ResearchCompendia Documentation

Release 1.3.1

ResearchCompendia Contributors

June 03, 2014

Contents

Contents:

CHAPTER 1

ResearchCompendia

A proof of concept for a research compendia webapp.

1.1 Introduction and Goals

This is a project to allow scientists to create research compendium [1] comprising all relevant narrative, code, and data to make their research truly reproducible. Our goal is allow and teach researchers to document the computational portions of their research methods as thoroughly as they would document a tabletop experiment. We want our tools to fulfill these goals:

The application has the following goals.

- We will make it possible to archive all of the data, codes, documentation, parameters, and environmental settings linked with published research in a versioned form.
- We will support the verification and validation processes by providing for the execution of shared code and the visualization of results.
- We want to help and encourage researchers to manage their research in a way that makes it mixable and executable.
- Most of all we wish to make these tools heavily automated, and easy to access and utilize to lessen the exertion required from already overburdened academic researchers in the process of publishing fully reproducible work.

Imagine if all the materials in a research project could be continuously packaged and deployed with no snags preventing use and refinement by anyone. We could help make research accessible to everyone.

1.2 Project Structure

This is a django project with the following structure.

- *home*: this handles the landing page, faq, and similar concerns that don't call for separate apps.
- *users*: this handles users and profiles by using django-allauth and cookiecutter-django's user template
- *compendia*: this handles the archiving and representation of a compendium.
- *lib*: this holds code that does not call for an app
- *api*: this handles our service apis.

[1] Gentleman, Robert, and Duncan Temple Lang. 2007. "Statistical Analyses and Reproducible Research." Journal of Computational and Graphical Statistics 16 (1): 1–23. doi:10.1198/106186007X178663. http://www.tandfonline.com/doi/abs/10.1198/106186007X178663.

1.3 Resources

- Free software: MIT License
- Technical Documentation: http://researchcompendia.readthedocs.org
- Issue tracker: https://github.com/researchcompendia/researchcompendia/issues
- Wiki: https://github.com/researchcompendia/researchcompendia/wiki
- IRC: #researchcompendia

1.3.1 Development Environments

- http://researchcompendia.org
- http://labs.researchcompendia.org

1.4 Acknowledgements

Make a separate acknowledgements page?

1.5 References

CHAPTER 2

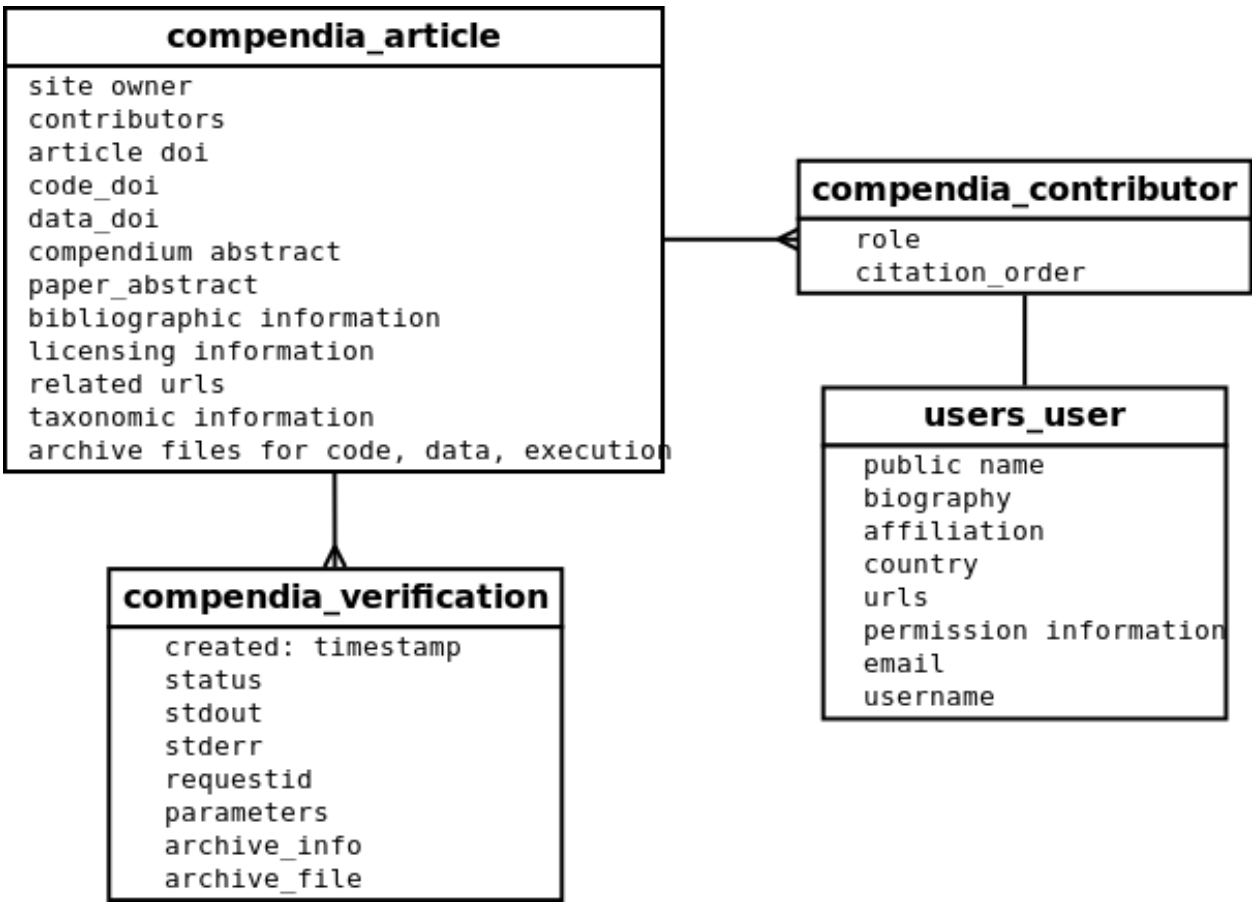
Project Structure

ResearchCompendia is a Django project that contains apps for organizing users and their compendia. It also has the basic functionality provided by Django and many common Django packages.

- *compendia*: this handles the presentation and archiving of a research compendium.
- *api*: this handles our service apis.
- *lib*: this holds utilities that do not call for an app
- *home*: this handles the landing page, faq, and similar concerns that don't call for separate apps.
- *users*: this handles users and profiles

2.1 Compendia

For this prototype, we've started with a few simple models to represent research compendia and related metadata. Here are a few of the most salient models.



The starting point for a compendium is an Article. An Article contains high level information about a compendium.

- reference information to the original material for use in citations
- a "code and data abstract" description
- links to archives of compendium materials such as code archives, data archives, and documentation
- link to archive of the *verification package*. This is only editable by admins.
- DOI for the original material as well as DOIs for the compendium materials.
- taxonomic information
- licenses for code and data

Articles with verification packages can be run via the verification service API. Each run of an article results in a Verification record that links back to its Article. Our prototype for now stores only a few fields. In the future, ResearchCompendia will use Sumatra as a service and will consume serialized Sumatra records for each run. At which point we will deprecate our own verification model.

2.2 API

2.2.1 /api/v1/dois/

This service accepts a json dictionary *{ 'doi': <doi> }* and performs a crossref request to return a json dictionary corresponding to compendium attributes. This is used on the compendium creation page to auto-populate values. I

made it an API rather than a part of the compendia creation view so that we wouldn't make the user wait on the django framework for populating the form in the backend before returning it to the template.

2.2.2 /api/v1/verification/*id*/

We have a primitive verification API implementation proof-of-concept that was done mostly to test out interface design rather than serve as an example of how to do verifications. See *Design Limitations* and *Future Plans* below for more details on the next verification proof-of-concept.

- POST: submits a request to create a new Verification for Article *id*

GET

GET returns a json representations of the most recent Verification results for Article *id*

curl -X GET "http://hostname/api/v1/verification/11/"

Example:

```
{"verifications":
  [
    {
      "archive_file_url": "/media/results/025a8ae48fdbd220fccccb879f4d1b4e-2014-04-03-15-42-37/verifi
      "archive_info": {
        "output_files": [ { "bytes": 101, "file": "pizza_order.json", "size": "101B" } ]
      },
      "created": "2014-04-03T20:42:37.288Z",
      "id": 52,
      "parameters": {},
      "requestid": "messageidnotusedyet",
      "status": "unknown",
      "stderr": "",
      "stdout": "{'attending': 33, 'pizzas': {'cheese': 3, 'meat': 3, 'vegan': 1, 'veg': 4}}\n\n
    },
    // and so on ...
  ]
}
```

POST

A POST request for an Article with a verification package will trigger an execution to create a new Verification.

This example request is a no-op since no new values are passed in the *parameters* field. It will just return a message about default parameters being used.

curl -X POST "http://hostname/api/v1/verification/11/":

```
{"message": "Request was made with default parameters. Fetched cached results."}
```

This example triggers an actual run even though the *parameters* field is empty since the functionality to check cached parameters is not yet built. If anything is passed, the code runs.

curl -v -X POST –data parameters="" "http://hostname/api/v1/verification/11/":

```
{
  "message": "ok",
  "output_dir": "/tmp/compendia4NlIJ9/hellopizza/compendiaoutput",
```

```
  "output_files": [],
  "path_to_zipped_output": "/tmp/compendia4NlIJ9/hellopizza/compendiaoutput.zip",
  "requestid": "messageidnotusedyet",
  "status": 201,
  "stderr": "Traceback (most recent call last):\n  File \"/tmp/compendia4NlIJ9/hellopizza/main\"
  "stdout": "",
  "zipbytes": 22,
  "zipsize": "22B"
}
```


Verification Package

Verification packages are created by administators based on the code and data archives provided by authors. Example verification packages can be found in our github repo, researchcompendia/meta-analyses. This repo is a fork of Tim Churches's repo containing meta-analyses on the benefits of reproducible research. This fork adds some verification scaffolding that we've packaged up to use as an example. A simpler example is available in this gist.

The structure of a verification package

- main: an executable called main that can be invoked
- default.json: a json file that contains default parameters
- compendiaoutput/: a directory where main deposits results
- A build mechanism that creates main (and is able to pull in specific dependencies)

Design Limitations

For now the verification service is not a real service. The logic lives in a verification utility inside of the django app.

The current implementation was done in a one-off demo to demonstrate a request/response. It has severe limitations, and is absolutely not production ready. For the demo, the api request kicks of a verification library call, and blocks until the job is finished. The result is persisted by our django ORM, and our django app archives the result files.

- It is synchronous and blocking.
- It does not enforce SLAs
- It does not use sandboxes.
- It only runs on the machine that the webapp is deployed to.
- It only works with the default django file storages system (problems with s3 backed storages)
- It requires manual work for installing dependencies.
- It requires manual work for creating verification packages, and this will be confusing to users.
- etc.

Future Plans

A sensible glimpse in to the future can see that the verification functionality will change such that

- it runs apart from the django app
- it supports asynchronous requests
- it uses sandboxing

- it has a saner method for dependency management
- it enforces SLAs

Moving the library to a service

Our verification library could be pulled out of this django project and turned in to a separate component that can be called as a service. The current django app passes a dictionary to the verification library since this is easily changed to a json message. For example, instead of calling a lib, it can be changed to make HTTP requests to the service.

Asynchronous requests

One approach for handling requests in a non-blocking fashion would be to use Celery for queuing tasks. We already use Celery for handling our link checking jobs, and I've been planning to do this for the next verification proof-of-concept.

Sandboxing and dependency management

We've been experimenting with using Docker for handling sandboxing as well as dependency management. For another verification proof-of-concept, we could create a lightweight service that talks to the docker api – actually, while I was spending time working on less fun features, someone already did an example of this and perhaps we could just take advantage of his project, Spin-docker.

VMs and containers aren't a silver bullet for deploying reproducible environments, and I should link to some discussion on all of this. TODO

Usability? Ha. I was thinking we'd need to do a lot of hand-holding at first, and also that we'd write up cookiecutters to help people generate skeleton packages (and also to automate that step when possible). We need to watch users attempting to use the system to advance from there.

Enforcing SLAs

If we continue to use the Django REST Framework we can take advantage of its ability to handle authentication, permissions, and throttling.

CHAPTER 3

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. You can contribute in many ways.

3.1 Contributing Code

Before you get started, remember that it is easier for us to accept pull requests that are narrow in scope and easy to review. Ideally we want to see pull requests with a commit that is a logical changeset so plan to work in this fashion.

When committing a change, please include the issue number in your commit comment. This helps us track the progress in the related issue. This works like so

> $ git commit CONTRIBUTING.rst -m 'improved the contributing docs for #1'

TODO: define some guidelines for what makes something easy to review.

Getting Started with Development

3.1.1 Pick up labeled tasks

We label some issues to guide contributions.

bitesized These bugs can be done by people with a beginning level of skill.

intermediate These bugs can be done easily by people with an intermediate level or skill or by patient beginners who get frequent review.

fly-by These can be done by people who are experts but don't have much time to devote to long term tasks.

brainstorming These are tasks where we welcome discussion about the ideas mentioned in the issue

If you see a task that is not already being worked on, feel free to claim it by leaving a comment and start working.

For more advanced tasks and tasks without these labels, please talk to us first.

3.2 Reviewing Code and Concepts

We are always learning. Review code in our repository and suggest improvements and alternatives to our approaches.

Look through pull requests and review the changes.

Help us by discussing issues we've tagged with the brainstorming label

3.3 Reporting Bugs

Report bugs in our issue tracker. https://github.com/researchcompendia/researchcompendia/issues.

If you are reporting a bug, please include:

- Your browser information.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

3.4 Give us feeback

Talk to us about features you'd like. Let us know how we are doing. You can send us email from the contact form or let us know by filing an issue in the issue tracker.

3.5 Write Documentation

We could always use more documentation, whether as part of the official docs, in docstrings, or even on the web in blog posts, articles, and such.

CHAPTER 4

Getting Started with Development

Note: If you have just started, this may seem confusing, and that is okay. No one starts out understanding how to do all of this. Additionally, anything that is confusing is a bug in our docs. Please open an issue to let us know where we need to improve things.

Ready to contribute code? Here's how to set up *ResearchCompendia* for local development.

4.1 Getting Code

Fork the repository and check out your fork and add our repo as a remote:

```
$ git clone https://github.com/YOURACCOUNT/researchcompendia.git
$ cd researchcompendia
$ git remote add parent https://github.com/researchcompendia/researchcompendia.git
```

4.2 Installing Dependencies

1. Install your local copy into a virtualenv. Here is one way to do it:

   ```
   $ cd researchcompendia/
   $ virtualenv venv
   $ source venv/bin/activate
   $ pip install -r requirements/ci.txt
   ```

   You can also use virtualenvwrapper if you have it is installed. It is convenient, though not required.

2. Set up environment variables. Once you have the environment varialbes set up, you are ready to set up the database.

3. We use Postgres as our database backend. We haven't written docs to walk though setting up Postgres yet. If you are unfamiliar with this process, you could consider using sqlite locally event thought our preferences is to use postgres in all environments. If you'd like to use sqlite, set up your DATABASE_URL environment variable to have a path to a file name that will hold your database. for example:

   ```
   'sqlite:////path/to/my/site/root/researchcompendia.db'
   ```

4.3 Setting up the Environment

These are the environment variables that are used in the site settings listed along with their defaults. For any default that doesn't apply to you, make an environment variable with your preferred setting.

You are required to create SECRET_KEY environment variable.

If you use fabric to provision a vagrant box, it will generate a SECRET_KEY for you.

Environment Variable	Default Setting
ADMINS	compendia@example.com
AWS_ACCESS_KEY_ID	''
AWS_SECRET_ACCESS_KEY	''
AWS_STORAGE_BUCKET_NAME	compendiaexamplebucket
DATABASE_URL	postgres://:5432/researchcompendia
DJANGO_BROKER_URL	hamqp://guest:guest@localhost:5672//
DJANGO_CELERY_DISABLE_RATE_LIMITS	True
DJANGO_CELERY_RESULT_BACKEND	cache+memcached://127.0.0.1:11211/
DJANGO_CELERY_RESULT_SERIALIZER	json
DJANGO_CELERY_TASK_SERIALIZER	json
DJANGO_CELERY_TIMEZONE	US/Central
CROSSREF_PID	''
DEBUG	True
DEFAULT_FILE_STORAGE	django.core.files.storage.FileSystemStorage
DEFAULT_FROM_EMAIL	compendia@example.com
DISQUS_API_KEY	'none'
DISQUS_WEBSITE_SHORTNAME	researchcompendiaorg
BONSAI_URL	http://127.0.0.1:9200
EMAIL_BACKEND	django.core.mail.backends.console.EmailBackend
MAILGUN_ACCESS_KEY	''
MAILGUN_SERVER_NAME	''
MEDIA_ROOT	normpath(join(PROJECT_ROOT, 'media'))
MEDIA_URL	if using the s3boto storages then '%s/media/' % S3_URL, otherwise /media/
REMOTE_DEBUG	False
SECRET_KEY	no default. will blow up if not set
SITE_ID	1
STATICFILES_STORAGE	django.contrib.staticfiles.storage.StaticFilesStorage
STATIC_ROOT	normpath(join(PROJECT_ROOT, 'staticfiles'))
STATIC_URL	if using the s3boto storages then '%s/static/' % S3_URL, otherwise /static/

There are a couple of places where settings are hard-coded in to templates. These need to be fixed. Meanwhile, you will need to change or remove the google analytics and addthis pubid codes.

- Our google analytics tracking code is hardcoded in templates/base.html.
- Our addthis pubid is hardcoded in templates/compendia/detail.html.

4.4 Preparing the Database

Set up the database by running:

```
$ cd companionpages
$ ./manage.py syncdb --migrate
$ ./manage.py loaddata fixtures/*
```

4.5 Launching ResearchCompendia

Once the database is set up, you can start the app:

```
$ ./manage.py runserver
```

Or perhaps you would like to have detailed stacktraces and messages:

```
$ ./manage.py runserver --traceback -v 3
```

4.6 Making Changes

Now that you have the code, a virtualenv, and the proper environment variables, you are ready to make your changes locally.

1. Make a topic branch for your changes. For example, if you wanted to add twitter logins to the site, you could make a branch named *twitterlogin*:

   ```
   $ git checkout -b twitterlogin
   ```

2. Periodically update your branch from the parent develop branch. Use git rebase (not git merge):

   ```
   $ git fetch parent
   $ git rebase parent/develop
   ```

   We prefer a pull request with one commit rather than many small commits. To avoid making a request with many commits, you can do an interactive rebase and use fixup.:

   ```
   $ git rebase -i parent/develop
   ```

3. Check that your changes pass style check and automated tests:

   ```
   $ make test
   ```

4. Demonstrate your changes. It can be helpful to share work you are running locally from your own machine so that other people can help test. PageKite is a free/libre open source software project that can do this for you. This QuickStart shows how.

5. Commit your changes and push your branch to up to your fork on GitHub.:

   ```
   $ git add .
   $ git commit -m "Adds twitter login for #123"
   $ git push origin twitterlogin
   ```

Now you are ready to make a pull request.

4.7 Reviewing Changes

Submit a pull request through the GitHub website to submit it for review. Before you submit a pull request, check that it meets these guidelines:

0. The pull request should be easy to review.
1. The pull request should include tests
2. Check https://travis-ci.org/researchcompendia/researchcompendia/pull_requests and make sure that the tests pass
3. If the pull request adds functionality, the docs and/or comments should be updated.

4.8 Trying out Vagrant

Note: This section is for developers who have experience with Vagrant and Fabric

If you want to use Vagrant clone the researchcompendia-deployment repo. It contains fabric files and a Vagrantfile that pulls down a debian wheezy VM from vagrantcound:

```
$ git clone https://github.com/researchcompendia/researchcompendia-deployment.git
$ cd researchcompendia-deployment
$ vagrant up
$ fab vagrant provision
```

Provision is not idempotent, so running it twice will probably fail in interesting ways. If you want to start over need to run *vagrant destroy* first.

Provision will set up the vagrant box in the same way that a production box is set up.

CHAPTER 5

Deployment

Note: If you are not a core developer you likely do not need these instructions. These are instructions for production deployment. You may be looking for: *Getting Started with Development*

Warning: This process is in flux and not fully automated.

5.1 Initial Setup

Clone the researchcompendia-deployment repo:

```
$ git clone https://github.com/researchcompendia/researchcompendia-deployment.git
$ cd researchcompendia-deployment
$ virtualenv venv
$ source venv/bin/activate
$ pip install -r requirements.txt
```

The top of the fabric.py file has settings for hostnames, port numbers, etc. Check these to verify that they apply to you. If they do not, change them as appropriate. Each fabric command should be prefaced with the name of the environment, *dev*, *staging*, *prod*, *vagrant* are available environments. If this is the first time you are setting up a box, run the provision task. This example provisions staging:

```
(venv)$ fab staging provision
```

Until the deployment and configuration process completely automated, there are manual steps to go through for a first install and deployment. You'll want to create a django superuser. Log in to the box, sudo su tyler, source the environment variables, run the createsuperuser command:

```
./manage.py createsuperuser
```

5.2 Creating A Release

Releases are created with the *git flow release* subcommands (part of git-flow).

Version numbers: We use a semantic versioning scheme.

All of the changes you want to go in to a release should be in the *development* branch. Once everything is there run (the version for this example is 1.0.0):

```
$ git flow release start 1.0.0
```

Once you've started a release, edit the HISTORY.rst file and bump the version in __init__.py and commit the changes. Then run:

```
$ git flow release finish 1.0.0
```

This will merge everything in develop to master and create a tag for you. Once that is done, you need to push the changes:

```
$ git checkout master
$ git push
$ git push --tags
$ git checkout develop
$ git push
```

5.3 Deploying a Release

Check release notes for any required updates to environment variables, database migrations, static files changes. Activate your researchcompendia-deployment virtualenv and run the deploy command:

```
(venv)$ fab staging deploy:1.0.0
```

That command is for the simplest case of a change. It doesn't migrate the database, for example. There is a fab command for that, *migrate*

5.4 Miscellanea

Convenient packages like *htop* and *multitail* are installed. Use *sudo htop* for a handy way to observe and control running processes.

The provision task creates a directory layout in the tyler user's home directory organized in the following way:

```
$ tree -L 2 site
site
-- bin
|   -- celeryworker.sh
|   -- check_downloads.sh
|   -- environment.sh
|   -- runserver.sh
-- logs
|   -- log_files.yml
-- tyler
```

Logs for nginx, celery, gunicorn, supervisor, cron, django are in the logs/ directory.:

```
logs
-- celery_worker.log         logs for celery and tasks
-- cron_checkdownloads.log   logs to see that the download link checker was called
-- gunicorn_supervisor.log   gunicorn/django console logs
-- log_files.yml             papertrail remote_syslog config file
-- tyler.access.log          nginx access log
-- tyler.error.log           nginx error log
```

Remote logging, the webapp, and celery are controled by supervisor. run *sudo supervisorctl status* to see a list of statuses.:

```
$ sudo supervisorctl status
celery                           EXITED     Jan 16 11:21 PM
remote_syslog                    RUNNING    pid 13411, uptime 1 day, 0:05:17
researchcompendia                RUNNING    pid 13828, uptime 1 day, 0:01:17
```

CHAPTER 6

Credits

6.1 Core Team

- Sheila Miguez <sheila@researchcompendia.org>
- Jennifer Seiler
- Victoria Stodden

6.2 Contributors

- @benmarwick
- You?

CHAPTER 7

History

7.1 1.3.1 (2014-06-03)

- Fixes #196, /search/ without query parameters caused a 500.

7.2 1.3.0

- Adds a Demos page where we can list ongoing demos. Currently we have a Table of Contents demo and some executability demos.
- The demo Table of Contents demo allows for an admin to generate a table of contents based on entries that categorize compendia types. The current demo shows a result card style with no stripped down informationc ompared to the main site result card style.
- Adds facets based on compendium type and primary research fields. These are stackable in the url, but the UI only drills down via links for now.
- Upgrades insecure requirements. Started tracking requirements via the https://requires.io service.
- Adds microformat to our header for rel-vcs as specified by https://joeyh.name/rfc/rel-vcs/
- template refactoring – pulled out some browse, facet, and pagination code in to separate files to be included in other templates. DRY
- minor style changes

7.3 1.2.1 (2014-05-12)

7.3.1 New Features and content

- Facetted search based on primary research field and compendium type #184
- Started Project Structure docs

7.3.2 Fixes

- User detail page broken due to old url pattern #182

7.4 1.2.0 (2014-04-14)

- First pull request from an external contributor! #168 fixes two typos in the FAQ. Thanks @benmarwick.
- First iteration with execution with some limited ability to do parameter passing, with execution history
- DOI minting for data and code

7.5 1.1.0 (2014-03-06)

7.5.1 New features and content

- Users can log in with Github and Persona
- Citations: compendia pages list citation information and users are reminded to cite code and data when they go to download code or data. #60
- Additional fields are auto-completed with the DOI auto-fill
- Added a Resources page with information about reproducible science and research compendia
- Compendia abstracts can use markdown

7.5.2 Fixes

- Citations show et. al. for papers with more than 5 authors #161
- Styling fixes for side navigation

7.6 1.0.0 (2013-12-18)

First talk! Now that we've had the first talk about this, let's have 1.0.0!

Not a lot of user-facing changes for this release. We've Renamed project ResearchCompendia from the pre-release *tyler* name, and renamed the repo to go with that.

- We have tagging on creation, but tags are not yet used for search and browsing.
- We have simple text search that searches through authors, title, abstract
- We have a simple compendium creation form with DOI autocompletition.
- We have some prelimary developer docs that discuss contributions and development.
- We have user profile pages so that users can view a list of their compendia.

CHAPTER 8

Indices and tables

- *genindex*
- *search*