web_reflectivity Documentation

Release 2.1.0.dev28+d202403222358

M. Doucet

Mar 22, 2024

CONTENTS

This Django application provides the web interface reflectivity.sns.gov to perform modeling of reflectivity data. The application gives users forms to set up their model and submit fitting jobs. To do so, it generates a python script to be executed either locally or on a remote compute resource. The generated script launches REFL1D, which does the actual minimization.

A description of reflectivity, an overview of the main features, and a real-life example of the use of the application are described in this article:

M. Doucet et al. SoftwareX 7 (2018) 287-293 https://doi.org/10.1016/j.softx.2018.09.001.

Please cite this article when using the web reflectivity interface.

CHAPTER

ONE

DEVELOPER DOCUMENTATION

1.1 Release notes

1.1.1 v2.0.0

The development team is very happy to announce this release which focussed on modernization and security fixes.

The main purpose of this release is to address security issues with how user's work is run on remote worker nodes. The new method of security creates a ssh key pair that is unique to the session and deleted when either the user logs out or abandons the session (idle for 24 hours). The other feature of the ssh key is that it is only accepted for connections from the host that created it.

Dependency changes

- python 2.7 to 3.8
- apache to nginx
- django 1.11 to 3.2
- jquery 1.72 to 3.6
- sqlite to postgress
- django-remote-submission

Other modernization changes include a heavily increased test coverage (currently 76%), building python wheels, and a sphinx site .

1.2 Developer Documentation

These pages contain the developer documentation. They are aimed at those who are modifying the source code of the project.

1.2.1 Contents:

Guide to Contributing

Contributions to this project are welcome. All contributors agree to the following:

- It is assumed that the contributor is an ORNL employee and belongs to the development team. Thus the following instructions are specific to ORNL development team's process.
- You have permission and any required rights to submit your contribution.
- Your contribution is provided under the license of this project and may be redistributed as such.
- All contributions to this project are public.

All contributions must be "signed off" in the commit log and by doing so you agree to the above.

Getting access to the main project

Direct commit access to the project is currently restricted to core developers. All other contributions should be done through pull requests.

Development procedure

1. A developer is assigned with a task during neutron status meeting and changes the task's status to **In Progress**.
2. The developer creates a branch off *next* and completes the task in this branch.
3. The developer creates a merge request (MR) off *next*.
4. The developer asks for another developer as a reviewer to review the MR. An MR can only be approved and merged by the reviewer.
5. The developer changes the task's status to **Complete** and closes the associated issue.

Contacting the Team

The best mechanism for a user to request a change is to contact the Reflectometry CIS. Please email Mathieu Doucet with your request.

Please state your change request as a:

- **Story** for any enhancement request
- **Defect** for any bug fix request.

Containerization

Web Reflectivity runs in Docker containers. The application is loosely coupled to ancillary services and easily deployable.

A quick look to file *docker-compose.local.yml* shows the following images to be generated:

- **web** runs the Web Reflectivity app. Image is built to use the host's network.
- **db** runs the posgreSQL database storing Web Reflectivity data like models and fits
- **redis** runs the queue manager, where fitting jobs requested by the user are queued as tasks to be run later.

- **nginx** runs the reverse proxy engine serving Web Reflectivity to the WWW
- **worker** runs the fitting engine *refl1D*. This image is used only when running in the developer's workstation.

Docker Volumes

Each image mentioned above uses volumes to mount certain directories or configuration files to the appropriate system location.

Named volumes:

- **web-static** is used by *webref* and *nginx* and is mounted to */var/www/web_reflectivity/static*
- **pgdata** is used by *db* and is mounted to */var/lib/postgresql/data*

Each image will mount its component logs to */var/log*.

List of Environment Variables

Deployment Configuration via Environment Variables

The values of the following variables are stored in a single GitLab environment variable of type *file*.

Settings in web reflectivity can be set using the following environment variables.

NOTE: entries deemed as secrets are in bold.

APP

VAR	SERVICE	DESCRIPTION
DJANGO_SETTINGS_MODULE	web	Controls which settings environment to use (*prod*, *envtest*, *unittest*)
DJANGO_SUPERUSER_USERNAME	web	
DJANGO_SUPERUSER_PASSWORD	web	
DJANGO_SUPERUSER_EMAIL	web	
APP_SECRET	web	
WEBREF_IP_ADDRESS	web	IP address of service *web* as seen from worker JOB_HANDLING_HOST

DATABASE

VAR	SERVICE	DESCRIPTION
DATABASE_NAME	db, web	Name of the Postgres database
DATABASE_USER	db, web	Owner of the database
DATABASE_PASS	db, web	
DATABASE_HOST	db, web	
DATABASE_PORT	db, web	

LIVE DATA SERVER

VAR	SERVICE	DESCRIPTION
LIVE_DATA_SERVER	web	URL template for retrieving data from remote server
LIVE_DATA_SERVER_DOMAIN	web	
LIVE_DATA_SERVER_PORT	web	
LIVE_PLOT_SECRET_KEY	web	
LIVE_DATA_API_USER	web	
LIVE_DATA_API_PWD	web	
LIVE_DATA_USER_UPLOAD_URL	web	URL template for uploading plots to remote data server
LIVE_DATA_USER_FILES_URL	web	URL template for retrieving list of user files

CELERY

VAR	SERVICE	DESCRIPTION
C_FORCE_ROOT	web	When *true* Celery workers will run as root
CELERY_LOG_LEVEL	web	

More details about the Celery configuration and how it is started can be found here.

REMOTE WORKER

VAR	SERVICE	DESCRIPTION
REFL1D_JOB_DIR	web	Absolute path for job output.
JOB_HANDLING_HOST	web	Hostname used for launching remote jobs. Can be *localhost* for local environments.
JOB_HANDLING_PORT	web	Port to connect on remote host.
JOB_HANDLING_INTERPRETER	web	Python interpreter to use for jobs submitted to *JOB_HANDLING_HOST* (e.g. *python3*)

LDAP

VAR	SERVICE	DESCRIPTION
LDAP_SERVER_URI	web	
LDAP_DOMAIN_COMPONENT	web	
LDAP_CERT_FILE	web	Path to CA certificate file

ICAT

VAR	SERVICE	DESCRIPTION
ICAT_DOMAIN	web	
ICAT_PORT	web	
CATALOG_URL	web	ONCat URL
CATALOG_ID	web	
CATALOG_SECRET	web	

(This page was reprocuded from https://code.ornl.gov/sns-hfir-scse/deployments/web-reflectivity-deploy/-/blob/main/docs/env.md)

External Service Dependencies

List of Dependencies:

- *Gravatar*
- *LDAP*
- *Live Data Service*
- *NGINX*
- *ONCat*
- *PostgreSQL*
- *Redis*

Gravatar

Gravatar is a service for providing globally unique avatars. This is used along user specific content.

LDAP

LDAP (Lightweight Directory Access Protocol), a central form of authentication. It is required for this service to determine access level of users when requesting experiment data. Once a user is logged in, the application will submit jobs to your compute resources on the user's behalf, through celery. This service is run in a dedicated server.

Live Data Service

Stores and provides live reduction data for use with REFL1D scripts generated with this application. This service is hosted at livedata.sns.gov.

NGINX

Reverse Proxy stood in between client and service. This service is run in a local docker container.

ONCat

Provides catalog data used internally at ORNL. Used for scripts and local data processing. This is hosted at oncat.ornl.gov.

PostgreSQL

Relational database used by Django. Stores different form data submitted by the user for modeling. This is run in a local docker container.

Redis

An in-memory data structure store, used as a distributed, in-memory key-value database, cache and message broker. In this application it is used to send fitting jobs to the remote worker and remove expired sessions and associated SSH keys. This is run in a local docker container.

Setting Up and Working in the Local Environment

Development of the application starts by being able to build and run Web Reflectivity in the developer's computer, termed the *Local Environment*, or just *LOCAL*.

The top-level *Makefile* contains macros (make targets) that liberates the developer from having to remember the various terminal *conda* and *docker* commands. Typing *make help* will show the list of macros with a short description.

```
$ make help
$ build-docker  build the image for the package under src/
$ clean         deletes containers, network, volumes, and images
$ conda         installs, then activates conda environment webrefl with all dependencies
$ dev           installs the local dev environment for the first time. Requires sudo␣
↪privileges if the 'docker' group doesn't exist or if you don't belong nto such group.
$ docs          create HTML docs under docs/_build/html/. Requires activation of the␣
↪webrefl conda environment
$ redev         reinstalls the local dev environment after the cleaning step
$ startdev      invoke docker-compose to create images, instantiate containers, and␣
↪start services
$ test          run unit tests
```

Web Reflectivity is fully containerized so nothing except the docker engine and docker-compose is needed to install in the developer's computer in order to compose and run the software. However, it is highly recommended to create first the conda environment containing the dependencies and do the development work in that environment. Command *make conda* will create conda environment *webrfl* with all needed dependencies. In addition, it will install the *src/web_reflectivity* package in development mode.

Configuration

As part of the configuration setp, the developer needs to export the following environment variables to the shell:

- *LDAP_SERVER_URI*
- *LDAP_USER_DN_TEMPLATE*
- *LDAP_DOMAIN_COMPONENT*

These value of these variables are secrets allowing the developer to login to Web Reflectivity with their UCAMS account. The developer must contact the development team to procure themselves with these variables. A convenient place to store them is in an *.envrc* file at the root level of the source code. See utility direnv for how *.envrc* files are used. A minimalistic *.envrc* file would look like this:

```
$ export LDAP_SERVER_URI=*****
$ export LDAP_USER_DN_TEMPLATE=*****
$ export LDAP_DOMAIN_COMPONENT=*****
```

where the ***** are placeholders for the actual values.

Next, command *make dev* is the entrypoint to configuring *LOCAL* and starting it for the first time.

- create a "docker" UNIX group if non-existent. This requires *sudo* privileges.
- add the developer's username to the "docker" group so that the developer can run docker commands.
- login to the code.ornl.gov container registry. You will need the UCAMS credentials for this.
- create directories under */tmp/log/web_reflectivity/* to mount the */var/log/* directories of each service. This allows the developer to peruse the logs without having to log into each container.
- compose and start all services specified in *cfg/docker-compose.localdev.yml*

For details regarding configuring Docker in a Linux box, see Docker post-installation. Subsequent builds of the software should be not performed with *make dev*.

Running the Application

After Web Reflectivity starts, the developer can point the browser to *localhost* and start using the service. See *the manual fit session example* for usage.

There are several ways to stop the running service:

- *Ctrl-C* will gently stop the containers, nothing more.
- *docker-compose down -v* will stop and remove the containers, as well as networks and volumes.
- *make clean* will invoke *docker-compose down -v* and then remove all images.

To restart the service, one can:

- *docker-compose up –build*
- *make startdev* will overwrite the *docker-compose.yml* file with *docker-compose.local.yml*, then invoke *docker-compose up –build*.
- *make redev* will invoke *make clean*, then *make startdev*

Recreating the images is time consuming so the typical development cycle is:

1. *docker-compose down -v*
2. make some changes to the code

3. *docker-compose up –build*

Notice, however, that *src/* is mounted on the container running *Web Reflectivity* so any changes to the python source will automatically reflect in the application. Hence, there will be no need to stop, then restart the service for the changes to take effect.

Unit Testing

To run the unit tests, activate the *webrefl* conda environment. To run all tests, invoke *make test*.

To run individual tests:

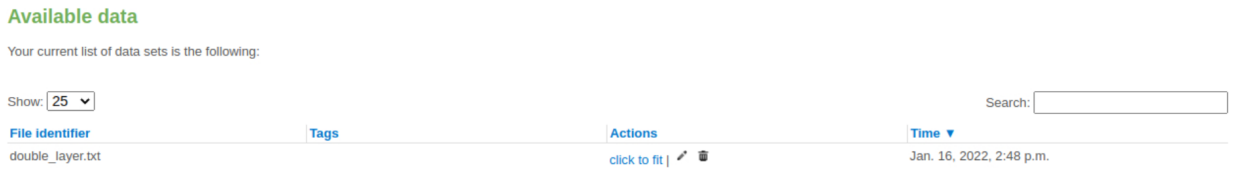
```
$ cd src
$ DJANGO_SETTINGS_MODULE='web_reflectivity.settings.unittest' pytest fitting/tests.py -k
↪'test_file_list'
```

Use Cases for QA

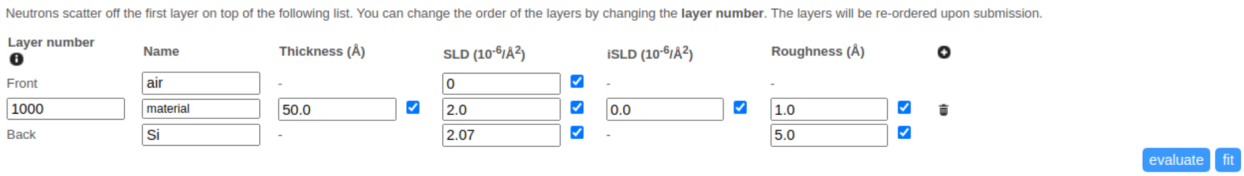
A Manual Fit Session

This page describes the steps one takes when fitting 1D reflectivity data on the web application, starting at the point when one logs in the application.

After login in, click in *Choose File* in order to select file *double_layer.txt* found in directory *test/data/*. After selecting the file, click in *Submit* in order to upload the file. The file will show up in the list of uploaded files.



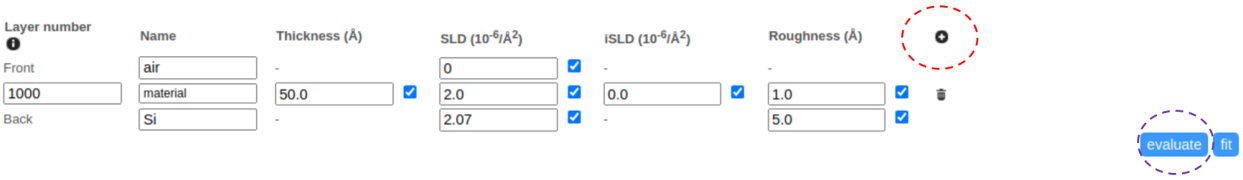
Click in the link *click to fit*. A default initial layer structure will be shown near the bottom of the model page:



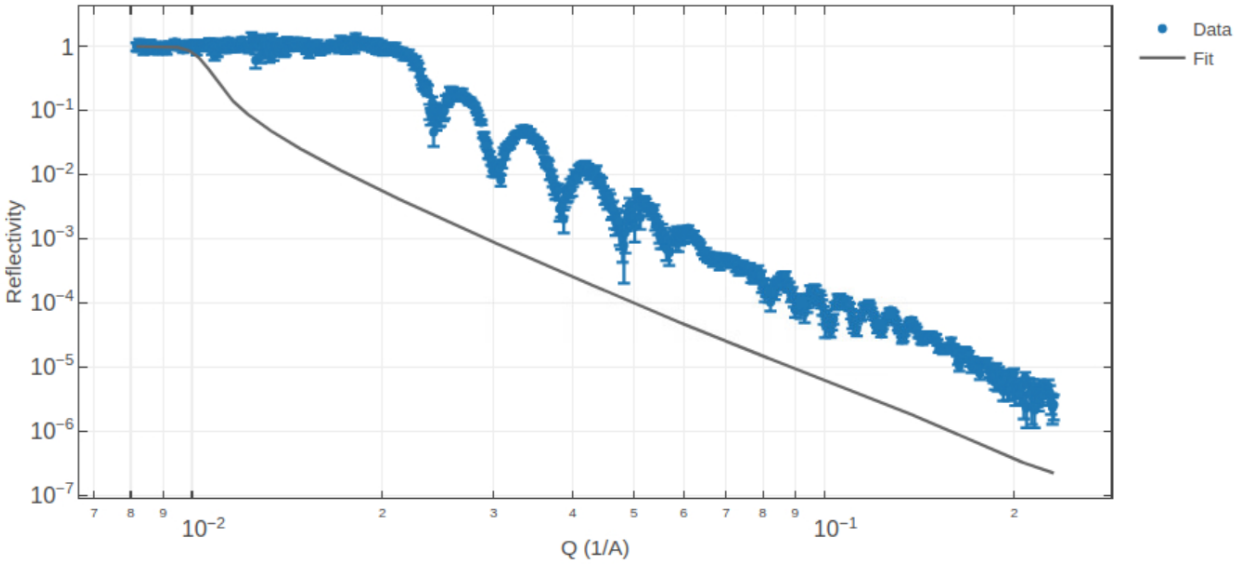
The default layer structure has only one layer, limited by air and Si on both sides. The solution layer structure contains two layers with specific thickness and other properties, namely

Layer number	Name	Thickness (Å)	SLD (10^{-6}/Å^2)	iSLD (10^{-6}/Å^2)	Roughness (Å)
Front	air	-	0.0	-	-
2	B	44.75	5.912	0.0	1.0
1	A	577.6	9.436	0.0	9.977
Back	Si	-	2.07	-	1.566

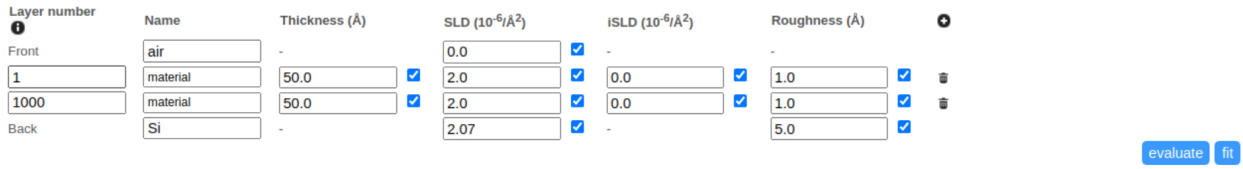
Thus, we must include an additional layer into the default initial layer structure. We use the "+" button for that (in the below picture, enclosed in the dashed-line red circle). However, before adding a layer we must evaluate or fit the current model. Press on the *Evaluate* button (in the below picture, enclosed in the dashed-line purple circle)



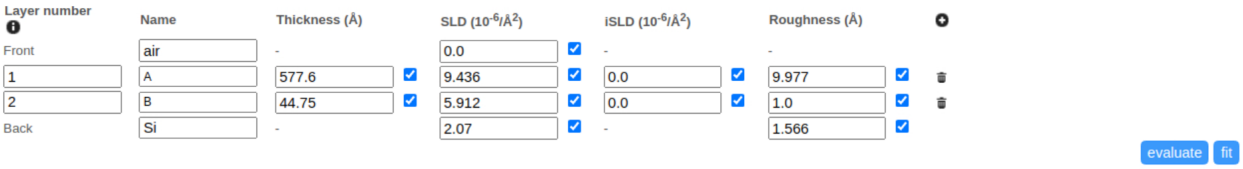
The evaluation prints the current model curve on top of the data. The plot is located near the top of the model page. As one would expect, the initial model is a poor fit of the data:



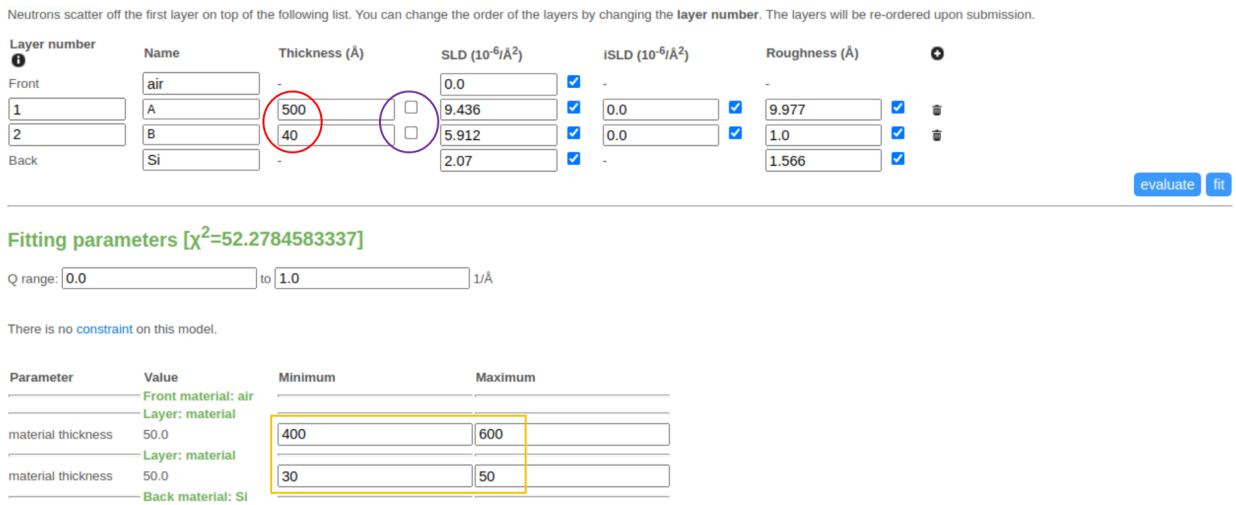
The poor fit is of little consequence now. Let's click in the "+" button now to add a new layer. This is the current model with two layers:



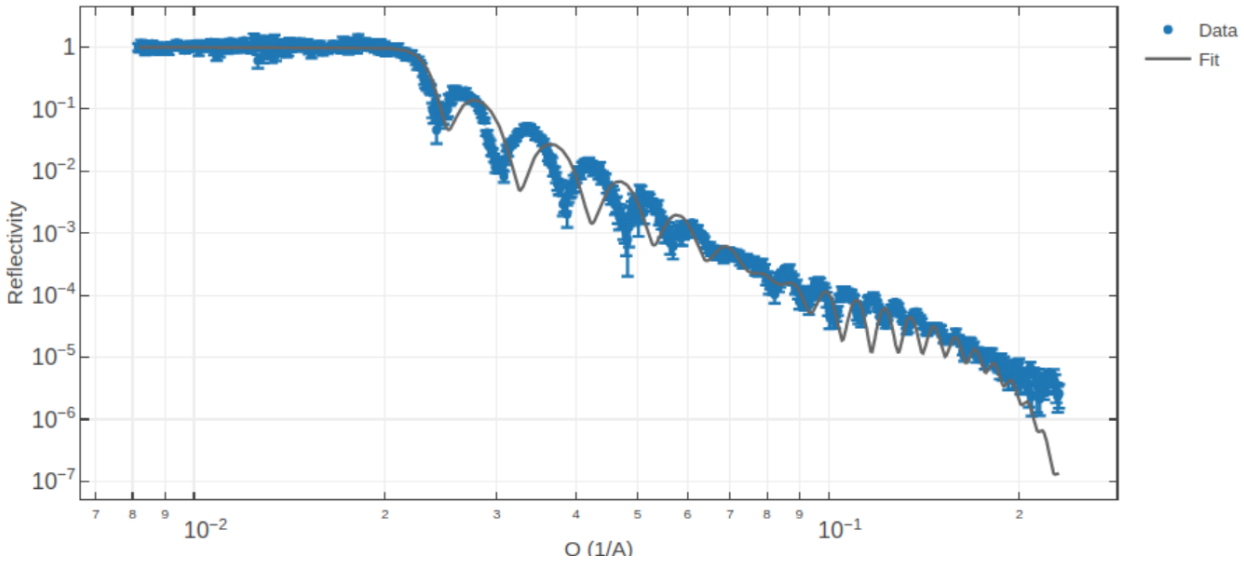
We enter the parameters of the solution layer structure into the model:



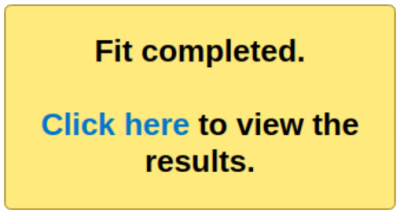
This should provide a model curve closely fitting the data. We're going to create an initial guess for the fit starting with these optimal parameters. We just distort the two thickness parameters for layers A and B. The goal is to recover the solution thickness when we fit the model. In this case we entered initial values of 500 and 40 Angstroms for layers *A* and *B*, respectively (red circle in the below picture). Also, we removed the nearby checkmarks, indicating these are fit parameters (purple circle in the below picture). When a parameter is unchecked, a *min* and *max* range becomes available (range rectangle in the below picture). We have adjusted the ranges so that our initial guesses fall right in the middle of the ranges:



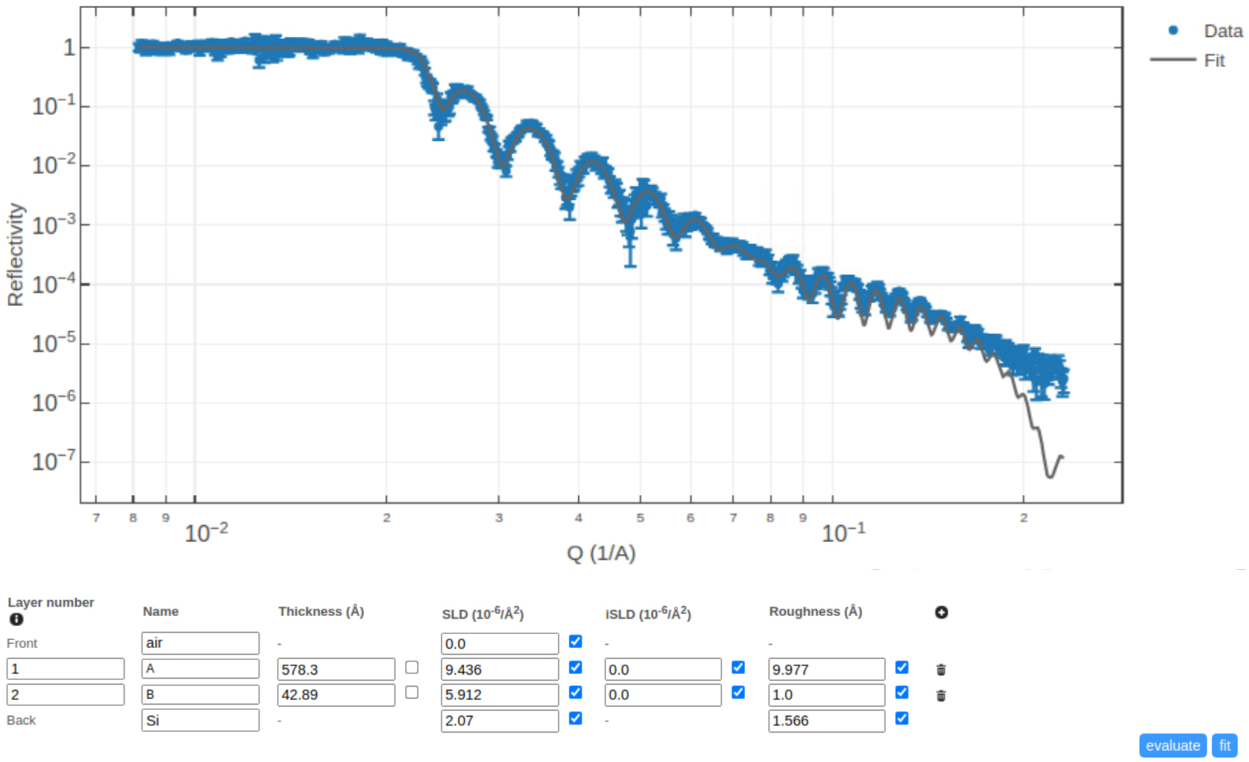
Click the *Evaluate* button to see how our initial guess aligns with the data. We find an "out-of-phase" fit:



We now fit the model by clicking in the *Fit* button. The job is submitted to the (local or remote) work server and fits results will be available once the fit job is finished. A message at the top of the page indicates this is so, with a link to display the fit:

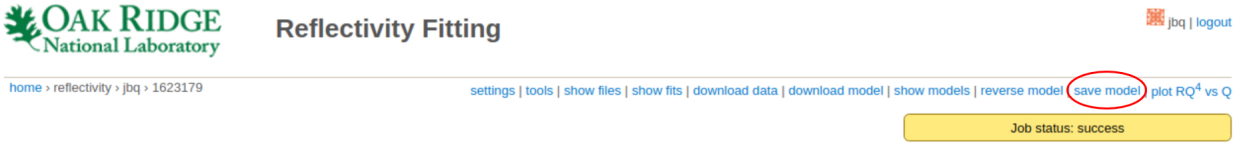


After clicking in the link, we can inspect the fit to the data and the fit parameters

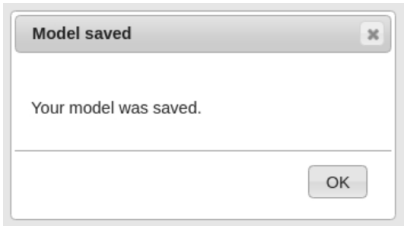


We obtain thicknesses *578.3* and *42.89* (compare to given solution parameters *577.6* and *44.75*).

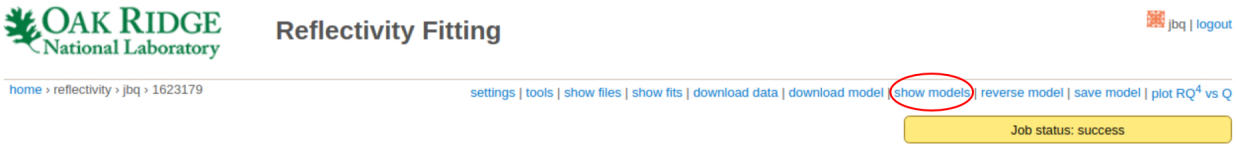
If so desired, we can save the fit by clicking in the *save model* link (red circle in the below picture):



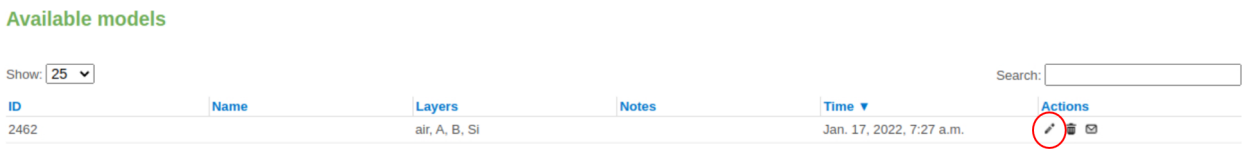
A pop-up will confirm the model was saved:



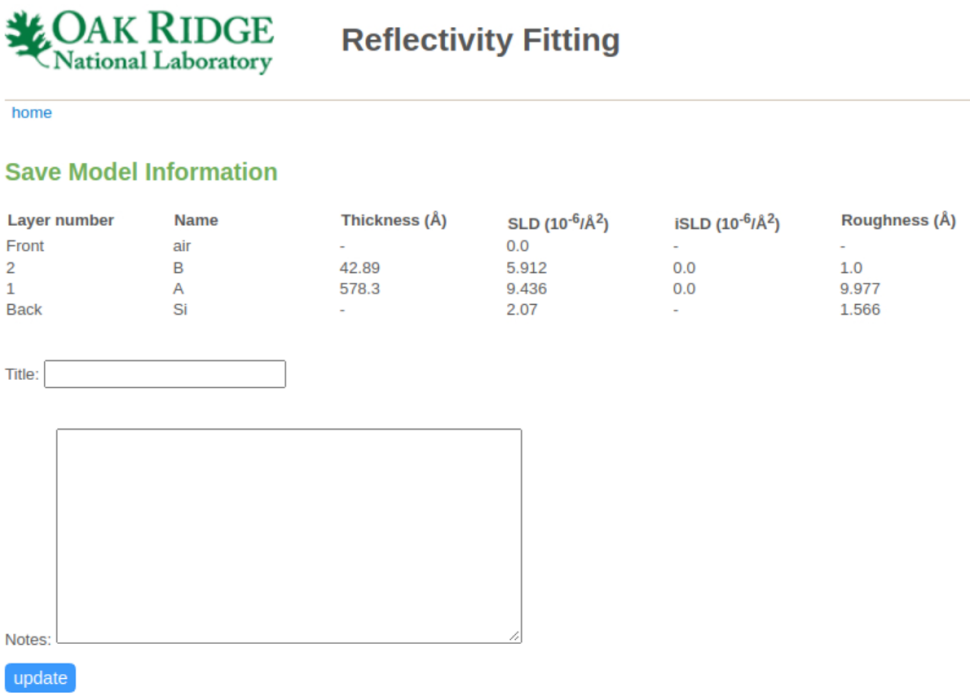
Close the pop-up by clicking in the *OK* button. Then, click in the *show models* button (red circle in the below picture):



The model will show at the top of the list of available models:



Clicking in the pencil button (red circle in the picture above) will show the fit parameters:



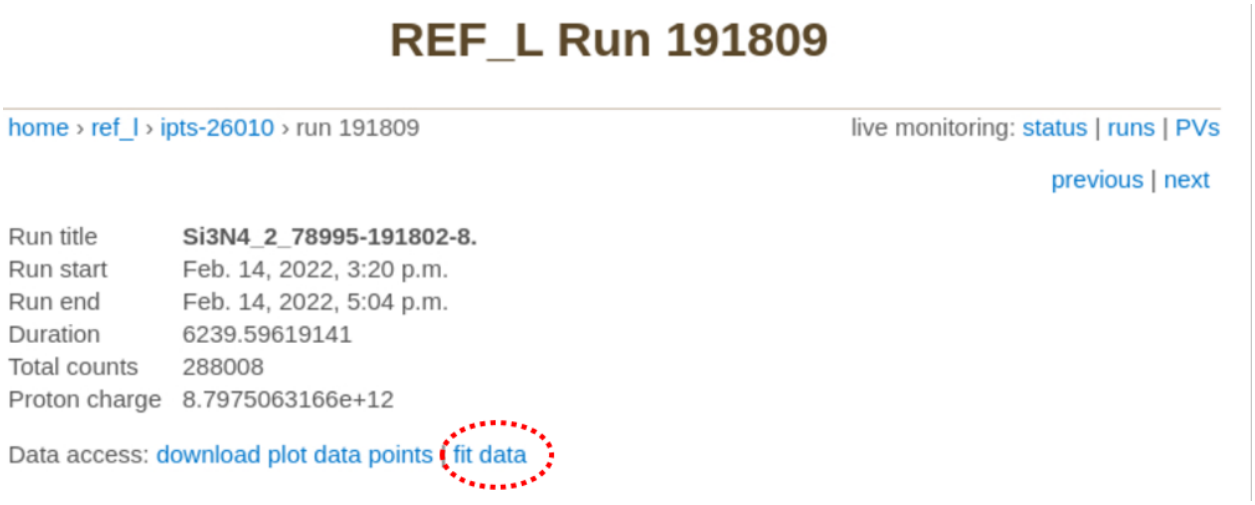
This finishes the manual fit session!

Fitting auto-reduced data

This page describes the steps one takes when fitting 1D reflectivity data on the web application, starting when looking at the auto-reduced data for a particular run in the web monitor app.

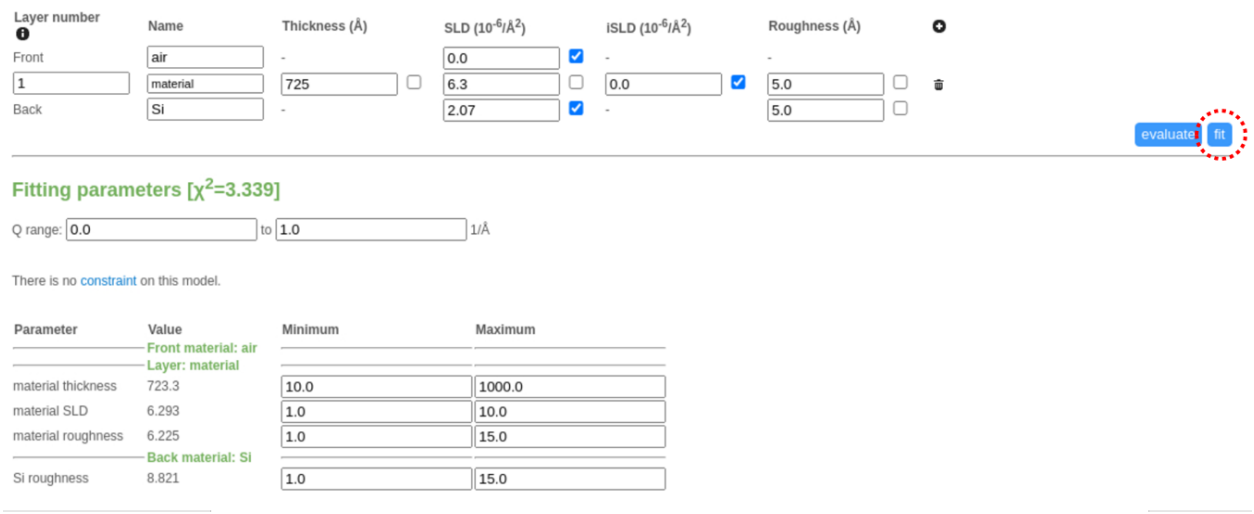
Once at the web monitor, go to the page of a run that was auto-reduced, e.g.: https://monitor.sns.gov/report/ref_l/191809/

Above the plot, there's a link to the fitting application. Click it: https://reflectivity.sns.gov/fit/ref_l/191809

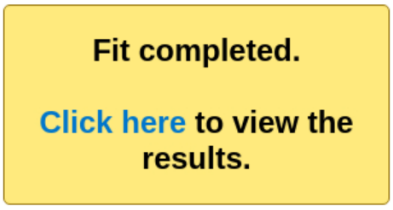


You should now see the same data. The data that was shown on the web monitor is also shown on the reflectivity fitting application.

In the *Layer model* section, you will now be able to define a model and perform a fit. In the layer that's called *material*, try entering a thickness of 725 Å and an SLD of 6.3. The uncheck the boxes for that layers thickness, SLD, and roughness. Also uncheck the Si roughness. Fitting parameters will now appears at the bottom of the page. Enter a maximum value of 1000 for the thickness, 10 for the SLD, and 15 for the roughness parameters. Then click *fit*.

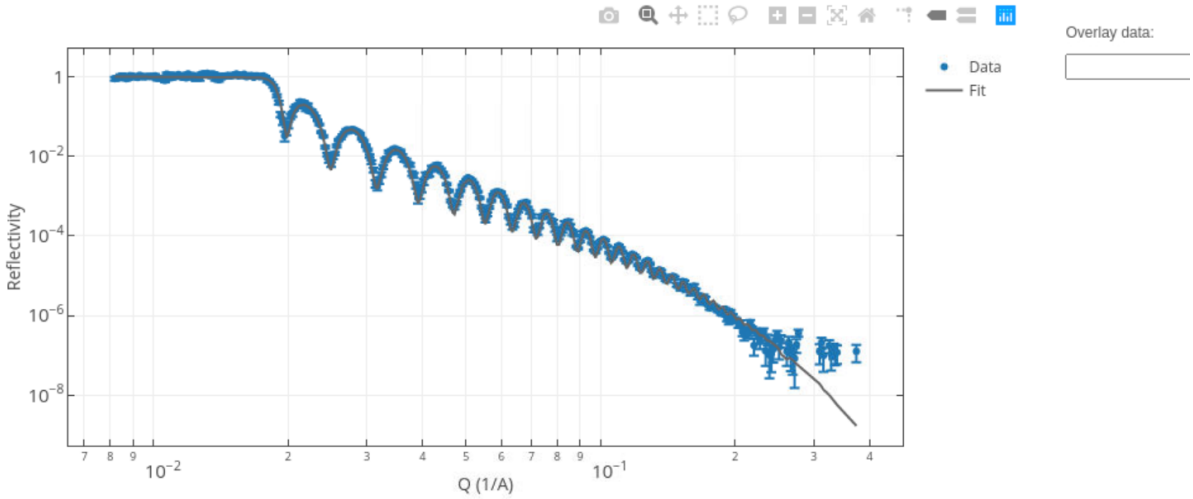


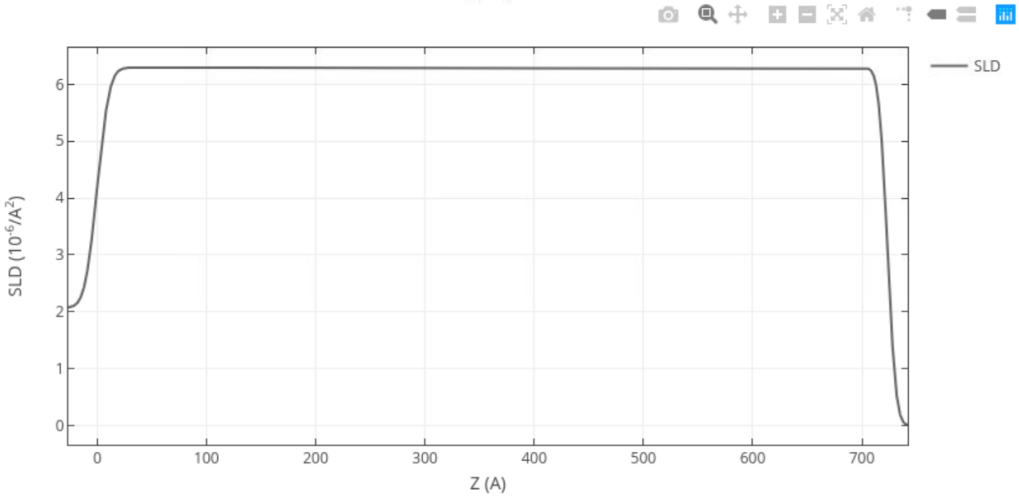
After a few seconds a message will appear to let you know the fit is done. Click that link:



Fit graph and optimal parameters should look like this:

Si3N4_2_78995-191802-8.





Layer model

Checked parameters will be kept fixed during the fitting procedure. You can also choose a model from your saved models.

Data	ref_l/191809	
Scale	1.0	☑
Background	0.0	☑

Neutrons scatter off the first layer on top of the following list. You can change the order of the layers by changing the **layer number**. The layers will be re-ordered upon submission.

Layer number	Name	Thickness (Å)	SLD (10^{-6}/Å^2)	iSLD (10^{-6}/Å^2)	Roughness (Å)
Front	air	-	0.0 ☑	-	-
1	material	723.3 ☐	6.293 ☐	0.0 ☑	6.225 ☐
Back	Si	-	2.07 ☑	-	8.821 ☐

DevOps Guide

Environment "testenv"

This environment allow developers and power users to run additional testing such as automated system tests and manual tests. One of its main purposes is to uncover bugs and defects not detected by the CI.

One host machine acting as a GitLab runner runs the instructions specified in the *deploy* job of the GitLab CI. There's only one specific machine allowed to pick and run the *deploy* job so that environment *testenv* is always deployed to the same machine. The app running in *testenv* is exposed to the WWW as *reflectivity-test.sns.gov*.

Currently, the database in this environment is not persistent, meaning a new deployment will erase whatever data has been stored since the last deployment. Also, *testenv* is currently listening and writing to *livedata.sns.gov*, as well as listening to *oncat.ornl.gov*. Development of a test environment for the web monitor will bring about test substitutes for both servers.

Database Management

We rely on Django's *manage.py* for dumping the *source* database and loading it into the *recipient* database. Django's *manage.py* allows for a fine control of what tables to dump and the command is agnostic of the database flavor (mysql, postgresql, sqlite) for both the source and recipient databases.

Dumping the Old Database

The database from reflectivity.sns.gov has quite a different schema than the database of the modernized application because models for the app and its dependencies have evolved.

Login to reflectivity.sns.gov and then:

```
cd /var/www/web_reflectivity/app
dumpfile=/tmp/webreflect_$(date +%F).json  # e.g. /tmp/webreflect_2022-05-01.json
python manage.py dumpdata --verbosity 3 --natural-foreign --natural-primary -e␣
↪contenttypes -e auth.Permission -e django_auth_ldap -e django_celery_results --indent␣
↪2 --database default --traceback > ${dumpfile}
```

For loading the resulting *JSON* file into the recipient database, jump to *Loading onto the Modernized Database*.

Dumping the Modernized Database

It is assumed that the container running the *web_reflectivity* app, as well as the container running the database are up and running.

The name of the container running the *web_reflectivity* app should be *test_webref_1* if running in the TEST environment. One can make sure by listing the running containers:

```
$> docker container ls
CONTAINER ID  MAGE                                                                    ␣
↪    COMMAND               CREATED      STATUS               PORTS               NAMES
e71b9b6c4a4e  code.ornl.gov:4567/reflectometry/web_reflectivity/web_reflectivity:latest-
↪dev  /usr/bin/docker-ent...  6 hours ago  Up 6 hours (healthy) 22/tcp, 8000/tcp   test_
↪webref_1
```

In this particular case, the name of the container is *test_webref_1*, and we can use the CONTAINER ID *e71b9b6c4a4e* in place of this name.

Open a shell to the container running the *web_reflectivity* app and execute the *dumpdata* make target:

```
$> docker exec -it test_webref_1 bash
(webrefl)$ make dumpdata  # e.g. creates /tmp/webreflect_2022-05-20.json
```

A *JSON* dump file */tmp/webreflect_$(date +%F).json* is generated in the container's */tmp* directory. An easy way to make it available to the host machine is to move this file to directory */var/log/* because this directory is mounted in the host machine as directory */tmp/log/web_reflectivity/web/*

Loading onto the Modernized Database

We need to make the *JSON* dump accessible from within the container running the app. An easy way is to place the file in the host machine directory */tmp/log/web_reflectivity/web/* because is bind-mounted to container's directory */var/log/*.

Assuming we have file */tmp/log/web_reflectivity/web/webreflect_2022-05-01.json* in the host machine, we need to open a shell in the *running* container servicing the application and execute the *make loaddata* target.

Details on how to find out the name of the running container are laid out in the previous section *Dumping the Modernized Database*.

```
docker exec -it test_webref_1 bash
(webrefl)$ make fixturefile=/var/log/webreflect_2022-05-01.json loaddata
```

This will update the recipient database. The command takes minutes to execute because it translates the *JSON* file into a large set of python objects. These objects are in turn translated into a long list of postgres commands to be executed on the recipient database.

Deployment for Testing

Deployment for the testing environment is handled by the web-relectivity-deploy repo. For details on how to deploy, read the Guide for the deployment to Testing Environment.

Access to the Test Environment

Access to the test server *reflectivity-test.sns.gov* via SSH is granted on a per user *and* per client-machine basis. Contact the DevOps engineer assigned to maintenance of the server. After granted access, SSH to the machine as user *cloud*

```
ssh cloud@reflectivity-test.sns.gov
```

The goal of accessing the test server is to troubleshoot any problems by directly manipulating the deployed application.

Deployment in Production

This deployment is scheduled to happen after a new version of the software is released and thoroughly tested in deployment *testenv*. For details on how to deploy, read the Guide for the deployment to Production Environment.

Gitlab Ci Job Descriptions

This page is here to provide a description of current jobs ran on the Gitlab CI/CD pipeline.

Job Name	Time (Minute)
buildtestimage	39
static-analysis	5
test	6
docs	1
userdocs	~
wheel	.5
webrefimg	23
deploy	~

Images can be found at https://code.ornl.gov/rse/images

And the images for this repo are uploaded here: code.ornl.gov:4567/reflectometry/web_reflectivity/

buildtestimage

First this job performs the [func_rse_docker_cleanup](https://code.ornl.gov/rse-deployment/rse-sharables/raw/master/rse-bash-modules.sh) action, then builds a test docker image for the package under src/ and pushes it to gitlab with the 'latest' tag This is performed first so that subsequent jobs may reuse the same image and avoid unnecessary builds.

static-analysis

This job pulls the latest docker image and runs the battery of checks normally performed by the pre-commit hook At the time of writing this includes the following:

- trailing-whitespace
- check-docstring-first
- check-json
- check-added-large-files
- check-yaml
- debug-statements
- requirements-txt-fixer
- check-merge-conflict
- end-of-file-fixer
- sort-simple-yaml
- black
- flake8

For the an accurate list of hooks, please refer to the `.pre-commit-config.yaml` file

test

This job pulls the latest docker image with tag containing string `buildimage`. It then performs db migrations, runs unit tests, generating a coverage report and finally builds the wheel to confirm it can successfully and stores it for later.

docs

This job pulls the latest docker image and generates docs in the /docs/ folder using Sphinx

userdocs

Using a POST request, this job signals the readthedocs.org site to pull and publish docs from the latest image.

The instance for this project is located here: https://web-reflectivity.readthedocs.io/en/latest/

wheel

This job pulls the latest docker image and publishes the wheel created during the test step using the `publish_wheel.sh` script The script is just a `python -m twine upload` with credential checks, failing the job if data is missings.

The filename is configured in the pyproject.toml. The naming convention for the generated wheel is `PREFIX-VERSION(.devDISTANCE)` where

- `PREFIX`: `web-reflectivity`
- `VERSION`: is the most recent tag given by `git describe`. For developer versions this is one minor version ahead of the last release. `VERSION` will be of the form `MAJOR.MINOR.PATCH(rcCANDIDATE)` where `rcCANDIDATE` is missing from a full release.
- `DISTANCE`: number of commits since latest git tag

An example name is `web_reflectivity-1.2.0.dev507-py3-none-any.whl`. This job only executes on protected branches such as `next`, `qa`, or `main`

webrefimg

This job builds the production docker image for Web Reflectivity It then pushes the image with the date appened to the tag and again with *:latest-dev* appended instead. i.e. this single image will have two tags associated with it, the former being its permanent tag, and the later a temporary tag. The temporary tag always refers to the latest version of this image. Finally it cleans up the images locally. This job only executes on protected branches such as `next`, `qa`, or `main`

deploy

This job attempts to deploy the docker image for the environment associated with the branch that trigged it.

Code Walkthroughs

These pages contain various descriptions and explanations of the code based on a particular topic/technology.

Celery Walkthrough

Table of Contents

Celery implements a *Task queue* to:

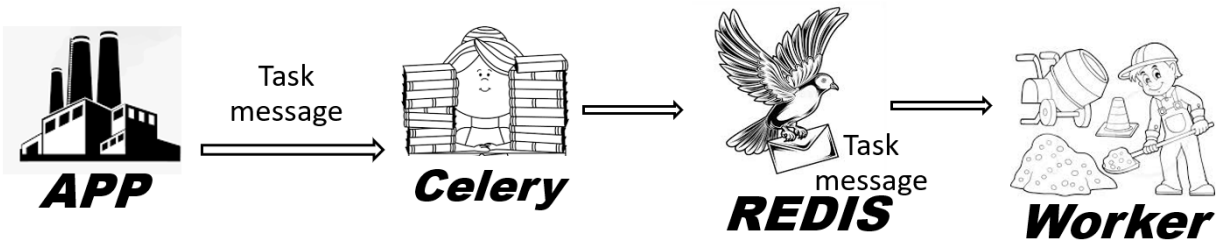
- execute instructions asynchronously
- run periodic tasks
- run cron tasks

...and much more. Celery docs at https://docs.celeryq.dev/

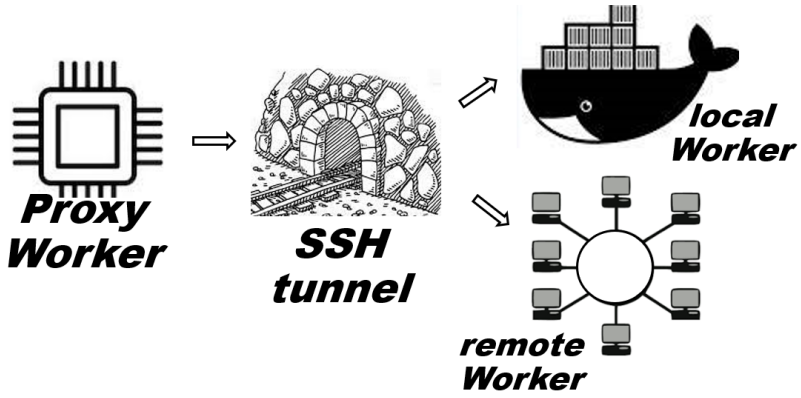
Celery requires a *message broker** capable of "outsorcing" tasks.

Main Ingredients

- **message** has input data and name of the task.
- **task** has executing instructions.
- **queue** stores the messages.
- **broker** fetches messages from the queue and delivers them to workers (not Celery)
- **worker** is the task executor (thread, VM, container...)



NOTE: in *web_reflectivity*, the worker is a CPU thread that delegates work to an external resource, such as the *worker* docker container or a computer node in the anaysis cluster.



Use of Celery in web_reflectivity

Celery, in concert with the message broker Redis, is used within app *web_reflectivity* for:

- send fitting jobs to the remote worker (*submit_job_to_server()*)
- remove expired sessions and associated SSH keys (*clean_expired_sessions*)

Additional Celery tasks are invoked not as task to be added to the queue but as *pure python functions*.

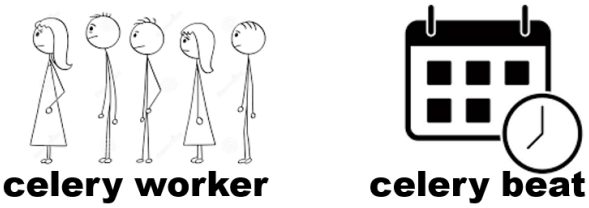
- establish passwordless SSH tunnels (*copy_key_to_server()*, *delete_key_from_server()*)

Where does the whole thing start?

The entry point src/docker-entrypoint.sh to the container startup creates two Celery instances:

```
celery --app fitting.celery worker --loglevel=${CELERY_LOG_LEVEL} --logfile=${CELERY_
↪WORKER_LOG_PATH} --detach
celery --app fitting.celery beat --scheduler django_celery_beat.
↪schedulers:DatabaseScheduler --loglevel=${CELERY_LOG_LEVEL} --logfile=${CELERY_BEAT_
↪LOG_PATH} --detach
```

- *fitting.celery* worker is the startup module
- *celery worker* instantiates the *Task* queue where new tasks can be added
- *celery beat* instantiates the *Task* scheduler to store period tasks and cron task to run at specific times.



Most of the startup module src/fitting/celery.py contents is boilerplate code:

```
app = Celery("web_reflectivity")

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "web_reflectivity.settings.develop")
app.config_from_object("django.conf:settings", namespace="CELERY")

app.autodiscover_tasks()
```

Configuration discovery is here accomplished with parsing attribute *settings* of module *django.conf*, which points to src/web_reflectivity/settings/develop.py and src/web_reflectivity/settings/base.py.

```
#####
# CELERY CONFIGURATION
#####
CELERY_RESULT_BACKEND = "django-db"
CELERY_BROKER_URL = "redis://redis:6379"
CELERY_TASK_SERIALIZER = "pickle"
CELERY_ACCEPT_CONTENT = ["pickle"]
```

Task discovery is here accomplished scanning the source code of *web_reflectivity* and any of its "installed apps". These are other Django apps inserted as dependencies.

```
users/task.py

django_remote_submission/task.py

...
```

autodiscover_tasks

For every installed app, Celery will check whether the app's source contains a "task.py" file. If so, it will parse the file searching for task functions.

Periodic Tasks

Celery has a flexible scheduling for task creation:

- happening at regular intervals
- happening at a specific time of the day every certain days of the week (*crontab*)

Scheduling can happen at *compile time* or at *runtime*.

Creating Tasks at Compile time

Scheduling at compile time is defined in the *CELERY_BEAT_SCHEDULE* setting. In src/web_reflectivity/settings/base.py

```
CELERY_BEAT_SCHEDULE = {
    "clean-expired-sessions": {
        "task": "users.tasks.clean_expired_sessions",
        "schedule": SESSION_COOKIE_AGE,
    },
}
```

task users.tasks.clean_expired_sessions occurs every *SESSION_COOKIE_AGE* seconds. It cleans browser sessions that had no activity for *SESSION_COOKIE_AGE* seconds or more.

```
@shared_task
def clean_expired_sessions() -> None:
    # ..body of the function..
```

The *@shared_task* decorator ensures the task is made available to every Celery instance (*web_reflectivity* has *two*). Tasks to be made available to specific Celery instances require decorating the task with attribute *task* said specific instance. One (hopefully) clarifying example:

```
# Two Celery instances initialized in myapp/celery.py
app1 = Celery("web_reflectivity")
app2 = Celery("web_reflectivity")

# Two tasks defined in myapp/tasks.py
from myapp.celery import app1

@app1.task
def task_specific():
    pass  # specific_task is made available to app1

@shared_task
def task_general():
    pass  # task_general is made available to app1 and app2
```

Notice that the shared task require that the Celery instances are instantiated *before* the *myapp/task.py* file is interpreted, as well as imported in the namespace of *myapp*. This is accomplished with boiler-plate code in *myapp/__init__.py*:

```
from .celery import app as celery_app
__all__ = ["celery_app"]
```

The same boiler-place code is in src/fitting/__init__.py

Creating Tasks at Runtime

Dependency django_celery_beat stores tasks in the app's database and exposes them in the admin site. Besides showing them, the app admin can **edit** them as well as **create** new tasks using anyone of the registered tasks.

The scheduler is specified when the Celery instance is created:

```
celery --app fitting.celery beat --scheduler django_celery_beat.
↪schedulers:DatabaseScheduler --loglevel=${CELERY_LOG_LEVEL} --logfile=${CELERY_BEAT_
↪PATH} --detach
```

The vanilla scheduler stores the scheduled tasks in a separate file, more appropriate when we're not supposed to mess with them not schedule new tasks.

Checking Executed Tasks

Dependency django_celery_results collects pieces of information from executed tasks (e.g. the returning value), store them in the database, and exposes them in the admin website. Useful for debugging.

Also, print and logged messages are be redirected to log file */var/log/celery.log* in the filesystem of the container running the *web* service. In *web_reflectivity* the directory */var/log* is bind-mounted to directory /tmp/log/web_reflectivity/web of the host machine.

One-off Tasks

Tasks to be run once in asynchronous mode are invoked with the *.delay* attribute

```
# in myapp/task.py
@shared_task
def my_task(greeting, target="World"):
    print(f"{greeting}, {target}!")

# in myapp/views.py
from myapp.task import my_task
# queue the task for asynchronous execution with the `delay` attribute
my_task.delay("Hello")  # will print "Hello, World!"
```

In *web_reflectivity*, django_remote_submission.task.submit_job_to_server is the only task invoked in this fashion.

```
submit_job_to_server.delay(
    job_pk=job.pk,
    key_filename=key_filename,
    username=username,
    log_policy=LogPolicy.LOG_TOTAL,
    store_results="",
    remote=(not settings.JOB_HANDLING_HOST == "localhost"),
)
```

Notice that the first positional argument to *submit_job_to_server()* is the table index in the database storing the state for an instance of class django_remote_submission.models.Job.

When passing information to a task:

- pass the python object if:

  - you want the task to use the state of the object at task **creation**.
  - the selected serializer (*pickle*) can serialize the object.
- pass the table index if:
  - you want the task to use the state of the object at task **execution**.
  - the worker has access to the database.

Tasks invoked as functions

Tasks invoked as functions run in the main thread (synchronous mode). Functions decorated with Celery-related decorators can still be calls as pure python functions.

```
# in myapp/task.py
@shared_task
def my_task(greeting, target="World"):
    print(f"{greeting}, {target}!")

# in myapp/views.py
from myapp.task import my_task
# queue the task for asynchronous execution with the `delay` attribute
my_task("Hello")  # will print "Hello, World!"
```

In *web_reflectivity*, django_remote_submission.task.copy_key_to_server and django_remote_submission.task.delete_key_from_server are the only tasks invoked in this fashion.

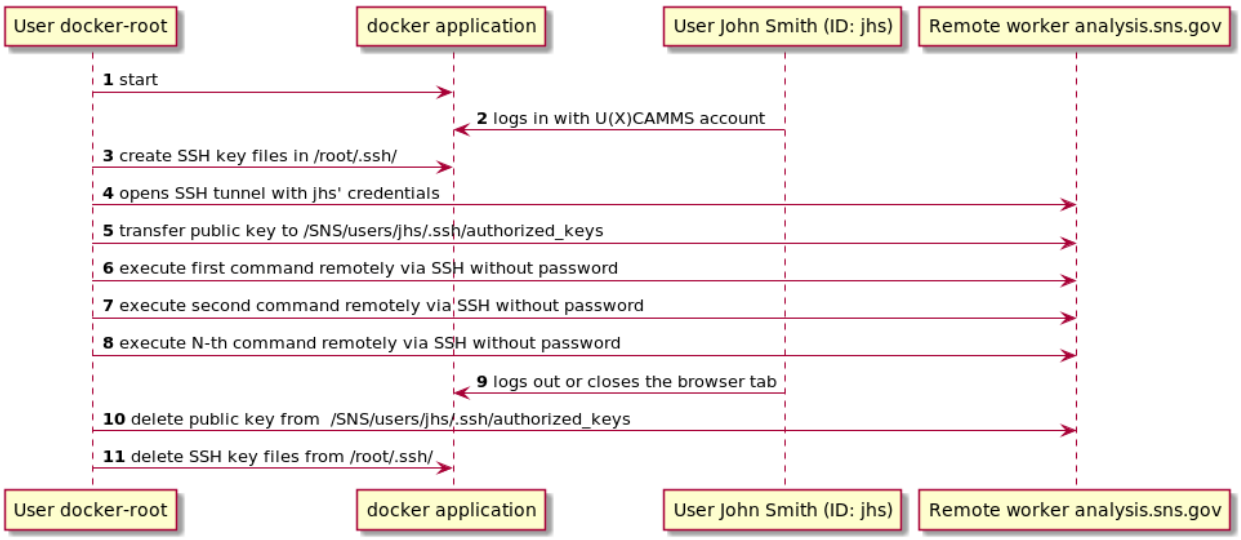
```
delete_key_from_server(
    public_key_filename=idfile.public,
    username=idfile.executor,
    password=None,
    key_filename=idfile.private,
    hostname=settings.JOB_HANDLING_HOST,
    port=settings.JOB_HANDLING_PORT,
    remote=True,
)
idfile.delete()
```

Notice that attributes of *idfile* are passed to *delete_key_from_server()* so it needs to run *before idfile* is deleted. We can be assured if we run *delete_key_from_server()* on the same thread.

The Job Control Layer

SSH-Tunnel to Remote Worker

Sequence diagram showing actors' roles (user "root" running the docker application and one person using the application) in the step that generates temporary SSH keys to establish a tunnel to the remote worker in charge of running the fitting calculations (usually, analysis.sns.gov)



Modules API

fitting

General fitting application

Fitting.forms

Forms for web reflectivity

class fitting.forms.**ConstraintForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

Simple form to select a data file on the user's machine

property media

Return all media required to render the widgets on this form.

class fitting.forms.**LayerForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Reflectivity model layer

get_layer()

Get layer info in a format we can send to refl1d

get_materials()

C60 = SLD(name='C60', rho=1.3, irho=0.0)

get_ranges(*sample_name='sample'*)

sample['C60'].interface.range(0, 20) sample['C60'].material.rho.range(0, 3) sample['C60'].thickness.range(1, 300)

has_free_parameter()

Check that we have a least one free parameter, otherwise the fitter will complain.

info_complete()

Return True of this layer should be used

property media

Return all media required to render the widgets on this form.

class fitting.forms.**LayerModelForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Form created from the ReflectivityLayer class

class Meta

Define how we use the model to create a form

__weakref__

list of weak references to the object (if defined)

model

alias of *ReflectivityLayer*

clean_name()

Refl1D doesn't like layer names that look like equations.

property media

Return all media required to render the widgets on this form.

class fitting.forms.**ReflectivityFittingForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Model parameters, excluding layers

get_materials()

C60 = SLD(name='C60', rho=1.3, irho=0.0)

get_predefined_intensity_range(*delta=0.001*, *probe_name='probe'*)

Since refl1d only fits, evaluating a model has to mean fitting in a tiny range.

get_ranges(*sample_name='sample'*, *probe_name='probe'*)

probe.intensity=Parameter(value=1.0,name="unity") probe.background.range(1e-8,1e-5)

get_sample_template()

Return a template for the sample description

has_free_parameter()

Check that we have a least one free parameter, otherwise the fitter will complain.

property media

Return all media required to render the widgets on this form.

class fitting.forms.**ReflectivityFittingModelForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Form created from the ReflectivityModel class

class Meta

Define how we use the model to create a form

__weakref__

list of weak references to the object (if defined)

model

alias of *ReflectivityModel*

clean_back_name()

Refl1D doesn't like layer names that look like equations.

clean_front_name()

Refl1D doesn't like layer names that look like equations.

property media

Return all media required to render the widgets on this form.

class fitting.forms.**SimultaneousModelForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

For to let users specify data to overlay or fit together

property media

Return all media required to render the widgets on this form.

class fitting.forms.**UploadFileForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

Simple form to select a data file on the user's machine

property media

Return all media required to render the widgets on this form.

class fitting.forms.**UserDataUpdateForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Form to update the information about an uploaded file

class Meta

Defining a form for the UserData model

__weakref__

list of weak references to the object (if defined)

model

alias of *UserData*

property media

Return all media required to render the widgets on this form.

Fitting.job_handling

Abstraction layer for handling fitting jobs

fitting.job_handling.**create_model_file**(*data_form*, *layer_forms*, *data_file=None*, *ascii_data=''*, *output_dir='/tmp'*, *fit=True*, *options={}*, *constraints=[]*, *template='reflectivity_model.py.template'*, *sample_name='sample'*, *probe_name='probe'*, *expt_name='expt'*)

Create a refl1d model file from a template

fitting.job_handling.**assemble_data_setup**(*data_list*)

Write the portion of the job script related to data files

fitting.job_handling.**assemble_job**(*model_script*, *data_script*, *expt_names*, *data_ids*, *options*, *work_dir*, *output_dir='/tmp'*)

Write the portion of the job script related to data files

Fitting.models

Data models

class fitting.models.**Constraint**(**args*, ***kwargs*)

Fitting parameter constraints

exception DoesNotExist

exception MultipleObjectsReturned

__str__()

Return str(self).

apply_constraint(*fit_problem*)

Apply the constraint to a fit problem

get_constraint_function(*alternate_name=None*)

Generate the constraint function

get_ranges(*sample_name='sample'*, *probe_name='probe'*)

Return the constraint code for the refl1d script

classmethod validate_constraint(*constraint_code*, *variables*)

Validate user-submitted constraint code.

class fitting.models.**FitProblem**(**args*, ***kwargs*)

Reflectivity model

exception DoesNotExist

exception MultipleObjectsReturned

__str__()
Return str(self).

delete(**args*, ***kwargs*)
Delete method to clean up related objects

model_to_dicts()
Return a dict with all the data values

show_layers()
Useful method to return the layers as a concise string

class fitting.models.**FitterOptions**(**args*, ***kwargs*)
Reflectivity model

exception DoesNotExist

exception MultipleObjectsReturned

get_dict()
Return an options dictionary

class fitting.models.**ReflectivityLayer**(**args*, ***kwargs*)
One layer of a reflectivity model

exception DoesNotExist

exception MultipleObjectsReturned

__str__()
Return str(self).

class fitting.models.**ReflectivityModel**(**args*, ***kwargs*)
Main reflectivity parameters

exception DoesNotExist

exception MultipleObjectsReturned

__str__()
Return str(self).

class fitting.models.**SavedModelInfo**(**args*, ***kwargs*)
Additional information attached to a saved model

exception DoesNotExist

exception MultipleObjectsReturned

class fitting.models.**SimultaneousConstraint**(**args*, ***kwargs*)
Constraint to tie parameters from two data sets in a simultaneous fit. #TODO: rewrite and merge this with Constraint when we are ready to write it as functions.

exception DoesNotExist

exception MultipleObjectsReturned

classmethod create_from_encoded(*fit_problem*, *par_to*, *par_from*, *user*)
Create a simultaneous constraint from encoded parameters

encode()

Encode an object into info that can be passed to a template

get_constraint(*sample_name='sample'*)

Return the constraint code for the refl1d script

Example: sample123['SiOx'].material.rho = sample345['SiOx'].material.rho

class fitting.models.**SimultaneousFit**(**args*, ***kwargs*)

Top level entry for a simultaneous fit. The FitProblem referenced here is the parent problem with which we can find the related data sets.

exception DoesNotExist

exception MultipleObjectsReturned

__str__()

Return str(self).

class fitting.models.**SimultaneousModel**(**args*, ***kwargs*)

Data sets to be addded to a FitProblem for simultaneous fitting

exception DoesNotExist

exception MultipleObjectsReturned

__str__()

Return str(self).

class fitting.models.**UserData**(**args*, ***kwargs*)

User data information

exception DoesNotExist

exception MultipleObjectsReturned

Fitting.parsing

Parsing utilities for REFL1D output files.

fitting.parsing.refl1d.**update_with_results**(*fit_problem*, *par_name*, *value*, *error*)

Update a mode with a parameter value.

Parameters

- **fit_problem** (FitProblem) – fit problem object to update
- **par_name** (*str*) – parameter name
- **value** (*float*) – parameter value
- **error** (*float*) – parameter error

fitting.parsing.refl1d.**find_error**(*layer_name*, *par_name*, *value*, *error_output*, *tolerance=0.001*, *pretty_print=False*)

Find the error of a parameter in the list of output parameters. @param layer_name: name of the layer @param par_name: name of the parameter @param value: output value, so we can recognize the entry @param error_output: list of fit output parameters from the DREAM output

The output parameter list should be in the format: [[parameter name, value, error], ...] The DREAM outputs are not grouped by sample/experiment, so we have to use the parameter values to determine which is which.

Because of constraints, parameters can have any name, so key on the parameter value to assign the errors, but don't change the reported value in case we incorrectly assign errors.

fitting.parsing.refl1d.**update_model_from_dict**(*fit_problem*, *experiment*, *error_output=None*, *pretty_print=False*)

Parse a json representation of the experiment :param FitProblem fit_problem: FitProblem-like ojbect :param dict experiment: dictionary representation of the fit problem read from the json output :param list error_output: list of DREAM output parameters, with errors. :param bool pretty_print: if True, the value will be turned into a value +- error string

fitting.parsing.refl1d.**update_model_from_json**(*content*, *fit_problem*)

Update a model described by a FitProblem object according to the contents of a REFL1D log.

Parameters

- **content** (*str*) – log contents
- **fit_problem** (FitProblem) – fit problem object to update

fitting.parsing.refl1d.**update_model**(*content*, *fit_problem*)

Update a model described by a FitProblem object according to the contents of a REFL1D log.

Parameters

- **content** (*str*) – log contents
- **fit_problem** (FitProblem) – fit problem object to update

fitting.parsing.refl1d.**extract_multi_data_from_log**(*log_content*)

Extract data block from a log. For simultaneous fits, an EXPT_START tag precedes every block:

EXPT_START 0 REFL_START

Parameters
log_content (*str*) – string buffer of the job log

fitting.parsing.refl1d.**extract_multi_sld_from_log**(*log_content*)

Extract multiple SLD profiles from a simultaneous REFL1D fit.

Parameters
log_content (*str*) – string buffer of the job log

fitting.parsing.refl1d.**extract_multi_json_from_log**(*log_content*)

Extract multiple JSON blocks from a REFL1D fit log.

fitting.parsing.refl1d.**parse_single_param**(*line*)

Parse a line of the refl1d DREAM output log 1 intensity 1.084(31) 1.0991 1.1000 [1.062 1.100] [1.000 1.100] 2 air rho 0.91(91)e-3 0.00062 0.00006 [0.0001 0.0017] [0.0000 0.0031]

Fitting.simultaneous

Handle multiple FitProblem objects for simultaneous fitting.

fitting.simultaneous.model_handling.**get_simultaneous_models**(*request*, *fit_problem*, *setup_request=False*)

Find related models and return a list of dictionary representing them.

Parameters

- **request** – http request object
- **fit_problem** (FitProblem) – FitProblem object
- **setup_request** (*bool*) – if True, the model will get set up from related fit problems

fitting.simultaneous.model_handling.**assemble_plots**(*request*, *fit_problem*, *result_fitproblems=None*)

Find all that needs to be plotted for this fit problem.

Parameters

- **request** – http request object
- **fit_problem** (FitProblem) – FitProblem object
- **result_fitproblems** (*list*) – list of FitProblem-like objects

fitting.simultaneous.model_handling.**compute_asymmetry**(*data_1*, *data_2*)

Compute asymmetry between two data sets.

Parameters

- **data_1** – data array
- **data_2** – data array

Fitting.view_util

Utilities for fitting views. Utilities for modeling application

fitting.view_util.**extract_ascii_from_div**(*html_data*)

Extract data from a plot . Only returns the first one it finds.

Parameters
html_data (*str*) – string

TODO: This should be refactored. When storing data locally, as opposed to using the external ORNL plot server, we can simply store the data as json. This function then needs to determine which approach to take.

fitting.view_util.**check_permissions**(*request*, *run_id*, *instrument*)

Verify that the user has the permissions to access the data

Parameters

- **run_id** (*str*) – run identifier (usually a number)
- **instrument** (*str*) – instrument name, or user name

fitting.view_util.**get_fit_problem**(*request*, *instrument*, *data_id*)

Get the latest FitProblem object for an instrument/data pair

Parameters

- **data_id** (*str*) – run identifier (usually a number)
- **instrument** (*str*) – instrument name, or user name

fitting.view_util.**get_model_as_csv**(*request*, *instrument*, *data_id*)

Return an ASCII block with model information to be loaded in third party applications.

Parameters

- **data_id** (*str*) – run identifier (usually a number)
- **instrument** (*str*) – instrument name, or user name

fitting.view_util.**get_results**(*request*, *fit_problem*)

Get the model parameters for a given fit problem

Parameters
fit_problem (FitProblem) – FitProblem object

fitting.view_util.**get_plot_from_html**(*html_data*, *rq4=False*, *fit_problem=None*)

Process html data and return plot data

Parameters

- **html_data** (*str*) – stored json for plotted data
- **rq4** (*bool*) – if True, the plot will be in R*Q^4
- **fit_problem** (FitProblem) – if supplied, a theory curve will be added

fitting.view_util.**assemble_plots**(*request*, *instrument*, *data_id*, *fit_problem*, *rq4=False*)

Find all that needs to be plotted for this fit problem.

Parameters

- **instrument** (*str*) – instrument name, or user name
- **data_id** (*str*) – run identifier (usually a number)
- **fit_problem** (FitProblem) – FitProblem object
- **rq4** (*bool*) – if True, the plot will be in R*Q^4

fitting.view_util.**find_overlay_data**(*fit_problem*)

Find extra data to be over-plotted for a given fit problem.

Parameters
fit_problem (FitProblem) – FitProblem object

fitting.view_util.**is_fittable**(*data_form*, *layers_form*)

Return True if a fit problem (comprised of all its forms) is fittable or not. To be fittable, refl1d requires at least one free parameter.

fitting.view_util.**evaluate_model**(*data_form*, *layers_form*, *html_data*, *fit=True*, *user=None*, *run_info=None*, *session=None*)

Protected version of the call to refl1d

fitting.view_util.**evaluate_simultaneous_fit**(*request*, *instrument*, *data_id*, *run_info*)

Assemble all the information for co-refinement

fitting.view_util.**save_fit_problem**(*data_form*, *layers_form*, *job_object*, *user*)

Save the state of the model forms

fitting.view_util.**apply_model**(*fit_problem*, *saved_model*, *instrument*, *data_id*)

Apply a saved model to a fit problem

fitting.view_util.**model_hash**(*fit_problem*)

Return a secret hash for a given fit problem

fitting.view_util.**copy_fit_problem**(*fit_problem*, *user*)

Make a duplicate copy of a FitProblem object

fitting.view_util.**plot1d**(*data_list*, *data_names=None*, *x_title=''*, *y_title=''*, *x_log=True*, *y_log=True*, *show_dx=False*)

Produce a 1D plot :param data_list: list of traces [[x1, y1], [x2, y2], ...] :param data_names: name for each trace, for the legend

fitting.view_util.**parse_ascii_file**(*request*, *file_name*, *raw_content*)

Process an uploaded data file :param request: http request object :param str file_name: name of the uploaded file :param str raw_content: content of the file

fitting.view_util.**get_user_files**(*request*)

Get list of uploaded files

Parameters

request – http request object

fitting.view_util.**parse_data_path**(*data_path*)

Parse a data path of the form <instrument>/<data>

fitting.view_util.**reverse_model**(*fit_problem*)

Reverse a layer model

Fitting.views

Definition of views

class fitting.views.**ConstraintView**(***kwargs*)

View for data fitting

get(*request*, *instrument*, *data_id*, *const_id=None*, **args*, ***kwargs*)

Process GET

post(*request*, *instrument*, *data_id*, *const_id=None*, **args*, ***kwargs*)

Process POST

class fitting.views.**FileView**(***kwargs*)

Process a file request

form_class

alias of *UploadFileForm*

get(*request*, **args*, ***kwargs*)

Process a GET request

post(*request*, **args*, ***kwargs*)

Process a POST request

class fitting.views.**FitAppend**(***kwargs*)

Append data to fit problem, usually for overlaying or simultaneous fitting.

get(*request*, *instrument*, *data_id*, **args*, ***kwargs*)

There is no get action, so just redirect to the fit list

post(*request*, *instrument*, *data_id*, **args*, ***kwargs*)

Add a data set to this fit problem

class fitting.views.**FitListView**(***kwargs*)

List of fits

get_context_data(***kwargs*)

Get the context for this view.

get_queryset()

Return the list of items for this view.

The return value must be an iterable and may be an instance of *QuerySet* in which case *QuerySet* specific behavior will be enabled.

model

alias of *FitProblem*

class fitting.views.**FitProblemDelete**(***kwargs*)

View to update the refl1d options

get_object(*queryset=None*)

Ensure that the object is owned by the user.

model

alias of *FitProblem*

class fitting.views.**FitterOptionsUpdate**(***kwargs*)

View to update the refl1d options

get(*request*, ***kwargs*)

Handle GET requests: instantiate a blank version of the form.

get_object(*queryset=None*)

Return the object the view is displaying.

Require *self.queryset* and a *pk* or *slug* argument in the URLconf. Subclasses can override this to return any object.

model

alias of *FitterOptions*

class fitting.views.**FitView**(***kwargs*)

View for data fitting

get(*request*, *instrument*, *data_id*, **args*, ***kwargs*)

Process GET :param request: request object :param instrument: instrument name :param data_id: data set identifier

post(*request*, *instrument*, *data_id*, **args*, ***kwargs*)

Process POST :param request: request object :param instrument: instrument name :param data_id: data set identifier

class fitting.views.**ModelListView**(***kwargs*)

View for data fitting

#TODO: Add option to upload a Motofit model

get(*request*, **args*, ***kwargs*)

Process GET

class fitting.views.**SaveModelDelete**(***kwargs*)

View to update the refl1d options

get_object(*queryset=None*)

Ensure that the object is owned by the user.

model

alias of *SavedModelInfo*

class fitting.views.**SaveModelUpdate**(***kwargs*)

View to update the refl1d options

get_object(*queryset=None*)

Ensure that the object is owned by the user.

model

alias of *SavedModelInfo*

class fitting.views.**SimultaneousView**(***kwargs*)

Set up the correlated parameters between two data sets for simultaneous fitting

get(*request*, *instrument*, *data_id*, **args*, ***kwargs*)

Process GET request

post(*request*, *instrument*, *data_id*, **args*, ***kwargs*)

Process POST request

class fitting.views.**UserDataDelete**(***kwargs*)

View to delete user data

get_object(*queryset=None*)

Ensure that the object is owned by the user.

model

alias of *UserData*

class fitting.views.**UpdateUserDataView**(***kwargs*)

View for modifying the information about an uploaded data file.

get(*request*, *instrument*, *data_id*, **args*, ***kwargs*)

Show current information about a user file

post(*request*, *instrument*, *data_id*, **args*, ***kwargs*)

Update information

fitting.views.**remove_constraint**(*request*, *instrument*, *data_id*, *const_id*)

Remove a constraint :param request: request object :param instrument: instrument name :param data_id: data set identifier :param const_id: pk of the constraint object to delete

fitting.views.**private**(*request*)

Return the page telling the user that the data is private.

fitting.views.**is_completed**(*request*, *job_id*)

AJAX call to know whether a job is complete. :param job_id: pk of the Job object

fitting.views.**download_reduced_data**(*request*, *instrument*, *data_id*)

Download reduced data from live data server :param request: http request object :param instrument: instrument name :param run_id: run number

fitting.views.**download_model**(*request*, *instrument*, *data_id*)

Download reduced data and fit data from latest fit :param request: http request object :param instrument: instrument name :param run_id: run number

fitting.views.**reverse_model**(*request*, *instrument*, *data_id*)

Download reduced data and fit data from latest fit :param request: http request object :param instrument: instrument name :param run_id: run number

fitting.views.**apply_model**(*request*, *instrument*, *data_id*, *pk*)

Download reduced data and fit data from latest fit :param request: http request object :param instrument: instrument name :param data_id: run number :param pk: primary key of model to apply

fitting.views.**save_model**(*request*, *instrument*, *data_id*)

AJAX call to save a model

#TODO: Save constraints too.

Parameters

- **request** – http request object
- **instrument** – instrument name
- **run_id** – run number

fitting.views.**remove_simultaneous_model**(*request*, *pk*)

Remove a data set/model from a simultaneous fit :param request: request object :param pk: SimultaneousModel object id

fitting.views.**update_simultaneous_params**(*request*, *instrument*, *data_id*)

Ajax call to process simultaneous fit model updates

Fitting.data_server

Data handling layer. Takes care of either storing and retrieving data locally or from a remote server.

fitting.data_server.data_handler.**generate_key**(*instrument*, *run_id*)

Generate a secret key for a run on a given instrument

Parameters

- **instrument** (*str*) – instrument name
- **run_id** (*int*) – run number

fitting.data_server.data_handler.**append_key**(*input_url*, *instrument*, *run_id*)

Append a live data secret key to a url

Parameters

- **input_url** (*str*) – url to modify
- **instrument** (*str*) – instrument name
- **run_id** (*int*) – run number

fitting.data_server.data_handler.**store_user_data**(*request*, *file_name*, *plot*)

Store user data

Parameters

- **request** – Django request object
- **file_name** (*str*) – name of the uploaded file
- **plot** (*str*) – user data, as a plotly json object

fitting.data_server.data_handler.**get_plot_data_from_server**(*instrument*, *run_id*, *data_type='html'*)

Retrieve data

Parameters

- **instrument** (*str*) – instrument or user name
- **run_id** (*int*) – run id, usually the run number
- **data_type** (*str*) – type of data, always HTML but kept here for API compatibility

fitting.data_server.data_handler.**get_user_files_from_server**(*request*, *filter_file_name=None*)

Get a list of the user's data on the live data server and update the local database

Parameters

- **request** – request object
- **filter_file_name** (*str*) – If this parameter is not None, we will only update the entry with that file name

datahandler

Local data handler, used for testing. In production, one would use a data server like the one here: https://github.com/neutrons/live_data_server

class datahandler.models.**Instrument**(**args*, ***kwargs*)

Table of instruments

exception DoesNotExist

exception MultipleObjectsReturned

__str__()

Return str(self).

class datahandler.models.**DataRun**(**args*, ***kwargs*)

Table of runs

exception DoesNotExist

exception MultipleObjectsReturned

__str__()

Return str(self).

class datahandler.models.**PlotData**(**args*, ***kwargs*)

Table of plot data. This data can either be json or html

exception DoesNotExist

exception MultipleObjectsReturned

__str__()

Return str(self).

tools

Convenience tools for planning and fitting reflectivity. Those tools consist of an SLD calculator and an electrode capacity calculator for energy storage measurements.

class tools.views.**ChargeRateView**(***kwargs*)

Compute capacity and charge rates.

form_class

alias of *ChargeRateForm*

get(*request*, **args*, ***kwargs*)

Process a GET request

post(*request*, **args*, ***kwargs*)

Process a POST request

class tools.forms.**ChargeRateForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

Input form for the capacity calculator

capacity()

Calculate capacity [micro Ah]

The charge packing refers, for instance, to the maximum x in Li_x:Si.

To test: Li_15 Si_4 -> 3579 mAh/g

Parameters

- **electrode** – electrode composition [string]
- **radius** – electrode radius [cm]
- **thickness** – electrode thickness [nm]
- **packing** – charge packing
- **valence_change** – change in oxidation state of the carrier

property media

Return all media required to render the widgets on this form.

users

Module to deal with authenticating users and verifying access

users.views.**perform_login**(*request*)

Perform user authentication

users.views.**perform_logout**(*request*)

Logout user, deleting temporary agent SSH keys and entry from worker's authorized_keys

users.view_util.**fill_template_values**(*request*, ***template_args*)

Fill the template argument items needed to populate side bars and other satellite items on the pages.

Only the arguments common to all pages will be filled.

users.view_util.**is_experiment_member**(*request*, *instrument*, *experiment*)

Determine whether a user is part of the given experiment.

Parameters

- **Requestrequest** – request object
- **instrument** (*str*) – Instrument name
- **experiment** (*str*) – IPTS name

web_reflectivity package

Submodules

web_reflectivity.settings module

The settings are split into difference ones directed at particular usage. They can be controlled using the DJANGO_SETTINGS_MODULE environment variable. Each settings module makes some small changes based on particular runtime environments. More information on django settings can be found at the django documentation site.

- web_reflectivity.settings.base which is, generally, the super-set of all other settings. This should never be assigned to DJANGO_SETTINGS_MODULE.
- web_reflectivity.settings.unittest is used for running the unit tests and while building the sphinx site
- web_reflectivity.settings.develop is used for development
- web_reflectivity.settings.envtest is used for remote test environment
- web_reflectivity.settings.prod is used for production environment

General settings

SESSION_COOKIE_AGE: int=60*60*24

How long to expire abandoned sessions. Default of one day, in seconds. Taken from the environment variable SESSION_COOKIE_AGE

SECRET_KEY: str="UNSET_SECRET"

Taken from the environment variable APP_SECRET

INSTALLATION_DIR: str="/var/www/"

Taken from the environment variable REFL_INSTALL_DIR and converted to a pathlib.Path

DEBUG: bool

This is True for all settings except web_reflectivity.settings.prod.

Settings for LDAP

LDAP_DOMAIN_COMPONENT: str

Taken from the environment variable LDAP_DOMAIN_COMPONENT

AUTH_LDAP_SERVER_URI: str

Taken from the environment variable LDAP_SERVER_URI

AUTH_LDAP_CERT_FILE: str

Taken from the environment variable LDAP_CERT_FILE. Failing to specify this results in not verifying certificates for the LDAP connection.

Settings for database

These are ignored for web_reflectivity.settings.unittest which is hard coded for sqlite3.

DATABASES: dict

There are 5 environment variables that are used for configuring the database connection. Failing to specify any of these will result in a mis-configured system. The environment variables are DATABASE_NAME, DATABASE_USER, DATABASE_PASS, DATABASE_HOST, and DATABASE_PORT.

Settings for live data server

LIVE_DATA_SERVER: str

Taken from the environment variable LIVE_DATA_SERVER

LIVE_DATA_SERVER_DOMAIN: str

Taken from the environment variable LIVE_DATA_SERVER_DOMAIN

LIVE_DATA_SERVER_PORT:: int

Taken from the environment variable LIVE_DATA_SERVER_PORT

LIVE_PLOT_SECRET_KEY: str

Taken from the environment variable LIVE_PLOT_SECRET_KEY

LIVE_DATA_API_USER: str

Taken from the environment variable LIVE_DATA_API_USER

LIVE_DATA_API_PWD: str

Taken from the environment variable LIVE_DATA_API_PWD

LIVE_DATA_USER_UPLOAD_URL: str

Taken from the environment variable LIVE_DATA_USER_UPLOAD_URL

LIVE_DATA_USER_FILES_URL: str

Taken from the environment variable LIVE_DATA_USER_FILES_URL

Settings for fitting server

REFL1D_JOB_DIR: str="/tmp"

Taken from the environment variable REFL1D_JOB_DIR and converted to a pathlib.Path

JOB_HANDLING_HOST: str="localhost"

Taken from the environment variable JOB_HANDLING_HOST

JOB_HANDLING_PORT: int=22

Taken from the environment variable JOB_HANDLING_PORT

JOB_HANDLING_INTERPRETER: str="python"

Taken from the environment variable JOB_HANDLING_INTERPRETER

Settings for OnCAT

CATALOG_URL: str

Taken from the environment variable CATALOG_URL

CATALOG_ID: str

Taken from the environment variable CATALOG_ID

CATALOG_SECRET: str

Taken from the environment variable CATALOG_SECRET

Settings for local development

Local development uses a specific local worker which needs a user configured.

TEST_REMOTE_USER: str

Taken from the environment variable TEST_USER_NAME

TEST_REMOTE_PASSWD: str

Taken from the environment variable TEST_USER_PASSWD

web_reflectivity.routing module

Not currently documented

web_reflectivity.urls module

Not currently documented

web_reflectivity.wsgi module

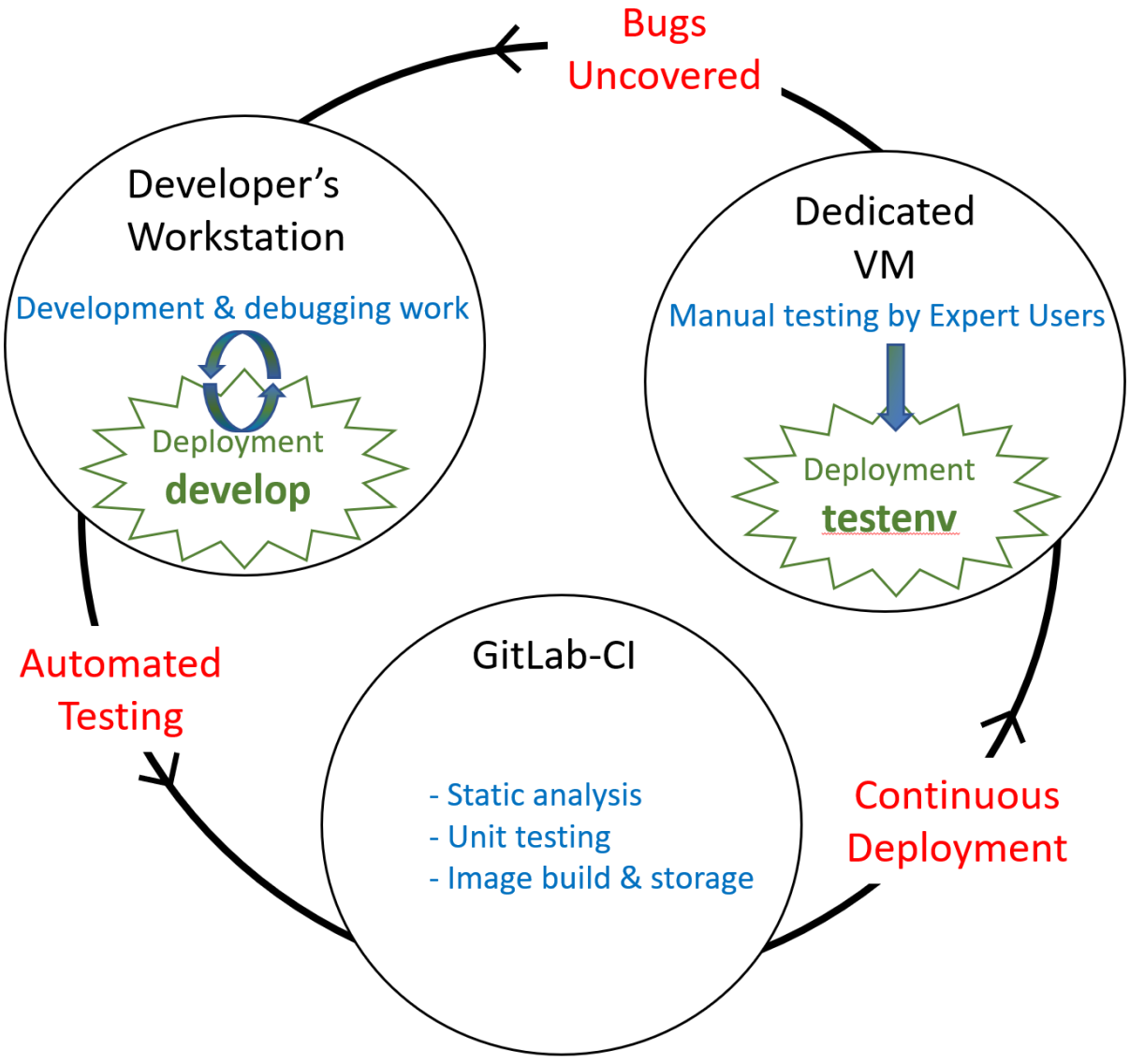
WSGI config for web_reflectivity project.

It exposes the WSGI callable as a module-level variable named `application`.

For more information on this file, see https://docs.djangoproject.com/en/3.2/howto/deployment/wsgi/

Development of the application is carried out by building and testing the software in three different environments:

- *locally* at the developer's workstation.
- at *GitLab CI* for automated testing.
- remotely at server *reflectivity-test.ornl.gov* for manual testing, termed environment *testenv*.



In all cases, building is implemented with *containerization* of the application in concert with *additional containers and services* providing the necessary services to successfully run the application, such as a database.

1.3 Links to Hardware provisioning and Deploy repositories

- neutrons-test-environment
- web-reflectivity-deploy

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

C

D

E

F

G

H

I

T

U

V

W