Spackmon Documentation

Release 0.0.1

Vanessa Sochat

Dec 08, 2021

GETTING STARTED

1 Support **3**

2 Resources **5**

2.1 Getting Started . . . 5
2.1.1 Bringing Up Containers . . . 5
2.1.2 Authentication . . . 8
2.1.3 Application Programming Interface . . . 11
2.1.4 API Tutorial . . . 19
2.1.5 Data Application Programming Interface . . . 22
2.1.6 Interfaces . . . 23
2.1.7 Settings . . . 25
2.2 Development Guide . . . 26
2.2.1 Design of the Models . . . 26
2.2.2 Development Setup . . . 27
2.2.3 Tables . . . 27
2.2.4 Documentation . . . 29
2.2.5 Setup . . . 29
2.2.6 Test Smeagle . . . 29
2.2.7 Change Compiler . . . 30
2.2.8 Test Symbolator . . . 30
2.3 Deployment Guide . . . 30
2.3.1 Amazon Web Services (AWS) . . . 31

Spack Monitor, or "spackmon" is a monitoring service and database for Spack. It will allow you to capture complete output, error, specs, and other metadata associated with spack builds, either to explore or further query.

CHAPTER

ONE

SUPPORT

- In case of **questions**, please post on stack overflow.
- To **discuss** with other spack users, you can use the mailing list. **Please do not post questions there. Use stack overflow for questions.**
- For **bugs and feature requests**, please use the issue tracker.
- For **contributions**, visit Spack Monitor on Github.

CHAPTER

TWO

RESOURCES

Spack Documentation The main page of Spack, with links to code, documentation, and community resources.

Spack Monitor Repository Spack Monitor on GitHub

Spack Repository The main spack code base.

2.1 Getting Started

Spack Monitor (spackmon) will allow you to programatically interact with a database to keep track of your spack builds. This means that we store entire configurations with package specs, along with error and output for the builds. The guide here should walk you through basic usage of spackmon. If you have other questions or find something missing, please let us know We recommend that you start by *Bringing Up Containers*, which includes specifics for starting with containers.

2.1.1 Bringing Up Containers

Installation comes down to bringing up containers. You likely want to start with your database in a container, and for a more substantial service, update the database credentials to be for something more production oriented. For using Spackmon you will need:

- Docker
- docker-compose

Once you have these dependencies, you'll first want clone the repository.

```
$ git clone git@github.com:spack/spack-monitor.git
$ cd spack-monitor
```

Then you can build your container, which we might call *spack/spackmon*. To do this with docker-compose:

```
$ docker-compose build
```

Note that if you just do `docker-compose up -d` without building, it will build it for you. It's good practice to keep these steps separate so you can monitor them more closely. Once you have the base container built, you can bring it up (which also will pull the other containers for nginx and the database).

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

You can verify they are running without any exit error codes:

```
$ docker-compose ps
        Name                       Command               State         Ports
-------------------------------------------------------------------------------------
spack-monitor_db_1      docker-entrypoint.sh postgres    Up      5432/tcp
spack-monitor_nginx_1   /docker-entrypoint.sh ngin ...   Up      0.0.0.0:80->80/tcp
spack-monitor_uwsgi_1   /bin/sh -c /code/run_uwsgi.sh    Up      3031/tcp
```

And you can look at logs for the containers as follows:

```
$ docker-compose logs
$ docker-compose logs uwsgi
$ docker-compose logs db
$ docker-compose logs nginx
```

Great! Now you are ready to start interacting with your Spack Monitor. Before we do that, let's discuss the different ways that you can interact.

Spackmon Interactions

There are two use cases that might be relevant to you:

- you have existing configuration files that you want to import
- you have a spack install that you want to build with, and directly interact

Import Existing Specs

For this case, we want to generate and then import a custom configuration. But to be clear, a configuration is simply a package with it's dependencies, meaning that the unique id for it is the `full_hash`. Let's make that first. As noted in the *Design of the Models* section, there is a script provided that will make it easy to generate a spec, and *note that we generate it with dependency (package) links using a full instead of a build hash*. Let's do that first. Since we need spack (on our host) we will run this outside of the container. Make sure that you have exported the spack bin to your path:

```
$ export PATH=$PATH:/path/to/spack/bin
```

From the repository, generate a spec file. There is one provided for Singularity if you want to skip this step. It was generated as follows:

```
 $ mkdir -p specs
                                  # lib       # outdir
 $ ./script/generate_random_spec.py singularity specs
...
wont include py-cython due to variant constraint +python
Success! Saving to /home/vanessa/Desktop/Code/spack-monitor/specs/singularity-3.6.4.
↪json
```

Important If you want to generate this on your own, you must use a full hash, as this is what the database uses as a unique identifier for each package.

```
spack spec --json --hash-type full_hash singularity
```

Your containers should already be running. Before we shell into the container, let's grab the spack version, which we will need for the request.

```
$ echo $(spack --version)
$ 0.16.0-1379-7a5351d495
```

Let's now shell into the container, where we can interact directly with the database.

```
$ docker exec -it spack-monitor_uwsgi_1 bash
```

The script `manage.py` provides an easy interface to run custom commands. For example, here is how to do migrations and setup the database (this is done automatically for you when you first bring up the container in `run_uwsgi.sh`, but if you need to change models or otherwise update the application, you'll need to run these manually in the container:

```
$ python manage.py makemigrations main
$ python manage.py makemigrations users
$ python manage.py migrate
```

When the database is setup (the above commands are run, by default) we can run a command to do the import. Note that we are including the spec file and the spack version (so you should have it on your path):

```
$ python manage.py import_package_configuration specs/singularity-3.6.4.json 0.16.0-
↪1379-7a5351d495
```

The package is printed to the screen, along with it's full hash.

```
Filename                          specs/singularity-3.6.4.json
Spack Version                     0.16.0-1379-7a5351d495
Status                            created
singularity v3.6.4                p64nmszwer36ly7pnch5fznni4cnmndg
```

You could run the same command externally from the container (and this extends to any command) by doing:

```
$ docker exec -it spack-monitor_uwsgi_1 python manage.py import_package_configuration␣
↪specs/singularity-3.6.4.json
```

If you do this twice, however, it's going to tell you that it already exists. We use the `full_hash` of the package to determine if it's already there.

```
$ docker exec -it spack-monitor_uwsgi_1 python manage.py import_package_configuration␣
↪specs/singularity-3.6.4.json $(spack --version)
Filename                          specs/singularity-3.6.4.json
Spack Version                     0.16.0-1379-7a5351d495
Status                            exists
singularity v3.6.4                xttimnxa2kc4rc33axvrcpzejiil6wbn
```

Note that these commands will work because the working directory is `/code` (where the specs folder is) and `./code` is bound to the host at the root of the repository. If you need to interact with files outside of this location, you should move them here. Note that this interaction is intended for development or testing. If you want to interact with the database from spack, the avenue will be via the *Application Programming Interface*.

Databases

By default, Spackmon will deploy with it's own postgres container, deployed via the docker-compose.yml. If you want to downgrade to sqlite, you can set `USE_SQLITE` in your `spackmon/settings.yml` file to a non null value. This will save a file, `db.sqlite3` in your application root. If you want to update to use a more production database, you can remove the `db` section in your docker-compose.yml, and then export variables for your database to the environment:

```
export DATABASE_ENGINE=django.db.mysql # this is the default if you don't set it
export DATABASE_HOST=my.hostname.dev
export DATABASE_USER=mydatabaseuser
export DATABASE_PASSWORD=topsecretbanana
export DATABASE_NAME=databasename
```

We have developed and tested with the postgres database, so please report any issues that you find if you try sqlite. If you want to try the application outside of the containers, this is possible (but not developed or documented yet) so please open an issue. Now that you have your container running and you've import a spec, you should read the *Application Programming Interface* docs to create a user and learn how to interact with your application in a RESTful, authenticated manner.

2.1.2 Authentication

This section will walk you through creating a user and getting your token to authenticate you to the *Application Programming Interface* so you can walk through the *API Tutorial*. You should already have your containers running (see *Bringing Up Containers* if you do not).

OAuth2 Accounts

To support allowing a user to create their own account, Spack Monitor has support to login via OAauth2 from GitHub, which means that as the admin of the server you'll need to setup an OAuth2 application, and as a user you'll need to authenticate with GitHub to login. Setup of that requires the following. For users to connect to Github, you need to register a new application and export the key and secret to your environment as follows:

```
# http://psa.matiasaguirre.net/docs/backends/github.html?highlight=github
export SOCIAL_AUTH_GITHUB_KEY='xxxxxxxxxxxx'
export SOCIAL_AUTH_GITHUB_SECRET='xxxxxxxxxxxx'
```

To provide these secrets via docker-compose, you can add an `environment` section to your yaml (that should not be placed in version control!)

```
uwsgi:
  restart: always
  build: .
  environment:
    - SOCIAL_AUTH_GITHUB_KEY=xxxxxxxxxxxx
    - SOCIAL_AUTH_GITHUB_SECRET=xxxxxxxxxxxxxxxx
  volumes:
    - .:/code
    - ./static:/var/www/static
    - ./images:/var/www/images
  links:
    - db
```

And then in the settings.yml, update `ENABLE_GITHUB_AUTH` to true. The server will not start if the credentials are not found in the environment. When you register the application, the callback url should be in the format

http://127.0.0.1/complete/github/, and replace the localhost address with your domain. See the Github Developers pages to browse more information on the Github APIs.

Legacy Account Creation

Before supporting user accounts, a user token could be generated on the command line to associate with a build. This is still supported for anyone that doesn't want to use (or cannot use) GitHub, however it requires a server admin to manually do the work, which isn't ideal for all deployments. For example, if we want to add a user:

```
$ docker exec -it spack-monitor_uwsgi_1 python manage.py add_user vsoch
Username: vsoch
Enter Password:
User vsoch successfully created.
```

You can then get your token (for the API here) as follows:

```
$ docker exec -it spack-monitor_uwsgi_1 python manage.py get_token vsoch
Username: vsoch
Enter Password:
50445263afd8f67e59bd79bff597836ee6c05438
```

TADA! We will export this token as `SPACKMON_TOKEN` in the environment to authenticate via the API, a flow that will generate us a temporary bearer token (that expires after a certain amonut of time). This is the authentication flow, discussed next.

The Authentication Flow

We are going to use a "docker style" OAuth 2 (as described here, with more details provided in this section.

The User Request

When you make a request to the API without authentication, this first request will return a 401 "Unauthorized" response The server knows to return a `Www-Authenticate` header to your client with information about how to request a token. That might look something like:

Note that realm is typically referring to the authentication server, and the service is the base URL for the monitor service. In the case of Spack Monitor they are one and the same (e.g., both on localhost) but this doesn't have to be the case. You'll see in the settings that you can customize the authentication endpoint.

The requester then submits a request to the realm with those variables as query parameters (e.g., GET) and also provides a basic authentication header, which for Spack Monitor, is the user's username and token associated with the account (instructions provided above for generating your username and token). We put them together as follows:

We then base64 encode that, and add it to the http Authorization header.

That request then goes to the authorization realm, which determines if the user has permission to access the service for the scope needed. Note that scope is currently limited to just build, and we also don't specify a specific resource. This could be extended if needed.

The Token Generation

Given that the user account is valid, meaning that we check that the username exists, the token is correct, and the user has permission for the scopes requested (true by default), we generate a jwt token that looks like the following:

```
{
  "iss": "http://127.0.0.1/auth",
  "sub": "vsoch",
  "exp": 1415387315,
  "nbf": 1415387015,
  "iat": 1415387015,
  "jti": "tYJCO1c6cnyy7kAn0c7rKPgbV1H1bFws",
  "access": [
    {
      "type": "build",
      "actions": [
        "build"
      ]
    }
  ]
}
```

If you are thinking that the type and actions are redundant, you are correct. We currently don't need to do much checking in terms of scope or actions. The "exp" field is the timestamp for when the token expires. The nbf says "This can't be used before this timestamp," and iat refers to the issued at timestamp. You can read more about jwt here. We basically use a python jwt library to encode this into a long token using a secret on the server, and return this token to the calling client.

```
{"token": "1sdjkjf....xxsdfser", "issued_at": "<issued timestamp>", "expires_in": 600}
```

Retrying the Request

The client then retries the same request, but adds the token to the Authorization header, this time with Bearer.

```
{"Authorization": "Bearer <token>"}
```

And then hooray! The request should be successful, along with subsequent requests using the token until it expires. The expiration in seconds is also defined in the settings.yml config file.

Disable Authentication

You can see in the *Settings* that there is a configuration variable to disable authentication, `DISABLE_AUTHENTICATION`. This usually isn't recommended. If you disable it, then views that require authentication will not look for the bearer token in the header.

If you want to interact with the API, we next recommend doing the *API Tutorial*, or just read more about the endpoints at *Application Programming Interface*.

2.1.3 Application Programming Interface

Spackmon implements a set of endpoints that make it possible for spack to communicate with the database via RESTful requests. This document outlines these endpoints, which we call the Spack Monitor Schema. You should read about *Authentication* if you want to first create a user to interact with these endpoints, and then check out the *API Tutorial* for a walkthrough.

Overview

Introduction

The **Spack Monitor Specification** defines an API protocol to standardize the requests and responses for spack to communicate with a monitoring server. It is created in the same spirit as the opencontainers distribution spec.

Definitions

The following terms are used commonly in this document, and a list of definitions is provided for reference:

- **server**: a service that provides the endpoints defined in this specification
- **spack**: a local spack installation where you intend to monitor builds of software

Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in RFC 2119. (Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997).

Conformance

Currently, we don't have any tools to test conformance, and the requirements are outlined here.

Determining Support

To check whether or not the server implements the spack monitor spec, the client SHOULD perform a *GET* request to the `/ms1/` (service info) endpoint. If the response is `200 OK`, then the server implements the specification. This particular endpoint MAY be used for authentication, however authentication is outside of the scope of this spec.

For example, given a url prefix of `http://127.0.0.0` the client would issue a *GET* request to:

```
GET http://127.0.0.1:5000/ms1/
```

And see the service info section below for more details on this request and response. This endpoint serves to either return a successful response to the calling spack client, or direct the client to use a differently named endpoint.

All of the following would be valid:

```
https://spack.org/ms1/
http://spack.org/ms1/
http://localhost/ms1/
http://127.0.0.1/ms1/
http://127.0.0.1:8282/ms1/
```

For each of the above, the client implementing the spec would be provided the url before `/ms1/` (e.g., https://spack.org/) and then use that to assemble all the endpoints (e.g., https://spack.org/ms1/).

Endpoint Requirements

Servers conforming to the Spack Monitor specification (like Spack Monitor) must provide the following endpoints:

1. **Service Info** (`GET /ms1/`) endpoint with a 200 or 30* response.

Response Details

Generally, a response will return a json object that shows a message, and a return code. For example, a successful response will have a message of "success" to go along with a 200 or 201 response code, while an unsuccessful response will have a message indicating the error, and an error code (e.g., 400, 500, etc.). Error responses may not have data. Successful reponses will have metadata specific to the endpoint.

```
{"message": "success", "data" {...}, "code": 201}
```

Generally, endpoint data will return a lookup of objects updated or created based on the type. For example, the new configuration endpoint has metadata about the spec created under a `spec` attribute of the data:

```
{
    "message": "success",
    "data": {
        "spec": {
            "full_hash": "p64nmszwer36ly7pnch5fznni4cnmndg",
            "name": "singularity",
            "version": "3.6.4",
            "spack_version": "1.0.0",
            "specs": {
                "cryptsetup": "tmi4pf6umhalop7mi6zyiv7xjpalyzgb",
                "go": "dehg3ddu6gacrmnoexbxhjv2i2d76yq6",
                "libgpg-error": "4cvsg42wxksiup6x74mlabu6un55wjzc",
                "libseccomp": "kfx6zyjxzudw77e3xk6i73bcgi2cavgh",
                "pkgconf": "al2hlnux3cchfhwiv2sbejnxvnogibac",
                "shadow": "aozeq6ybtsnrs5phtonutwes7fe6yhcy",
                "squashfs": "vpemhhpzqqf7mvpzdvcg6szfah6mwt2q",
                "util-linux-uuid": "g362jjpzlfp3qhfm7gdery6v3xgeh3lg"
            }
        },
        "created": true
    },
    "code": 201
}
```

Timestamps

For all fields that will return a timestamp, we are tentatively going to use the stringified version of a `datetime.now()`, which looks like this:

```
2020-12-15 11:43:24.811860
```

Endpoint Details

Service Info

`GET /ms1/`

This particular Endpoint exists to check the status of a running monitor service. The client should issue a `GET` request to this endpoint without any data, and the response should be any of the following:

- 404: not implemented
- 200: success (indicates running)
- 503: service not available
- 302: found, change namespace
- 301: redirect

As the initial entrypoint, this endpoint also can communicate back to the client that the prefix (ms1) has changed (e.g., response 302 with a Location header). More detail about the use case for each return code is provided below. For each of the above, the minimal response returned should include in the body a status message and a version, both strings:

```
{"status": "running", "version": "1.0.0"}
```

Service Info 404

In the case of a 404 response, it means that the server does not implement the monitor spec. The client should stop, and then respond appropriately (e.g., giving an error message or warning to the user).

```
{"status": "not implemented", "version": "1.0.0"}
```

Service Info 200

A 200 is a successful response, meaning that the endpoint was found, and is running.

```
{"status": "running", "version": "1.0.0"}
```

Service Info 503

If the service exists but is not running, a 503 is returned. The client should respond in the same way as the 404, except perhaps trying later.

```
{"status": "service not available", "version": "1.0.0"}
```

Service Info 302

A 302 is a special status intended to support version changes in a server. For example, let's say that an updated specification API is served at `/ms2/` and by default, a client knows to send a request to `/ms1/`. To give the client instruction to use `/ms2/` for all further interactions, the server would return a 302 response

```
{"status": "multiple choices", "version": "1.0.0"}
```

with a location header to indicate the updated url prefix:

```
Location: /m2/
```

And the client would update all further prefixes accordingly.

Service Info 301

A 301 is a more traditional redirect that is intended for one off redirects, but not necessarily indicatig to change the entire client namespace. For example, if the server wanted the client to redirect `/ms1/` to be `/service-info/` (but only for this one case) the response would be:

```
{"status": "multiple choices", "version": "1.0.0"}
```

With a location header for just this request:

```
Location: /service-info/
```

For each of the above, if the server does not return a Location header, the client should issue an error.

New Spec

`POST /ms1/specs/new/`

If you have a spec configuration file, you can load it into Python and issue a request to this endpoint. The response can be any of the following:

- 404: not implemented
- 201: success (indicates created)
- 503: service not available
- 400: bad request
- 403: permission denied
- 200: success (but the config already exists)

New Config Created 201

If the set of specs are created from the configuration file, you'll get a 201 response with data that includes the configuration id (the full_hash) along with full hashes for each package included:


```
{
    "message": "success",
    "data": {
        "spec": {
            "full_hash": "p64nmszwer36ly7pnch5fznni4cnmndg",
            "name": "singularity",
            "version": "3.6.4",
            "spack_version": "1.0.0",
            "specs": {
                "cryptsetup": "tmi4pf6umhalop7mi6zyiv7xjpalyzgb",
                "go": "dehg3ddu6gacrmnoexbxhjv2i2d76yq6",
                "libgpg-error": "4cvsg42wxksiup6x74mlabu6un55wjzc",
                "libseccomp": "kfx6zyjxzudw77e3xk6i73bcgi2cavgh",
                "pkgconf": "al2hlnux3cchfhwiv2sbejnxvnogibac",
                "shadow": "aozeq6ybtsnrs5phtonutwes7fe6yhcy",
                "squashfs": "vpemhhpzqqf7mvpzdvcg6szfah6mwt2q",
                "util-linux-uuid": "g362jjpzlfp3qhfm7gdery6v3xgeh3lg"
            }
        },
        "created": true
    },
    "code": 201
}
```


All of the above are full hashes, which we can use as unique identifiers for the builds.

New Config Already Exists 200

If the configuration in question already exists, you'll get the same data response, but a status code of 200 to indicate success (but not create).

New Build

`POST /ms1/builds/new/`

This is the endpoint to use to get or lookup a previously done build, and retrieve a build id that can be used for further requests. A new build means that we have a spec, an environment, and we are starting a build! The `Build` object can be either created or retrieved (if the comibination already exists), and it will hold a reference to the spec, the host build environment, build phases, and (if the build is successful) a list of objects associated (e.g., libraries and other binaries produced).

- 404: not implemented or spec not found
- 200: success
- 201: success
- 503: service not available
- 400: bad request
- 403: permission denied

In either case of success (200 or 201) the response data is formatted as follows:

```
{
    "message": "Build get or create was successful.",
    "data": {
        "build_created": true,
        "build_environment_created": true,
        "build": {
            "build_id": 1,
            "spec_full_hash": "p64nmszwer36ly7pnch5fznni4cnmndg",
            "spec_name": "singularity"
        }
    },
    "code": 201
}
```

New Build Created 201

When a new build is created, the status will be 201 to indicate that.

New Build Success 200

If a build is re-run, it may already have been created. The status will be 200 to indicate this.

Update Build Status

`POST /ms1/builds/update/`

When Spack is running builds, each build task associated with a spec and host environment can either succeed or fail, or something else. In each case, we need to update Spack Monitor with this status. The default status for a build task is `NOTRUN`. Once the builds start, given a failure, this means that the spec that failed is marked as `FAILURE`, and the main spec along with the other specs that were not installed are marked as `CANCELLED`. In the case of success for any package, we mark with `SUCCESS`. If Spack has a setting to "rollback" we will need to account for that (not currently implemented).

- 404: not implemented or spec not found
- 200: success
- 503: service not available
- 400: bad request
- 403: permission denied

Build Task Updated 200

When you want to update the status of a spec build, a successful update will return a 200 response.

```
{
    "message": "Status updated",
    "data": {
        "build": {
            "build_id": 1,
            "spec_full_hash": "p64nmszwer36ly7pnch5fznni4cnmndg",
            "spec_name": "singularity"
        }
    },
    "code": 200
}
```

Update Build Phase

`POST /ms1/builds/phases/update/`

Build Phases are associated with builds, and this is when we have output and error files. The following responses re valid:

- 404: not implemented or spec not found
- 200: success
- 503: service not available
- 400: bad request
- 403: permission denied

The request to update the phase should look like the following - we include the build id (created or retrieved from the get build endpoint) along with simple metadata about the phase, and a status.

```
{
    "build_id": 47,
    "status": "SUCCESS",
    "output": null,
    "phase_name": "autoreconf"
}
```

Update Build Phase 200

When a build phase is successfully updated, the response data looks like the following:

```
{
    "message": "Phase autoconf was successfully updated.",
    "code": 200,
    "data": {
        "build_phase": {
            "id": 1,
            "status": "SUCCESS",
            "name": "autoconf"
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Analyze Builds Metadata

`POST /ms1/analyze/builds/`

Analyze endpoints correspond with running `spack analyze`, and are generally for taking some metadata (environment, install files, etc.) from the installed package directory and adding them to the server. When a spec is finished installing, we have a metadata folder, usually within the spack root located at `opt/<system>/<compiler>/<package>/.spack` with one or more of the following files:

- spack-configure-args.txt'
- spack-build-env.txt'
- spec.yaml
- archived-files
- spack-build-out.txt
- install_manifest.json
- install_environment.json
- repos
- errors.txt

The `install_environment.json` can easily be used to look up the build id, and then any kind of metadata can be added. The data keys that you send will correspond to where the metadata is added:

- environment_variables: indicates a list of environment variables to link to a build
- install_files: indicates a list of install files to be created as objects
- config_args: the content of `spack-configure-args.txt`

Any other attribute is assumed to be a lookup of key value pairs, indexed by an object.

As a user, you are allowed to send as many of these keys and data to the server as you see fit, meaning you can do multiple kinds of analyses at once and then update the monitor server. A complete example of sending a build environment and install files is shown below:

```
{
    "environ": {
        "SPACK_CC": "/usr/bin/gcc",
        "SPACK_CC_RPATH_ARG": "-Wl,-rpath,",
        "SPACK_COMPILER_SPEC": "gcc@9.3.0",
        "SPACK_CXX": "/usr/bin/g++",
        "SPACK_CXX_RPATH_ARG": "-Wl,-rpath,",
        ...
        "SPACK_TARGET_ARGS": "'-march=skylake -mtune=skylake'"
    },
    "config": "",
    "manifest": {
        "/home/vanessa/Desktop/Code/spack/opt/spack/linux-ubuntu20.04-skylake/gcc-9.3.
↪0/diffutils-3.7-2tm6lq6qmyrj6jjiruf7rxb3nzonnq3i/.spack": {
            "mode": 17901,
```

(continues on next page)

(continued from previous page)


```
            "owner": 1000,
            "group": 1000,
            "type": "dir"
        },
        ...
        "/home/vanessa/Desktop/Code/spack/opt/spack/linux-ubuntu20.04-skylake/gcc-9.3.
↪0/diffutils-3.7-2tm6lq6qmyrj6jjiruf7rxb3nzonnq3i": {
            "mode": 17901,
            "owner": 1000,
            "group": 1000,
            "type": "dir"
        }
    },
    "full_hash": "5wdhxf5usk7g6gznwhydbwzmdibxqhjp"
}
```


The environment is read in, filtered to a list to include only `SPACK_*` variables, and split into key value pairs, and the package full hash is provided to look up. If any information does not exist, it is simply not sent. A full request might look like the following: The response can then be any of the following:

- 404: not implemented
- 503: service not available
- 400: bad request
- 403: permission denied
- 200: success

Unlike other endpoints, this one does not check if data is already added for the build, it simply re-writes it. This is under the assumption that we might re-do an analysis and update the metadata associated. The response is brief and tells the user that the metadata for the build has been updated:


```
{
    "message": "Metadata updated",
    "data": {
        "build": {
            "build_id": 1,
            "spec_full_hash": "p64nmszwer36ly7pnch5fznni4cnmndg",
            "spec_name": "singularity"
        }
    },
    "code": 200
}
```


2.1.4 API Tutorial

For this tutorial, you should have already started containers (*Bringing Up Containers*) and created your user account and token (*Authentication*).

API Examples

In the scripts folder in the repository, you'll find a file `spackmoncli.py` that provides an example client for interacting with a Spack Monitor database. The `api-examples` folder also includes these examples in script form. Before doing this tutorial, you should have already started containers (*Bringing Up Containers*), and created a username and token (*Application Programming Interface*).

The only dependency you should need for these examples is to install requests, which we use here because it's easier than urllib (in spack we don't want to add a dependency so we stick to urllib). You can use the requirements.txt in the examples folder to do this.

```
$ pip install -r script/api-examples/requirements.txt
```

Service Info Example

Let's first create a client, and use it to get service info for the running server. This section will also show us how to create a client, which we will do across all examples here. This particular request does not require a token.

```
from spackmoncli import SpackMonitorClient
```

If we are using the server running on localhost, and the default endpoint, we don't need to customize the arguments.

```
client = SpackMonitorClient()
<spackmoncli.SpackMonitorClient at 0x7f24545fdb80>
```

However you could easily customize them as follows:

```
client = SpackMonitorClient(host="https://servername.org", prefix="ms2")
<spackmoncli.SpackMonitorClient at 0x7f24545fdb80>
```

Next, let's ping the service info endpoint.

```
client.service_info()
 {'id': 'spackmon',
 'status': 'running',
 'name': 'Spack Monitor (Spackmon)',
 'description': 'This service provides a database to monitor spack builds.',
 'organization': {'name': 'spack', 'url': 'https://github.com/spack'},
 'contactUrl': 'https://github.com/spack/spack-monitor/issues',
 'documentationUrl': 'https://spack-monitor.readthedocs.io',
 'createdAt': '2021-02-10T10:40:19Z',
 'updatedAt': '2021-02-11T00:06:06Z',
 'environment': 'test',
 'version': '0.0.1',
 'auth_instructions_url': 'https://spack-monitor.readthedocs.io/en/latest/getting_
↪started/auth.html'}
```

Note that we provide this example script service_info.py in the repository so you should be able to just run it to produce the example above:

```
$ python script/api-examples/service_info.py
```

Also take notice that we are running these scripts *outside of the container* as you'd imagine would be done with a service.

Upload Config Example

While most interactions with the API are going to come from spack, we do provide an equivalent example and endpoint to upload a spec file, verbatim. For this interaction, since we are modifying the database, you are required to export your token and username first:

```
$ export SPACKMON_TOKEN=50445263afd8f67e59bd79bff597836ee6c05438
$ export SPACKMON_USER=vsoch
```

For this example upload_config.py in the repository you'll see that by way of the spackmon client we find this token in the environment, and add it as a base64 encoded authorization header.

```
$ python script/api-examples/upload_config.py specs/singularity-3.6.4.json $(spack --
↪version)
```

If you run this inside the container, you can grab the version of spack from the host and use directly as a string:

```
$ echo $(spack --version)
$ python script/api-examples/upload_config.py specs/singularity-3.6.4.json 0.16.0-
↪1379-7a5351d495
```

If you haven't added it yet (the full hash of the first package in the file is the unique id) you'll see that it was added:

```
$ python script/api-examples/upload_config.py specs/singularity-3.6.4.json
The package was successfully created.
{
    "message": "success",
    "data": {
        "full_hash": "p64nmszwer36ly7pnch5fznni4cnmndg",
        "name": "singularity",
        "version": "3.6.4",
        "spack_version": "0.16.0-1379-7a5351d495",
        "specs": {
            "cryptsetup": "tmi4pf6umhalop7mi6zyiv7xjpalyzgb",
            "go": "dehg3ddu6gacrmnoexbxhjv2i2d76yq6",
            "libgpg-error": "4cvsg42wxksiup6x74mlabu6un55wjzc",
            "libseccomp": "kfx6zyjxzudw77e3xk6i73bcgi2cavgh",
            "pkgconf": "al2hlnux3cchfhwiv2sbejnxvnogibac",
            "shadow": "aozeq6ybtsnrs5phtonutwes7fe6yhcy",
            "squashfs": "vpemhhpzqqf7mvpzdvcg6szfah6mwt2q",
            "util-linux-uuid": "g362jjpzlfp3qhfm7gdery6v3xgeh3lg"
        }
    }
}
```

That's a hint of the metadata that can be returned to a calling client. In the context of spack, we actually don't need to pass around this metadata, because spack always carries a representation of a package's full hash and dependencies. If you've already added the package, you'll see:

```
$ python script/api-examples/upload_config.py specs/singularity-3.6.4.json $(spack --
↪version)
This package already exists in the database.
```

Local Save Upload Example

When you run spack install and ask the monitor to save local:

```
$ spack install --monitor --monitor-save-local singularity
```

This will generate a dated output directory in `~/.spack/reports/monitor` that you might want to upload later. You'll again want to export your credentials:

```
$ export SPACKMON_TOKEN=50445263afd8f67e59bd79bff597836ee6c05438
$ export SPACKMON_USER=vsoch
```

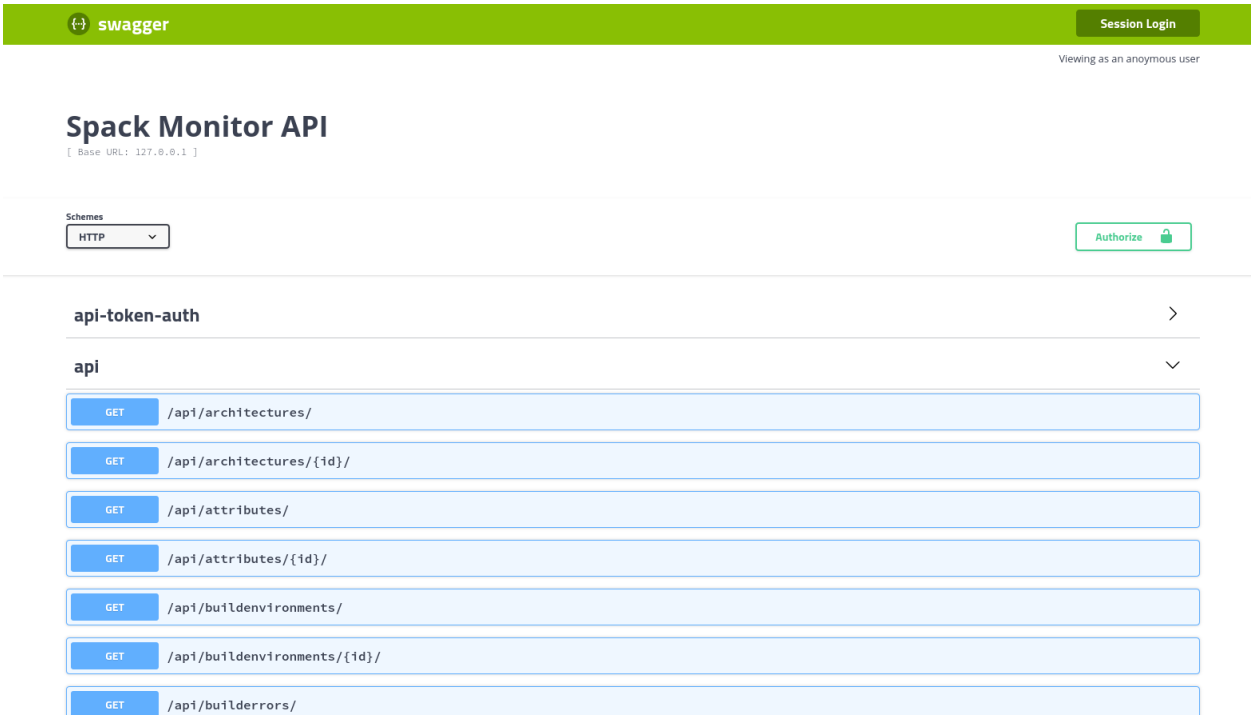
For this example upload_save_local.py in the repository you'll see that by way of the spackmon client we can do this upload.

```
$ python script/api-examples/upload_save_local.py ~/.spack/reports/monitor/2021-06-14-
↪17-02-27-1623711747/
```

In the above we run the script and provide the path to the directory we want to upload results for. The script will upload spec objects, then retrieve the build id, and finish up with phase logs and build statuses.

2.1.5 Data Application Programming Interface

Unlike the API that is used to interact with Spack Monitor from Spack, the data API exists only to expose data in Spack Monitor. For the time being, it is entirely public. If you want to explore data in spack monitor (or see endpoints that you can use from a client like Python or curl) you should browse to `api/docs` to see a full schema:



Generally, clicking on "api" and then the GET endpoint of your choosing will give you a preview, and then you can mimic the curl command or reproduce in Python. For example, for the builds endpoint, I can do this curl request for a local spack monitor:

Or in Python

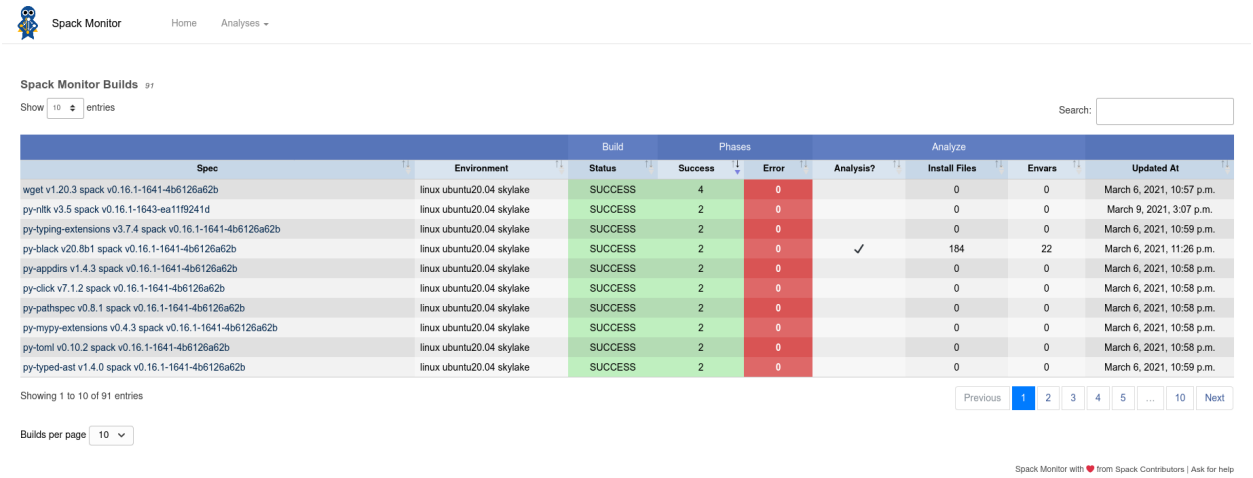
If appropriate, this can eventually be made into a client.

2.1.6 Interfaces

Spackmon currently has two simple views that will be expanded upon based on our needs.

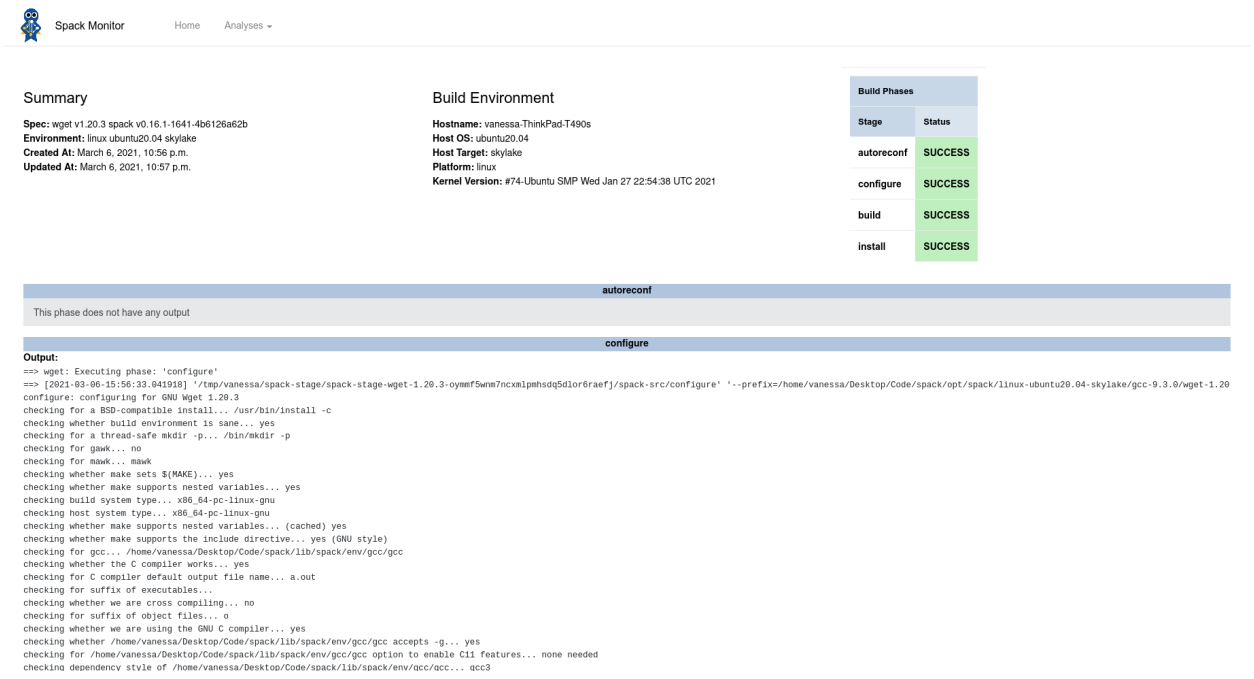
Home

The home view shows a table of current builds. This is rendered in the web page, and will be switched to server side rendering as the number of builds gets large. You can distinguish whether a build was just run (meaning it has phases and output) vs. added via an analysis (meaning we don't have build phase output, but analysis results) or both based on the Analysis column. In the future we might want a better way to distinguish these different types.



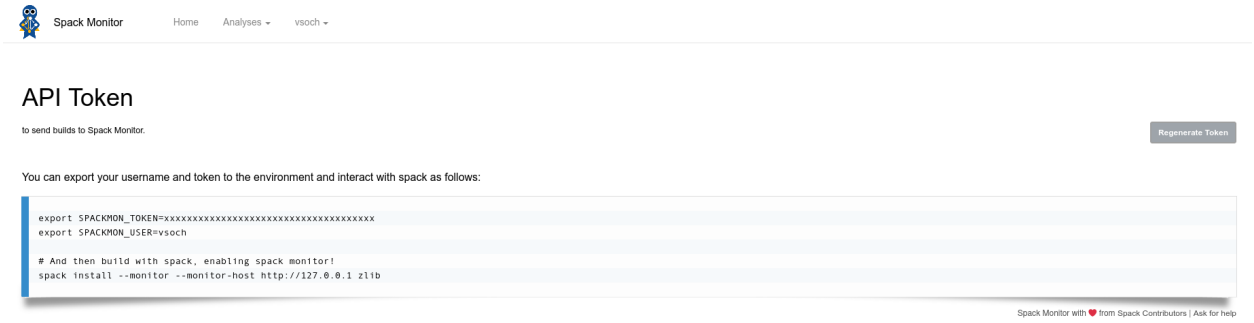
Build Interface

The build interface shows basic summary, install environment, and phase information, along with complete output for each phase (given that the build has associated phases).



User Account

When you login, you can view your token page, which shows how to export your username and token to the environment, and then interact with the Spack Monitor server from spack:



2.1.7 Settings

Settings are defined in the settings.yml file, and are automatically populated into Spack Monitor.

Table 1: Title

Name	Description	Default
GOOGLE_ANALYTICS_SITE	The site of your website for Google Analytics, if desired	None
GOOGLE_ANALYTICS_ID	The identifier for Google Analytics, if desired	None
TWITTER_USERNAME	A Twitter username to link to in the footer.	spackpm
GITHUB_REPOSITORY	A GitHub repository to link to in the footer	https://github.com/spack/spack-monitor
GITHUB_DOCUMENTATION	GitHub documentation (or other) to link to in the footer	https://spack-monitor.readthedocs.io
USE_SQLITE	Use an sqlite database instead of the postgres container (set to non null)	true
DISABLE_AUTHENTICATION	Don't require the user to provide a token in requests (set to non null)	None
ENVIRONMENT	The global name for the deployment environment (provided in service info metadata)	test
SENDGRID_API_KEY	Not in use yet, will allow sending email notifications	None
SENDGRID_SENDER_EMAIL	Not in use yet, will allow sending email notifications	None
DOMAIN_NAME	The server domain name, defaults to a localhost address	http://127.0.0.1
CACHE_DIR	Path to directory to use for cache, defaults to "cache" in root of directory	None
DISABLE_CACHE	Don't cache front end views	true
API_URL_PREFIX	The prefix to use for the API	ms1
API_TOKEN_EXPIRES_SECONDS	The expiration (in seconds) of an API token granted	ms1
AUTH_SERVER	Set to non null to define a custom authentication server	None
ENABLE_GITHUB_AUTH	Enable GitHub OAuth2 Authentication (requires environment secrets)	True
AUTH_INSTRUCTIONS	A link for the user to get authentication instructions	https://spack-monitor.readthedocs.io/en/latest/getting_started/auth.html

2.2 Development Guide

This is the Spackmon development guide, which will help you to use Spackmon as a developer, and provide background as to how it was originally development. If you have a question please let us know

2.2.1 Design of the Models

This document provides background information as to the technique that was used to design the original models.

Early Discussion

The first discussion between vsoch and tgamblin talked about how spack doesn't currently allow deployment of something it wasn't built with (but it's a work in progress. We'd want to do something called splicing, or cloning a node spec and then pointing it to a different dependency version, all the while preserving the provenance. Once this works, we would be able to do combinatorial builds and deployments. The discussion then moved into how we'd want to be able to put "all this stuff" into a database with some indexing and query strategy. On a high level, we want to say:

- Every graph is a configuration
- We can query by all dependencies that share dependency, or other parameters for a spec
- We want to index by, for example, the cray json document

An example query might be:

> Get me all records built with this version of package, deployed with this other version of package.

And it was also noted that eventually we will have database for abi, although this is another thing entirely. Later discussion with more team members we identified experiment information that would be important to represent:

- at least the spec
- status (success, or failure)
- the phase it failed
- errors and warnings
- parse environment to make models
- not the prefix, but possibly the hash
- .spack hidden folder in a package directory (note that if a build fails, we don't get a lot of metadata out.)
- granularity should be on level of package

For example, for each stage in configure, build, and test, we likely would want to store a spec,error, output, and possibly repos (or urls to them). For the install component, if it is successful, we might then have a manifest.

An *example script <https://github.com/spack/spack-buildspace-exploration/blob/main/spack_generate_random_specs.py>_* was provided that shows how to generate a random spec, and we modified this for the repository here to just save the spec to the filesystem. If you use this script, you should first have the spack bin on your path so the `spack-python` interpreter can be found. Then run the script providing a library name and output directory. E.g.,

We now have a spec (in json) that can be used to develop the models! The first goal would be to generate an endpoint that allows for uploading a model into the database. Once this basic structure is defined, we would want to review the models and discuss:

- Are the unique constraints appropriate for each model?
- Is the level of granularity appropriate (e.g., one model == one table, allowing for query)

- Are the `ON_DELETE` actions appropriate? (e.g., if I delete a model that depends on another, what happens?)
- Are the unique constraints appropriate? (e.g., this is what Django uses to tell if something already exists)
- Are the CharField lengths appropriate?
- Do any of the models need a "catch all" extra json field?

2.2.2 Development Setup

For using this software you will need:

- Docker
- docker-compose

Once you have these dependencies, you'll first want to build your container, which we might call *spack/spackmon*. You can reference *Bringing Up Containers* to see how to build and then start containers, and test basic import.

2.2.3 Tables

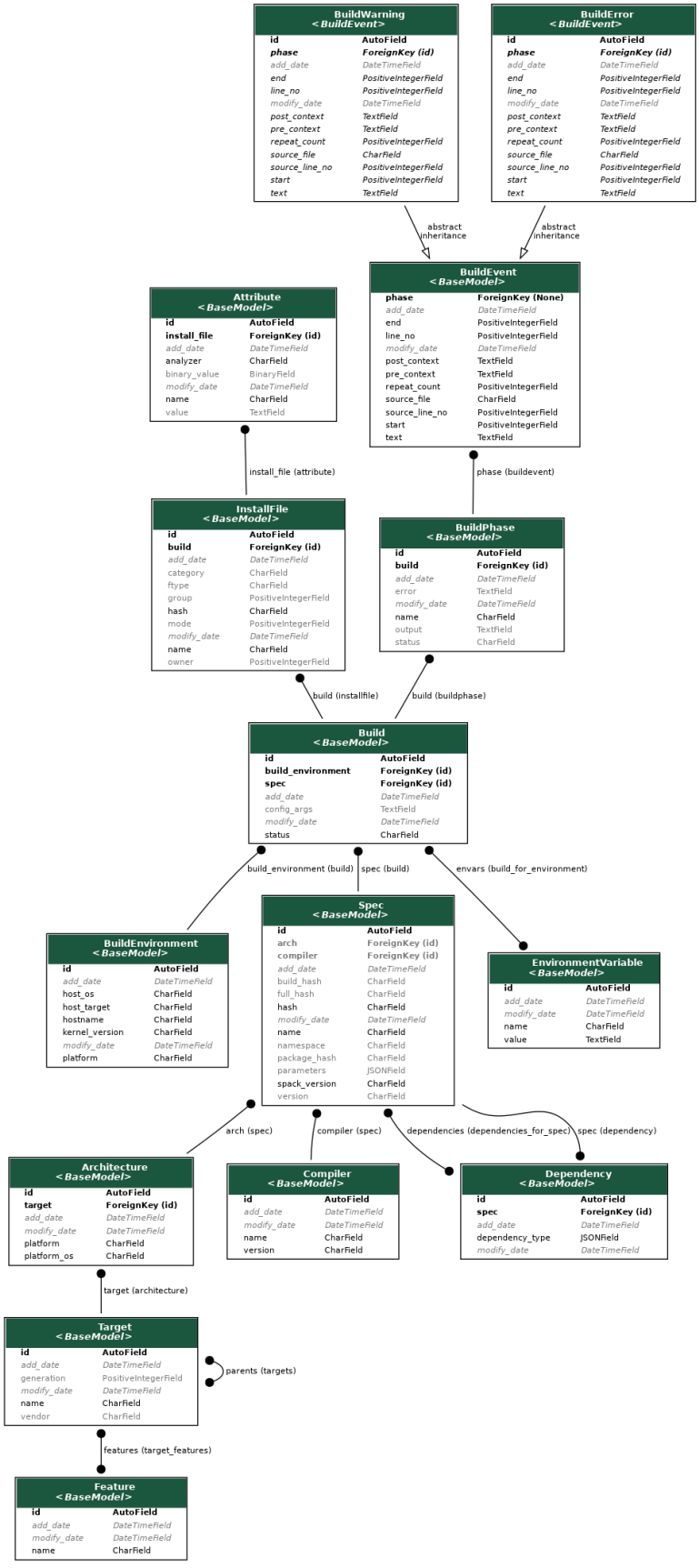
The table design is represented in the models.py file of each app. Once you have the application running, you can generate a graph as follows - we are only going to include models from main, and exclude the abstract BaseModel (which unnecessarily adds complexity to the diagram):

```
$ python manage.py graph_models main -X BaseModel -o tables.png
```

Or from the outside of the container:

```
$ docker exec -it spack-monitor_uwsgi_1 python manage.py graph_models -X BaseModel␣
↪main -o tables.png
$ mv tables.png docs/development/img/
```

The output looks like this:


BuildWarning
< BuildEvent>
id AutoField
phase ForeignKey (id)
add_date DateTimeField
end PositiveIntegerField
line_no PositiveIntegerField
modify_date DateTimeField
post_context TextField
pre_context TextField
repeat_count PositiveIntegerField
source_file CharField
source_line_no PositiveIntegerField
start PositiveIntegerField
text TextField
BuildError
< BuildEvent>
id AutoField
phase ForeignKey (id)
add_date DateTimeField
end PositiveIntegerField
line_no PositiveIntegerField
modify_date DateTimeField
post_context TextField
pre_context TextField
repeat_count PositiveIntegerField
source_file CharField
source_line_no PositiveIntegerField
start PositiveIntegerField
text TextField
abstract inheritance
abstract inheritance
BuildEvent
< BaseModel>
phase ForeignKey (None)
add_date DateTimeField
end PositiveIntegerField
line_no PositiveIntegerField
modify_date DateTimeField
post_context TextField
pre_context TextField
repeat_count PositiveIntegerField
source_file CharField
source_line_no PositiveIntegerField
start PositiveIntegerField
text TextField
Attribute
< BaseModel>
id AutoField
install_file ForeignKey (id)
add_date DateTimeField
analyzer CharField
binary_value BinaryField
modify_date DateTimeField
name CharField
value TextField
install_file (attribute)
phase (buildevent)
InstallFile
< BaseModel>
id AutoField
build ForeignKey (id)
add_date DateTimeField
category CharField
ftype CharField
group PositiveIntegerField
hash CharField
mode PositiveIntegerField
modify_date DateTimeField
name CharField
owner PositiveIntegerField
BuildPhase
< BaseModel>
id AutoField
build ForeignKey (id)
add_date DateTimeField
error TextField
modify_date DateTimeField
name CharField
output TextField
status CharField
build (installfile)
build (buildphase)
Build
< BaseModel>
id AutoField
build_environment ForeignKey (id)
spec ForeignKey (id)
add_date DateTimeField
config_args TextField
modify_date DateTimeField
status CharField
build_environment (build)
spec (build)
envars (build_for_environment)
BuildEnvironment
< BaseModel>
id AutoField
add_date DateTimeField
host_os CharField
host_target CharField
hostname CharField
kernel_version CharField
modify_date DateTimeField
platform CharField
Spec
< BaseModel>
id AutoField
arch ForeignKey (id)
compiler ForeignKey (id)
add_date DateTimeField
build_hash CharField
full_hash CharField
hash CharField
modify_date DateTimeField
name CharField
namespace CharField
package_hash CharField
parameters JSONField
spack_version CharField
version CharField
EnvironmentVariable
< BaseModel>
id AutoField
add_date DateTimeField
modify_date DateTimeField
name CharField
value TextField
arch (spec)
compiler (spec)
dependencies (dependencies_for_spec)
spec (dependency)
Architecture
< BaseModel>
id AutoField
target ForeignKey (id)
add_date DateTimeField
modify_date DateTimeField
platform CharField
platform_os CharField
Compiler
< BaseModel>
id AutoField
add_date DateTimeField
modify_date DateTimeField
name CharField
version CharField
Dependency
< BaseModel>
id AutoField
spec ForeignKey (id)
add_date DateTimeField
dependency_type JSONField
modify_date DateTimeField
target (architecture)
Target
< BaseModel>
id AutoField
add_date DateTimeField
generation PositiveIntegerField
modify_date DateTimeField
name CharField
vendor CharField
parents (targets)
features (target_features)
Feature
< BaseModel>
id AutoField
add_date DateTimeField
modify_date DateTimeField
name CharField

2.2.4 Documentation

The documentation here is generated by way of sphinx and a few other dependencies. You can generally cd into the docs folder, install requirements, and then build:

After build, the documentation will be in the `_build/html` folder. You can cd to that location and deploy a local webserver to view it:

For the above, you would open to port 9999 to see the rendered docs. This short guide is me taking notes to prepare for an analysis that can look across versions and compilers. For my first test I want to install zlib across several versions of a package and compilers. Note that this requires this branch.

2.2.5 Setup

Before doing this, you should have exported a spack monitor token and username in your envrionment.

```
SPACKMON_TOKEN=xxxxxxxxxxxxxxxxxxxxxxxxxxxx
SPACKMON_USER=vsoch
```

We also need to ensure that everything builds with debug.

```
$ . /share/spack/setup-env.sh
export SPACK_ADD_DEBUG_FLAGS=true
```

2.2.6 Test Smeagle

And then here is how to install zlib with using spack monitor (locally) and then running the analyzer for the same set:

```
# Install ALL versions of zlib with default compiler
$ spack  install --monitor --all --monitor-tag smeagle zlib

# Analyze all versions of zlib plus recursive dependents
$ spack analyze --monitor run --analyzer smeagle --recursive --all zlib
```

I did a sanity check to see the results in the database:

```
$ docker exec -it uwsgi bash
p(sm) root@8a3433dedba8:/code# python manage.py shell
Python 3.8.12 | packaged by conda-forge | (default, Oct 12 2021, 21:59:51)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.28.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from spackmon.apps.main.models import Attribute

In [2]: Attribute.objects.all()
Out[2]: <QuerySet [<Attribute: Attribute object (1)>, <Attribute: Attribute object␣
↪(2)>]>

In [3]: Attribute.objects.first()
Out[3]: <Attribute: Attribute object (1)>

In [4]: Attribute.objects.first().json_value
Out[4]:
{'library': '/data/libz.so.1.2.11',
 'locations': [{'function': {'name': 'free',
```

(continues on next page)

(continued from previous page)

```
    'type': 'Function',
    'direction': 'import'}},
  {'function': {'name': '__errno_location',
    'type': 'Function',
    'direction': 'import'}},
  {'function': {'name': 'write', 'type': 'Function', 'direction': 'import'}},
  {'function': {'name': 'strlen', 'type': 'Function', 'direction': 'import'}},
```

Yes!

2.2.7 Change Compiler

And now we want to install the same versions of zlib with different compilers.

```
# Install ALL versions of zlib
$ spack  install --monitor --all --monitor-tag smeagle zlib %gcc@8.4.0
$ spack  install --monitor --all --monitor-tag smeagle zlib %gcc@7.5.0

# Analyze all versions of zlib plus recursive dependents
$ spack analyze --monitor run --analyzer smeagle --recursive --all zlib%gcc@8.4.0
$ spack analyze --monitor run --analyzer smeagle --recursive --all zlib%gcc@7.5.0
```

2.2.8 Test Symbolator

Let's now do the same, but using the symbolator analyzer (we already have them installed):

```
# Analyze all versions of zlib plus recursive dependents
$ spack analyze --monitor run --analyzer symbolator --recursive --all zlib
$ spack analyze --monitor run --analyzer symbolator --recursive --all zlib%gcc@8.4.0
$ spack analyze --monitor run --analyzer symbolator --recursive --all zlib%gcc@7.5.0
$ spack analyze --monitor run --analyzer symbolator --recursive --all curl
$ spack analyze --monitor run --analyzer symbolator --recursive --all curl%gcc@8.4.0
$ spack analyze --monitor run --analyzer symbolator --recursive --all curl%gcc@7.5.0
```

For splice analysis examples, see the **`api-examples folder _<https://github.com/spack/spack-monitor/tree/main/script/api-examples>`_**.

2.3 Deployment Guide

This section includes deployment information for Spackmon, outside of your host with docker-compose. If you have a question please let us know

2.3.1 Amazon Web Services (AWS)

This tutorial requires an AWS account. You'll need the following dependencies on the instance:

- Docker
- docker-compose

Setting up AWS

Amazon allows you to run instances on a service called EC2 or lightsail.

Lightsail

Lightsail is much easier to setup. You can select to create a new Linux instance, selecting to choose the OS and then choosing Ubuntu 20.04, and then download the default key. You'll need to change permissions for it:

```
$ chmod 400 ~/.ssh/Lightsail*.pem
```

And then follow the instruction to ssh to it (the username is ubuntu)

```
$ ssh -v -i "~/.ssh/Lightsail-<keyhname>.pem" ubuntu@<ipaddress>
```

You'll want to register a static IP, from the Networking tab, otherwise we cannot associate a domain. You'll also want to add an HTTPS rule to networking, unless you don't plan on setting up https. You can now jump down to the Install Dependencies section.

EC2

You'll want to navigate in your cloud console to Compute -> EC2 and then select "Launch Instance."

- Select a base OS that you are comfortable with. I selected "Ubuntu Server 20.04 LTS (HVM), SSD Volume Type - ami-042e8287309f5df03 (64-bit x86) / ami-0b75998a97c952252 (64-bit Arm)"
- For size I selected t2.xlarge.
- Select Step 3. Configure Instance Details. In this screen everything is okay, and I selected to enable termination protection. Under tenancy I first had wanted "Dedicated - Run a Dedicated Instance" but ultimately chose the default (Shared) as Dedicated was not an allowed configuration at the time.
- Select Step 4. Add Storage. For this I just increased the root filesystem to 100 GB. This instruction is for a test instance so I didn't think we needed to add additional filesystems.
- IAM roles: I created a new one with the role that was for an llnl admin, with access to EC2.
- Under Step 5. Tags - you can add tags that you think are useful! I usually add service:spack-monitor and creator (my name).
- Step 6. Configure Security group: give the group a name that will be easy to associate (e.g., spack-monitor-security-group). Make sure to add rules for exposing ports 80 and 443 for the web interface.

When you click "Review and Launch" there will be a popup generated about a keypair. If you don't have one, you should generate it and save to your local machine. You will need this key pair to access the instance with ssh. I typically move the pem keys to my ~/.ssh folder.

Finally, you'll want to follow instructions here to generate a static ip address and associate it with your instance. This will ensure that if you map a domain name to it, the ip address won't change if you stop and restart.

To connect to your instance, the easiest thing to do is click "View Instances," and then search for your instance by metadata or tags (e.g., I searched for "spack-monitor"). You can also rename your instance to call it something more meaningful (e.g., spack-monitor-dev). If you then click the instance ID, it will take you to a details page, and you can click "Connect" and then the tab for SSH. First, you will be instructed to change the permissions of your key:

```
$ chmod 400 ~/.ssh/myusername-spack-monitor.pem
```

Important if you are on VPN, you wil need to request a firewall exception to connect via ssh. Otherwise, you can follow the instructions on this page to:

1. stop the instance
2. edit user data to reset the ssh service
3. start the instance
4. connect via a session

Sessions are in the browser instead of a terminal, but can work if you are absolutely desparate! Otherwise, if you are off VPN or have an exception, you can do:

```
$ ssh -v -i "~/.ssh/username-spack-monitor.pem" ubuntu@ec2-XX-XXX-XXX-X.compute-1.
↪amazonaws.com
```

Install Dependencies

Once connected to the instance, here is a basic script to t your dependencies installed. If you log in via ssh, you should be able to exit and connect again and not need sudo.

Build and Start Spack Monitor

It looks like nginx is installed on the instance by default, so you will need to stop it.

```
$ sudo service nginx stop
```

And then build and bring up spack monitor:

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

After creation, if you need an interactive shell:

```
$ docker exec -it spack-monitor_uwsgi_1 bash
```

or to see logs

```
$ docker-compose logs uwsgi
```

You can then check that the instance interface is live (without https). When that is done, use this script to set up https. This means that you've registered a static IP, have a domain somewhere where you've associated it, and then are able to generate certificates for it. After generating the certificates, you will want to use the docker-compose.yml and nginx.conf in the https folder.

You can reference *Bringing Up Containers* for more details.

Update Settings

Finally, you'll want to ensure that you update the list of allowed hosts to include localhost and your custom domain, turn the application on debug mode, ensure https only is used, and (if you want) enable timezone support to the `settings.py` file.

```
ALLOWED_HOSTS = ["localhost", "127.0.0.1", "monitor.spack.io"]
USE_TZ = True
SECURE_SSL_REDIRECT = True
DEBUG = False
```