



JuliaBase

Release 1.0c

Torsten Bronger

February 02, 2015

1	Introduction	1
1.1	Technical overview	2
1.2	Getting started	2
2	A walk through JuliaBase	3
2.1	The demo site	3
2.2	The demo accounts	3
2.3	Rosalee: The everyday work	4
2.4	Juliette: The assigner of work	10
2.5	Nick: Technical service for others	12
2.6	Sean: The team leader	13
3	Installation	15
3.1	Prerequisites	15
3.2	Linux configuration	15
3.3	PostgreSQL	16
3.4	Django	16
3.5	JuliaBase	16
3.6	Apache	17
4	Programming	19
4.1	Organizing your source code	19
4.2	Creating a new Django app	20
4.3	Adding a new process module	21
4.4	A more complex example: Writing a deposition module	24
4.5	Process glossary	26
5	Model permissions	27
5.1	Semantics and conventions	27
5.2	Omitting permissions	28
5.3	Django’s default permissions	28
6	Class-based views	29
6.1	The API	29
6.2	Main classes	31
6.3	Mixins	32
7	The remote client	35
7.1	Use cases	35
7.2	Extending the remote client	36

7.3	Local installation and usage	37
7.4	Other programming languages	37
7.5	Classes and functions	38
8	Settings reference	43
8.1	General JuliaBase settings	43
8.2	Settings for LDAP	45
8.3	Django settings with special meaning in JuliaBase	47
9	Sample names	51
9.1	Name format properties	51
9.2	Provisional sample names	52
9.3	Initials	52
9.4	Name prefix templates	53
10	Hacking on JuliaBase	55
10.1	Architecture	55
10.2	Coding guidelines	55
11	Utilities	57
11.1	Common helpers	57
11.2	Feed reporting	65
11.3	Form field classes	70
11.4	Form classes	71
11.5	Plots	72
11.6	URLs	74
12	Template tags and filters	77
12.1	JuliaBase core	77
12.2	Samples	78
13	Markdown	83
13.1	Paragraphs	83
13.2	Emphasis	83
13.3	Escaping characters	83
13.4	Special characters	84
13.5	Math equations	84
13.6	Links	84
13.7	Lists	84
13.8	Line breaks	85
14	The JuliaBase project	87
14.1	Licenses	87
14.2	Short project history	88
	Index	89

INTRODUCTION

Your scientific institute or working group creates lots of samples, and your team needs a tool to keep track of them? JuliaBase is made for exactly that! It is a database solution for samples, their processing and their characterization, with the following features:

- intuitive browser-based interface, fully working even on mobile devices
- maximal flexibility for being adapted perfectly to your production and measurement setups, and to your workflows
- possibility to manage more than one department in a single database
- fine-grained access control
- connects to your LDAP server for user management
- keeps track of samples across sample splits
- support for pre-evaluating raw data and creating plots
- automatic notification of changes in your samples
- sample management by sample series, topics, and tags
- arbitrarily complex searches made easy, e.g. “find all samples with infrared measurements, deposited together with a sample on glass substrate with a conductivity greater than 10^{-6} S/cm; oh yes, and only from this year and made by John”
- export to spreadsheets
- automatic lab notebooks
- database interaction from own programs, e.g. for connecting your measurement setup directly to the database
- fully translatable; core is available in English and German so far
- fully brandable; adjust the look to your corporate identity and/or your taste
- mature codebase
- compliance with state-of-the-art Web standards and security considerations
- fully open source

We believe that the database should adapt to the people and the existing workflows rather than the other way round!

However, there is no free lunch . . . JuliaBase’s flexibility comes at a cost. You have to create the Python code to describe your setups and apparatuses. Leaving out fancy things, this is copy,

paste, and modify of < 100 lines of code for each apparatus. JuliaBase contains code for typical processing and measurement setups that you can use as a starting point.

1.1 Technical overview

For better evaluation, here is a short list of the technical aspects of JuliaBase:

- JuliaBase is built on top of the [Django web framework](#).
- JuliaBase is written 100% in the Python programming language (version 2.7 or 3.2 onward).
- Although other setups are possible, the easiest server installation bases on Linux, PostgreSQL, and Apache.
- Hardware requirements are very low; a 100 people institute could be served by a single ordinary desktop computer.

1.2 Getting started

If you want to give JuliaBase a try, visit the [demo site](#). If you like it, the next step could be [installing it](#). Its installation includes the demo site, so you have immediately something up and running, and you can evaluate it even better.

If you consider actually using JuliaBase, have a look at the full [table of contents](#). This documentation also exists [in PDF format](#).

Finally, see [The JuliaBase project](#) for how to get in touch with the JuliaBase community.

A WALK THROUGH JULIABASE

2.1 The demo site

At <https://demo.juliabase.org>, a demo of JuliaBase is installed so that you can play with it a little bit. You can log in with various accounts with different levels of permissions, add samples, processes, tasks etc., have a look at sample data sheets or a lab notebook, and much more.

However, since this site is accessible to everyone, it may become chaotic over time. In order to prevent that, all the data is reset every hour on the hour. So, don't be surprised if you have to log in again after some time – all session data is reset, too. There is nothing to worry though as you may log in as often as you wish.

This demo site is also the default target of the remote client code shipped with JuliaBase. You are encouraged to use the demo as a test bed for your client code.

2.2 The demo accounts

The demo site is the JuliaBase installation of the “Institute of Nifty New Materials” (INM). It's a very small institute with only six employees. All accounts have the password “12345”.

2.2.1 The boss

Sean Renard (s.renard) is the lead scientist and director of this institute. Accordingly, his JuliaBase account allows him to view all samples, but he has got other privileges, too. More about that later.

2.2.2 The technical staff

Nick Burkhardt (n.burkhardt) is a technician in the INM for a very long time. He is responsible for the *PDS setup* (photothermal deflection spectroscopy), a measurement setup. He performs measurements for researchers. He would never let another person use his PDS.

Hank Griffin (h.griffin) is also a technician. He is responsible for the *solarsimulator*, another measurement setup. He performs measurements for researchers, but after proper instructions by him, other people may use the apparatus, too.

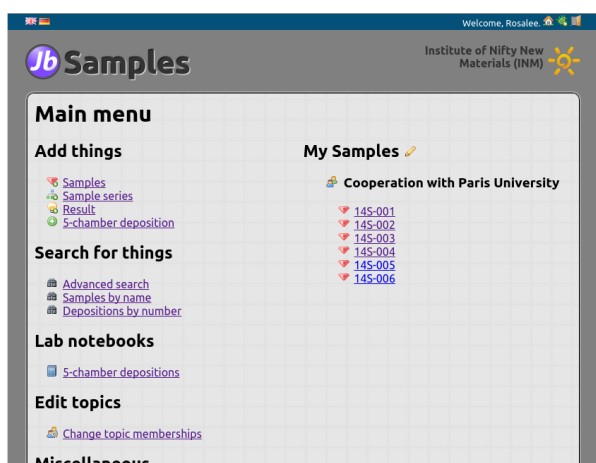
Eddie Monroe (e.monroe) is a deposition operator. This is a technician who manages a deposition system, in his case, the *cluster tool deposition*. Here too, other institute members may use this system after proper instructions.

2.2.3 The scientific staff

Rosalee Calvert (r.calvert) is a tenure scientist and creates samples by herself in the *5-chamber deposition setup*. Currently, she's the only one using this setup. Afterwards, she measures the samples in the solarsimulator. His current project is a cooperation with the University of Paris.

Juliette Silverton (j.silverton) is a PhD student with a lot of work. Thus, she is unable to do sample preparation and measurements herself, and let others do it. Consequently, she makes intensive use of JuliaBase's "task lists" feature in order to commission the work.

2.3 Rosalee: The everyday work



Log in as `r.calvert`, the typical ordinary user. We will see how she gets her work done.

2.3.1 The "My Samples" list

In the main menu, you can see Rosalee's "My Samples" on the right hand side. This list usually contains not *all* samples of a user but only those that are *currently of interest* to him/her. Still, this list may become quite long, and is therefore structured by *topics* and *sample series*. You may click on the bullet icons (👤 or 📁) in the list to fold and unfold sections that you want to hide.

2.3.2 Topics

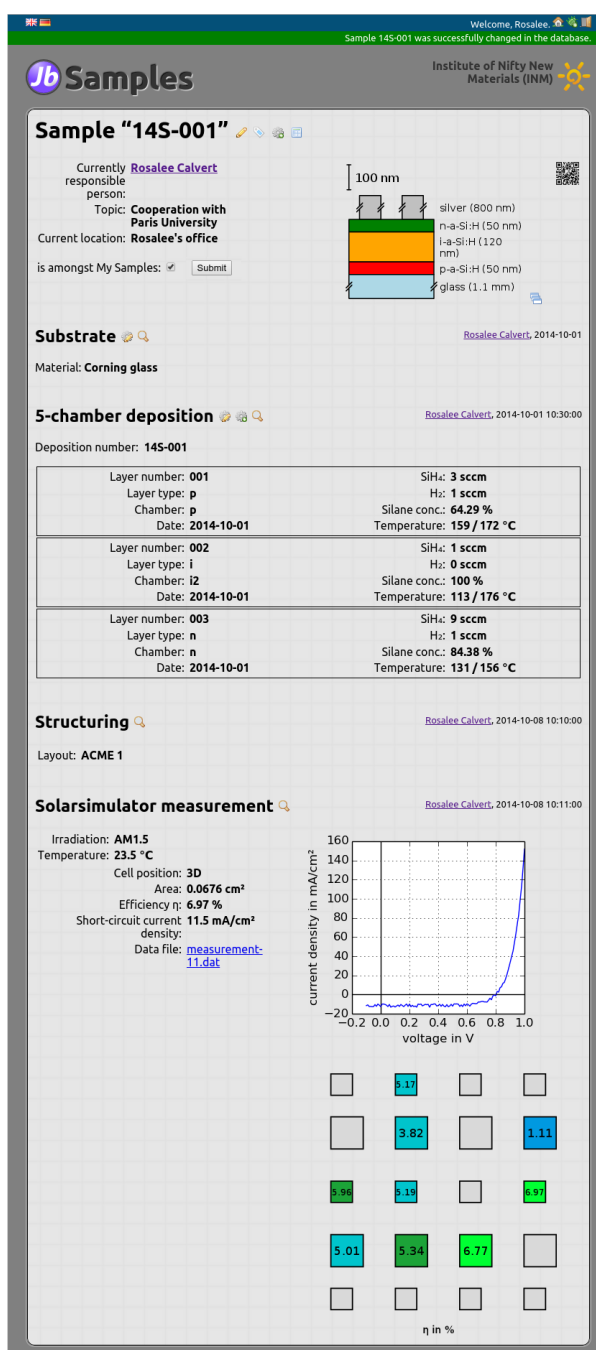
One sample usually belongs to exactly one topic. This helps to organise samples. For one thing, one can give topics expressive names, which makes the samples' purpose clear. In Rosalee's list, all samples belong to the topic "Cooperation with Paris University".

But even more important is that topics define who can see the sample. The prime directive in JuliaBase is: *You can only see samples of your topics*.

If a sample is in no topic, it is completely unprotected. Everyone can see it and take possession of it. There may be use cases for that but usually, you should put all your samples in topics.

People can be in an arbitrary lot of topics at the same time, but a sample is in exactly one topic. It may change it during its lifetime, though. Senior team members may have the permission to see all samples, whether in their topics or not. Sean Renard is such a person.

2.3.3 Sample data sheet



Let's visit a sample by clicking on "14S-001". You see its data sheet. At the top, it contains some general information like the currently responsible person and the topic. Then, you see a list of everything that has been done with this sample, in chronological order. It starts with the substrate, is continued with the deposition of the silicon layers, and ends with a measurement


in the solar simulator.

Every such step is called a “process” in JuliaBase. Even the “Substrate” is a process, albeit a slightly odd one. Every process has an operator and a timestamp. You can fold processes that would otherwise clobber the data sheet by clicking on the heading.


The main work when adapting JuliaBase to a new institute is programming all the processes that the institute needs. The solarsimulator measurement at the bottom is a good example why it is worth the effort. Just click on the colored squares, and you see how the data and the plot change immediately. Such features are lacking in off-the-shelf databases. This high degree of adaptability and flexibility is the primary strength of JuliaBase.

Let’s scroll back to the top. You see a schematic cross section of the sample, called an “informal stack” because it may not be totally accurate. This is not part of JuliaBase’s core because it may not be useful for every institute. But it’s part of the source code and you may use it. If you click on it, you get it as a PDF. This is also true for all plots in JuliaBase.


2.3.4 Edit samples

You can edit a sample by clicking on the pencil icon  next to the sample’s name. “Editing a sample” refers *only* to the data at the top of the data sheet. In particular, no processes are affected. Very often when you change somethin in JuliaBase, you have to describe your changes shortly at the bottom right. It may be tedious sometimes, but it may be very helpful to others who get notified by your changes.

2.3.5 Add processes

Also on the top of the sample data sheet there is the gear icon , which is to append a new process to the sample. If you click on it, you are asked which kind of process you’d like to add. Let’s have a look at two possibilities: Splitting and result process.

Split a sample

If you split samples into pieces, you surely want to keep track of that. For doing so, you click on the gear icon  and select “sample split”. Then, you can enter the new names of the samples. Usually, they extend their parent’s name. When looking at a sample’s data sheet, you also see all processes of the parents.

Result process

Result processes, often simply called “result”, are a handy ad-hoc way to append something to a sample’s data sheet. If you want to add a measurement result for which no dedicated process has been programmed so far, or if you want to add a plot, a picture, or a comment, then create a result. It’s the Swiss-Army-knife process if nothing else fits. Because it’s so flexible, be careful not to have spelling errors in order to keep searching and exporting easy.

2.3.6 Advanced search

Advanced search

sample:

name:

currently responsible person:

current location:

purpose:

tags: (separated with commas, no whitespace)

topic: explicitly empty: ☐

containing:

operator:

external operator:

timestamp:

comments:

irradiation: (in °C)

temperature:

containing:

cell position:

data file: (only the relative path below)

area: (in cm²)

efficiency η : (in %)

short-circuit current density: (in mA/cm²)


containing:

containing:

Submit

• 14S-002
• 14S-003

add samples

Rosalee wants to see her best samples. For this, go back to the main menu (the house icon  on the top right, or the big text “Samples”) and select “Search for things – Advanced search”. Now, perform the following steps, clicking on “Submit” after each step:

1. Select “sample” in the drop down menu.
2. Enter “calvert” in “currently responsible person” and select “solarsimulator measurement” in the drop-down menu “containing”.
3. Select “AM1.5” in “irradiation”, and select “solarsimulator cell measurement” in the inner drop-down menu “containing”.
4. Enter in “efficiency η ” the value “8”.

You get the result as in the image next to this text: Two of her samples match the criteria, namely “14S-002” and “14S-003”. This means, both samples have at least one solarsimulator measurement under AM 1.5 irradiation, with at least one cell with an efficiency greater than 8%.

Note: You may bookmark advanced searches and revisit them as often as you wish. Every time, you get new results for your old search criteria.

2.3.7 Data export

containing:

Column groups:

- sample
- substrate
- 5-chamber deposition
- 5-chamber deposition, 5-chamber layer
- 5-chamber deposition, 5-chamber layer #2
- 5-chamber deposition, 5-chamber layer #3
- structuring
- solarsimulator measurement
- solarsimulator measurement #2

Columns:

- SC%
- T/°C (1)
- T/°C (2)
- solarsimulator measurement
- timestamp
- operator
- comments
- irradiation
- temperature/°C
- η of best cell/%

Below, you see a preview of the table. If you export it by clicking on the button, you get the table in CSV format. This should be importable by any table-processing program. It has the following properties, which you may have to specify when importing the data:

1. The columns are *tabulator-separated* ("TAB").
2. The file is encoded in *UTF-8*.

Note that depending on the MS Excel version number, it may be easier to import the table into Excel by saving the file with the extension ".txt" before importing it.

		η of best cell/% (solarsimulator measurement)	SiH ₄ /sccm (5-chamber deposition, 5-chamber layer #2)
✓	145-002	8.83	0.000
✓	145-003	10.4	1.000

Submit

Rosalee needs the data in her spreadsheet program. So, click yet another time on "Submit". You may select the processes on the sample data sheet that should be included into the output. Select the second layer of the 5-chamber deposition and the first solarsimulator measurement. Click on "Submit". Now, you may select the fields of these processes that should be included into the output. Select "SiH₄/sccm" (the silane flux, by the way) of the layer and " η of best cell/%" of the solarsimulator measurement. Click on "Submit".


The result is shown in the screenshot. The table comprises all the data that will be included into the output. Click one last time on "Submit", and you can download that table as a CSV file ready-to-be-opened with your favourite spreadsheet program. When opening it, take care that columns are separated *only* by tabstops.

2.3.8 Add samples

From the main menu, you can click on "Add things – Samples" to add samples. Note that this page is quite institute-specific. Your institute may not have the concept of substrates, for example, and surely not something like a "cleaning number". Anyway, you must enter the number of samples as well as their current location. Add a couple of samples, but don't rename them yet.

Fresh samples have a provisional name in JuliaBase. It looks like "*00034", i.e., an asterisk followed by a five-digit number. Never use these names on sample boxes or in lab notebooks. They are meant to be replaced by a real name as quickly as possible. Rosalee's samples get their names after the first deposition of silicon, so let's do that now.

2.3.9 Lab notebooks

From the main menu, open the lab notebook of the five-chamber deposition. You see six depositions of October 2014. Select one of them. JuliaBase shows you a page containing the details of only this deposition. At the top of it, click on the gear icon  in order to duplicate this deposition.

2.3.10 Add new deposition process

Rosalee duplicates old depositions because she doesn't vary much. This way, she adds new depositions without fuss. In the page for the new deposition, she only has to select the samples for the deposition (which are the samples recently added, with these "...-a" names), change some other things that were different in this run, and click on "Submit".

Old sample name	New name	Pieces	New sample name
*00001	14S-007-a	1	14S-007
*00002	14S-007-b	1	14S-007
*00003	14S-007-c	1	14S-007

New current location: 5-chamber deposition lab (for all samples; leave empty for no change)

Submit

Now, it's a habit in the Institute of Nifty New Materials to give the sample the same name as the deposition. Therefore, immediately after having added the deposition, you are redirected to a page where you can check and change the new sample names. JuliaBase suggests the deposition's name for all samples (in the case of the screenshot, three of them). However, names must be unique, so Rosalee appends "...-a", "...-b", and "...-c" (see screenshot, second column). Click on "Submit", that's it! The newly deposited samples appear with their proper names under "My Samples" on the main menu page.

Of course, your institute may have another workflow without such renaming, which is a bad idea anyway – names in a database should never change. So you can just leave out the renaming page in your own code.

2.3.11 Change permissions for processes

	can add	can view all	can edit all	can change permissions
5-chamber depositions	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Submit

As already mentioned, Rosalee is responsible for the 5-chamber deposition setup. But let's assume Eddie wants to do such depositions, too, and gets an introduction? Then, he should also be allowed to add such depositions to JuliaBase.

Rosalee visits "Miscellaneous – Permissions to processes" from the main menu, selects Eddie from the drop-down menu and clicks on "Submit". She puts tick marks into the first two checkboxes and clicks again on "Submit". Now Eddie has got the following additional permissions:

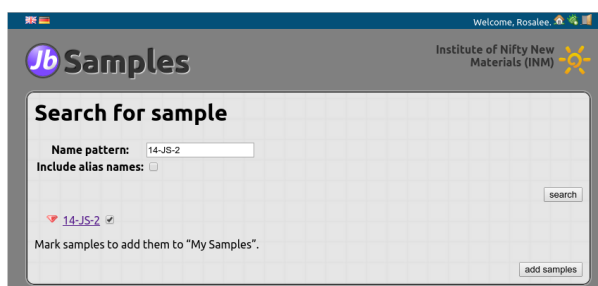
- He can add new 5-chamber depositions.
- He can edit his own 5-chamber depositions (those that he's the operator of).

- He can view all 5-chamber depositions. In particular, this implies that he can view the lab notebook.

2.3.12 Claims of samples

There is one of Juliette's samples that Rosalee wants to acquire possession of. In principle, Juliette could set the sample's "currently responsible person" to Rosalee, but Juliette is reluctant to do work that also other could do (more on that later).

Moreover, sometimes you must acquire possession of orphaned samples; or of samples that were imported as legacy data without any ownership information. In these cases, it is really *necessary* to be able to claim samples. This is done in two steps.

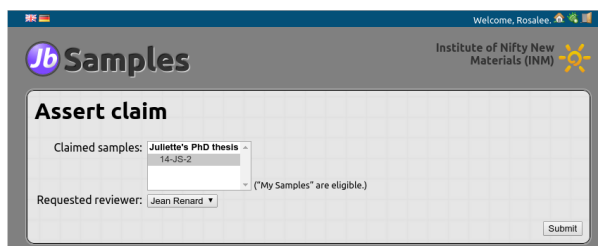


The screenshot shows a web interface for 'Jb Samples' with a header 'Welcome, Rosalee.' and 'Institute of Nifty New Materials (INM)'. The main section is titled 'Search for sample'. It contains a 'Name pattern:' input field with '14-JS-2' entered, and an 'Include alias names:' checkbox which is checked. A 'search' button is on the right. Below the search area, there is a red heart icon next to '14-JS-2' and a link icon. A text prompt says 'Mark samples to add them to "My Samples".' and an 'add samples' button is at the bottom right.

Adding a sample to My Samples

Rosalee clicks on "Search for things – Samples by name" in the main menu and enters the name "14-JS-2" into the field. She puts a tick into the checkbox and clicks on "add samples". This way, the sample 14-JS-2 is added to Rosalee's "My Samples".

The actual claim



The screenshot shows the 'Assert claim' form. It has a 'Claimed samples:' dropdown menu with 'Juliette's PhD thesis' and '14-JS-2' selected. Below it is a 'Requested reviewer:' dropdown menu with 'Jean Renard' selected. A note says '("My Samples" are eligible.)'. A 'Submit' button is at the bottom right.




Next, Rosalee visits "Miscellaneous – Sample claims" from the main menu, and on the next page, selects "existing samples". Here, she selects 14-JS-2, and "Sean Renard" as the reviewer of the claim (he's the only choice anyway). That's it. The next page lets Rosalee review the claim. But now she has to wait for Sean (who got an automatic email) for approving it.

2.4 Juliette: The assigner of work

Juliette has a lot to do and cannot deal with such things as sample preparation and characterization itself. Thus, she assigns tasks to other people and analyses the results. Logout and re-login as j.silverton/12345.


2.4.1 Adding a task

Let us assume Juliette wants to have a PDS measurement for her sample 14-JS-1. Therefore, visit “Miscellaneous – Task lists” from the main menu. There, first set up the page by selecting the processes that you’re interested in. Select “PDS measurements” and click on “Submit”.

Now you add a new task for the PDS setup by clicking on the plus icon  for PDS measurements. Select the sample 14-JS-1, click on “Submit” and you’re finished. You can see the new task in the list of tasks. There, you may withdraw it by clicking on the minus icon , or edit it by clicking on the pencil icon .

2.4.2 Sending a sample to another user

Juliette wants to show the sample 14-JS-1 to Nick so that he can have a look at it. Of course, Nick could look for the sample himself, but since the sample is in the topic “Juliette’s PhD thesis” and Nick isn’t, he cannot view the sample’s data sheet.

To send the sample to Nick, click on the pencil icon  next to “My samples” on the main menu page. Select the sample 14-JS-1 on the left. Then, on the right, select Nick in the multiple choice “Copy to user” and enter, say, “Please have a look at this sample” at “Comment for recipient”. Finally, set “Clearance” to “all processes up to now”, because Juliette wants Nick to be able to see the whole data sheet of 14-JS-1.

2.5 Nick: Technical service for others

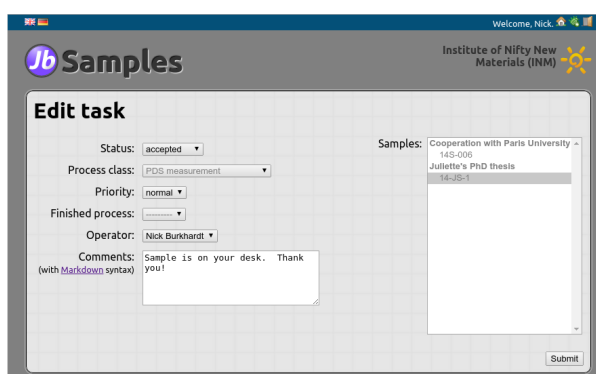
Now login as n.burkhardt/12345. You can see 14-JS-1 under “My Samples”, and you can view its data sheet. The transfer has worked.

2.5.1 The newsfeed

Moreover, Nick has been notified by the transfer in “Main menu – Miscellaneous – Newsfeed”. There, he can also see that Juliette has filed a new task for PDS measurements because Nick has the necessary permissions for the PDS. The newsfeed contains all important news for the respective user: Changes in their samples, new samples in their topics, samples transferred to them, new tasks, and much more.


The newsfeed is not really intended to be view in the browser. You may do so, but it is a little bit awkward. Rather, use a program capable of RSS feeds like Thunderbird. It is able to show you which entries in the feed are really new.

2.5.2 Tasks



Since Nick has read that Juliette had filed a new PDS task, he visits the “Task lists” page himself. When doing this the first time, you have to select the PDS and click on “Submit” to make PDS tasks viewable to Nick.

In general, there are more than one person working at a setup like the PDS. Sometimes, people are absent (holidays, illness, etc). Therefore, it is not a-priori clear who will actually do a task, and the task must be explicitly accepted by someone and assigned to someone. In order to do this, click on the pencil icon to edit it. Set the “status” to “accepted” and “operator” to Nick himself. Juliette will get notified of this.

Nick can edit all PDS tasks by clicking on the pencil icon . When Nick is actually doing the measurement, he may set the task’s status to “in progress”, and after that, to “finished”. A finished task may even be connected with the concrete PDS measurement.

Some of these steps are optional. It depends on your workflow. An operator might only set finished tasks to “finished” without further ado. Or he may use all of the features offered by task lists. Or anything in between.

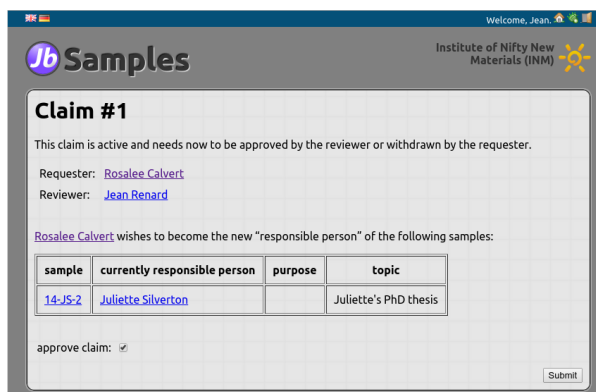
2.6 Sean: The team leader

Login as s.renard/12345. Sean, being the team leader, has extended permissions. They are:

- View all samples
- Create new topics
- Change memberships in all topics
- Grant and revoke permissions to all setups
- Approve or reject sample claims

The Institute of Nifty New Materials only has two levels: The team leader and the rest. You may add further levels in your institution, and you may set the permissions in a different way. However, we’ve made the experience that complex permission policies are a burden that should be avoided.

2.6.1 Approve a sample claim



Claim #1

This claim is active and needs now to be approved by the reviewer or withdrawn by the requester.

Requester: [Rosalee Calvert](#)
Reviewer: [Jean Renard](#)

[Rosalee Calvert](#) wishes to become the new “responsible person” of the following samples:

sample	currently responsible person	purpose	topic
14-JS-2	Juliette Silverton		Juliette's PhD thesis

approve claim: ☒

[Submit](#)

Visit on the main menu “Miscellaneous – sample claims”. At the bottom of this page, you see the sample claim of Juliette. Click on it. Sean can now review it in detail, and approve or reject it.

INSTALLATION

3.1 Prerequisites

Basically, you only need the current version of the [Django web framework](#) together with its prerequisites. Typically, this will be a computer with a Linux operating system, and with Apache and PostgreSQL running on it. However, Django is flexible. It also runs on a Windows server, and it may be combined with different web servers and database backends. See [Django's own installation guide](#) for more information. Still, in this document, we assume the default setup, which we also strongly recommend: Linux, Apache, PostgreSQL. We deliberately avoid mentioning any particular Linux distribution because we assume that at least their server flavours are similar enough. For what it's worth, the authors run Ubuntu Server.

Mostly, no sophisticated finetuning of the components is necessary because JuliaBase deployments will serve only a few (< 1000) people. In particular, PostgreSQL and Apache can run in default configuration by and large. On the other hand, the computer should be a [high-availability system](#), possibly realized with virtual machines. In our own installation, we manage a three-nines system, i.e. 99.9 % availability. Additionally, regular backups are a must! To set up these things, however, is beyond the scope of this document. Your IT department may turn out to be very helpful with this.

In the following, we'll show you how to get JuliaBase up and running quickly. While our way is already useful for a production system, you may wish or need to do it in a different way. Thus, consider the following a good starting point for your own configuration.

3.2 Linux configuration

Additionally to the software that is running on any recent and decent Linux operating system by default anyway, you must install:

- Apache2
- PostgreSQL (and the Python module "psycopg2" for it)
- memcached (and the Python module for it)
- matplotlib
- reportlab

3.3 PostgreSQL

If you have PostgreSQL and Apache on the same computer, PostgreSQL's default configuration should work for you. The defaults are quite restrictive, so they can be considered secure. Otherwise, if you need to change something, it is probably in `pg_hba.conf` (where the [user authentication](#) resides) or `postgresql.conf` (where the [general configuration](#) resides), both of which are typically found in `/etc/postgresql/version/main/`.

Anyway, you create a PostgreSQL user with this:

```
username@server:~$ sudo -u postgres psql
psql (9.3.4)
Type "help" for help.

postgres=# CREATE USER username WITH PASSWORD 'topsecret' CREATEDB;
CREATE ROLE
postgres=# \q
```

In this snippet, you have to replace `username` with your UNIX user name, and `topsecret` with a proper password, which shouldn't be your UNIX login password. Finally, create the database with:

```
username@server:~$ createdb juliabase
```

3.4 Django

A certain version of JuliaBase works only with a certain version of Django. Currently, this is Django 1.7. Install it according to [Django's own instructions](#). Not further configuration is necessary.

3.5 JuliaBase

JuliaBase is organized in a public [Git repository on GitHub](#). So far, there is no public release of JuliaBase 1.0. However, the master branch in the repository is a release candidate, which can be cloned locally with

```
username@server:~$ git clone https://github.com/juliabase/juliabase.git
```

It contains three Django apps:

1. `jb_common`
2. `samples`
3. `institute`

"`jb_common`" implements the basic JuliaBase functionality. On top of that, "`samples`" implements the actual samples database. And on top of that, "`institute`" implements code that is specific to the specific institution or department or work group that wants to use JuliaBase. "`institute`" implements a *generic* institute. You will replace "`institute`" with your own app.

3.6 Apache

Add to your Apache configuration something like the following:

```
<VirtualHost *:80>
  ServerName juliabase.example.com
  WSGIScriptAlias / /home/username/myproject/mysite/wsgi.py
  XSendFile on
  XSendFilePath /
  Alias /media /var/www/juliabase/media
  <Location "/">
    Order allow,deny
    Allow from all
    Require all granted
  </Location>
</VirtualHost>
```

This snippet contains several parts that highly probably need to be adjusted by you, in particular `juliabase.example.com`, `username`, and all paths in general. But this should be obvious. The proper place for it depends on your Linux variant. It may be the (new) file `/etc/apache2/httpd.conf`, or a new file in `/etc/apache2/conf.d`, or a new file in `/etc/apache2/sites-available` with a symlink in `/etc/apache2/sites-enabled`.

PROGRAMMING

This document explains JuliaBase for the programmer who wants to adapt it to their institute, research department, or scientific group. It contains an overview of the process as a whole, and refers to other pages with the details. We hope that it serves as a gentle tutorial which makes the adaption process as easy as possible. Feedback is welcomed!

For the adaption process, you should be familiar with several technologies:

1. *Python*. You should have advanced experience in this language. This includes the standard library; you should at least know what it can do and how to find information about it.
2. *Django*. You must have mastered the tutorial of the Django web framework.
3. *HTML*. Basic knowledge should be enough.

Furthermore, some admin skills are necessary to get everything running.

4.1 Organizing your source code

It would be a bad idea to download JuliaBase's source code and modify it directly to your needs because then, any JuliaBase update would destroy your changes. Instead, you make a structure according to this:

```
myproject/  
  manage.py  
  juliabase/  
    {the original JuliaBase release}  
  mysite/  
    __init__.py  
    settings.py  
    urls.py  
    wsgi.py  
  institute/  
    __init__.py  
    admin.py  
    urls.py  
    migrations/  
    models/  
    static/  
    templates/  
    views/  
    ...
```

Thus, follow these steps:

1. Create the directory `myproject/`
2. Copy a JuliaBase release to `myproject/juliabase`

3. Copy the file `myproject/juliabase/manage.py` to `myproject/`.
4. Create `myproject/mysite/`
5. Copy the files `settings.py`, `urls.py`, and `wsgi.py` from `myproject/juliabase/` to `myproject/mysite/`
6. Create an empty file `myproject/mysite/__init__.py`
7. Copy recursively the directory `myproject/juliabase/institute` to `myproject/`.

4.1.1 Git subtree

If you're using Git, you may consider using the `subtree` command to get JuliaBase in your repo: You structure your repository like above but without the `juliabase` subdirectory. Then, you say:

```
username@server:~/myproject$ git subtree add --prefix juliabase --squash \
https://github.com/juliabase/juliabase.git v1.0
```

A new version is pulled into your repo with:

```
username@server:~/myproject$ git subtree pull --prefix juliabase --squash \
https://github.com/juliabase/juliabase.git v2.0
```

4.2 Creating a new Django app

Okay, now let's dive into the serious stuff.

You created your Django project, you added JuliaBase to it as explained in [Organizing your source code](#). Furthermore, you set up everything as explained in [Installation](#) (Apache is not yet needed). Let's try to get it running with

```
username@server:~/myproject$ ./manage.py migrate
username@server:~/myproject$ ./manage.py loaddata demo_accounts
username@server:~/myproject$ ./manage.py collectstatic
username@server:~/myproject$ ./manage.py runserver
```

This should make the site accessible locally at the URL <http://127.0.0.1:8000>.

The institute app that is used is `myproject/institute`. For far, it is a 1:1 copy of the JuliaBase app of the same name. The plan is to transform it into what *you* need by pruning and modifying it. The primary tasks when adapting the app to your group or institution are:

1. Branding.
2. Adapting the “add new samples” page.
3. Manage the physical processes.

4.2.1 Branding

For the time being, we will stay with the app name “institute” to keep the number of changes small. Remember that it is only the app name; you may re-brand the webpages to whatever you like. The central point of doing so is `institute/templates/jb_base.html`. There,

you may change the name of the institution as well as its logo. The logo file should be placed in `institute/static/institute/`.

Moreover, every JuliaBase installation must have at least one department. It needs to be created only once and should be named appropriately.

4.2.2 The “add new samples” view

JuliaBase respects that creating new samples is a rather institute-specific procedure and therefore does not include a view for this. Instead, you must create one, but you may use the INM’s view in `institute/views/samples/sample.py` as a comprehensive starting point (in particular, the function `institute.views.samples.sample.add()`).

In the INM, every sample starts its life with a *substrate*. This is a physical process that is always the very first one in the sample’s history. Therefore, the web page where you can add new samples also asks for the substrate data, and creates the samples together with their substrates. You may or may not wish to have substrates, too.

The second big issue is sample names. Most institutions have quite idiosyncratic ideas about the sample naming policy. But JuliaBase is very flexible regarding this, see [Sample names](#). In the “add new samples” view, you may let the user input (a pattern for) the new samples right away, or you may give the names totally automatically. Or, you may do it similarly to the INM: Let the user decide between some options, and possibly redirect to a bulk-rename view after having added the samples with provisional names.

4.2.3 Physical processes

Physical processes are the thing that a JuliaBase programmer will spend most of its time on. They represent everything physically available in your institution: Measurement setups, deposition setups, clean room processes, chemical treatment, etc.

The INM app “institute” ships with some examples. You may convert them to what you need, but you can also remove them. For the latter, visit `institute/urls.py` and have a look at the following part (at the bottom):

```
pattern_generator = PatternGenerator(urlpatterns, "institute.views.samples")
pattern_generator.deposition("ClusterToolDeposition", views={"add", "edit"})
pattern_generator.deposition("FiveChamberDeposition", "5-chamber_depositions")
pattern_generator.physical_process("PDSMeasurement", "number")
pattern_generator.physical_process("Substrate", views={"edit"})
pattern_generator.physical_process("Structuring", views={"edit"})
pattern_generator.physical_process("SolarsimulatorMeasurement")
```

Here, you can simply remove a line and the process is gone. Well, not entirely: You still need to remove its views module, templates, and models in order to have everything neat and clean. But removing the URL is enough for the moment.

4.3 Adding a new process module

So you want to add a new measurement device or manufacturing process to your JuliaBase installation. You do so by adding new models, views, URLs, and possibly an electronic lab notebook to your app “institute”.

I will show how to do that step-by-step. In this example case, we write the code for layer thickness measurements.

4.3.1 Overview

The following steps are necessary for creating a physical process:

1. Create a database model in `institute/models.py`.
2. Create links in `urls.py`.
3. Create a view module in `samples/views/`. Fill the view module with an “EditView” class.
4. Create an “edit” and a “show” template in `templates/`.
5. *(Optional)* Create an electronic lab notebook.
6. *(Optional)* Create support for the new process in the Remote Client.
7. *(Optional)* Import legacy data.

In general, you will not do all of this from scratch. Instead, you will copy-and-paste from an already existing process which is as similar to the new one as possible.

4.3.2 Creating the database models

A “database model” or simply a “model” is a class in the Python code which represents a table in the database. It defines which things need to be stored for every thickness measurement. Since a model is a very Django-specific construction, see the [Django model documentation](#) for the details.

Let us assume that your thickness measurements need two fields: The measured thickness and the method that was used to measure the thickness. For the method, you want to give the user the choice between five pre-set methods.

Thus, add the following code to your `models.py`:

```
method_choices=(("profilers&edge", _(u"profilers + edge")),
                ("ellipsometer", _(u"ellipsometer")),
                ("calculated", _(u"calculated from deposition parameters")),
                ("estimate", _(u"estimate")),
                ("other", _(u"other")))

class ThicknessMeasurement(PhysicalProcess):
    thickness = models.DecimalField(_(u"layer thickness"), max_digits=6,
                                   decimal_places=2, help_text=_(u"in nm"))
    method = models.CharField(_(u"measurement method"), max_length=30,
                              choices=method_choices, default="profilers&edge")
```

The first part defines the five choices – note that it defines pairs of strings, namely the internal name, which will be written to the hard disk, and the descriptive name, which will be shown to the user. The descriptive name is enclosed by `_(...)` to make it translatable to various languages.

Try to be as restrictive as is sensible when defining your models. In particular, mark only those fields as optional that are really optional, set minimal and maximal values for numeric fields where applicable, and restrict the number of digits for decimal fields. This not only forces users to enter plausible values, it also helps debugging.

Schema migration

After you add (or change) database models, you must do a so-called schema migration. This means that the tables in the database PostgreSQL are actually changed, so that Django can use this new structure (a.k.a. schema).

It is a good idea to test a schema migration first on a test server.

The schema migration is created and applied by saying:

```
./manage.py makemigrations institute
./manage.py migrate institute
```

The first line will create a new file in `institute/migrations/`. It should be added to your repository.

4.3.3 Creating the URLs

The following work is done in `institute/urls.py`. This step is fairly simple. For the thickness measurement, you add:

```
pattern_generator.physical_process("LayerThicknessMeasurement")
```

4.3.4 Creating the view

Typically, the view is the most complex task when creating a new kind of process. The Python file containing it must be called `process_class.py`, thus in the current example, `layer_thickness_measurement.py`. It contains two parts:

1. The form(s).
2. The class `EditView` function. This is mandatory.

The form

For such a simple process class, this is simple:

```
class LayerThicknessForm(samples.utils.views.ProcessForm):
    class Meta:
        model = LayerThicknessMeasurement
        fields = "__all__"
```

View class

You only need to create a view class for *editing*, which can also be used to *adding*. (The *display* of an existing process is handled by JuliaBase.) This view function must be defined like this:

```
class EditView(ProcessView):
    form_class = LayerThicknessForm
```

For such a simple process like layer thickness measurement, that's it! For more complex processes, you may have to define further form classes, or do additional validation in an `is_referentially_valid()` method in the view class. For the full API reference, see [Class-based views](#). For more examples, see the view modules of the `institute` app in JuliaBase's source distribution.

4.3.5 Creating the templates

You need two templates per process, one that is called `edit_process_name.html` and the other that is called `show_process_name.html`. Copy them from the process which is most closely related to the one you're editing and apply the necessary modifications. Put them into the directory `institute/templates/samples/`.

4.4 A more complex example: Writing a deposition module

I will show how to write a module for a deposition system by creating an example module step-by-step. The crucial difference to the simple measurement process from above is that depositions consist of *layers*, and there can be arbitrarily many of them. Every process class that needs some sort of sub-model is more complicated, as explained in the following.

4.4.1 The models

A deposition system typically needs two models: One for the deposition data and one for the layer data. The layer data will carry much more fields than the deposition, and it will contain a pointer to the deposition it belongs to. This way, deposition and layers are kept together. This pointer is represented by a “foreign key” field.

The deposition model is derived from `Deposition`, which in turn is a `Process`:

```
class FiveChamberDeposition(samples.models.Deposition):
    class Meta(samples.models.PhysicalProcess.Meta):
        permissions = generate_permissions(
            {"add", "change", "view_every", "edit_permissions"}, "FiveChamberDeposition")
```

It contains a full set of permissions to limit “add” and “edit” access to certain users. Moreover the `view_every` makes a lab notebook possible. See [Model permissions](#) for further information.

In contrast, the layer model is derived from `Layer`, which in turn is an ordinary Django model (not a `Process`):

```
class FiveChamberLayer(samples.models.Layer):
    deposition = models.ForeignKey(
        FiveChamberDeposition, related_name="layers", verbose_name=_("deposition"))
    layer_type = models.CharField(
        _("layer type"), max_length=2, choices=five_chamber_layer_type_choices, blank=True)
    chamber = models.CharField(
        _("chamber"), max_length=2, choices=five_chamber_chamber_choices)
    sih4 = model_fields.DecimalField(
        "SiH4", max_digits=7, decimal_places=3, unit="sccm", null=True, blank=True)
    h2 = model_fields.DecimalField(
        "H2", max_digits=7, decimal_places=3, unit="sccm", null=True, blank=True)
    temperature_1 = model_fields.DecimalField(
        _("temperature 1"), max_digits=7, decimal_places=3, unit="°C", null=True, blank=True)
    temperature_2 = model_fields.DecimalField(
        _("temperature 2"), max_digits=7, decimal_places=3, unit="°C", null=True, blank=True)

    class Meta(samples.models.Layer.Meta):
        unique_together = ("deposition", "number")
```

The most important thing here is the `deposition` field which points to the deposition this layer belongs to. It forms part of a `unique_together` declaration. The other fields are ordinary data fields.

4.4.2 Populating user context

In order to enable users to duplicate existing depositions, you should override `get_context_for_user()` method in the deposition model:

```
def get_context_for_user(self, user, old_context):
    context = old_context.copy()
    if permissions.has_permission_to_add_physical_process(user, self.__class__):
        context["duplicate_url"] = "{0}?copy_from={1}".format(
            django.core.urlresolvers.reverse("add_five_chamber_deposition"),
            urlquote_plus(self.number))
    else:
        context["duplicate_url"] = None
    return super(FiveChamberDeposition, self).get_context_for_user(user, context)
```

This method is used when the HTML for a process (in this case a deposition) is created. Its return value is a dictionary which is combined with the dictionary sent to the `show_process_class.html` template. This way, additional program logic can be used to generate the HTML. In case of depositions, a “duplicate” button can be added, depending on the user’s permissions.

4.4.3 The view

In the view module which must be called `five_chamber_deposition.py`, the main form gets additional cleaning methods:

```
class DepositionForm(samples.utils.views.DepositionForm):

    class Meta:
        model = institute.models.FiveChamberDeposition
        fields = "__all__"

    def clean_number(self):
        number = super(DepositionForm, self).clean_number()
        return form_utils.clean_deposition_number_field(number, "S")

    def clean(self):
        cleaned_data = super(DepositionForm, self).clean()
        if "number" in cleaned_data and "timestamp" in cleaned_data:
            if cleaned_data["number"][:2] != cleaned_data["timestamp"].strftime("%y"):
                self.add_error("number", _("The first two digits must match the year of the deposition."))
        return cleaned_data
```

The view class must override `get_next_id()` because the ID of a deposition (its number) is non-numerical in the INM:

```
class EditView(RemoveFromMySamplesMixin, DepositionView):
    form_class = DepositionForm
    layer_form_class = LayerForm

    def get_next_id(self):
        return institute.utils.base.get_next_deposition_number("S")
```

4.4.4 The lab notebook

There are two things to set up for electronic lab notebooks: The URL and the template.

Adding the URL for depositions is trivial as the method `samples.utils.urls.PatternGenerator.deposition()` by default also creates a lab notebook URL:

```
pattern_generator.deposition("FiveChamberDeposition", "5-chamber_depositions")
```

For normal processes, you need to request the lab notebook URL explicitly:

```
pattern_generator.physical_process("ConductivityMeasurement",  
    views={"add", "edit", "lab_notebook"})
```

Finally, you need to create a template called `lab_notebook_process_class.html`. It contains the processes to be displayed in a lab notebook table in the context variable `processes`.

4.5 Process glossary

process Anything that contains information about a sample. This can be a process in the literal meaning of the word, i.e. a deposition, an etching, a clean room process etc. It can also be a measurement or a result. However, even the substrate, sample split, and sample death are considered processes in JuliaBase.

It may have been better to call this “history item” or just “item” instead of “process”. The name “process” is due to merely historical reasons, but there we go.

measurement A special kind of *process* which contains a single measurement. It belongs to the class of *physical processes*.

physical process A deposition or a measurement process. Its speciality is that only people with the right permission for a certain physical process are allowed to add and edit physical processes.

result A result – or result process, as it is sometimes called in the source code – is a special process which contains only a remark, a picture, or a table with result values.

MODEL PERMISSIONS

Permissions defined in the `Meta` class of models are used in Django to define user permissions connected with this model. JuliaBase also makes use of this permissions framework. There are, however, some peculiarities to be taken into account when defining model permissions for JuliaBase model classes.

5.1 Semantics and conventions

For models derived from `PhysicalProcess`, there are four permissions with a special meaning to JuliaBase. Their codenames must follow the following naming conventions so that they have effect.

`add_classname` Means that a user is allowed to add new processes, and to edit unfinished processes.

`edit_permissions_for_classname` Means that a user is allowed to edit the permissions of other users for this process class.

`view_every_classname` Means that a user is allowed to view all processes. In particular, such users are allowed to read the lab notebook.

`change_classname` Means that a user is allowed to edit *all* processes.

Further rules:

- *classname* must be given in lowercase letters without underscores.
- If a user has the permission `add_classname`, this user can edit processes he/she is the operator of.
- You can view processes of samples that you can view.
- For obvious reasons, the `edit_permissions_for_classname` permission implies all the others. Usually, the users in charge of this setup or apparatus have this permission.

5.1.1 Example

The following code snippet defines the permissions for the `ClusterToolDeposition`:

```
class Meta(samples.models.PhysicalProcess.Meta):
    permissions = (("add_clustertooldeposition", "Can add cluster tool depositions"),
                  ("edit_permissions_for_clustertooldeposition",
                   "Can edit perms for cluster tool I depositions"),
                  ("view_every_clustertooldeposition",
                   "Can view all cluster tool depositions"),
```

```
("change_clustertooldeposition",  
"Can edit all cluster tool depositions"))
```

Using `jb_common.utils.base.generate_permissions()`, this can be heavily simplified:

```
class Meta(samples.models.PhysicalProcess.Meta):  
    permissions = generate_permissions(  
        {"add", "change", "view_every", "edit_permissions"}, "ClusterToolDeposition")
```

5.2 Omitting permissions

You may define all four permissions above. However, if you omit some of them, this has influence on JuliaBase's treatment of that process class. The obvious effect of omitting a permission is that no user can have that permission. But there are also more subtle effects.

If you omit the `add_...` permission, *every* user is allowed to add such a process. This may be suitable for things like specimen tempering, etching, or thickness measurements that are not bound to a specific apparatus.

If you omit the `edit_permissions_for_...` permission, the process class will not appear in the “*Permissions to processes*” page. Moreover, no email is sent to a person in charge of the setup if a user creates his/her very first process of that kind.

5.3 Django's default permissions

By default, Django generates an `add_...`, `change_...`, and `delete_...` permission for every model. You can switch it off for a certain model by saying

```
class Meta:  
    default_permissions = ()
```

For physical processes, this has been done already — this is the reason why we derived our Meta class from `samples.models.PhysicalProcess.Meta` in the above [Example](#).

We recommend you to switch off Django's default permissions globally for your project. This way, it's much easier to control which permissions exist for a certain model. You switch them off by saying in your `manage.py`:

```
import django.contrib.auth.management  
def _get_only_custom_permissions(opts, ctype):  
    return list(opts.permissions)  
django.contrib.auth.management._get_all_permissions = _get_only_custom_permissions
```


CLASS-BASED VIEWS

Class-based views are highly practical for the add/edit view of physical processes because they keep code duplication at a minimum. In some cases, you get away with only a few lines of code. Mixin classes reduce the redundancy further. Although it is still possible to have ordinary view *functions* for physical processes, we do not recommend this. If you follow the convention of calling your view class “`EditView`” and place it in a module called `class_name.py`, the `PatternGenerator` will detect it and create the URL dispatch for it.

6.1 The API

The API of JuliaBase’s class-based view classes is best described by discussing the attributes and methods of the common base class `ProcessWithoutSamplesView`. Not only if you derive your views, but also if you need to define your own *abstract* view class, you should derive it from one of the concrete classes presented in the next section, though, because you probably want to re-use part of their functionality.

This class is found in the module `samples.utils.views.class_views`.

class `ProcessWithoutSamplesView` (*kwargs*)**

Abstract base class for the classed-based views. It deals only with the process per se, and in particular, with no samples associated with this process. This is done in the concrete derived classes `ProcessView` (one sample) and `ProcessMultipleSamplesView` (multiple samples). So, you should never instantiate this one.

The methods that you most likely want to redefine in you own concrete class are, with decreasing probability:

- `is_referentially_valid()`
- `save_to_database()`
- `get_next_id()`
- `build_forms()`
- `get_title()`
- `get_context_data()`

Note that for `is_referentially_valid()`, `save_to_database()`, `build_forms()`, and `get_context_data()`, it is necessary to call the inherited method.

Since you connect forms with the view class, the view class expects certain constructor signatures of the forms. As for the process model form, it must accept the logged-in user as the first argument. This is the case for `ProcessForm` and `DepositionForm`, so this should not be a problem. The derived class (see below) may impose constraints on their external forms either.

Variables

- **form_class** – The model form class of the process of this view.
- **model** – The model class of the view. If not given, it is derived from the process form class.
- **class_name** – The name of the model class, e.g. "clustertooldeposition".
- **process** – The process instance this view is about. If we are about to add a new process, this is `None` until the respective form is saved.
- **forms** – A dictionary mapping template context names to forms, or lists of forms. Mandatory keys in this dictionary are "process" and "edit_description". (Derived classes add "sample", "samples", "remove_from_my_samples", "layers", etc.)
- **data** – The POST data if we have a POST request, or `None` otherwise.
- **id** – The ID of the process to edit, or `None` if we are about to add a new one. This is the recommended way to distinguish between editing and adding.
- **preset_sample** – The sample with which the process should be connected by default. May be `None`.
- **request** – The current request object. This is inherited from Django's view classes.
- **template_name** – The file name of the rendering template, with the same path syntax as in the `render()` function.
- **identifying_field** – The name of the field in the process which is the poor man's primary key for this process, e.g. the deposition number. It is taken from the model class.

build_forms()

Fills the `forms` dictionary with the forms, or lists of them. In this base class, we only add "process" itself and "edit_description". Note that the dictionary key is later used in the template as context variable.

This method has no parameters and no return values, `self` is modified in-situ. It is good habit to check for a key before setting it, allowing derived methods to set it themselves without doing double work.

get_context_data(kwargs)**

Generates the template context. In particular, we inject the forms and the title here into the context. This method is part of the official Django API.

Return the context dict

Return type dict

get_next_id()

Gets the next identifying value for the process class. In its default implementation, it just takes the maximal existing value and adds 1. This needs to be overridden if the identifying field is non-numeric.

Return The next untaken of the identifying field, e.g. the next free deposition number.

Return type object

get_title()

Creates the title of the response. This is used in the <title> tag and the <h1> tag at the top of the page.

Return the title of the response

Return type unicode

is_all_valid()

Checks whether all forms are valid. Unbound forms – which may occur also in POST requests – are not checked. Moreover, this method guarantees that the `is_valid()` method of every bound form is called in order to collect all error messages.

Return whether all forms are valid

Return type bool

is_referentially_valid()

Checks whether the data of all forms is consistent with each other and with the database. This is the partner of `is_all_valid()` but checks the inter-relations of data.

This method is frequently overridden in concrete view classes.

Note that a `True` here does not imply a `True` from `is_all_valid()`. Both methods are independent of each other. In particular, you must check the validity of forms that you use here.

Return whether the data submitted to the view is valid

Return type bool

save_to_database()

Saves the data to the database.

Return the saved process instance

Return type `samples.models.PhysicalProcess`

startup()

Fetch the process to-be-edited from the database and check permissions. This method has no parameters and no return values, `self` is modified in-situ.

6.2 Main classes

The following names are found in the module `samples.utils.views`.

class ProcessView (**kwargs)

View class for physical processes with one sample each. The HTML form for the sample is called `sample` in the template. Typical usage can be very short:

```
from samples.utils.views import ProcessForm, ProcessView

class LayerThicknessForm(ProcessForm):
    class Meta:
        model = LayerThicknessMeasurement
        fields = "__all__"

class EditView(ProcessView):
    form_class = LayerThicknessForm
```

class ProcessMultipleSamplesView (**kwargs)

View class for physical processes with one or more samples each. The HTML form for the sample list is called `samples` in the template. The usage is analogous to `ProcessView`.

class DepositionView (**kwargs)

View class for depositions. The layers of the deposition must always be of the same type. If they are now, you must use `DepositionMultipleTypeView` instead. Additionally to `form_class`, you must set the `layer_form_class` class variable to the form class to be used for the layers.

The layer form should be a subclass of `SubprocessForm`.

class DepositionMultipleTypeView (**kwargs)

View class for depositions the layers of which are of different types (i.e., different models). You can see it in action in the module `institute.views.samples.cluster_tool_deposition`. Additionally to the class variable `form_class`, you must set:

Variables

- **layer_form_classes** – This is a tuple of the form classes for the layers
- **short_labels** – (*optional*) This is a dict mapping a layer form class to a concise name of that layer type. It is used in the selection widget of the add-layer form.

class SubprocessForm (view, *args, **kwargs)

Model form class for subprocesses and deposition layers. Its only purpose is to eat up the `view` parameter to the constructor so that you need not redefine the constructor every time.

6.3 Mixins

class RemoveFromMySamplesMixin (**kwargs)

Mixin for views that like to offer a “Remove from my samples” button. In the template, they may add the following code:

```
{{ remove_from_my_samples.as_p }}
```

This mixin must come before the main view class in the list of parents.

class SubprocessesMixin (**kwargs)

Mixing for views that represent processes with subprocesses. Have a look at

`institute.views.samples.solarsimulator_measurement` for an example. For this to work, you must define the following additional class variables:

- `subform_class`: the model form class for the subprocesses
- `process_field`: the name of the field of the parent process in the subprocess model
- `subprocess_field`: the `related_name` parameter in the field of the parent process in the subprocess model

You should derive the model form class of the subprocess from `SubprocessForm`. This is not mandatory but convenient, see there.

In the template, the forms of the subprocesses are available in a list called `subprocesses`. Furthermore, you should include

```
{{ number.as_p }}
```

in the template so that the user can set the number of subprocesses.

This mixin must come before the main view class in the list of parents.

THE REMOTE CLIENT

The “remote client” is a programming library for contacting JuliaBase over the network. Being a library, it is not a program per se but used by your own code for the bidirectional communication with the database. The remote client communicates through a REST HTTP interface implemented in the JuliaBase server. This interface could be used directly, however, the remote client makes it much easier.

In the following, we will describe what the remote client can do and how to get it running. For the API itself, we have to refer to the documentation in the sources of the Python files in `remote_client/`. Note especially the examples in `remote_client/examples/`.

7.1 Use cases

The remote client has many applications.

7.1.1 Crawlers

Crawlers are programs that go through a bunch of data or log files and transmit their content to the database. For example, a measurement setup may write a data file for every measurement run. Every hour, a crawler is called, which scans for new files. It reads the new files and uses the content to add new measurements processes to the database. This way, new measurements automatically end up in the database. You may have more than one crawler regularly running in your institute.

The very same crawler can also be used to import legacy data. The crawler can be programmed in such a way that during its first run, it imports thousands of data sets of recent years, and from there on, it imports newly added data sets every hour.

Since crawlers run unattendedly, they must be very robust. If a data file is invalid, or if the central server is temporarily unavailable, it must gently exit and try again next time, without forgetting to add something, and without adding things twice.

Moreover, legacy data mostly is poorly structured and organized. If you think your data is the exception, think again. During writing the crawler, you will be surprised how many odd ideas and mistakes your colleagues are capable of.

Be that as it may, crawlers are the primary use case for the remote client. The remote client contains functions especially for making writing them as easy as possible.

7.1.2 Connecting the setup with the database

A more direct way than the crawler for bringing processing and measurement data into the database, albeit not always feasible, is to extend the control program of the experimental setup. For this, you must have access to the source code of the control program. Then, you can use the remote client for writing new runs immediately into the database.

However, the communication needn't be one-way. You can also use the remote client to authenticate the user, to check whether he or she is allowed to use this setup, and to check whether the sample names belong to valid and for the user accessible samples.

The remote client is written in Python, however, it is easy to use it from programs written in other programming languages, and JuliaBase ships with a Delphi binding.

7.1.3 Data mining and analysis

Everyone with browser access to the JuliaBase database can use the remote client for accessing, too. The permissions will be the same in both cases, of course. This can be used by researchers capable of programming to get data from the database and do something with it, e.g. creating statistics or evaluating raw data.

Imagine the following: A researcher makes many sample series and measures their temperature-dependent conductivity. The researcher can write a program that extracts the names of the raw data files from the database, then creates a plot for each sample series with the conductivity curves of all samples of that series, and writes these plots back to the database. There, one can see them on the series' pages, and in every sample's data sheet.

7.2 Extending the remote client

As for JuliaBase itself, also the remote client should be extended with the functionality special to your institution. This is not absolutely necessary, but without it, you can only use the very basic functionality. However, extending the remote client is *much* easier than extending JuliaBase. In JuliaBases' source code, the default remote client resides in the directory `remote_client/`, and it has the following structure:

```
remote_client/
  jb_remote_inm.py
  jb_remote/
    __init__.py
    common.py
    samples.py
    settings.py
    ...
```

Replace the file `jb_remote_inm.py` with your own institute's code, and give your file a name derived from your institution's name. Your file should start with something like:

```
from __future__ import absolute_import
from jb_remote import *

settings.ROOT_URL = settings.TESTSERVER_ROOT_URL = "https://juliabase.my-institute.edu/"
```

From there on, you are totally free how to program your incarnation of the remote client. You may use `jb_remote_inm.py` as a source of inspiration, of course.

7.2.1 Settings

The following settings in the remote client are available and should be set in your `jb_remote_inm.py`:

ROOT_URL The URL of the production server. It must end in a slash. Default: None

TESTSERVER_ROOT_URL The URL of the test server. It must end in a slash. Default: `"https://demo.juliabase.org/"`

SMTP_SERVER The DNS name of the SMTP server used for outgoing mail. It may be used in crawlers to send success or error emails. You may add a port number after a colon. Default: `"mailrelay.example.com:587"`

SMTP_LOGIN The login name of the SMTP server. If empty, no login is performed. If not empty, TLS is used. Default: `"username"`

SMTP_PASSWORD The password used to login to the SMTP server through TLS. Default: `"password"`

EMAIL_FROM The sender used for outgoing mails. Default: `"me@example.com"`

EMAIL_TO The recipient of outgoing mails. Default: `"admins@example.com"`

7.3 Local installation and usage

Once you finished extending the default remote client, *your* remote client consists of the files listed [above](#), with `jb_remote_inm.py` substituted by your file called, say, `jb_remote_my_institute.py`. This set of files needs to be copied to the machines where the remote client is supposed to be used, and the top directory (the one with your `jb_remote_my_institute.py`) should be in `PYTHONPATH`.

Then, you use the remote client with a Python script as easy as this:

```
from jb_remote_my_institute import *

setup_logging("console")
login("juliabase", "12345")

sample = Sample("14-JS-1")
sample.current_location = "Sean's office"
sample.edit_description = "location changed"
sample.submit()

logout()
```

By the way, the files are organized in a way that you can update very conveniently: If a new version of JuliaBase is released, you simply have to replace the `jb_remote/` subdirectory with the new one.

7.4 Other programming languages

You can communicate with a JuliaBase server using any programming language. You can implement the HTTP communication directly in most modern languages. However, it is probably much easier to write a thin wrapper around the Python implementation.

For Delphi, this has already been done in `remote_client/delphi/juliabase.pas`. The code is only 150 lines long. Using it is as simple as

```
program juliabase_example;

{$APPTYPE CONSOLE}

uses
  SysUtils, juliabase;
var
  result : String;
begin
  execute_jb('juliabase', '12345',
    'sample = Sample("14-JS-1");' +
    'sample.current_location = "Sean''s office";' +
    'sample.edit_description = "location changed";' +
    'sample.submit()');
end.
```

`execute_jb` adds some boilerplate code and calls the Python interpreter, which executes the commands you passed as a string. You must install the Python interpreter and the Python version of the remote client on the machine, but the avoidance of code duplication makes up for it.

See the top comment in `remote_client/delphi/juliabase.pas` for further instructions.

7.5 Classes and functions

As explained in [Extending the remote client](#), the following names become available by saying

```
from jb_remote import *
```

connection

The connection to the database server. It is of the type `JuliaBaseConnection`.

primary_keys

A dict-like object of type `PrimaryKeys` that is a mapping of identifying keys to IDs. Possible keys are:

"users" mapping user names to user IDs.

"external_operators" mapping external operator names to external operator IDs.

"topics" mapping topic names to topic IDs.

login (*username*, *password*, *testserver=False*)

Logins to JuliaBase.

Parameters

- **username** (*unicode*) – the username used to log in
- **password** (*unicode*) – the user's password
- **testserver** (*bool*) – whether the testserver should be user. If `False`, the production server is used.

class JuliaBaseConnection

Class for the routines that connect to the database at HTTP level. This is a singleton class, and its only instance resides at top-level in this module.

open (*relative_url*, *data=None*, *response_is_json=True*)

Do an HTTP request with the JuliaBase server. If *data* is not *None*, its a POST request, and GET otherwise.

Parameters

- **relative_url** (*str*) – the non-domain part of the URL, for example `"/samples/10-TB-1"`. “Relative” may be misleading here: only the domain is omitted.
- **data** (*dict mapping unicode to unicode, int, float, bool, file, or list*) – the POST data, or *None* if it’s supposed to be a GET request.
- **response_is_json** (*bool*) – whether the content type of the response must be JSON

Returns the response to the request

Return type *object*

Raises

- **JuliaBaseError** – if JuliaBase couldn’t fulfill the request because it contained errors. For example, you requested a sample that doesn’t exist, or the transmitted measurement data was incomplete.
- **urllib.error.URLError** – if a lower-level error occurred, e.g. the HTTP connection couldn’t be established.

exception JuliaBaseError (*error_code*, *message*)

Exception class for high-level JuliaBase errors.

Variables

- **error_code** – The numerical error code. See `jb_common.utils` for further information, and the root `__init__.py` file of the various JuliaBase apps for the tables with the error codes.
- **error_message** – A description of the error. If `error_code` is 1, it contains the URL to the error page (without the domain name).

logout ()

Logs out of JuliaBase.

class PrimaryKeys

Dictionary-like class for storing primary keys. I use this class only to delay the costly loading of the primary keys until they are really accessed. This way, GET-request-only usage of the Remote Client becomes faster. It is a singleton.

Variables components – set of types of primary keys that should be fetched. For example, it may contain `"external_operators=*"` if all external operator’s primary keys should be fetched. The modules that are part of `jb_remote` are supposed to populate this attribute in top-level module code.

setup_logging (*destination=None*)

If the user wants to call this in order to enable logging, he must do so before logging in. Note that it is a no-op if called a second time.

Parameters destination (*str*) – Where to log to; possible values are:

"file" Log to `/tmp/jb_remote.log` if `/tmp` is existing, otherwise (i.e. on Windows), log to `jb_remote.log` in the current directory.

"console" Log to `stderr`.

None Do not log.

parse_timestamp (*timestamp*)

Convert a timestamp coming from the server to a Python `datetime` object. The server serialises with the `DjangoJSONEncoder`, which in turn uses the `.isoformat()` method. As long as the server does not handle timezone-aware timestamps (i.e., `USE_TZ=False`), this works.

Parameters **timestamp** (*str*) – the timestamp to parse, coming from the server. It must have ISO 8601 format without timezone information.

Returns the timestamp as a Python `datetime` object

Return type `datetime.datetime`

double_urlquote (*string*)

Returns a double-percent-quoted string. This mimics the behaviour of Django, which quotes every URL retrieved by `django.core.urlresolvers.resolve`. Because it does not quote the slash `"/"` for obvious reasons, I have to quote sample names, sample series names, deposition numbers, and non-int process "identifying fields" *before* they are fed into `resolve` (and quoted again).

format_timestamp (*timestamp*)

Serializes a timestamp. This is the counter function to `parse_timestamp`, however, there is an asymmetry: Here, we don't generate ISO 8601 timestamps (with the "T" inbetween). The reason is that Django's `DateTimeField` would not be able to parse it.

Parameters **timestamp** (*datetime.datetime*) – the timestamp to format

Returns the timestamp in the format `"YYYY-MM-DD HH:MM:SS"`.

Return type `str`

sanitize_for_markdown (*text*)

Convert a raw string to Markdown syntax. This is used when external (legacy) strings are imported. For example, comments found in data files must be sent through this function before being stored in the database.

Parameters **text** (*unicode*) – the original string

Returns the Markdown-ready string

Return type `unicode`

class Result (*id_=None, with_image=True*)

Class representing result processes.

Parameters

- **id** (*int or unicode*) – if given, the instance represents an existing result process of the database. Note that this triggers an exception if the result ID is not found in the database.
- **with_image** (*bool*) – whether the image data should be loaded, too

submit ()

Submit the result to the database.

Returns the result process ID if succeeded.

Return type int

class Sample (*name=None, id_=None*)

Class representing samples.

Parameters

- **name** (*unicode*) – the name of an existing sample; it is ignored if *id_* is given
- **id** (*int*) – the ID of an existing sample

submit ()

Submit the sample to the database.

Returns the sample's ID if succeeded.

Return type int

class TemporaryMySamples (*sample_ids*)

Context manager for adding samples to the “My Samples” list temporarily. This is used when editing or adding processes. In order to be able to link the process with samples, they must be on your “My Samples” list.

This context manager should be used like this:

```
with TemporaryMySamples(sample_ids):  
    ...
```

The code at ... can safely assume that the *sample_ids* have been added to “My Samples”. After having executed this code, those samples that hadn't been on “My Samples” already are removed from “My Samples”. This way, the “My Samples” list is unchanged eventually.

Parameters *sample_ids* (*list of int or int*) – the IDs of the samples that must be on the “My Samples” list; it may also be a single ID

SETTINGS REFERENCE

In order to have the default values for the following settings available, you must start your `settings.py` with:

```
from jb_common.settings_defaults import *
from samples.settings_defaults import *
```

We recommend to use the `settings.py` in JuliaBase's root directory as a starting point.

8.1 General JuliaBase settings

8.1.1 ADD_SAMPLES_VIEW

Default: "" (Empty string)

Name of the view in Python's dot notation which points to the view to add samples. For example:

```
ADD_SAMPLES_VIEW = "institute.views.samples.sample.add"
```

This view function must have exactly one parameter, namely the `request` instance.

8.1.2 CACHE_ROOT

Default: `"/tmp/juliabase_cache"`

The path where dispensable (in the sense of re-creatable) files are stored. JuliaBase mostly uses this directory to store images, e.g. plot files. If the path doesn't exist when the JuliaBase service is started, it is created. The default value should be changed to e.g. `"/var/cache/juliabase"`. Note that such a path needs to be created by you because JuliaBase doesn't have the necessary permissions. Also note that such a path needs to be writable by the webserver process JuliaBase is running on.

8.1.3 CRAWLER_LOGS_ROOT

Default: "" (Empty string)

Path to the crawlers' log files. In this directory, the log file for a particular process class is called `class_name.log`. Mind the spelling: `MyProcessClassName` becomes `my_process_class_name.log`.

8.1.4 CRAWLER_LOGS_WHITELIST

Default: `()` (Empty tuple)

List of process classes for which the crawler log is public, i.e. not restricted to users that are allowed to add new processes of that kind.

8.1.5 DEBUG_EMAIL_REDIRECT_USERNAME

Default: `""` (Empty string)

Username of a user to which all outgoing email should be sent if the Django setting `DEBUG=True`. If this name is invalid, in particular if it is empty, no emails are sent at all in debugging mode. This prevents embarrassment caused by emails sent to other people while merely debugging your code.

8.1.6 INITIALS_FORMATS

Default:

```
INITIALS_FORMATS = \
    {"user": {"pattern": r"[A-Z]{2,4}|[A-Z]{2,3}\d|[A-Z]{2}\d{2}",
              "description": _("The initials start with two uppercase letters. "
                               "They contain uppercase letters and digits only. "
                               "Digits are at the end.")},
     "external_contact": {"pattern": r"[A-Z]{4}|[A-Z]{3}\d|[A-Z]{2}\d{2}",
                          "description": _("The initials start with two uppercase letters. "
                                           "They contain uppercase letters and digits only. "
                                           "Digits are at the end. "
                                           "The length is exactly 4 characters.")}}
}
```

This maps the kind of initials to their properties. It must contain exactly the two keys `"user"` and `"external_contact"`. See *Initials* for more information.

8.1.7 JAVASCRIPT_I18N_APPS

Default: `("django.contrib.auth", "samples", "jb_common")`

Tuple containing all apps which contain translations that should be used in JavaScript code. The apps are named as in the Django setting `INSTALLED_APPS`. See the [Django documentation](#) for further information.

8.1.8 MERGE_CLEANUP_FUNCTION

Default: `""` (Empty string)

Name of the view in Python's dot notation which points to a function which is called after each sample merge. This function must take exactly two parameters, the sample that is merged and the sample that this sample is merged into. It is possible to leave this settings empty; then, nothing special is called.

8.1.9 NAME_PREFIX_TEMPLATES

Default: `()` (Empty tuple)

List of string templates that define possible sample name prefixes. See *Sample names* for more information.

8.1.10 SAMPLE_NAME_FORMATS

Default:

```
SAMPLE_NAME_FORMATS = {"provisional": {"possible_renames": {"default":}},  
                        "default":      {"pattern": r"[-A-Za-z_/0-9#()]*"}}
```

This setting defines which sample names are allowed in your database. It maps the names of the formats to their properties. See *Sample names* for more information.

8.1.11 THUMBNAIL_WIDTH

Default: 400

This number represents the width in pixels of the thumbnails of plots and images that are generated for the sample data sheet.

8.1.12 USE_X_SENDFILE

Default: `False`

If `True`, JuliaBase assumes that the web server is interpreting the `X-Sendfile` header and serves a static file (e.g. an image) itself. Otherwise, JuliaBase serves the file, which may be slower and more time-consuming. In most cases, it doesn't matter, though.

8.2 Settings for LDAP

8.2.1 LDAP_ACCOUNT_FILTER

Default: `"(! (userAccountControl:1.2.840.113556.1.4.803:=2))"`

LDAP filter for filtering LDAP members that are eligible for JuliaBase access. The default filter finds any member which is not inactive. The default value works well for Active Directory domain controllers.

8.2.2 LDAP_ADDITIONAL_ATTRIBUTES

Default: `()` (Empty tuple)

JuliaBase limits the attributes it receives for every user to a certain subset, e.g. the user's real name and their department. If your code needs additional LDAP attributes, put their names into this tuple. Note that its members must be byte strings. An example might be:

```
LDAP_ADDITIONAL_ATTRIBUTES = (b"telephoneNumber", b"msExchUserCulture",  
                              b"physicalDeliveryOfficeName")
```

8.2.3 LDAP_ADDITIONAL_USERS

Default: {} (Empty dict)

Dictionary mapping user names to JuliaBase department names. This contains users that are in the LDAP directory but are not in one of the departments listed in the setting `LDAP_DEPARTMENTS` explained below. The use case is that some people working in the organization but not in the department(s) may still be eligible for database access. By putting them in `LDAP_ADDITIONAL_USERS`, they are allowed to login. They are associated with the department they are mapped to.

8.2.4 LDAP_DEPARTMENTS

Default: {} (Empty dict)

Dictionary mapping LDAP department names to JuliaBase department names. If your LDAP directory data sets contain the “department” attribute, this setting determines which department get access to JuliaBase. If this setting is empty, all LDAP members get access.

If the LDAP doesn’t contain the “department” attribute, this setting should be empty.

8.2.5 LDAP_GROUPS_TO_PERMISSIONS

Default: {} (Empty dict)

Dictionary mapping LDAP group names to sets of Django permission names. Use the Django codename of the permission, without any app label. An example might be:

```
LDAP_GROUPS_TO_PERMISSIONS = {  
    "TG_IEF-5_teamleaders": {"view_every_sample", "adopt_samples",  
                             "edit_permissions_for_all_physical_processes",  
                             "add_externaloperator",  
                             "view_every_externaloperator",  
                             "add_topic", "change_topic"}  
}
```

Note that you should not change permissions in JuliaBase’s admin interface that occur in `LDAP_GROUPS_TO_PERMISSIONS`. They will be overwritten during the next synchronization with the LDAP directory (in particular, at next user login). Consider these permissions being managed exclusively automatically.

8.2.6 LDAP_LOGIN_TEMPLATE

Default: "{username}"

This pattern is used to bind to (a.k.a. login into) the LDAP server. JuliaBase uses this binding only to check whether the user’s credentials (login, password) are valid. {username} is replaced by the username of the user that tries to login into JuliaBase. A typical value for this setting is

```
LDAP_LOGIN_TEMPLATE = "{username}@mycompany.com"
```

8.2.7 LDAP_SEARCH_DN

Default: "" (Empty string)

The “distinguished name” (DN) which should be used as the base of the search for user details in the LDAP directory. It is typically something like:

```
LDAP_SEARCH_DN = "DC=ad,DC=mycompany,DC=com"
```

8.2.8 LDAP_URLS

Default: () (Empty tuple)

List of URLs of LDAP directories. If you want to use LDAP, this must contain at least one URL. It may contain more if there are multiple redundant LDAP servers. In this case, JuliaBase will try each of them until it finds a working one. An example value may be:

```
LDAP_URLS = ["ldaps://dc-e01.ad.mycompany.com:636"]
```

Here, 636 is the port number of LDAP-over-TLS.

8.3 Django settings with special meaning in JuliaBase

Note that JuliaBase does not change the meaning or the default value of Django settings.

8.3.1 LANGUAGES

This settings determines which flags to offer at the top of the screen. Since JuliaBase is available in English and German so far, a sensible value may be:

```
LANGUAGES = (("de", _("German")), ("en", _("English")))
```

Note that the `_(...)` makes the language names themselves translatable. To get this working, you must import `gettext_lazy` into `settings.py`:

```
from django.utils.translation import gettext_lazy as _
```

8.3.2 CACHES

JuliaBase makes heavy use of Django’s cache framework. Thus, we recommend to configure an efficient caching backend like memcache:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.MemcachedCache",
        "LOCATION": ["localhost"],
        "TIMEOUT": 3600 * 24 * 28
    }
}
```

8.3.3 DEBUG

JuliaBase behaves slightly differently if `DEBUG=True`. In particular, all outgoing emails are redirected to `DEBUG_EMAIL_REDIRECT_USERNAME`.

8.3.4 DEFAULT_FROM_EMAIL

JuliaBase uses this Django setting also for its own outgoing emails.

8.3.5 INSTALLED_APPS

The minimal set of installed apps for JuliaBase is:

```
INSTALLED_APPS = (
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.admin",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "institute",
    "samples",
    "jb_common"
)
```

Of course, you must replace `j_institute` by your own institute's app. Furthermore, you may add as many apps as you like, as long as the inner order is preserved.

8.3.6 LOGIN_URL

The default URL configuration of JuliaBase puts the login view so that you should say:

```
LOGIN_URL = "/login"
```

8.3.7 LOGIN_REDIRECT_URL

JuliaBase assumes that this setting contains the home page of the database application. It is used in the default templates if you click on the "JuliaBase" name on the top. You may simply set it to `"/"`.

8.3.8 MIDDLEWARE_CLASSES

The following is a minimal set of middleware JuliaBase is working with:

```
MIDDLEWARE_CLASSES = (
    "django.middleware.common.CommonMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "jb_common.middleware.MessageMiddleware",
    "django.middleware.locale.LocaleMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "jb_common.middleware.LocaleMiddleware",
    "samples.middleware.juliabase.ExceptionsMiddleware",
)
```

Note that while you may add further middleware, you must not change the inner ordering of existing middleware.

8.3.9 SECRET_KEY

Note that in the `settings.py` file shipped with JuliaBase, the `SECRET_KEY` is read from the file `~/juliabase_secret_key`. If this file doesn't exist, it is generated. This is good for quickly getting things running and not insecure per se, but you should be aware of this. In particular, once generated, you must not lose this file in your production environment because it would result in data loss, and nobody else should have access to it.

Of course, alternatively, you may set the `SECRET_KEY` in a completely different way.

8.3.10 TEMPLATE_CONTEXT_PROCESSORS

Make sure that you add `"jb_common.context_processors.default"` to the list of context processors.

8.3.11 TEMPLATE_DIRS

So that you can override JuliaBase's templates with own templates, you should set:

```
TEMPLATE_DIRS = (os.path.abspath(os.path.join(os.path.dirname(__file__), "../juliabase")),)
```

You may add further paths, but if your project layout is structured according to *Organizing your source code*, this one must be present. Then, you can extend Juliabase templates by beginning your template with e.g.

```
{% extends "samples/templates/samples/list_claims.html" %}
```

8.3.12 TEMPLATE_LOADERS

In conjunction with `TEMPLATE_DIRS` you must make sure that Django will look for templates first in the app directories, and then in the filesystem. With activated template caching, this looks like:

```
TEMPLATE_LOADERS = (
    ("django.template.loaders.cached.Loader",
     ("django.template.loaders.app_directories.Loader",
      "django.template.loaders.filesystem.Loader")),)
```

and without caching, like:

```
TEMPLATE_LOADERS = ("django.template.loaders.app_directories.Loader",
                    "django.template.loaders.filesystem.Loader")
```

8.3.13 USE_TZ

You may set it to `True` (the default for newly created Django projects). This may or may not work for you since JuliaBase is not tested with timezone-awareness. In particular, the remote client will surely fail. Therefore, we recommend to leave it at the default value `False`.

SAMPLE NAMES

The naming scheme for samples is an important aspect of your samples database. In order to support you with that, JuliaBase allows the definition of so-called name formats. These are string patterns that describe all valid sample names. They are defined in `settings.py` like this:

```
SAMPLE_NAME_FORMATS = {
    "provisional": {"possible_renames": {"new"}},
    "old":         {"pattern": r"{short_year}[A-Z]-\d{{3,4}}([-A-Za-z_]/[-A-Za-z_0-9#()])?",
                    "possible_renames": {"new"}},
    "new":         {"pattern": r"({short_year}-{user_initials}|{external_contact_initials})"
                    r"-[-A-Za-z_0-9#()]+"}
}
```

In this example, three name formats are defined, namely “provisional”, “old”, and “new”. They are mapped to dictionaries which contain their properties.

Note: There is a length limit of 30 characters for sample names. This is hard-wired in JuliaBase.

9.1 Name format properties

9.1.1 "pattern"

This is the [regular expression pattern](#) for this name format. It should match sample names of this format *and only these*. In other words, every sample name should be identified unambiguously with a name format. JuliaBase appends an implicit `\Z` so that the whole sample name must match.

You can use some placeholder in the pattern that are interpreted by JuliaBase to enforce additional constraints to new sample names. These are:

{year} the current year as a four-digit number

{short_year} the current year as a two-digit number

{user_initials} the initials of the currently logged-in user

{external_contact_initials} the initials of an external contact (a.k.a. external operator) of the currently logged-in user

{combined_initials} any of the initials above

Because placeholders are embraced with `{...}`, you have to double any curly braces used in the pattern itself, as can be seen at the `{{3,4}}` in the example above.

9.1.2 "possible_renames"

Default: `set()` (empty set)

This is a set containing all name formats into which this name format can be renamed by the user. Of course, Python code can rename a sample to anything – although it will cause trouble if the new name is not matched by any name format. However, this property sets limits to what the *user* can *explicitly* do. In the JuliaBase code, it affects the bulk-rename view as well as the split-and-rename view. And you may enforce it in your own code.

9.1.3 "verbose_name"

This property contains a human-friendly name for the format name. You should enclose it with `_("...")` to make it translatable.

9.2 Provisional sample names

There is a special name format in JuliaBase called “provisional”. It has a fixed pattern `r"*\d{5}$"` and a default verbose name. It is used for newly created samples and usually immediately replaced by something real in the “bulk rename” view. You should never use a provisional name outside JuliaBase, e.g. on sample boxes, in lab notebooks, or emails! And, you should never allow renaming of any format into provisional names. In other words, one should get rid of a provisional name as quickly as possible.

9.3 Initials

Initials are a way to generate namespaces for samples. This way, name collisions may be prevented. For example, the probability that the researcher John Doe and the researcher Paula Poe both call their sample “sample-1” is pretty high. However, if they put their initials somewhere in the sample name, the samples may be called “JD-sample-1” and “PP-sample-1”; problem solved.

When it comes to choosing initials, JuliaBase follows the first come first served principle. If John Doe chooses initials after Jane Doe has picked “JD”, he must use “JD1” or “JDOE” or whatever. They cannot be changed anymore by the user.

Initials are at most 4 characters long. Apart from that, JuliaBase is configurable with the setting `INITIALS_FORMATS`. It is a dictionary mapping “user” and “external_contact” to a properties dictionary. The allowed properties are the following:

9.3.1 "pattern"

This is a [regular expression pattern](#). The whole initials must match it. In contrast to `SAMPLE_NAME_FORMATS`, this pattern cannot contain placeholders, and therefore, you must not double any curly brackets.

For example, if you want to restrict user initials to two or three uppercase letters, you simply add to `settings.py`:


```
INITIALS_FORMATS["user"] = {  
    "pattern": "[A-Z]{2,3}",  
    "description": _("The initials consist of two or three uppercase letters.")}
```

9.3.2 "description"

This is translatable string describing the pattern in a Human-friendly way. It is used for error messages.

9.4 Name prefix templates

In the bulk rename sample view (the view to which the user is redirected after having created new samples, in order to give them a name), the user can select a name prefix, which is then prepended to every new name. This is partly for convenience and partly for name policy enforcement. With the setting `NAME_PREFIX_TEMPLATES`, you can configure this behaviour:

```
NAME_PREFIX_TEMPLATES = ("{short_year}-{user_initials}-", "{external_contact_initials}-")
```

This example defines two prefixes. You may use the following placeholders in the templates:

{year} the current four-digit year

{short_year} the current two-digit year

{user_initials} the initials of the currently logged-in user

{external_contact_initials} the initials of any external contact of the currently logged-in user

You may also add the empty string "" as a template. Then, the user may also choose to not use any prefix, and the new names are taken as is.

HACKING ON JULIABASE

This *unfinished* chapter explains how to contribute to the JuliaBase project itself instead of adapting it to your institution.

10.1 Architecture

Since JuliaBase is based on the Django Web framework, it consists of several Django apps. The core app is called “jb_common”. It provides functionality which is essential for all JuliaBase components. On top of that, the app “samples” contains all features of a samples database. However, it does not contain institute-specific code in order to remain generic and flexible. This institute-specific code resides in an app of its own and must be created by a skilled programmer.

JuliaBase is shipped together with an example institute app called “inm”. It provides not only a demo for JuliaBase, it also is a good starting point for your own app. Besides, essential testing of JuliaBase can only be done on top of inm.

10.2 Coding guidelines

JuliaBase source code modules should not exceed 1000 lines of code. You should stick to [PEP 8](#) and the [Django coding guidelines](#). String literals are double-quoted unless double quotes appear in them:

```
"This is a string literal"  
'This is "another" string literal'
```

Never use `u` in front of a string literal. Instead, JuliaBase code uses the `unicode_literals` future import.

JuliaBase makes one exception from PEP 8: It allows lines with 125 columns instead of only 80.

All variables and source code comments should be in English.

Note: I skip all docstrings in the code examples in this document because otherwise, the examples would be too bloated. However, write rich Docstrings for all non-trivial functions and methods. Write them in [ReST format](#).

Internationalization is a very important point in JuliaBase. All strings exposed to the user should be marked as translatable by putting them in `_("...")` unless you have very good reason not to do so (e.g. for some proper names). Note that in code which is executed at

module load time (e.g. model and form fields), `_` should stand for `ugettext_lazy`, whereas within functions and methods which are executed on each request, it should be `ugettext`. You may achieve this by setting `_` to `ugettext_lazy` at the beginning of the module, and to `ugettext` at the end.

10.2.1 Python 3

JuliaBase runs on both Python 2.7 and Python 3.2+. Take care that your contributions do the same. We found the [Django documentation on this topic](#) extremely useful. The remote client comes with its own copy of the `six` module in order to keep external dependencies at a minimum.

10.2.2 Boilerplate code

Start every file with:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
# {Licence}

"""{Module docstring}
"""

from __future__ import absolute_import, unicode_literals, division
{Python 3 compatibility statements}

{Python standard library imports}
{Non-standard imports (Numpy, Scipy, matplotlib etc)}
{Django imports}
{JuliaBase imports}
```

The “Python 3 compatibility statements” may be lines like:

```
import django.utils.six as six
from django.utils.six.moves import cStringIO as StringIO
from django.utils.encoding import python_2_unicode_compatible
```

UTILITIES

In the following sections, the most important functions, classes, and exception classes are mentioned and explained. This covers everything that is used in the “institute” app. It should give you a solid starting point for creating an own institute app. Nevertheless, the utility modules contain even more functions and classes.

11.1 Common helpers

11.1.1 String-related

The following names are found in the module `jb_common.utils.base`.

`format_lazy(*args, **kwargs)`

Implements a lazy variant of the `format` string method. For example, you might say:

```
verbose_name = format_lazy(_(u"Raman {0} measurement"), 1)
```

Here, “`_`” is `gettext_lazy`.

`format_enumeration(items)`

Generates a pretty-printed enumeration of all given names. For example, if the list contains `["a", "b", "c"]`, it yields “a, b, and c”.

Parameters `items` (*iterable of unicode*) – iterable of names to be put into the enumeration

Returns human-friendly enumeration of all names

Return type unicode

`camel_case_to_underscores(name)`

Converts a CamelCase identifier to one using underscores. For example, “MySamples” is converted to “my_samples”, and “PDSMeasurement” to “pds_measurement”.

Parameters `name` (*str*) – the camel-cased identifier

Returns the identifier in underscore notation

Return type str

`camel_case_to_human_text(name)`

Converts a CamelCase identifier to one intended to be read by humans. For example, “MySamples” is converted to “my samples”, and “PDSMeasurement” to “PDS measurement”.

Parameters `name` (*str*) – the camel-cased identifier

Returns the pretty-printed identifier

Return type `str`

11.1.1.2 File-related

The following names are found in the module `jb_common.utils.base`.

`find_file_in_directory` (*filename, path, max_depth=None*)

Searches for a file in a directory recursively to a given depth.

Parameters

- **`filename`** (*str*) – The file to be searched for. Only the basename is required.
- **`path`** (*str*) – The path from the top level directory where the searching starts.
- **`max_depth`** (*int*) – The maximum recursion depth; if `None`, there is no limit.

Returns The relative path to the searched file, or `None` if not found.

Return type `str`

`check_filepath` (*filepath, default_root, allowed_roots=frozenset([]), may_be_directory=False*)

Test whether a certain file is openable by JuliaBase.

Parameters

- **`filepath`** (*str*) – Path to the file to be tested. This may be absolute or relative. If empty, this function does nothing.
- **`default_root`** (*str*) – If `filepath` is relative, this path is prepended to it.
- **`allowed_roots`** (*iterable of str*) – All absolute root paths where `filepath` is allowed. `default_root` is implicitly added to it.
- **`may_be_directory`** (*bool*) – if `True`, `filepath` may be a readable directory

Returns the normalised `filepath`

Return type `str`

Raises `ValidationError` if the `text` contained forbidden syntax elements.

`is_update_necessary` (*destination, source_files=[], timestamps=[], additional_inaccuracy=0*)

Returns whether the destination file needs to be re-created from the sources. It bases of the timestamps of last file modification. If the union of `source_files` and `timestamps` is empty, the function returns `False`.

Parameters

- **`destination`** (*unicode*) – the path to the destination file

- **source_files** (*list of unicode*) – the paths of the source files
- **timestamps** (*list of datetime.datetime*) – timestamps of non-file source objects
- **additional_inaccuracy** (*int or float*) – When comparing file timestamps across computers, there may be trouble due to inaccurate clocks or filesystems where the modification timestamps have an accuracy of only 2 seconds (some Windows FS'es). Set this parameter to a positive number to avoid this. Note that usually, JuliaBase *copies existing* timestamps, so inaccurate clocks should not be a problem.

Returns whether the destination file needs to be updated

Return type bool

Raises **OSError** if one of the source paths is not found

remove_file (*path*)

Removes the file. If the file didn't exist, this is a no-op.

Parameters **path** (*str*) – absolute path to the file to be removed

Returns whether the file was removed; if `False`, it hadn't existed

Return type bool

makedirs (*path*)

Creates a directory and all of its parents if necessary. If the given path doesn't end with a slash, it's interpreted as a filename and removed. If the directory already exists, nothing is done. (In particular, no exception is raised.)

Parameters **path** (*str*) – absolute path which should be created

11.1.3 Generating responses

The following names are found in the module `jb_common.utils.base`.

exception JSONRequestException (*error_number, error_message*)

Exception which is raised if a JSON response was requested and an error in the submitted data occurred. This will result in an HTTP 422 response with a JSON-encoded (*error code, error message*) body. For example, in a JSON-only view function, you might say:

```
if not request.user.is_staff:
    raise JSONRequestException(6, "Only admins can access this resource.")
```

The ranges for the error codes are:

0–999 special codes, codes common to all applications, and JuliaBase-common

1000–1999 JuliaBase-samples

2000–2999 institute-specific extensions to JuliaBase-samples

3000–3999 JuliaBase-kicker

The complete table with the error codes is in the main `__init__.py` of the respective app.

is_json_requested(*request*)

Tests whether the current request should be answered in JSON format instead of HTML. Typically this means that the request was made by the JuliaBase Remote Client or by JavaScript code.

Parameters *request* (*HttpRequest*) – the current HTTP Request object

Returns whether the request should be answered in JSON

Return type bool

respond_in_json(*value*)

The communication with the JuliaBase Remote Client or to AJAX clients should be done without generating HTML pages in order to have better performance. Thus, all responses are Python objects, serialised in JSON notation.

The views that can be accessed by the Remote Client/AJAX as well as normal browsers should distinguish between both by using `is_json_requested`.

Parameters *value* (object (an arbitrary Python object)) – the data to be sent back to the client that requested JSON.

Returns the HTTP response object

Return type *HttpResponse*

static_file_response(*filepath*, *served_filename*=None)

Serves a file of the local file system.

Parameters

- **filepath** – the absolute path to the file to be served
- **served_filename** – the filename the should be transmitted; if given, the response will be an “attachment”

Returns the HTTP response with the static file

Type `django.http.HttpResponse`

The following name is found in the module `samples.utils.views`.

successful_response(*request*, *success_report*=None, *view*=None, *kwargs*={},
query_string=u'', *forced*=False, *json_response*=True)

After a POST request was successfully processed, there is typically a redirect to another page – maybe the main menu, or the page from where the add/edit request was started.

The latter is appended to the URL as a query string with the `next` key, e.g.:

`/juliabase/6-chamber_deposition/08B410/edit/?next=/juliabase/samples/08B410a`

This routine generated the proper *HttpResponse* object that contains the redirection. It always has HTTP status code 303 (“see other”).

If the request came from the JuliaBase Remote Client, the response is a pickled `json_response`. (Normally, a simple `True`.)

Parameters

- **request** (*HttpRequest*) – the current HTTP request
- **success_report** (*unicode*) – an optional short success message reported to the user on the next view

- **view** (*str or function*) – the view name/function to redirect to; defaults to the main menu page (same when `None` is given)
- **kwargs** (*dict*) – group parameters in the URL pattern that have to be filled
- **query_string** (*unicode*) – the *quoted* query string to be appended, without the leading `"?"`
- **forced** (*bool*) – If `True`, go to view even if a “next” URL is available. Defaults to `False`. See `bulk_rename.bulk_rename` for using this option to generate some sort of nested forwarding.
- **json_response** (*object*) – object which is to be sent as a pickled response to the remote client; defaults to `True`.

Returns the HTTP response object to be returned to the view’s caller

Return type `HttpResponse`

11.1.1.4 Sample-related

The following names are found in the module `samples.utils.views`.

dead_samples (*samples, timestamp*)

Determine all samples from `samples` which are already dead at the given `timestamp`.

Parameters

- **samples** (list of `samples.models.Sample`) – the samples to be tested
- **timestamp** (*datetime.datetime*) – the timestamp for which the dead samples should be found

Returns set of all samples which are dead at `timestamp`

Return type set of `samples.models.Sample`

sample_name_format (*name, with_match_object=False*)

Determines which sample name format the given name has. It doesn’t test whether the sample name is existing, nor if the initials are valid.

Parameters **name** (*unicode*) – the sample name

Returns The name of the sample name format and the respective match object. The latter can be used to extract groups, for example. `None` if the name had no valid format.

Return type (`unicode`, `re.MatchObject`) or `NoneType`.

get_sample (*sample_name*)

Lookup a sample by name. You may also give an alias. If more than one sample is found (can only happen via aliases), it returns a list. Matching is exact.

Parameters **sample_name** (*unicode*) – the name or alias of the sample

Returns the found sample. If more than one sample was found, a list of them. If none was found, `None`.

Return type `samples.models.Sample`, list of `samples.models.Sample`, or `NoneType`

does_sample_exist (*sample_name*)

Returns `True` if the sample name exists in the database.

Parameters **sample_name** (*unicode*) – the name or alias of the sample

Returns whether a sample with this name exists

Return type `bool`

normalize_sample_name (*sample_name*)

Returns the current name of the sample.

Parameters **sample_name** (*unicode*) – the name or alias of the sample

Returns The current name of the sample. This is only different from the input if you gave an alias.

Return type `unicode`

lookup_sample (*sample_name, user, with_clearance=False*)

Looks up the `sample_name` in the database (also among the aliases), and returns that sample if it was found *and* the current user is allowed to view it. Shortened provisional names like “*2” are also found. If nothing is found or the permissions are not sufficient, it raises an exception.

Parameters

- **sample_name** (*unicode*) – name of the sample
- **user** (*django.contrib.auth.models.User*) – the currently logged-in user
- **with_clearance** (*bool*) – whether also clearances should be searched for and returned

Returns the single found sample; or the sample and the clearance instance if this is necessary to view the sample and `with_clearance=True`

Return type `samples.models.Sample` or `samples.models.Sample, samples.models.Clearance`

Raises

- **Http404** – if the sample name could not be found
- **AmbiguityException** – if more than one matching alias was found
- **samples.permissions.PermissionError** – if the user is not allowed to view the sample

remove_samples_from_my_samples (*samples, user*)

Remove the given samples from the user’s MySamples list

Parameters

- **samples** (list of `samples.models.Sample`) – the samples to be removed.
- **user** (*django.contrib.auth.models.User*) – the user whose MySamples list is affected

extract_preset_sample (*request*)

Extract a sample from a query string. All physical processes as well as result processes may have an optional parameter in the query string, namely the sample to which they should be applied (results even a sample series, too). If such a parameter is present, the given sample – if existing – must be added to the list of selectable samples, and it must be the initially marked sample.

This routine is used in all views for creating physical processes. It is not used for result processes because they need a given sample *series*, too, and this would have been over-generalisation.

This routine extracts the sample name from the query string and returns the sample. If nothing was given or the sample non-existing, it returns `None`.

Parameters *request* (*HttpRequest*) – the current HTTP Request object

Returns the sample given in the query string, if any

Return type `samples.models.Sample` or `NoneType`

restricted_samples_query (*user*)

Returns a `QuerySet` which is restricted to samples the names of which the given user is allowed to see. Note that this doesn't mean that the user is allowed to see all of the samples themselves necessarily. It is only about the names. See the `samples.views.sample.search()` view for further information.

Parameters *user* (*django.contrib.auth.models.User*) – the user for which the allowed samples should be retrieved

Returns a queryset with all samples the names of which the user is allowed to know

Return type `QuerySet`

11.1.5 Miscellaneous

The following names are found in the module `samples.utils.views`.

convert_id_to_int (*process_id*)

If the user gives a process ID via the browser, it must be converted to an integer because this is what's stored in the database. (Well, actually SQL gives a string, too, but that's beside the point.) This routine converts it to a real integer and tests also for validity (not for availability in the database).

Parameters *process_id* (*str*) – the pristine process ID as given via the URL by the user

Returns the process ID as an integer number

Return type `int`

Raises `Http404` if the *process_id* didn't represent an integer number.

table_export (*request*, *data*, *label_column_heading*)

Helper function which does almost all work needed for a CSV table export view. This is not a view per se, however, it is called by views, which have to do almost nothing anymore by themselves. See for example `sample.export`.

This function return the data in JSON format if this is requested by the `Accept` header field in the HTTP request.

Parameters

- **request** (*HttpRequest*) – the current HTTP Request object
- **data** (`samples.data_tree.DataNode`) – the root node of the data tree
- **label_column_heading** (*unicode*) – Description of the very first column with the table row headings, see `generate_table_rows`.

Returns the HTTP response object or a tuple with all needed forms to create the export view

Return type `HttpResponse` or tuple of `django.forms.Form`

The following names are found in the module `jb_common.utils.base`.

get_really_full_name (*user*)

Unfortunately, Django's `get_full_name` method for users returns the empty string if the user has no first and surname set. However, it'd be sensible to use the login name as a fallback then. This is realised here.

Parameters **user** (*django.contrib.auth.models.User*) – the user instance

Returns The full, human-friendly name of the user

Return type `unicode`

check_markdown (*text*)

Checks whether the Markdown input by the user contains only permitted syntax elements. I forbid images and headings so far.

Parameters **text** – the Markdown input to be checked

Raises **ValidationError** if the `text` contained forbidden syntax elements.

help_link (*link*)

Function decorator for views functions to set a help link for the view. The help link is embedded into the top line in the layout, see the template `base.html`. The default template `jb_base.html` prepends `"http://www.juliabase.org/"`. But you may change that by overriding the `help_link` block in your own `jb_base.html`.

Parameters **link** (*str*) – the relative URL to the help page.

send_email (*subject, content, recipients, format_dict=None*)

Sends one email to a user. Both subject and content are translated to the recipient's language. To make this work, you must tag the original text with a dummy `_` function in the calling content, e.g.:

```
_ = lambda x: x
send_mail(_("Error notification"), _("An error has occurred."), user)
_ = gettext
```

If you need to use string formatting à la

```
"Hello {name}".format(name=user.name)
```

you must pass a dictionary like `{"name": user.name}` to this function. Otherwise, translating wouldn't work.

Parameters

- **subject** (*unicode*) – the subject of the email
- **content** (*unicode*) – the content of the email; it may contain substitution tags
- **recipients** – the recipients of the email
- **format_dict** (*dict mapping unicode to unicode*) – the substitutions used for the `format` string method for both the subject and the content

round (*value, digits*)

Method for rounding a numeric value to a fixed number of significant digits.

Parameters

- **value** (*float*) – the numeric value
- **digit** – number of significant digits

Returns rounded value**Return type** str**generate_permissions** (*permissions, class_name*)

Auto-generates model permissions. It may be used in physical process classes – but not only there – like this:

```
class Meta(samples.models.PhysicalProcess.Meta):
    permissions = generate_permissions(
        {"add", "change", "view_every", "edit_permissions"}, "ModelClassName")
```

Parameters

- **permissions** (*set of unicode*) – The permissions to generate. Possible values are "add", "change", "view_every", and "edit_permissions".
- **class_name** (*str*) – python class name of the model class, e.g. "LayerThicknessMeasurement".

Returns the permissions tuple**Return type** tuple of (unicode, unicode)

11.2 Feed reporting

The following name is found in the module `samples.utils.views`.**class Reporter** (*originator*)

This class contains all feed-generating routines as methods. Their names start with `report_...`. The main reason for putting them into a class is that this class assures that no user gets two feed entries. Therefore, if you want to report a certain database change to the users, create an instance of `Reporter` and call all methods that are related to the database change. Call the most meaningful method first, and the most general last. For example, when changing the data of a sample, the respective view calls the following methods in this order:

```
report_new_responsible_person_samples
report_changed_sample_topic
report_edited_samples
```

Of course, the first two are only called if the respective data change really happend. Thus, the new responsible person is signalled first, then all people about a possible topic change, and if this didn't happen, all so-far un-signalled users get a general message about changed sample data.

If you want to signal something to all possibly interested users, no matter on which feed entries they already have received, just create a new instance of `Reporter`.

Mostly, you can call the method directly without binding the instance of `Reporter` to a name, as in:

```
feed_utils.Reporter(request.user).report_result_process(
    result, edit_description=None)
```

Variables

- **interested_users** – all users that get informed with the next generated feed entry by a call to `__connect_with_users`
- **already_informed_users** – All users who have already received a feed entry from this instance of `Reporter`. They won't get a second entry.
- **originator** – the user responsible for the database change reported by the feed entry of this instance of `Reporter`.

Class constructor.

Parameters **originator** (*django.contrib.auth.models.User*) – the user who did the database change to be reported; almost always, this is the currently logged-in user

report_changed_sample_series_topic (*sample_series*, *old_topic*, *edit_description*)

Generate a feed entry about a topic change for a sample series. All members of the former topic and the new topic are informed. Note that it is possible that further things were changed in the sample series at the same time (reponsible person, samples ...). They should be mentioned in the description by the one who changed it.

Parameters

- **sample_series** (list of *samples.models.SampleSeries*) – the sample series that went into a new topic
- **old_topic** (*jb_common.models.Topic*) – the old topic of the samples; may be `None` if they weren't in any topic before
- **edit_description** (dict mapping str to object) – The dictionary containing data about what was edited in the sample series (besides the change of the topic). Its keys correspond to the fields of `EditDescriptionForm`.

report_changed_sample_topic (*samples*, *old_topic*, *edit_description*)

Generate a feed entry about a topic change for sample(s). All members of the former topic (if any) and the new topic are informed. Note that it is possible that

further things were changed in the sample(s) at the same time (reponsible person, purpose...). They should be mentioned in the description by the one who changed it.

Parameters

- **samples** (list of `samples.models.Sample`) – the samples that went into a new topic
- **old_topic** (`jb_common.models.Topic`) – the old topic of the samples; may be `None` if they weren't in any topic before
- **edit_description** (dict mapping str to object) – The dictionary containing data about what was edited in the samples (besides the change of the topic). Its keys correspond to the fields of `EditDescriptionForm`.

report_changed_topic_membership (*users, topic, action*)

Generate one feed entry for changed topic memberships, i.e. added or removed users in a topic.

Parameters

- **users** (iterable of `django.contrib.auth.models.User`) – the affected users
- **topic** (`jb_common.models.Topic`) – the topic whose memberships have changed
- **action** (str) – what was done; "added" for added users, "removed" for removed users

report_copied_my_samples (*samples, recipient, comments*)

Generate a feed entry for sample that one user has copied to another user's "My Samples" list.

Parameters

- **samples** (list of `samples.models.Sample`) – the samples that were copied to another user
- **recipient** (`django.contrib.auth.models.User`) – the other user who got the samples
- **comments** (unicode) – a message from the sender to the recipient

report_edited_sample_series (*sample_series, edit_description*)

Generate a feed entry about an edited of sample series. All users who have watches samples in this series are informed, including the currently responsible person (in case that it is not the originator).

Parameters

- **sample_series** (list of `samples.models.SampleSeries`) – the sample series that was edited
- **edit_description** (dict mapping str to object) – The dictionary containing data about what was edited in the sample series. Its keys correspond to the fields of `EditDescriptionForm`.

report_edited_samples (*samples, edit_description*)

Generate a feed entry about a general edit of sample(s). All users who are allowed to see the sample and who have the sample on their “My Samples” list are informed.

Parameters

- **samples** (list of `samples.models.Sample`) – the samples that was edited
- **edit_description** (dict mapping str to object) – The dictionary containing data about what was edited in the samples. Its keys correspond to the fields of `EditDescriptionForm`.

report_new_responsible_person_sample_series (*sample_series, edit_description*)

Generate a feed entry for a sample series that changed their currently responsible person. This feed entry is only sent to that new responsible person. Note that it is possible that further things were changed in the sample series at the same time (topic, samples ...). They should be mentioned in the description by the formerly responsible person.

Parameters

- **sample_series** (list of `samples.models.SampleSeries`) – the sample series that got a new responsible person
- **edit_description** (dict mapping str to object) – Dictionary containing data about what was edited in the sample series (besides the change of the responsible person). Its keys correspond to the fields of `EditDescriptionForm`.

report_new_responsible_person_samples (*samples, edit_description*)

Generate a feed entry for samples that changed their currently responsible person. This feed entry is only sent to that new responsible person. Note that it is possible that further things were changed in the sample(s) at the same time (topic, purpose ...). They should be mentioned in the description by the formerly responsible person.

Parameters

- **samples** (list of `samples.models.Sample`) – the samples that got a new responsible person
- **edit_description** (dict mapping str to object) – Dictionary containing data about what was edited in the samples (besides the change of the responsible person). Its keys correspond to the fields of `EditDescriptionForm`.

report_new_sample_series (*sample_series*)

Generate one feed entry for a new sample series.

Parameters **sample_series** (`samples.models.SampleSeries`) – the sample series that was added

report_new_samples (*samples*)

Generate one feed entry for new samples. If more than one sample is in the given list, they are assumed to have been generated at the same time, so they should share the same topic and purpose.

If the sample or samples are not in a topic, no feed entry is generated (because nobody listens).

Parameters **samples** (list of `samples.models.Sample`) – the samples that were added

report_physical_process (*process*, *edit_description=None*)

Generate a feed entry for a physical process (deposition, measurement, etching etc) which was recently edited or created. If the process is still unfinished, nothing is done.

Parameters

- **process** (`samples.models.Process`) – the process which was added/edited recently
- **edit_description** (dict mapping str to object) – The dictionary containing data about what was edited in the process. Its keys correspond to the fields of `EditDescriptionForm`. None if the process was newly created.

report_removed_task (*task*)

Generate one feed for a removed task. It is called immediately before the task is actually deleted.

Parameters **task** (`models.Task`) – the to-be-deleted task

report_result_process (*result*, *edit_description=None*)

Generate a feed entry for a result process which was recently edited or created.

Parameters

- **result** (`samples.models.Result`) – the result process which was added/edited recently
- **edit_description** (dict mapping str to object) – The dictionary containing data about what was edited in the result. Its keys correspond to the fields of `EditDescriptionForm`. None if the process was newly created.

report_sample_split (*sample_split*, *sample_completely_split*)

Generate a feed entry for a sample split.

Parameters

- **sample_split** (`samples.models.SampleSplit`) – sample split that is to be reported
- **sample_completely_split** (*bool*) – whether the sample was completely split, i.e. no piece of the parent sample is left

report_status_message (*process_class*, *status_message*)

Generate one feed entry for new status messages for physical processes.

Parameters

- **process_class** (`django.contrib.contenttypes.models.ContentType`) – the content type of the physical process whose status has changed
- **status_message** (`samples.models.StatusMessage`) – the status message for the physical process

report_task (*task*, *edit_description=None*)

Generate one feed entry for a new task or an edited task.

Parameters

- **task** (`models.Task`) – the task that was created or edited
- **edit_description** (dict mapping str to object or None) – The dictionary containing data about what was edited in the task. Its keys correspond to the fields of `EditDescriptionForm`. None if the task was newly created.

report_withdrawn_status_message (*process_class*, *status_message*)

Generate one feed entry for a withdrawn status message for physical processes.

Parameters

- **process_class** (`django.contrib.contenttypes.models.ContentType`) – the content type of the physical process one of whose statuses was withdrawn
- **status_message** (`samples.models.StatusMessage`) – the status message for the physical process

11.3 Form field classes

The following names are found in the module `jb_common.utils.views`.

class UserField (*choices=()*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=u''*, **args*, ***kwargs*)

Form field class for the selection of a single user. This can be the new currently responsible person for a sample, or the person you wish to send “My Samples” to.

set_users (*user*, *additional_user=None*)

Set the user list shown in the widget. You *must* call this method (or `set_users_without()`) in the constructor of the form in which you use this field, otherwise the selection box will remain empty. The selection list will consist of all currently active users, plus the given additional user if any.

Parameters

- **user** (`django.contrib.auth.models.User`) – The user who wants to see the user list
- **additional_user** (`django.contrib.auth.models.User`) – Optional additional user to be included into the list. Typically, it is the current user for the process to be edited.

set_users_without (*user*, *excluded_user*)

Set the user list shown in the widget. You *must* call this method (or `set_users()`) in the constructor of the form in which you use this field, otherwise the selection box will remain empty. The selection list will consist of all currently active users, minus the given user.

Parameters

- **user** – The user who wants to see the user list

- **excluded_user** (*django.contrib.auth.models.User*) – User to be excluded from the list. Typically, it is the currently logged-in user.

class MultipleUsersField (*args, **kwargs)

Form field class for the selection of zero or more users. This can be the set of members for a particular topic.

set_users (user, additional_users=[])

Set the user list shown in the widget. You *must* call this method in the constructor of the form in which you use this field, otherwise the selection box will remain empty. The selection list will consist of all currently active users, plus the given additional users if any.

Parameters

- **user** – The user who wants to see the user list
- **additional_users** (iterable of *django.contrib.auth.models.User*) – Optional additional users to be included into the list. Typically, it is the current users for the topic whose memberships are to be changed.

class TopicField (choices=(), required=True, widget=None, label=None, initial=None, help_text=u'', *args, **kwargs)

Form field class for the selection of a single topic. This can be the topic for a sample or a sample series, for example.

set_topics (user, additional_topic=None)

Set the topic list shown in the widget. You *must* call this method in the constructor of the form in which you use this field, otherwise the selection box will remain empty. The selection list will consist of all currently active topics, plus the given additional topic if any. The “currently active topics” are all topics with at least one active user amongst its members.

Parameters

- **user** (*django.contrib.auth.models.User*) – the currently logged-in user
- **additional_topic** (*jb_common.models.Topic*) – Optional additional topic to be included into the list. Typically, it is the current topic of the sample, for example.

11.4 Form classes

The following names are found in the module `samples.utils.views`.

class ProcessForm (user, *args, **kwargs)

Abstract model form class for processes. It ensures that timestamps are not in the future, and that comments contain only allowed Markdown syntax.

Moreover, it defines a field “combined_operator” of the type `OperatorField`. In the HTML template, you should offer this field to non-staff, and the usual operator/external operator to staff.

Parameters **user** (*django.contrib.auth.models.User*) – the currently logged-in user

is_referentially_valid(*sample_form*)

Test whether the forms are consistent with each other and with the database. In its current form, it only checks whether the sample is still “alive” at the time of the measurement.

Parameters *sample_form* (SampleForm) – a bound sample selection form

Returns whether the forms are consistent with each other and the database

Return type bool

class DepositionForm(*user, data=None, **kwargs*)

Model form for depositions (not their layers).

is_referentially_valid(*sample_form*)

Test whether the forms are consistent with each other and with the database. In its current form, it only checks whether the sample is still “alive” at the time of the measurement.

Parameters *sample_form* (SampleForm) – a bound sample selection form

Returns whether the forms are consistent with each other and the database

Return type bool

class SampleSelectForm(*user, process_instance, preset_sample, *args, **kwargs*)

Form for the sample selection field. You can only select *one* sample per process (in contrast to depositions).

Parameters

- **user** (*django.contrib.auth.models.User*) – the current user
- **process_instance** (*samples.models.Process*) – the process instance to be edited, or None if a new is about to be created
- **preset_sample** (*samples.models.Sample*) – the sample to which the process should be appended when creating a new process; see `utils.extract_preset_sample`

class DepositionSamplesForm(*user, deposition, preset_sample, data=None, **kwargs*)

Form for the list selection of samples that took part in the deposition. This form has the special behaviour that it prevents changing the samples when editing an *existing* process.

class EditDescriptionForm(**args, **kwargs*)

Form for letting the user enter a short description of the changes they made.

class RemoveFromMySamplesForm(*data=None, files=None, auto_id=u'id_%s', prefix=None, initial=None, error_class=<class 'django.forms.utils.ErrorList'>, label_suffix=None, empty_permitted=False*)

Form for the question whether the user wants to remove the samples from the “My Samples” list after the process.

11.5 Plots

The following names are found in the module `samples.utils.plots`.

exception `PlotError`

Raised if an error occurs while generating a plot. Usually, it is raised in `samples.models.Process.draw_plot()` and caught in `samples.views.plots.show_plot()`.

`read_plot_file_beginning_at_line_number` (*filename*, *columns*, *start_line_number*, *end_line_number=None*, *separator=None*)

Read a datafile and returns the content of selected columns beginning at *start_line_number*. You shouldn't use this function directly. Use the specific functions instead.

Parameters

- **`filename`** (*str*) – full path to the data file
- **`columns`** (*list of int*) – the columns that should be read.
- **`start_line_number`** (*int*) – the line number where the data starts
- **`end_line_number`** (*int or None*) – the line number where the record should end. The default is `None`, means till end of file.
- **`separator`** (*str or None*) – the separator which separates the values from each other. Default is `None`

Returns List of all columns. Every column is represented as a list of floating point values.

Return type list of list of float

Raises `PlotError` if something goes wrong with interpreting the file (I/O, un-parseable data)

`read_plot_file_beginning_after_start_value` (*filename*, *columns*, *start_value*, *end_value=u''*, *separator=None*)

Read a datafile and return the content of selected columns after the *start_value* was detected. You shouldn't use this function directly. Use the specific functions instead.

Parameters

- **`filename`** (*str*) – full path to the data file
- **`columns`** (*list of int*) – the columns that should be read.
- **`start_value`** (*str*) – the *start_value* indicates the line after the data should be read
- **`end_value`** (*str*) – the *end_value* marks the line where the record should end. The default is the empty string
- **`separator`** (*str or None*) – the separator which separates the values from each other. Default is `None`

Returns List of all columns. Every column is represented as a list of floating point values.

Return type list of list of float

Raises `PlotError` if something goes wrong with interpreting the file (I/O, un-parseable data)

11.6 URLs

The following name is found in the module `samples.utils.urls`.

class `PatternGenerator` (*url_patterns*, *views_prefix*)

This class helps to build URL pattern lists for physical processes. You instantiate it once in your URLconf file. Then, you add URLs by calling `physical_process` for every physical process:

```
pattern_generator = PatternGenerator(urlpatterns, "institute.views.samples")
pattern_generator.deposition("ClusterToolDeposition", views={"add", "edit"})
pattern_generator.deposition("FiveChamberDeposition", "5-chamber_depositions")
pattern_generator.physical_process("PDSMeasurement", "number")
pattern_generator.physical_process("Substrate", views={"edit"})
```

Parameters

- **url_patterns** (list of `url()` instances) – The URL patterns to populate in situ.
- **views_prefix** (*unicode*) – the prefix for the view functions as a Python path, e.g. `"my_app.views.samples"`

deposition (*class_name*, *url_name*=None, *views*=set([`u'edit'`, `u'add'`, `u'lab_notebook'`]))

Add URLs for the views of the deposition process `class_name`. This is a shorthand for `physical_process` with defaults optimized for depositions: `identifying_field` is `"number"`, and the views include a lab notebook.

Parameters

- **class_name** (*unicode*) – Name of the deposition class, e.g. `"FiveChamberDeposition"`.
- **url_name** (*unicode*) – The URL path component to be used for this deposition. By default, this is the class name converted to underscores notation, with an `"s"` appended, e.g. `"thickness_measurements"`.
- **views** (*set of unicode*) – The view functions for which URLs should be generated. You may choose from `"add"`, `"edit"`, `"custom_show"`, and `"lab_notebook"`.

physical_process (*class_name*, *identifying_field*=None, *url_name*=None, *views*=set([`u'edit'`, `u'add'`]))

Add URLs for the views of the physical process `class_name`. For the `"add"` and the `"edit"` view, an `edit(request, process_class_name_id)` function must exist. In case of `"add"`, `None` is passed as the second parameter. For the `"custom show"` view, a `show(request, process_class_name_id)` function must exist. If there is an `identifying_field`, this is used for the second parameter name instead. If no URL for a custom show view is requested, a default one is generated using a generic view function (which is mostly sufficient).

Parameters

- **class_name** (*unicode*) – Name of the physical process class, e.g. "ThicknessMeasurement".
- **identifying_field** (*unicode*) – If applicable, name of the model field which serves as “poor man’s” primary key. If not given, the `id` is used.
- **url_name** (*unicode*) – The URL path component to be used for this process. By default, this is the class name converted to underscores notation, with an “s” appended, e.g. "thickness_measurements". It may contain slashes.
- **views** (*set of unicode*) – The view functions for which URLs should be generated. You may choose from "add", "edit", "custom_show", and "lab_notebook".

TEMPLATE TAGS AND FILTERS

12.1 JuliaBase core

You use these tags and filter with:

```
{% load juliabase %}
```

12.1.1 Tags

markdown_hint ()

Tag for inserting a short remark that Markdown syntax must be used here, with a link to further information.

input_field (*field*)

Tag for inserting a field value into an HTML table as an editable field. It consists of two `<td>` elements, one for the label and one for the value, so it spans two columns. This tag is primarily used in templates of edit views. Example:

```
{% input_field deposition.number %}
```

error_list (*form, form_error_title, outest_tag=u'<table>', colspan=1*)

Includes a comprehensive error list for one particular form into the page. It is an HTML table, so take care that the tags are nested properly. Its template can be found in the file `"error_list.html"`.

Parameters

- **form** (*forms.Form*) – the bound form whose errors should be displayed; if `None`, nothing is generated
- **form_error_title** (*unicode*) – The title used for general error messages. These are not connected to one particular field but the form as a whole. Typically, they are generated in the `is_referentially_valid` functions.
- **outest_tag** (*unicode*) – May be `"<table>"` or `"<tr>"`, with `"<table>"` as the default. It is the outmost HTML tag which is generated for the error list.
- **colspan** (*int*) – the width of the table in the number of columns; necessary because those &##\$# guys of WHATWG have dropped `colspan="0"`; see http://www.w3.org/Bugs/Public/show_bug.cgi?id=13770

12.1.2 Filters

get_really_full_name (*user*, *anchor_type=u'http'*, *autoescape=False*)

Unfortunately, Django's `get_full_name` method for users returns the empty string if the user has no first and surname set. However, it'd be sensible to use the login name as a fallback then. This is realised here. See also `jb_common.utils.get_really_full_name`.

The optional parameter to this filter determines whether the name should be linked or not, and if so, how. There are three possible parameter values:

"http" (default) The user's name should be linked with his web page on JuliaBase

"mailto" The user's name should be linked with his email address

"plain" There should be no link, the name is just printed as plain unformatted text.

markdown (**args*, ***kwargs*)

Filter for formatting the value by assuming Markdown syntax. Embedded HTML tags are always escaped. Warning: You need at least Python Markdown 1.7 or later so that this works.

FixMe: Before Markdown sees the text, all named entities are replaced, see `jb_common.utils.substitute_html_entities()`. This creates a mild escaping problem. `\&` becomes `&` instead of `\&`;. It can only be solved by getting python-markdown to replace the entities, however, I can't easily do that without allowing HTML tags, too.

fancy_bool (*boolean*)

Filter for coverting a bool into a translated "Yes" or "No".

urlquote (**args*, ***kwargs*)

Filter for quoting strings so that they can be used as parts of URLs. Note that also slashes `»/«` are escaped.

Also note that this filter is "not safe" because for example ampersands need to be further escaped.

urlquote_plus (**args*, ***kwargs*)

Filter for quoting URLs so that they can be used within other URLs. This is useful for added "next" URLs in query strings, for example:

```
<a href="{% process.edit_url %}?next={% sample.get_absolute_url|urlquote_plus %}"
  >{% trans 'edit' %}</a>
```

12.2 Samples

You use these tags and filter with:

```
{% load samples_extras %}
```

12.2.1 Tags

verbose_name (*parser*, *token*)

Tag for retrieving the descriptive name for an instance attribute. For example:

```
{% verbose_name Deposition.pressure %}
```

will print “pressure”. Note that it will be translated for a non-English user. It is useful for creating labels. The model name may be of any model in any installed app. If two model names collide, the one of the firstly installed app is taken.

value_field (*parser, token*)

Tag for inserting a field value into an HTML table. It consists of two `<td>` elements, one for the label and one for the value, so it spans two columns. This tag is primarily used in templates of show views, especially those used to compile the sample history. Example:

```
{% value_field layer.base_pressure "W" 3 %}
```

The unit (“W” for “Watt”) is optional. If you have a boolean field, you can give “yes/no” as the unit, which converts the boolean value to a yes/no string (in the current language). For gas flow fields that should collapse if the gas wasn’t used, use “sccm_collapse”. If not given but the model field has a unit set (i.e. `...QuantityField`), that unit is used.

The number 3 is also optional. However, if it is set, the unit must be at least “”. With this option you can set the number of significant digits of the value. The value will be rounded to match the number of significant digits.

split_field (*field1, field2, field3=None*)

Tag for combining two or three input fields which have the same label and help text. It consists of two or three `<td>` elements, one for the label and one for the input fields, so it spans multiple columns. This tag is primarily used in templates of edit views. Example:

```
{% split_field layer.voltage1 layer.voltage2 %}
```

The tag assumes that for from-to fields, the field name of the upper limit must end in “_end”, and for ordinary multiple fields, the verbose name of the first field must end in a space-separated number or letter. For example, the verbose names may be “voltage 1”, “voltage 2”, and “voltage 3”.

value_split_field (*parser, token*)

Tag for combining two or more value fields which have the same label and help text. It consists of two `<td>` elements, one for the label and one for the value fields, so it spans two columns. This tag is primarily used in templates of show views, especially those used to compile the sample history. Example:

```
{% value_split_field layer.voltage_1 layer.voltage_2 "V" %}
```

The unit (“V” for “Volt”) is optional. If you have a boolean field, you can give “yes/no” as the unit, which converts the boolean value to a yes/no string (in the current language). For gas flow fields that should collapse if the gas wasn’t used, use “sccm_collapse”. If not given but the model field has a unit set (i.e. `...QuantityField`), that unit is used.

12.2.2 Filters

round (*value, digits*)

Filter for rounding a numeric value to a fixed number of significant digits. The result may be used for the `quantity()` filter below.

quantity (*value*, *unit=None*, *autoescape=False*)

Filter for pretty-printing a physical quantity. It converts $3.4\text{e-}3$ into $3.4 \cdot 10^{-3}$. The number is the part that is actually filtered, while the unit is the optional argument of this filter. So, you may write:

```
{{ deposition.pressure|quantity:"mbar" }}
```

It is also possible to give a list of two values. This is formatted in a from-to notation.

get_really_full_name (*user*, *anchor_type=u'http'*, *autoescape=False*)

Unfortunately, Django's `get_full_name` method for users returns the empty string if the user has no first and surname set. However, it'd be sensible to use the login name as a fallback then. This is realised here. See also `samples.utils.views.get_really_full_name()`.

The optional parameter to this filter determines whether the name should be linked or not, and if so, how. There are three possible parameter values:

"http" (default) The user's name should be linked with his web page on JuliaBase

"mailto" The user's name should be linked with his email address

"plain" There should be no link, the name is just printed as plain unformatted text.

get_safe_operator_name (*user*, *autoescape=False*)

Return the name of the operator (with the markup generated by `get_really_full_name` and the "http" option) unless it is a confidential external operator.

timestamp (*value*, *minimal_inaccuracy=0*)

Filter for formatting the timestamp of a process properly to reflect the inaccuracy connected with this timestamp.

Parameters *value* (`samples.models.Process` or dict mapping str to object)
– the process whose timestamp should be formatted

Returns the rendered timestamp

Return type unicode

markdown_samples (**args*, ***kwargs*)

Filter for formatting the value by assuming Markdown syntax. Additionally, sample names and sample series names are converted to clickable links. Embedded HTML tags are always escaped. Warning: You need at least Python Markdown 1.7 or later so that this works.

FixMe: Before Markdown sees the text, all named entities are replaced, see `samples.utils.views.substitute_html_entities`. This creates a mild escaping problem. `\&` becomes `& amp;` instead of `\&`. It can only be solved by getting python-markdown to replace the entities, however, I can't easily do that without allowing HTML tags, too.

first_upper (**args*, ***kwargs*)

Filter for formatting the value to set the first character to uppercase.

sample_tags (*sample*, *user*)

Shows the sample's tags. The tags are shortened. Moreover, they are suppressed if the user is not allowed to view them.

camel_case_to_human_text (*value*)

See `jb_common.utils.base.camel_case_to_human_text` for documentation.

MARKDOWN

If you enter a comment, this is just plain text at first. However, JuliaBase supports a special syntax in these comments. This makes them more useful.

This special syntax was not invented for JuliaBase. Instead, it is a well-known markup syntax called “[Markdown](#)”. See also its [homepage](#) for the full story. However, for practical reasons, JuliaBase forbids image inclusion and headings in its comments. But you can use all the rest. Note that some characters (e.g. »_«, »*«, »[«) have a special meaning, so if you want to use them as-is, you have to prepend a backslash »\«.

JuliaBase even adds a nice feature itself: If you enter the name of a sample or a sample series, JuliaBase converts it to a clickable link automatically. Within names, prepending a backslash to »_« is not necessary.

If you like to test JuliaBase’s comment syntax, visit the [Markdown sandbox](#) on the demo site.

13.1 Paragraphs

Paragraphs are separated by an empty line:

```
First paragraph.  
  
Second paragraph.
```

Output:

First paragraph.
Second paragraph.

13.2 Emphasis

```
*italics*, **bold**. Alternatively: _italics_, __bold__
```

italics, **bold**. Alternatively: *italics*, **bold**

13.3 Escaping characters

If you like to use a character as-is but Markdown interprets it as something special, put a backslash before it:

In **italics**, and this is not in **italics**.

In *italics*, and this is not in **italics**.

Note that this is not necessary in sample names. For example, in “08B-410_a_3”, nothing gets italic (if this sample exists).

13.4 Special characters

```
&sigma; = e n &mu;  
&micro;c-Si:H
```

$\sigma = e n \mu$
 $\mu\text{c-Si:H}$

See the [Wikipedia entry](#) for a full list of these characters.

13.5 Math equations

You can use LaTeX equations between \dots :

```
 $\alpha = \frac{1}{\beta}$ 
```

$$\alpha = \frac{1}{\beta}$$

13.6 Links

```
[Homepage of IBM] (http://www.ibm.com), <http://www.fz-juelich.de>
```

Homepage of IBM, <http://www.fz-juelich.de>

Note that names of samples and sample series are converted to links automatically.

13.7 Lists

1. sputtering
2. etching
3. depositing

1. sputtering
2. etching
3. depositing

Yes, that’s not the same! ; –) For long lines, indentation is correct this way.

- * this
- * that
- * and this, too

- this

- that
- and this, too

13.8 Line breaks

Put two spaces at the end of the line:

```
First line.<SPC><SPC>  
Second line.
```

```
First line.  
Second line.
```


THE JULIABASE PROJECT

JuliaBase is an open-source project in the spirit of Free Software. As such, its community is equally open. Our first goal is to have a large community of people adapting JuliaBase to their institutions. While working with JuliaBase's source code, these people will probably make improvements. Our second goal is to encourage everyone to contribute these improvements to the public JuliaBase code. This way, everyone can benefit from the improvements of everyone else!

The home of JuliaBase is its [home page](#). The domain name "juliabase.org" is registered to Torsten Bronger. However, the content of the home page, including all logos, is part of JuliaBase's source code and licensed as Free Software. The current maintainer of JuliaBase is Torsten Bronger, bronger@physik.rwth-aachen.de. The home page is hosted on a server owned by the Research Centre Jülich, Germany.

- Most importantly, JuliaBase's full source code is organized as a [public source code repository](#).
- Also on GitHub is our [bug and feature tracker](#).
- We have a public [mailing list](#) at GoogleGroups. This is both for discussions about the development process as well as general questions. We recommend that you subscribe to this list before sending emails to it. (You don't need a Google account for this. Ask the maintainer if you have questions.)
- For Usenet fans, this mailing list is also available on Gmane as a [newsgroup](#).
- On irc.freenode.net is our IRC chat room called #juliabase.
- The translations are coordinated on [Transifex](#)

14.1 Licenses

JuliaBase's core is licensed under the terms of the [Affero GNU Public License](#) (AGPL).

The following files, however, are distributed under the terms of the less strict [GNU General Public License](#) (GPL):

- The top-level files `settings.py`, `wsgi.py`, `manage.py`, `urls.py`, and `log.py`
- All files below `institute/` (the "institute" app)
- `remote_client/jb_remote_inm.py`
- `remote_client/delphi/juliabase.pas`

- All files below `remote_client/examples/`

Finally, the “six” module and the “mimeparse” module are distributed under their own terms stated at the start of the respective file.

14.1.1 What does this mean?

It effectively means:

1. You can download, run, and modify JuliaBase freely.
2. You can use all files that serve as examples (in particular, the “institute” app) as a starting point for your adaption of JuliaBase.
3. You can offer a JuliaBase web service in your institute, company, or whatever, as long as you also offer the JuliaBase source code, including your modifications, for download for your users. You can fulfill this requirement by contributing your modifications to the JuliaBase project.
4. The GPL ensures that you do *not* need to offer the files of (2.) for download or to contribute them, as they may contain confidential material. Besides, they will change often, so it would be a hassle.

This rather elaborate licensing is done in order to have maximal convenience and flexibility for people who adapt and use JuliaBase, while strongly encourage them to contribute improvements of JuliaBase itself back to the community at large.

Thus, have fun using JuliaBase behind closed doors, but if you improve it, please send patches to the JuliaBase maintainers so that everyone benefits. Thank you!

14.2 Short project history

JuliaBase was started in 2008 in one institute of the Forschungszentrum Jülich under the name of “Chantal” by Torsten Bronger. In 2009, Marvin Goblet was hired as a full-time programmer for Chantal. In 2013/14, three further institutes in Jülich created Chantal deployments, and with them, further programmers joined the team. The core of the source code was separated from the institute-specific parts and re-branded as “JuliaBase”. This name is derived from *Iuliacum*, the Latin name of Jülich, where JuliaBase was contrived.

A

adapting, 17

add

process module, 21

sample, 8, 21

ADD_SAMPLES_VIEW, 43

advanced

search, 6

Apache

configuration, 16

architecture, 55

B

branding, 20

build_forms() (ProcessWithoutSamplesView
 method), 30**C**

CACHE_ROOT, 43

CACHES (setting), 47

camel_case_to_human_text() (in module
 jb_common.utils.base), 57camel_case_to_human_text() (in module sam-
 ples.templatetags.samples_extras),
 80camel_case_to_underscores() (in module
 jb_common.utils.base), 57

chat room, 85

check_filepath() (in module
 jb_common.utils.base), 58check_markdown() (in module
 jb_common.utils.base), 64

claim

sample, 13

coding

guidelines, 55

comments

process, 81

community, 85

configuration

Apache, 16

Django, 16

PostgreSQL, 15

connection (in module jb_remote.common),
 38convert_id_to_int() (in module
 samples.utils.views), 63

CRAWLER_LOGS_ROOT, 43

CRAWLER_LOGS_WHITELIST, 43

crawlers, 35

CSV

export, data, 7

D

data

CSV export, 7

data sheet

sample, 5

dead_samples() (in module
 samples.utils.views), 61

DEBUG (setting), 47

DEBUG_EMAIL_REDIRECT_USERNAME,
 44

DEFAULT_FROM_EMAIL, 48

Delphi, 37

demo, 3

deposition() (PatternGenerator method), 74

DepositionForm (class in
 samples.utils.views), 72DepositionMultipleTypeView (class in
 samples.utils.views), 32DepositionSamplesForm (class in
 samples.utils.views), 72DepositionView (class in
 samples.utils.views), 32

Django

configuration, 16

does_sample_exist() (in module
 samples.utils.views), 62double_urlquote() (in module
 jb_remote.common), 40

E

edit
 sample, 6
 EditDescriptionForm (class in
 samples.utils.views), 72
 environment variable
 PYTHONPATH, 37
 equations, 84
 error_list() (in module
 jb_common.templatetags.juliabase),
 77
 export
 data CSV, 7
 extract_preset_sample() (in module
 samples.utils.views), 62

F

fancy_bool() (in module
 jb_common.templatetags.juliabase),
 78
 feed, 12
 filters
 template, 75
 find_file_in_directory() (in module
 jb_common.utils.base), 58
 first_upper() (in module sam-
 ples.templatetags.samples_extras),
 80
 format_enumeration() (in module
 jb_common.utils.base), 57
 format_lazy() (in module
 jb_common.utils.base), 57
 format_timestamp() (in module
 jb_remote.common), 40
 formulae, 84
 functions
 utility, 56

G

generate_permissions() (in module
 jb_common.utils.base), 65
 get_context_data()
 (ProcessWithoutSamplesView
 method), 30
 get_context_for_user(), 24
 get_next_id() (ProcessWithoutSamplesView
 method), 30
 get_really_full_name() (in module
 jb_common.templatetags.juliabase),
 78
 get_really_full_name() (in module
 jb_common.utils.base), 64

get_really_full_name() (in module sam-
 ples.templatetags.samples_extras),
 80
 get_safe_operator_name() (in module sam-
 ples.templatetags.samples_extras),
 80
 get_sample() (in module
 samples.utils.views), 61
 get_title() (ProcessWithoutSamplesView
 method), 31
 git, 20
 guidelines
 coding, 55

H

help_link() (in module
 jb_common.utils.base), 64
 history
 project, 88

I

initials
 user, 52
 INITIALS_FORMATS, 44
 input_field() (in module
 jb_common.templatetags.juliabase),
 77
 installation, 13
 INSTALLED_APPS, 48
 irc, 85
 is_all_valid() (ProcessWithoutSamplesView
 method), 31
 is_json_requested() (in module
 jb_common.utils.base), 59
 is_referentially_valid() (DepositionForm
 method), 72
 is_referentially_valid() (ProcessForm
 method), 71
 is_referentially_valid()
 (ProcessWithoutSamplesView
 method), 31
 is_update_necessary() (in module
 jb_common.utils.base), 58

J

JAVASCRIPT_I18N_APPS, 44
 jb_remote.common (module), 38
 jb_remote.samples (module), 40
 JSONRequestException, 59
 JuliaBaseConnection (class in
 jb_remote.common), 38
 JuliaBaseError, 39

L

lab notebook, 8, 25
 LANGUAGES (setting), 47
 LDAP, 45
 LDAP_ACCOUNT_FILTER, 45
 LDAP_ADDITIONAL_ATTRIBUTES, 45
 LDAP_ADDITIONAL_USERS, 46
 LDAP_DEPARTMENTS, 46
 LDAP_GROUPS_TO_PERMISSIONS, 46
 LDAP_LOGIN_TEMPLATE, 46
 LDAP_SEARCH_DN, 47
 LDAP_URLS, 47
 license, 87
 login() (in module `jb_remote.common`), 38
 LOGIN_REDIRECT_URL, 48
 LOGIN_URL, 48
 logout() (in module `jb_remote.common`), 39
 lookup_sample() (in module `samples.utils.views`), 62

M

mailing list, 85
 main menu, 4
 markdown, 81
 markdown() (in module `jb_common.templatetags.juliabase`), 78
 markdown_hint() (in module `jb_common.templatetags.juliabase`), 77
 markdown_samples() (in module `samples.templatetags.samples_extras`), 80
 maths, 84
 measurement, 26
 MERGE_CLEANUP_FUNCTION, 44
 MIDDLEWARE_CLASSES, 48
 migration
 schema, 22
 mkdirs() (in module `jb_common.utils.base`), 59
 module
 add process, 21
 MultipleUsersField (class in `jb_common.utils.views`), 71
 My Samples, 4

N

name
 prefix templates, 53
 NAME_PREFIX_TEMPLATES, 44

names
 sample, 49
 sample provisional, 52
 newsfeed, 12
 newsgroup, 85
 normalize_sample_name() (in module `samples.utils.views`), 62
 notifications, 12

O

open() (JuliaBaseConnection method), 38

P

parse_timestamp() (in module `jb_remote.common`), 40
 PatternGenerator (class in `samples.utils.urls`), 74
 permissions, 4, 9, 13, 26
 user, 3
 physical process, 26
 physical_process() (PatternGenerator method), 74
 PlotError, 72
 PostgreSQL
 configuration, 15
 prefix
 templates, name, 53
 prerequisites, 15
 primary_keys (in module `jb_remote.common`), 38
 PrimaryKeys (class in `jb_remote.common`), 39
 privileges
 user, 3
 process, 6, 21, 26
 comments, 81
 module, add, 21
 template, 23
 unfinished, 27
 view, 23
 ProcessForm (class in `samples.utils.views`), 71
 ProcessMultipleSamplesView (class in `samples.utils.views`), 32
 ProcessView (class in `samples.utils.views`), 31
 ProcessWithoutSamplesView (class in `samples.utils.views.class_views`), 29
 project
 history, 88
 provisional
 names, sample, 52
 Python 3, 56
 PYTHONPATH, 37

Q

quantity() (in module samples.templatetags.samples_extras), 79

R

read_plot_file_beginning_after_start_value() (in module samples.utils.plots), 73
 read_plot_file_beginning_at_line_number() (in module samples.utils.plots), 73
 remote client, 33
 remove_file() (in module jb_common.utils.base), 59
 remove_samples_from_my_samples() (in module samples.utils.views), 62
 RemoveFromMySamplesForm (class in samples.utils.views), 72
 RemoveFromMySamplesMixin (class in samples.utils.views), 32
 report_changed_sample_series_topic() (Reporter method), 66
 report_changed_sample_topic() (Reporter method), 66
 report_changed_topic_membership() (Reporter method), 67
 report_copied_my_samples() (Reporter method), 67
 report_edited_sample_series() (Reporter method), 67
 report_edited_samples() (Reporter method), 67
 report_new_responsible_person_sample_series() (Reporter method), 68
 report_new_responsible_person_samples() (Reporter method), 68
 report_new_sample_series() (Reporter method), 68
 report_new_samples() (Reporter method), 68
 report_physical_process() (Reporter method), 69
 report_removed_task() (Reporter method), 69
 report_result_process() (Reporter method), 69
 report_sample_split() (Reporter method), 69
 report_status_message() (Reporter method), 69
 report_task() (Reporter method), 70
 report_withdrawn_status_message() (Reporter method), 70
 Reporter (class in samples.utils.views), 65
 repository, 85
 respond_in_json() (in module

jb_common.utils.base), 60
 restricted_samples_query() (in module samples.utils.views), 63
 result, 6, 26
 Result (class in jb_remote.samples), 40
 round() (in module jb_common.utils.base), 65
 round() (in module samples.templatetags.samples_extras), 79

S

sample
 add, 8, 21
 claim, 10, 13
 data sheet, 5
 edit, 6
 names, 49
 provisional names, 52
 send to user, 11
 series, 4
 split, 6
 Sample (class in jb_remote.samples), 41
 sample_name_format() (in module samples.utils.views), 61
 SAMPLE_NAME_FORMATS, 45, 51
 sample_tags() (in module samples.templatetags.samples_extras), 80
 SampleSelectForm (class in samples.utils.views), 72
 sanitize_for_markdown() (in module jb_remote.common), 40
 save_to_database() (ProcessWithoutSamplesView method), 31
 schema
 migration, 22
 SECRET_KEY, 48
 send to user
 sample, 11
 send_email() (in module jb_common.utils.base), 64
 series
 sample, 4
 set_topics() (TopicField method), 71
 set_users() (MultipleUsersField method), 71
 set_users() (UserField method), 70
 set_users_without() (UserField method), 70
 settings, 41
 setup_logging() (in module jb_remote.common), 39

- source code, 85
 - structure, 19
- split
 - sample, 6
- split_field() (in module samples.templatetags.samples_extras), 79
- startup() (ProcessWithoutSamplesView method), 31
- static_file_response() (in module jb_common.utils.base), 60
- structure
 - source code, 19
- submit() (Result method), 40
- submit() (Sample method), 41
- SubprocessesMixin (class in samples.utils.views), 32
- SubprocessForm (class in samples.utils.views), 32
- successful_response() (in module samples.utils.views), 60
- T**
- table_export() (in module samples.utils.views), 63
- tags
 - template, 75
- tasks, 12
- template
 - filters, 75
 - process, 23
 - tags, 75
- TEMPLATE_CONTEXT_PROCESSORS, 49
- TEMPLATE_DIRS, 49
- TEMPLATE_LOADERS, 49
- templates
 - name prefix, 53
- TemporaryMySamples (class in jb_remote.samples), 41
- THUMBNAIL_WIDTH, 45
- timestamp() (in module samples.templatetags.samples_extras), 80
- TopicField (class in jb_common.utils.views), 71
- topics, 4, 4
- U**
- unfinished
 - process, 27
- urlquote() (in module jb_common.templatetags.juliabase), 78
- urlquote_plus() (in module jb_common.templatetags.juliabase), 78
- URLs, 23
- USE_TZ, 49
- USE_X_SENDFILE, 45
- usenet, 85
- user
 - initials, 52
 - permissions, 3
 - privileges, 3
- user context, 24
- UserField (class in jb_common.utils.views), 70
- utility
 - functions, 56
- V**
- value_field() (in module samples.templatetags.samples_extras), 79
- value_split_field() (in module samples.templatetags.samples_extras), 79
- verbose_name() (in module samples.templatetags.samples_extras), 78
- view
 - process, 23