the crawl policy, then the crawler schedules the link to be fetched. Links are fetched roughly in order of cost, so lower-cost items are typically fetched before higher-cost items.

After the crawler downloads the document at `http://foo.com/bar`, it checks each link in that document against the URL rules in the policy. For example, if the link matches the regex in rule #1, then the link will be given a score of 2.0: the rule says to add 1.0 to the cost of its parent (which was 1.0).

If the link matches rule #2, then that rule says to multiply the parent's cost by zero. This results in the new cost being set to zero, and the crawler discards links where the cost is zero, so the link will not be followed.

Although the URL rules are a bit complicated at first, they turn out to be a very powerful way to guide the crawler. For example, if we step back a bit and consider the effect of the two rules above, we see that it follows links inside the seed domain and does not follow links outside the seed domain. In other words, this is a *deep crawl*!

If we replace the two rules here with just a single rule that says "Always add 1.0" , then that would result in a *broad crawl* policy! In fact, you can go look at the default *Broad Crawl* policy included in Starbelly to confirm that this is how it works.

## 5.6 User Agents

When the crawler downloads a resource, it sends a *User Agent* string in the headers. By default, Starbelly sends a user agent that identifies itself with a version number and includes a URL to its source code repository. You may customize what user agent is sent using the policy. If you include multiple user agent strings, one will be chosen at random for each request.

## 5.7 Proxy Rules

By default, the crawler downloads resources directly from their hosts. In some cases, you may want to proxy requests through an intermediary. The *Proxy Rules* specify which proxy server should be used for which request, similar to the *URL Rules* above.

## 5.8 MIME Type Rules

While *URL Rules* determine which links to follow, *MIME Type Rules* determine what types of resources to download. By default, the crawler only downloads resources that have a MIME type matching the regex `^text/`, which matches plain text and HTML resources. If you want the crawler to download images, for example, then you would add a new rule like `^image/*` that would match GIF, JPEG, and PNG resources.

The MIME type of a resource is determined by inspecting the `Content-Type` header, which means that *MIME Type Rules* are not applied until *after the crawler downloads headers* for a resource. If the crawler determines that a resource should not be downloaded, then the crawler closes the connection and discards any data that has already been downloaded.

## 5.9 Limits

The *Limits* policy specifies limits on how far and how long the crawl should run. If a limit is left blank, then that limit will not be applied to the crawl.

- Max cost: the crawler will not follow links that have a cost greater than the one specified here.
- Max duration: the maximum amount of time the crawler should run, in seconds.

- Max items: the maximum number of items that the crawler should download. This number includes successes, errors, and exceptions.

CHAPTER 6

# Administrator Guide

**Contents**



## 6.1 Overview

This section goes over some common tasks that you may need to perform as a Starbelly administrator. In the examples below, if a command prompt is prefixed with a container name, then that indicates that the command must be run inside a specific Docker container. For example, if you see this:

```
starbelly-dev-app:/starbelly# ls /usr/local/etc
jupyter
```

Then that command should be run inside of the `starbelly-dev-app` container. To obtain a shell inside that container, run:

```
$ docker exec -it starbelly-dev-app /bin/bash
starbelly-dev-app#
```

You can use the same technique to get a shell inside the `starbelly-dev-db` or `starbelly-dev-web` containers.

## 6.2 Clear Database

To clear all data from the database, including crawl data, job data, and other state:

```
starbelly-dev-app:/starbelly# python3 -m starbelly.clear
```

## 6.3 Change Password

Adding or changing passwords is covered in the *Installation Guide* under the "Security" section.

CHAPTER 7

# Developer Guide

**Contents**



## 7.1 Technologies Used

If you are thinking about helping out with Starbelly development, it will be useful to familiarize yourself with the different components and technologies being used:

- Starbelly Docker
  - Docker
  - Docker Compose
- Starbelly Protobuf
  - Protobuf
- Starbelly Server
  - Asyncio
  - Python 3
  - Restructed Text
  - RethinkDB
  - WebSockets

## 7.2 Getting Started

If you wish to contribute to Starbelly development, you should first make sure that you have gone through the Developer Installation in the *Installation Guide*. Once you have done that, go through the *API Documentation* and get familiar with how the client and server interact.

## 7.3 Common Tasks

As you start working on Starbelly code, you'll encounter some common tasks that you wish to perform. In the examples below, if a command prompt is prefixed with a container name, then that indicates that the command must be run inside a specific Docker container. For example, if you see this:

```
starbelly-dev-app:/starbelly# ls /usr/local/etc
jupyter
```

Then that command should be run inside of the `starbelly-dev-app` container. To obtain a shell inside that container, run:

```
$ docker exec -it starbelly-dev-app /bin/bash
starbelly-dev-app#
```

You can use the same technique to get a shell inside the `starbelly-dev-db` or `starbelly-dev-web` containers.

### 7.3.1 Build Documentation

This documentation that you are reading is written in RestructuredText format and stored in the main `starbelly` repo under `/docs`.

```
starbelly-dev-app:/starbelly/docs# make html
Running Sphinx v1.7.1
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
no targets are out of date.
build succeeded.
```

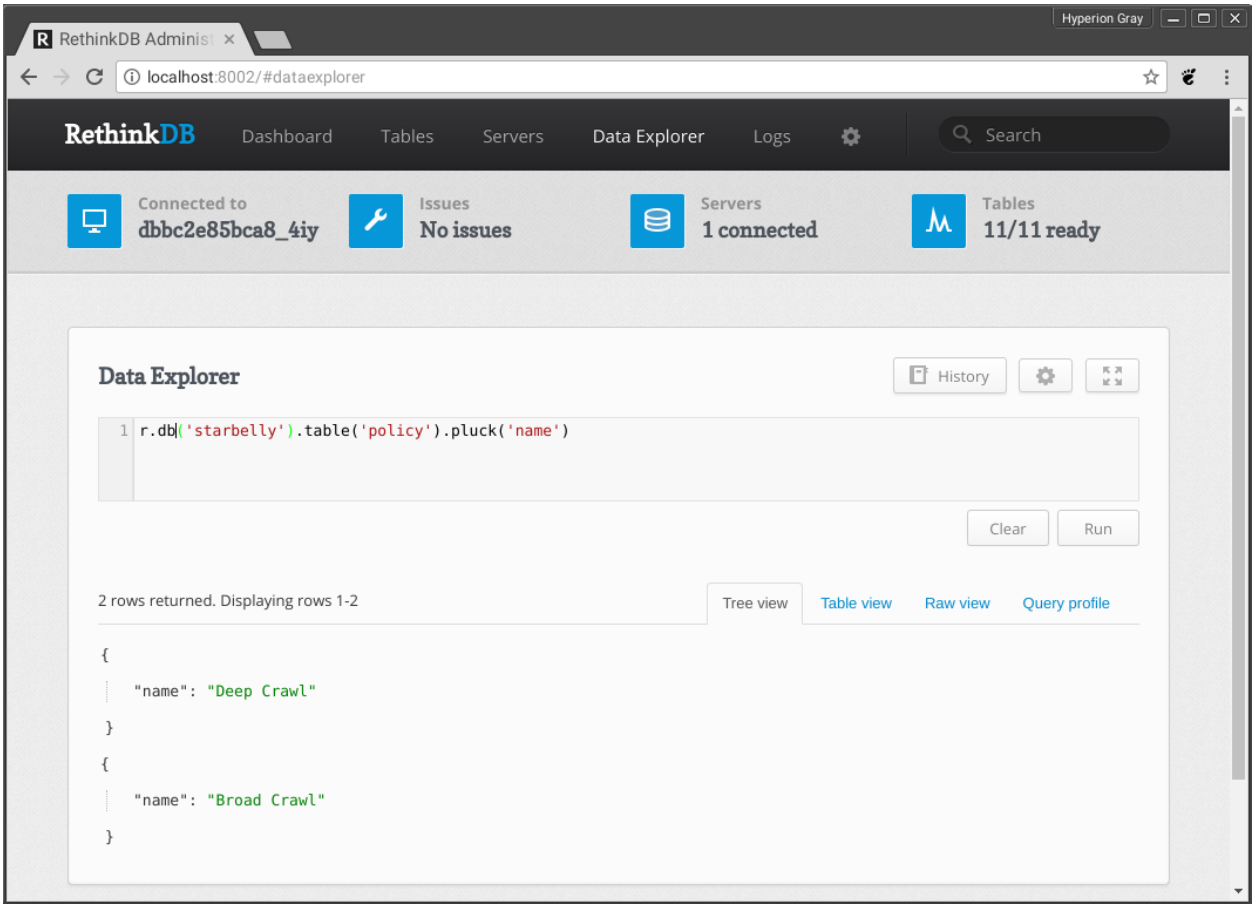
To view the documentation, use your web browser to navigate to the host's `starbelly/docs/_build/html/index.html`.

### 7.3.2 Clear Database

See the *Administrator Guide*.

### 7.3.3 Database Query

There are two ways to run RethinkDB queries. The easiest way is to access the RethinkDB GUI on port 8002 using your browser. You can browse lots of information about the database or use the "Data Explorer" to run queries. Note that this interface only allows JavaScript queries, so if you are trying to troubleshoot a Python query you will need to translate it into JavaScript.




If you want to run a Python query, you can use the Starbelly Shell instead, but it is a bit less pretty than the GUI:

```
>>> query = r.table('policy').pluck('name')
>>> cursor = qrun(query)
>>> qshow(cursor)
RethinkDB Cursor: [
    {'name': 'Deep Crawl'},
    {'name': 'Broad Crawl'},
]
```

The "Starbelly Shell" section contains more details about the shell.

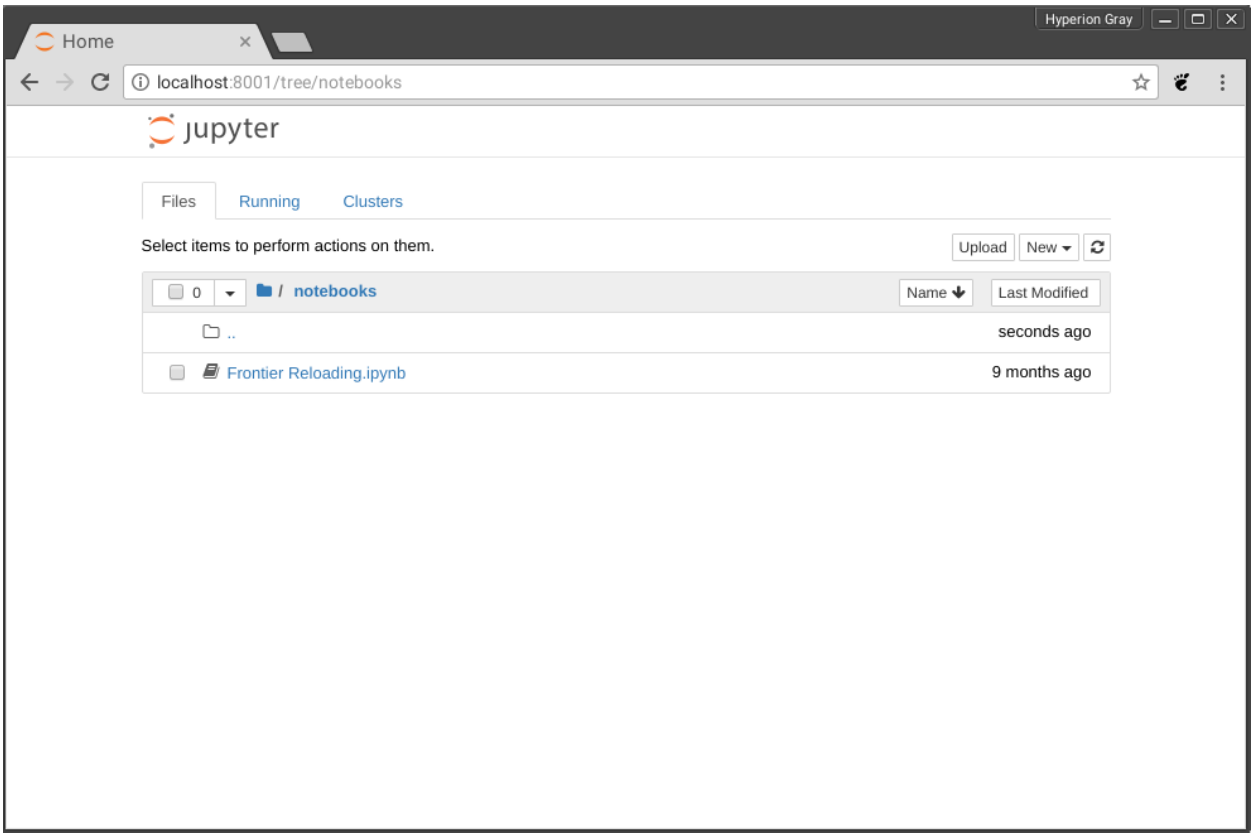
### 7.3.4 Notebook

A Jupyter server is included in the developer Docker images to make it easy to build a notebook for experiments and prototypes. A notebook can also be a slightly easier way to use the Starbelly Shell (see below).

Begin by running the Jupyter server:

```
starbelly-dev-app:/starbelly# jupyter notebook --allow-root --ip 0.0.0.0 --
↪NotebookApp.token=''
[W 20:44:35.840 NotebookApp] All authentication is disabled.  Anyone who can connect␣
↪to this server will be able to run code.
[I 20:44:35.850 NotebookApp] Serving notebooks from local directory: /starbelly
[I 20:44:35.850 NotebookApp] 0 active kernels
[I 20:44:35.850 NotebookApp] The Jupyter Notebook is running at:
[I 20:44:35.850 NotebookApp] http://0.0.0.0:8888/
[I 20:44:35.850 NotebookApp] Use Control-C to stop this server and shut down all␣
↪kernels (twice to skip confirmation).
```

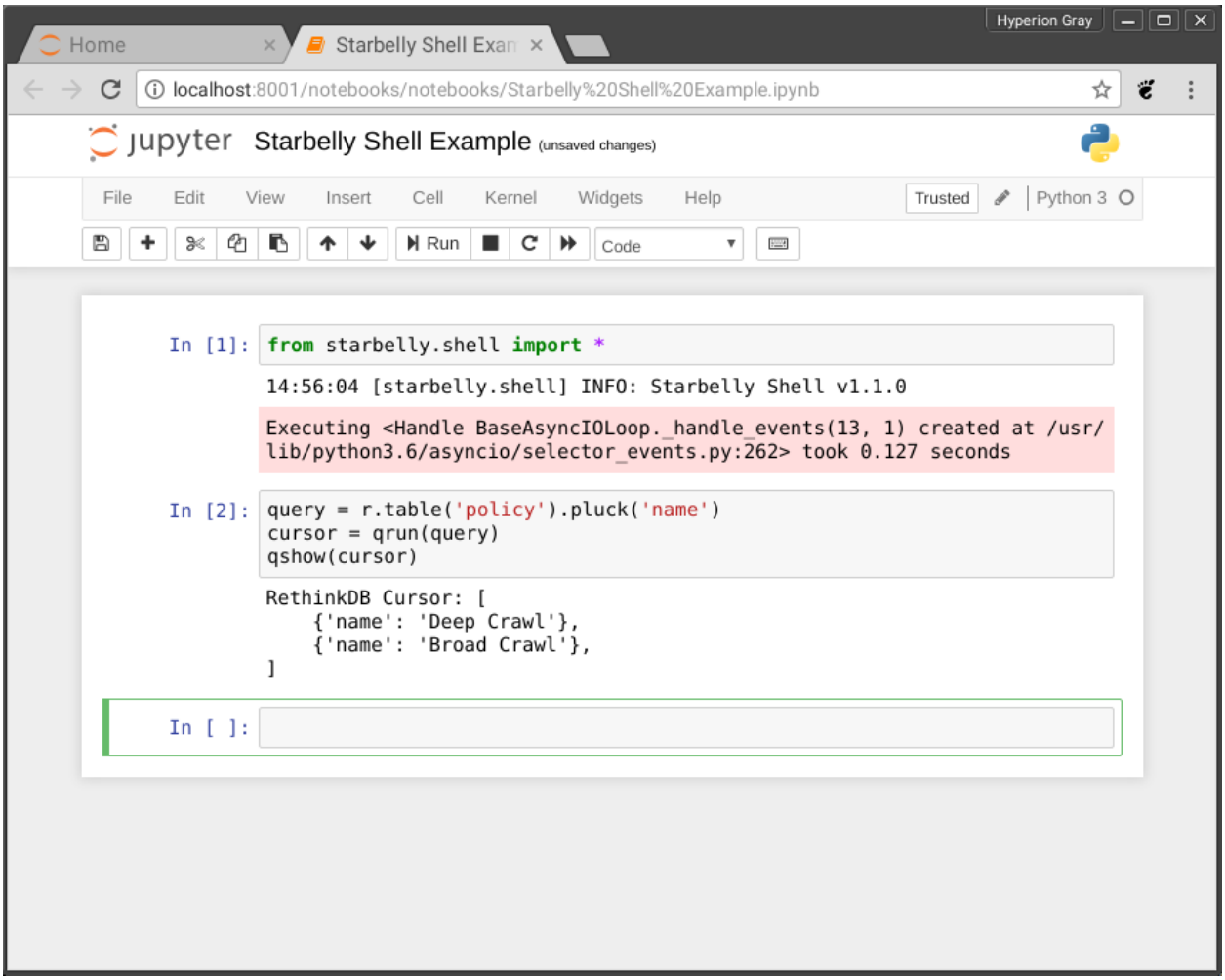
Now access the Jupyter server by going to `localhost:8001` in your browser.




You can create and manage your notebooks here. In each notebook, you'll probably want to begin by importing the shell functions:

```
from starbelly.shell import *
```

Then you can run commands inside the notebook just like you can from the Starbelly Shell:




When you are done, type <Ctrl>+C on the command line to shut down the Jupyter server. An example notebook is included in the repository.

### 7.3.5 Starbelly Shell

The Starbelly shell is an interactive Python interpreter that offers quick access to Starbelly's internal API, and it is a good place to debug little bits of code. The Shell cannot directly access the server's internal state at runtime, but it does use the same exact code paths as the server to do things like parsing configuration files and connecting to a database.

To start the shell:

```
starbelly-dev-app:/starbelly# python3 -im starbelly.shell
20:03:42 [starbelly.shell] INFO: Starbelly Shell v1.1.0
>>>
```

This Python prompt has a lot of additions to the global namespace and helpers for debugging code. For example, you can view the configuration:

```
>>> config['database']['user']
'starbelly-app'
```

Note that the name `config` (and many others) have already been imported into the shell's namespace.

You can run arbitrary coroutines with `crun()`:

```
>>> async def foo():
...   await asyncio.sleep(0.1)
...   print('done!')
...
>>> crun(foo())
done!
```

You can run a query against RethinkDB:

```
>>> query = r.table('policy').pluck('name')
>>> cursor = qrun(query)
>>> qshow(cursor)
RethinkDB Cursor: [
    {'name': 'Deep Crawl'},
    {'name': 'Broad Crawl'},
]
```

**Warning:** Showing the results of a query will exhaust the cursor object! If you try to do anything else with the cursor, you will find that it has no more data. You need to `qrun()` the query again to get a fresh cursor.

You can also run a callback on each row of a cursor to transform the data.

```
>>> def lower(row):
...   return {k:v.lower() for k,v in row.items()}
...
>>> query = r.table('policy').pluck('name')
>>> cursor = qrun(query)
>>> qiter(cursor, lower)
[{'name': 'deep crawl'}, {'name': 'broad crawl'}]
```

You can also use the Starbelly Shell functions inside of a Notebook. See the "Notebook" section above.

CHAPTER 8

# API Documentation

**Contents**



## 8.1 Overview

The crawler is controlled completely by an API. Clients connect to the crawler using websockets and exchange messages with the crawler using protobuf messages. The built-in GUI relies solely on this API, so everything that can be done in the GUI can also be done with the API – and more!

One of the central goals for the API is to enable clients to synchronize crawl results in real time. Most crawling systems are batch-oriented: you run the crawler for a period of time and then collect the results when the crawl is finished. Starbelly is streaming-oriented: it can send crawl results to a client as soon as it downloads them.

Let's imagine that a crawl has started running and already has 1,000 results. A client can connect to Starbelly and quickly fetch the first 1,000 results. Because the crawler is still running, new results will continue to stream in as the crawler downloads them. If either the server or the client needs to disconnect for some reason, the client is able to reconnect later and pick up the stream exactly where it left off.

## 8.2 Connecting to API

The API is exposed as a websocket service on port 443 at the path `/ws/`. For example, if starbelly is running on the host `starbelly.example.com`, then you should connect to the web socket using the URL `wss://starbelly.example.com/ws/`. By default, Starbelly uses HTTP basic authentication, so you need to include those credentials when you connect to the API.

## 8.3 Messages

Starbelly uses `protobuf` to encode messages sent between the client and the server. There are three types of message used in the API:

1. Request
2. Response
3. Event

The *request* and *response* messages are created in pairs: the client sends a *request* to the server and the server sends back exactly one *response* per request. The response indicates whether the request was successful and may include other data related to the request.

Although each request generates a response, the responses are not necessarily sent back in the same order that the requests are received. If the client sends two commands very quickly (call them A and B), it may get the responses back in either order, e.g. A→B or B→A. For this reason, the client should include a unique `request_id` with each request; the server will include the same `request_id` in its response so that the client can track which response goes with which request. The client can assign request IDs in any manner that it chooses, but one sensible approach would be to assign an incrementing sequence of integers.

The third type of message is an *event*, which is pushed from the server to the client. For example, the client can send a request to subscribe to job status. The server will send a response containing a subscription ID. Now, whenever a job has a status event, such as downloading a new resource, the server will send an event to the client containing the job status data and the corresponding subscription ID. The client can close the subscription by sending another request. The server will stop sending event messages and will send a response indicating that the subscription has been cancelled.

Protobuf is a binary serialization format that supports common data types like integers, strings, lists, and maps. It is similar in purpose to JSON, but protobuf is more efficient in terms of encoding overhead and serialization speed.

## 8.4 Example Session

This section shows a complete interaction where a client starts a crawl and synchronizes crawl results. To begin, the client sends a `RequestSetJob` request to the server that includes the seed URL, a policy identifier, and a crawl name.

```
Request {
    request_id: 1
    Command: RequestSetJob {
        run_state: RUNNING
        policy_id: d28b379ff3668322bfd5d56e11d4e34e
        seeds: "https://cnn.com"
        name: "My Crawl"
    }
}
```

The server will kick off a crawling job and will send a response telling the client that the job has started successfully and including an identifier for the new job.

```
Response {
    request_id: 1
    is_success: true
    Body: ResponseNewJob {
        job_id: 0514478baffd401546b755bf460b5997
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Notice that the response includes the request ID sent by the client, so we know that this is a response to the above request.

This response tells us that the crawl is starting, but we would like to keep track of the crawl's progress and know when it finishes. The next step is to send a subscription request for job status events.

```
Request {
    request_id: 2
    Command: RequestSubscribeJobStatus {
        min_interval: 3.0
    }
}
```

This subscription provides high-level job status for *all* crawl jobs, including data like how many items have been downloaded, how many pages had errors, how many pages results in exceptions, etc. Job status can change rapidly when the crawler is busy, because each item downloaded counts as a change in job status. The `min_interval` parameter specifies the minimum amount of time in between job status events sent by the server. In this example, if there are multiple job status events, the server will batch them together and send at most 1 event every 3 seconds for this subscription. On the other hand, if the crawl is very slow, then it may send events even less frequently than that.

The server will create the subscription and respond with a subscription identifier.

```
Response {
    request_id: 1
    is_success: true
    Body: ResponseNewSubscription {
        subscription_id: 300
    }
}
```

When the client first subscribes to job status, the crawler will send the complete status of each currently running job. For example, if the crawler has already downloaded one item, the job status may look like this:

```
Event {
    subscription_id: 300
    Body: JobList {
        jobs: {
            job_id: 0514478baffd401546b755bf460b5997
            seeds: "https://cnn.com"
            policy: d28b379ff3668322bfd5d56e11d4e34e
            name: "My Crawl"
            run_state: RUNNING
            started_at: "2017-11-03T10:14:42.194744"
            item_count: 1
            http_success_count: 1
            http_error_count: 0
            exception_count: 0
            http_status_counts: {
                200: 1
            }
        }
    }
}
```

After sending complete job status, the crawler will send small updates as the job status changes. For example, after the crawler downloads a second item, it will send an event like this:

```
Event {
    subscription_id: 300
    Body: JobList {
        jobs: {
            job_id: 0514478baffd401546b755bf460b5997
            item_count: 2
            http_success_count: 2
            http_status_counts: {
                200: 2
            }
        }
    }
}
```

Notice how the second message is much smaller: it only contains the fields that have changed since the previous event. This is how the job status subscription allows clients to efficiently keep track of the status of all jobs. This API is used in the GUI to power the Dashboard and Results screens.

For a complete list of API messages, look at the starbelly-protobuf repository.

## 8.5 Web Client

The crawler GUI is implemented as a stand-alone application written in Dart, and it interacts with the Starbelly server solely through the public API. Therefore, anything that you can do in the GUI can also be done through the API.

https://github.com/hyperiongray/starbelly-web-client

## 8.6 Python Client

A very basic and incomplete Python client library implementation is available:

https://github.com/hyperiongray/starbelly-python-client

This client library will be improved over time and made more stable, but for now it may be used as a reference implementation.

CHAPTER 9

# Changelog

## 9.1 v1.1.0 (2018-XX-XX)

- Upgrade Docker base image to Ubuntu:17.10.
- Upgrade Dart SDK to 1.24.0.
- Upgrade web client to Angular 4.0.0.

## 9.2 v1.0.0 (2017-11-03)

- Initial release.