CATMAID Documentation

Release 2021.12.21

CATMAID Developers

Mar 14, 2022

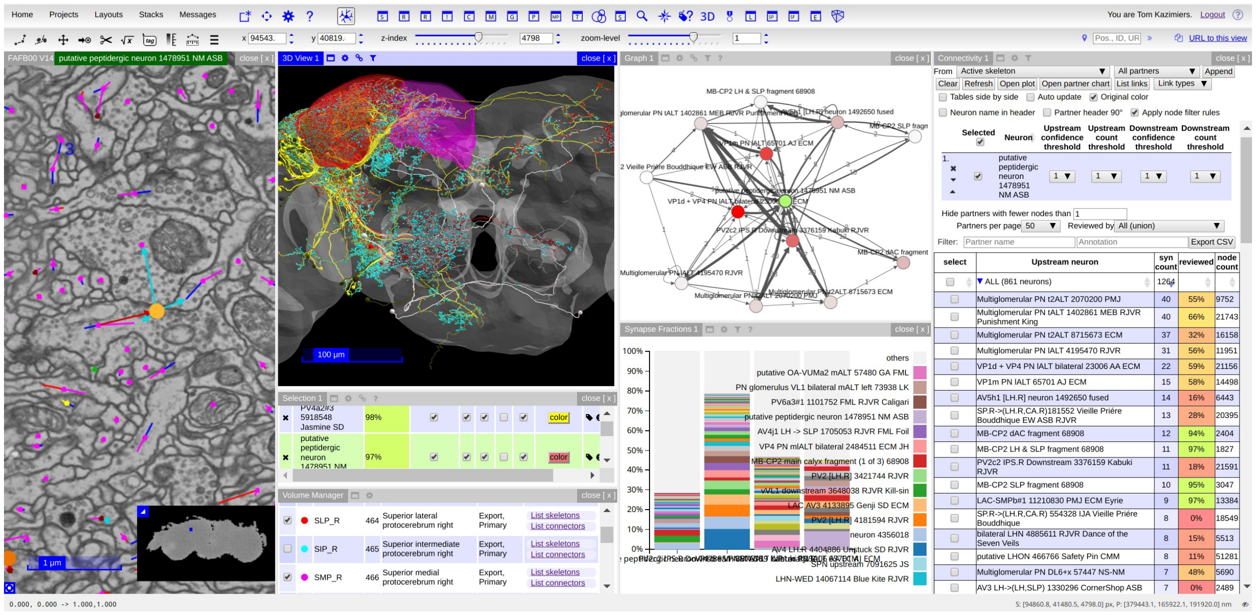
CONTENTS

The **C**ollaborative **A**nnotation **T**oolkit for **M**assive **A**mounts of **I**mage **D**ata

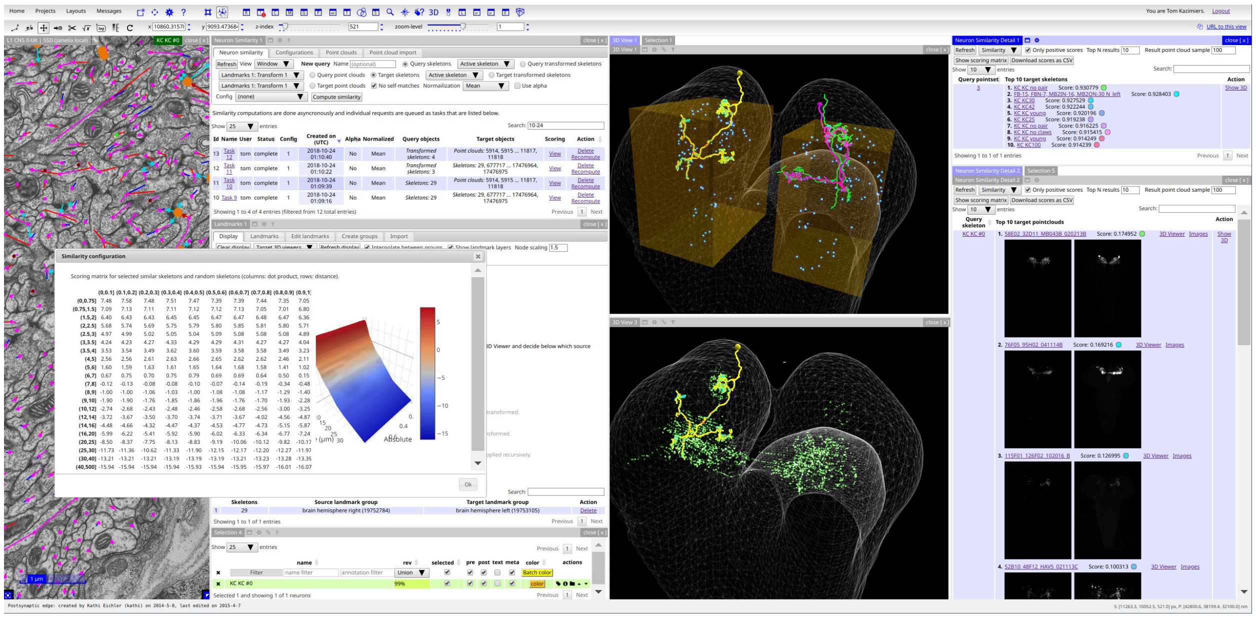
Features:

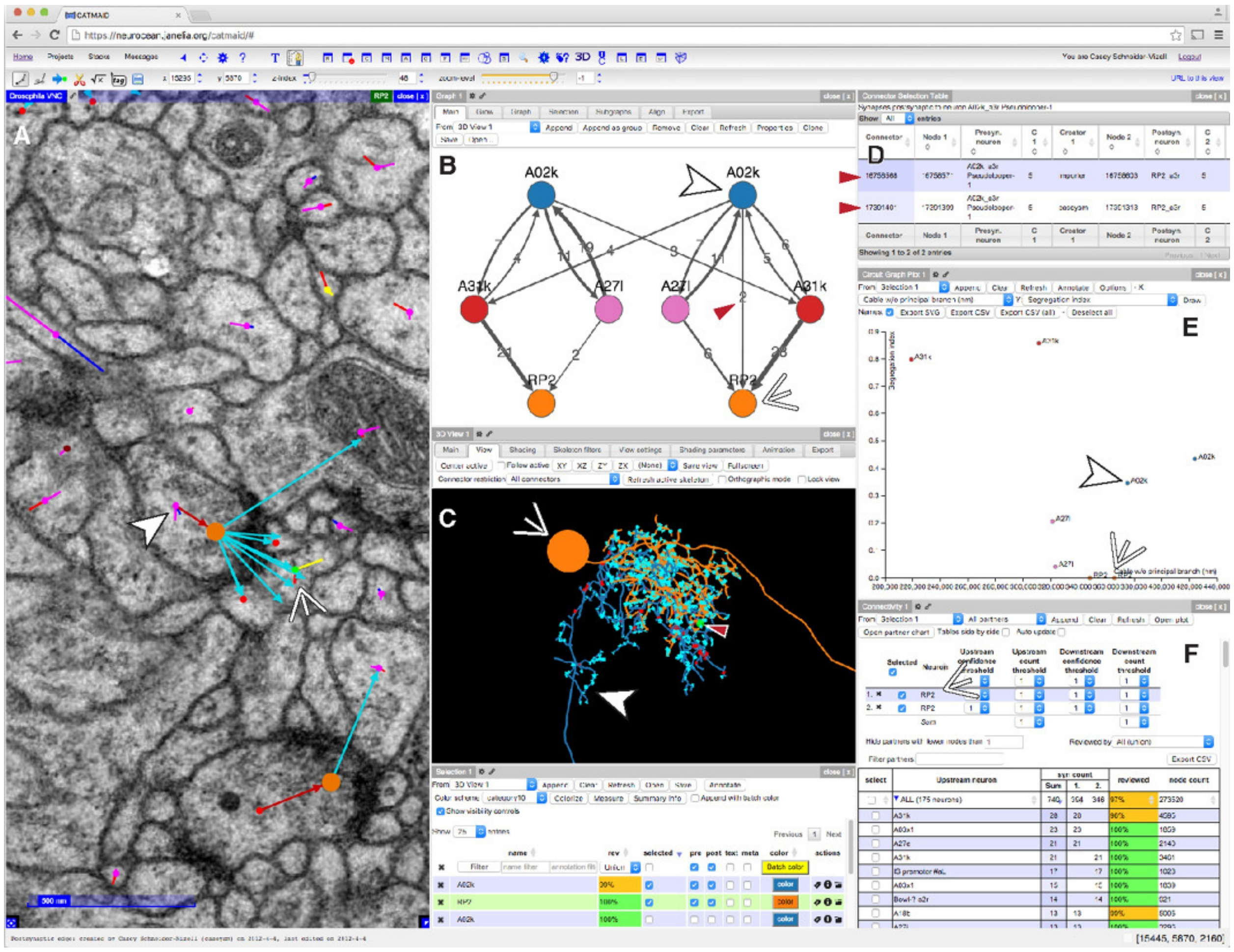
- Fast terabyte-scale image data browsing
- Collaborative microcircuit reconstruction and annotation
- Flexible hierarchical semantic annotation
- Multiple linked image stack display
- Neuron Catalog
- SVG and WebGL-based neuronal morphology viewer
- Open source software (GPLv3)

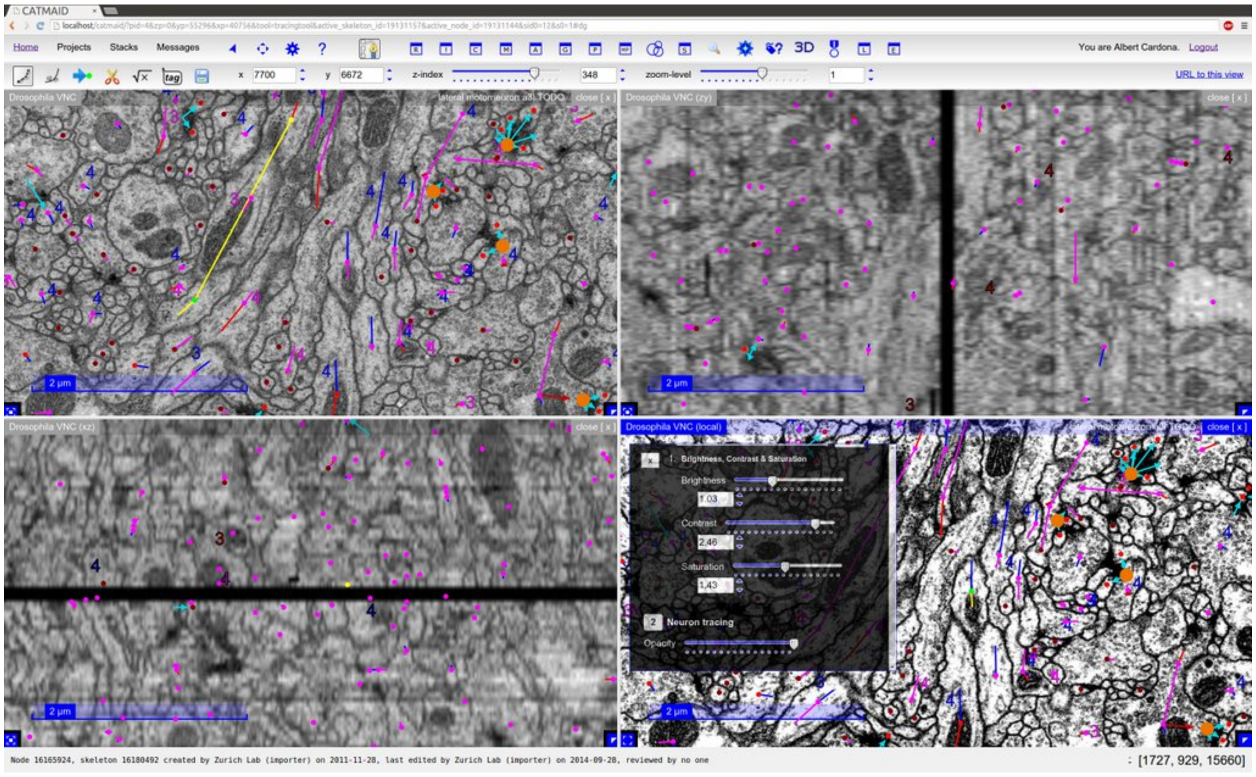
User interface



select	Upstream neuron	syn count	reviewed	node count
	ALL (861 neurons)	1264		
	Multiglomerular PN t2ALT 2070200 PMJ	40	55%	9752
	Multiglomerular PN tALT 1402861 MEB RJVR Punishment King	40	66%	21743
	Multiglomerular PN t2ALT 8715673 ECM	37	32%	16158
	Multiglomerular PN lALT 4195470 RJVR	31	56%	11951
	VP1d + VP4 PN lALT bilateral 23006 AA ECM	22	59%	21156
	VP1m PN lALT 65701 AJ ECM	15	58%	14498
	AV5h1 [LH.R] neuron 1492650 fused	14	16%	6443
	SP.R->(LH.R,CA.R)181552 Vieille Prière Bouddhique EW ASB RJVR	13	28%	20395
	MB-CP2 dAC fragment 68908	12	94%	2404
	MB-CP2 LH & SLP fragment 68908	11	97%	1827
	PV2c2 IPS.R Downstream 3376159 Kabuki RJVR	11	18%	21591
	MB-CP2 SLP fragment 68908	10	95%	3047
	LAC-SMPb#1 11210830 PMJ ECM Eyrie	9	97%	13384
	SP.R->(LH.R,CA.R) 554328 IJA Vieille Prière Bouddhique	8	0%	18549
	bilateral LHN 4885611 RJVR Dance of the Seven Veils	8	15%	5513
	putative LHON 466766 Safety Pin CMM	8	11%	51281
	Multiglomerular PN DL6+x 57447 NS-NM	7	48%	5690
	AV3 LH->(LH,SLP) 1330296 CornerShop ASB	7	0%	2489



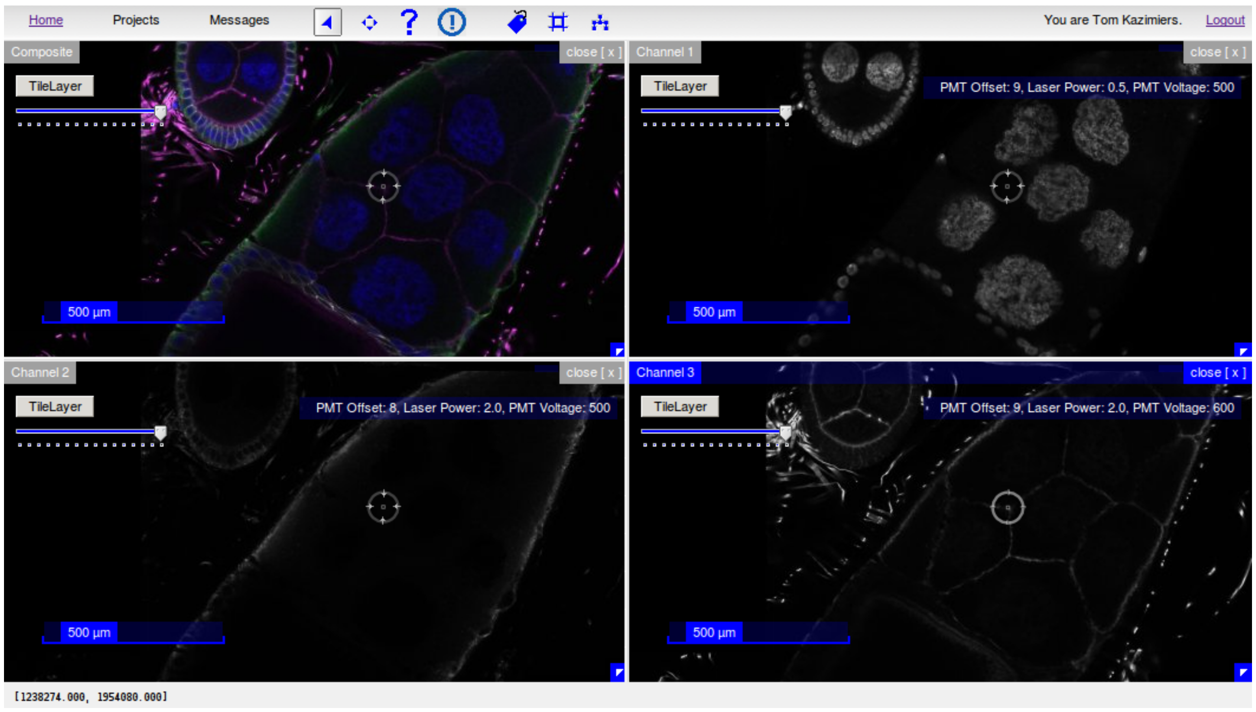

A
B
C
D
E
F
A02k
A31k
A27l
RP2
Segregation index
Cable w/o principal branch (nm)
500 nm



Brightness 1.03, Contrast 2.46, Saturation 1.43

Node 16165924, skeleton 16180492 created by Zurich Lab (importer) on 2011-11-28, last edited by Zurich Lab (importer) on 2014-09-28, reviewed by no one

[1727, 929, 15660]

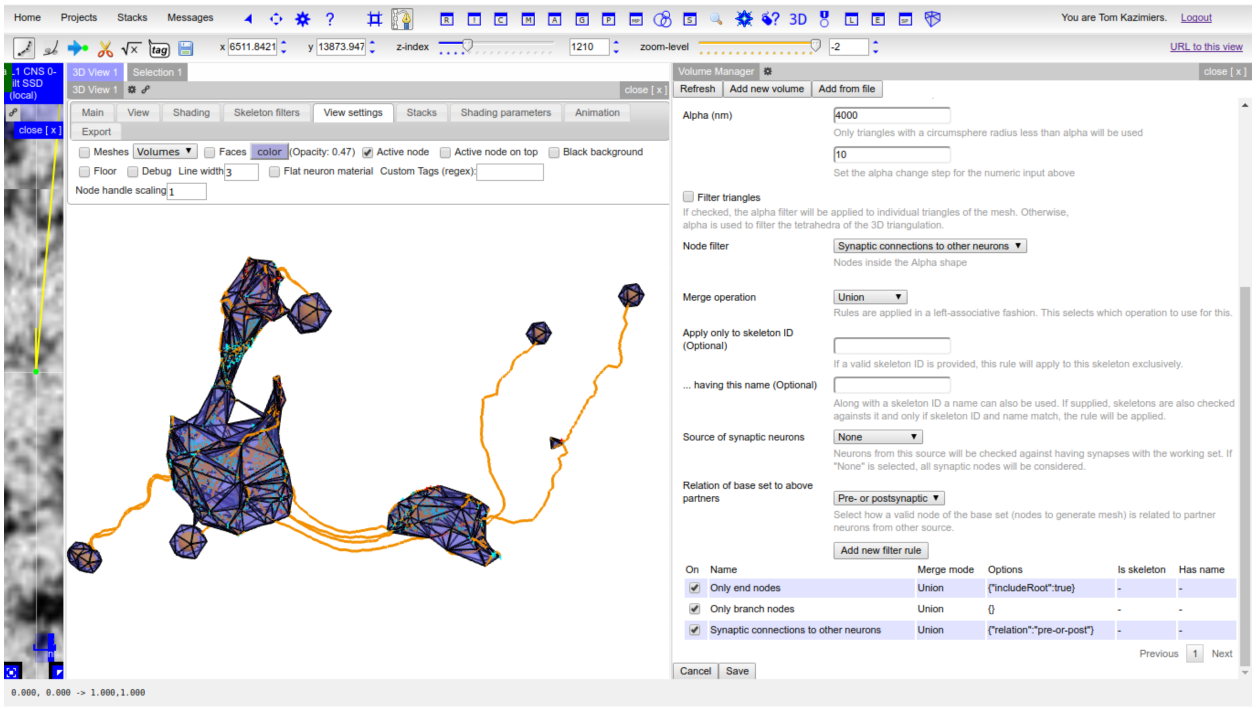


PMT Offset: 9, Laser Power: 0.5, PMT Voltage: 500

PMT Offset: 8, Laser Power: 2.0, PMT Voltage: 500

PMT Offset: 9, Laser Power: 2.0, PMT Voltage: 600

[1238274.000, 1954080.000]



x 6511.8421 y 13873.947 z-index 1210 zoom-level -2

Meshes [Volumes] Faces color (Opacity: 0.47) ✓ Active node Active node on top Black background
Floor Debug Line width 3 Flat neuron material Custom Tags (regex):
Node handle scaling 1

Alpha (nm) 4000
Only triangles with a circumsphere radius less than alpha will be used
10
Set the alpha change step for the numeric input above

Filter triangles
If checked, the alpha filter will be applied to individual triangles of the mesh. Otherwise, alpha is used to filter the tetrahedra of the 3D triangulation.

Node filter: Synaptic connections to other neurons
Nodes inside the Alpha shape

Merge operation: Union
Rules are applied in a left-associative fashion. This selects which operation to use for this.

Apply only to skeleton ID (Optional)
If a valid skeleton ID is provided, this rule will apply to this skeleton exclusively.

... having this name (Optional)
Along with a skeleton ID a name can also be used. If supplied, skeletons are also checked againsts it and only if skeleton ID and name match, the rule will be applied.

Source of synaptic neurons: None
Neurons from this source will be checked against having synapses with the working set. If "None" is selected, all synaptic nodes will be considered.

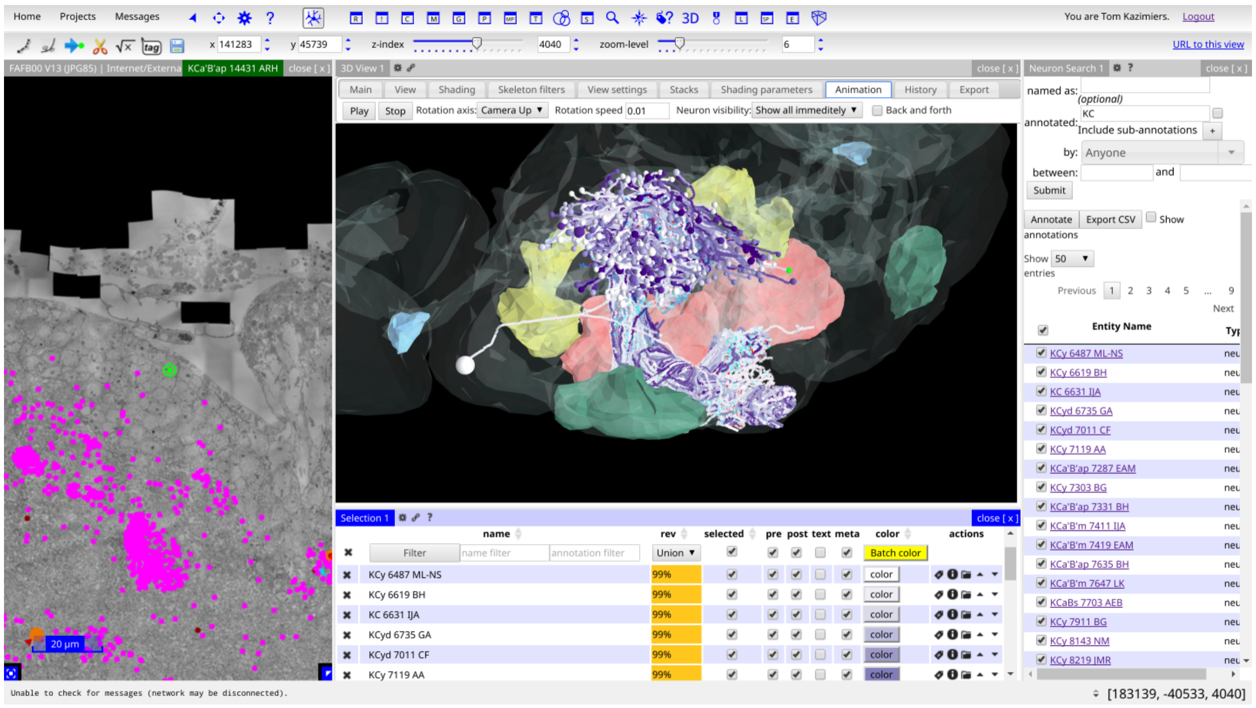
Relation of base set to above partners: Pre- or postsynaptic
Select how a valid node of the base set (nodes to generate mesh) is related to partner neurons from other source.

Add new filter rule

On	Name	Merge mode	Options	Is skeleton	Has name
✓	Only end nodes	Union	{"includeRoot":true}	-	-
✓	Only branch nodes	Union	{}	-	-
✓	Synaptic connections to other neurons	Union	{"relation":"pre-or-post"}	-	-

Previous 1 Next

0.000, 0.000 -> 1.000,1.000



x 141283 y 45739 z-index 4040 zoom-level 6

Play Stop Rotation axis: Camera Up Rotation speed 0.01 Neuron visibility: Show all immediately Back and forth

named as: (optional)
annotated: KC Include sub-annotations
by: Anyone
between: and
Submit
Annotate Export CSV Show annotations
Show 50 entries
Previous 1 2 3 4 5 ... 9 Next

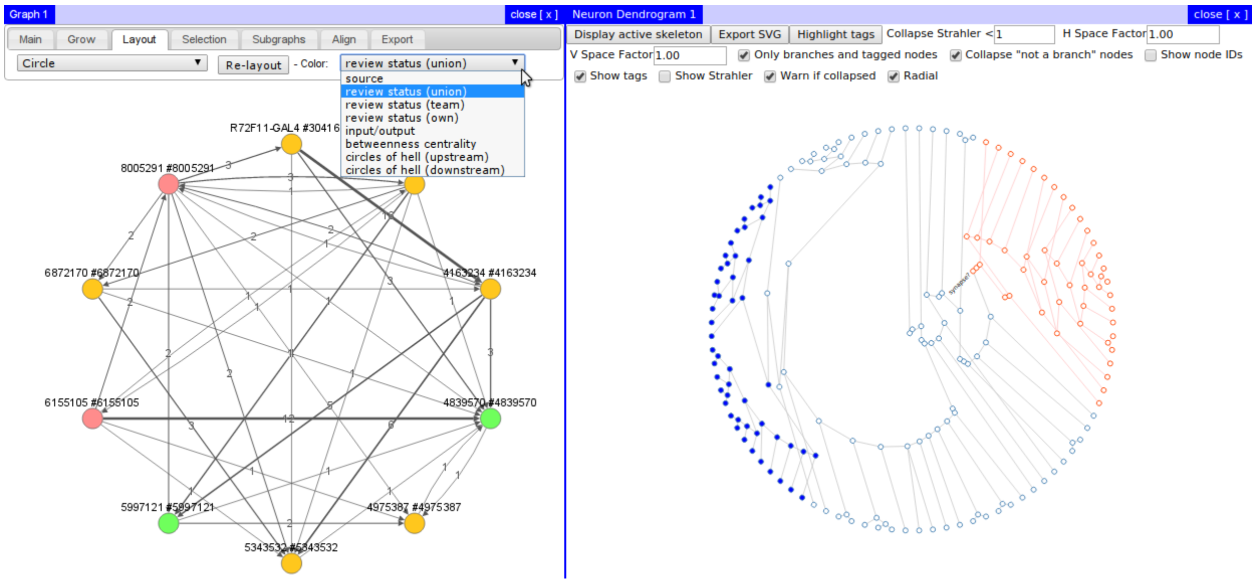
	Entity Name	Typ
✓	KCy 6487 ML-NS	neu
✓	KCy 6619 BH	neu
✓	KC 6631 IJA	neu
✓	KCyd 6735 GA	neu
✓	KCyd 7011 CF	neu
✓	KCy 7119 AA	neu
✓	KCa'B'ap 7287 EAM	neu
✓	KCy 7303 BG	neu
✓	KCa'B'ap 7331 BH	neu
✓	KCa'B'm 7411 IJA	neu
✓	KCa'B'm 7419 EAM	neu
✓	KCa'B'ap 7635 BH	neu
✓	KCa'B'm 7647 LK	neu
✓	KCaBs 7703 AEB	neu
✓	KCy 7911 BG	neu
✓	KCy 8143 NM	neu
✓	KCy 8219 JMR	neu

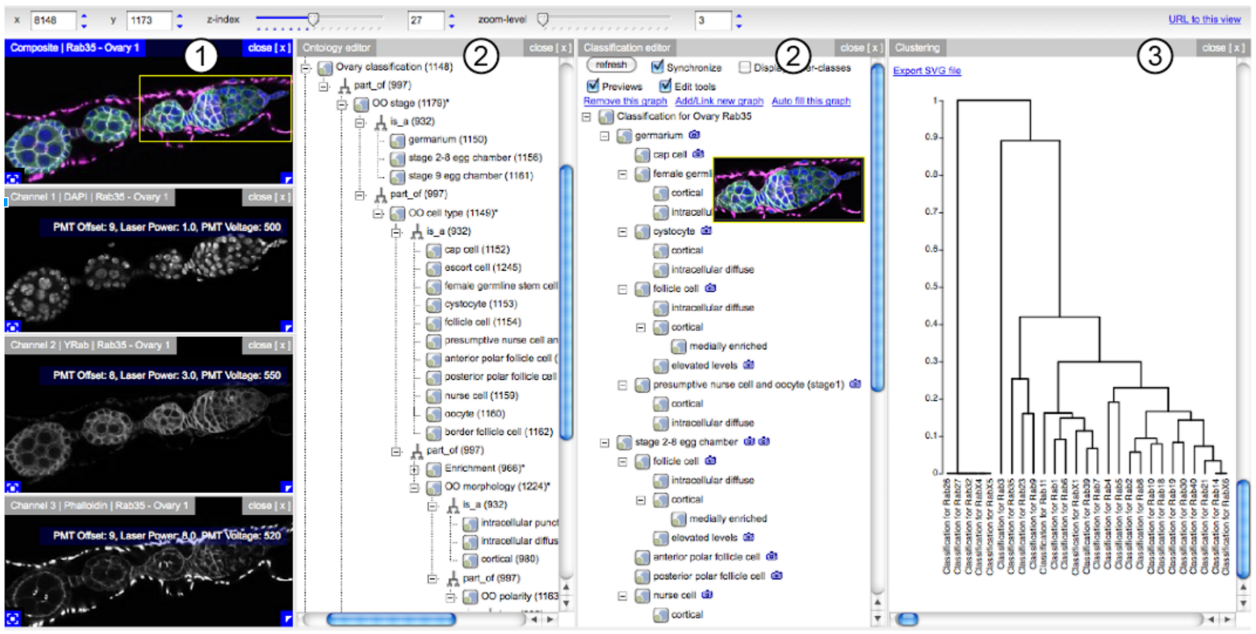
Selection 1

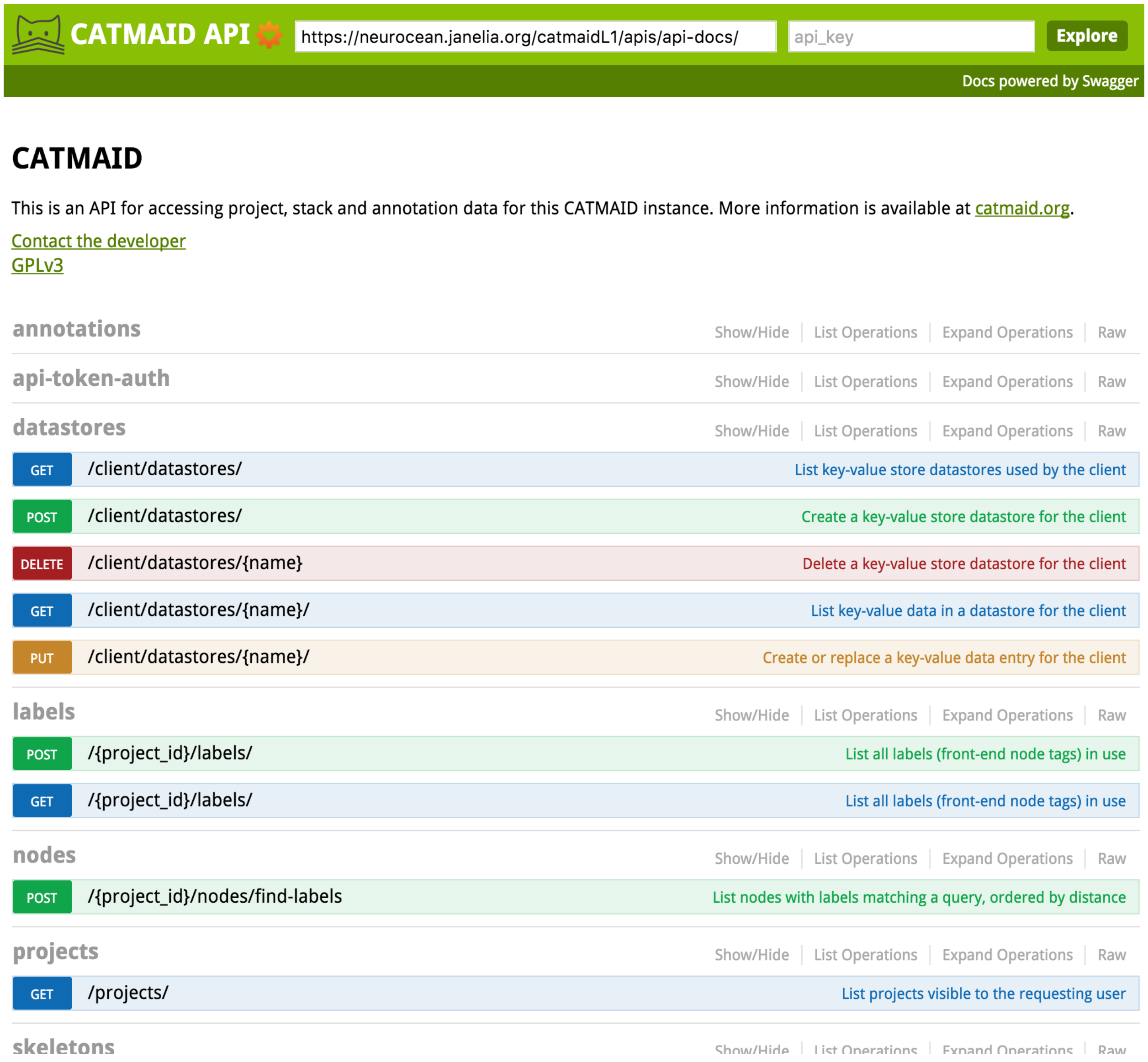
	name	rev	selected	pre	post	text	meta	color	actions
✖	Filter name filter annotation filter	Union	✓	✓	✓	☐	✓	Batch color	
✖	KCy 6487 ML-NS	99%	✓	✓	✓	☐	✓	color	
✖	KCy 6619 BH	99%	✓	✓	✓	☐	✓	color	
✖	KC 6631 IJA	99%	✓	✓	✓	☐	✓	color	
✖	KCyd 6735 GA	99%	✓	✓	✓	☐	✓	color	
✖	KCyd 7011 CF	99%	✓	✓	✓	☐	✓	color	
✖	KCy 7119 AA	99%	✓	✓	✓	☐	✓	color	

Unable to check for messages (network may be disconnected).

[183139, -40533, 4040]


Graph 1
close [x]
Main
Grow
Layout
Selection
Subgraphs
Align
Export
Circle
Re-layout
- Color:
review status (union)
source
review status (union)
review status (team)
review status (own)
input/output
betweenness centrality
circles of hell (upstream)
circles of hell (downstream)
R72F11-GAL4 #30416
8005291 #8005291
6872170 #6872170
4163234 #4163234
6155105 #6155105
4839570 #4839570
5997121 #5997121
4975387 #4975387
5343532 #5343532
Neuron Dendrogram 1
Display active skeleton
Export SVG
Highlight tags
Collapse Strahler <
H Space Factor
V Space Factor
Only branches and tagged nodes
Collapse "not a branch" nodes
Show node IDs
Show tags
Show Strahler
Warn if collapsed
Radial


URL to this view
z-index
zoom-level
Composite | Rab35 - Ovary 1
1
Channel 1 | DAPI | Rab35 - Ovary 1
PMT Offset: 9, Laser Power: 1.0, PMT Voltage: 500
Channel 2 | YRab | Rab35 - Ovary 1
PMT Offset: 8, Laser Power: 3.0, PMT Voltage: 550
Channel 3 | Phalloidin | Rab35 - Ovary 1
PMT Offset: 9, Laser Power: 8.0, PMT Voltage: 520
Ontology editor
2
Ovary classification (1148)
part_of (997)
OO stage (1178)*
is_a (932)
germarium (1150)
stage 2-8 egg chamber (1156)
stage 9 egg chamber (1161)
OO cell type (1149)*
cap cell (1152)
escort cell (1245)
female germline stem cell
cystocyte (1153)
follicle cell (1154)
nurse cell (1159)
oocyte (1160)
border follicle cell (1162)
Enrichment (966)*
OO morphology (1224)*
OO polarity (1163
Classification editor
2
refresh
Synchronize
Previews
Edit tools
Remove this graph
Add/Link new graph
Auto fill this graph
Classification for Ovary Rab35
germarium
cap cell
cortical
intracellular diffuse
follicle cell
medially enriched
elevated levels
presumptive nurse cell and oocyte (stage1)
stage 2-8 egg chamber
anterior polar follicle cell
posterior polar follicle cell
nurse cell
Clustering
3
Export SVG file
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0



CATMAID API https://neurocean.janelia.org/catmaidL1/apis/api-docs/ api_key Explore

Docs powered by Swagger

CATMAID

This is an API for accessing project, stack and annotation data for this CATMAID instance. More information is available at catmaid.org.

Contact the developer
GPLv3

annotations Show/Hide | List Operations | Expand Operations | Raw

api-token-auth Show/Hide | List Operations | Expand Operations | Raw

datastores Show/Hide | List Operations | Expand Operations | Raw

GET	/client/datastores/	List key-value store datastores used by the client
POST	/client/datastores/	Create a key-value store datastore for the client
DELETE	/client/datastores/{name}	Delete a key-value store datastore for the client
GET	/client/datastores/{name}/	List key-value data in a datastore for the client
PUT	/client/datastores/{name}/	Create or replace a key-value data entry for the client

labels Show/Hide | List Operations | Expand Operations | Raw

POST	/{project_id}/labels/	List all labels (front-end node tags) in use
GET	/{project_id}/labels/	List all labels (front-end node tags) in use

nodes Show/Hide | List Operations | Expand Operations | Raw

POST	/{project_id}/nodes/find-labels	List nodes with labels matching a query, ordered by distance

projects Show/Hide | List Operations | Expand Operations | Raw

GET	/projects/	List projects visible to the requesting user

skeletons Show/Hide | List Operations | Expand Operations | Raw

CHAPTER

ONE

DOCUMENTATION

1.1 User Documentation

1.1.1 Introduction

CATMAID is a Collaborative Annotation Toolkit for Massive Amounts of Image Data. It is designed to navigate, share and collaboratively annotate massive image data sets of biological specimens. The interface is inspired by GoogleMaps, with which it shares basic navigation concepts, enhanced to allow the exploration of 3D biological image data acquired by optical or physical sectioning microscopy techniques. The interface enables seamless sharing of regions of interest through bookmarks and synchronized navigation through multiple registered data sets.

With massive biological image data sets it is unrealistic to create a sustainable centralized repository. A unique feature of CATMAID is its partially decentralized architecture where the presented image data can reside on any Internet accessible server and yet can be easily cross-referenced in the central database. In this way no image data are duplicated and the data producers retain full control over their images.

CATMAID is intended to serve as data sharing platform for biologists using high-resolution imaging techniques to probe large specimens. Any high-throughput, high-content imaging project such as gene expression pattern screens would benefit from the interface for data sharing and annotation.

1.1.2 Screenshots and videos

1.1.3 Tools

CATMAID includes a set of tools to navigate, share and annotate large image data sets. These tools often contain some control elements to modify certain parameters. This could e.g. be sliders, buttons or input boxes. Most of the controls offer various ways for changing a value and so sliders and input boxes can be changed with the help of the mouse wheel, small buttons or direct input.

The tools visible to a user can be set in the admin interface on a per user basis. See the section about *user profiles* to learn how to modify the visible tools and how to set defaults suiting your use case.

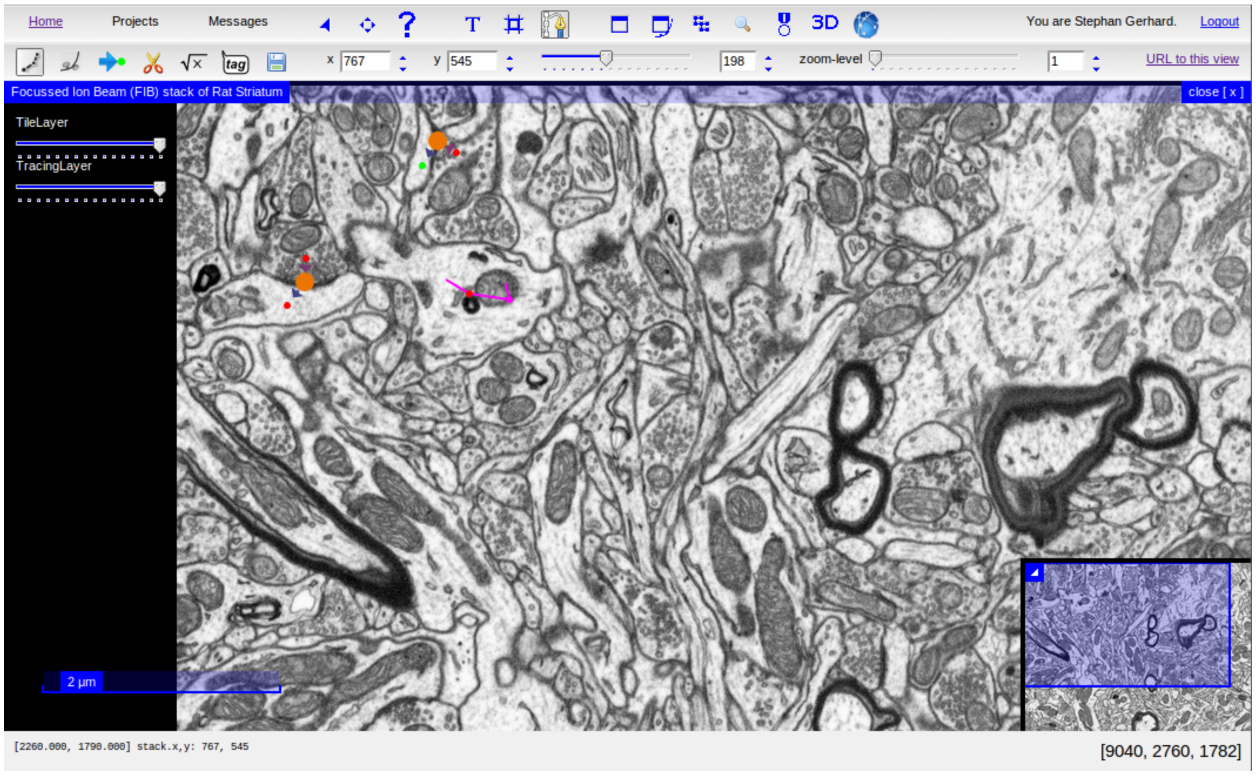


Fig. 1: The CATMAID User Interface with skeleton annotations and tags

Navigation

The navigation tool makes it easy to browse a data set. It allows you to specify the center of the current view and provides controls to change the current slice and zoom level. Keep in mind that a zoom-level defines how often the images dimension get divided by two (i.e. dim/2^zoom). A change of these controls will redraw the current view.

Clicking and dragging the view with the mouse can be used to move around on the current slice.

Tagging

Projects and stacks can be associated with tags. Tags are basically strings that capture some property of an object. They can contain spaces and any other alphanumerical character. CATMAID's tagging tool will allow you to view and modify the tags of an active stack and of the project it belongs to. Changes will only be applied when the check icon on the right is clicked.

Cropping

With the help of the cropping tool it is possible to extract sub-stacks out of the currently viewed stack and other stacks in the project. The region of interest can be specified by clicking with the left mouse button on the view and dragging the created rectangle to the desired shape. This rectangle can be created and adjusted as well with the help of the four input boxes in the cropping tool bar.

By default only one slice is created. However, by using the "top z-index" and "bottom z-index" sliders, the range in the Z dimension can be modified. The "zoom-level" slider denotes what the zoom level of the *output image* will be. It is perfectly fine to draw a crop box in a view with zoom level three and let the cropping tool create an output stack based on zoom level zero. Note that one can easily increase the output file size with operations like this.

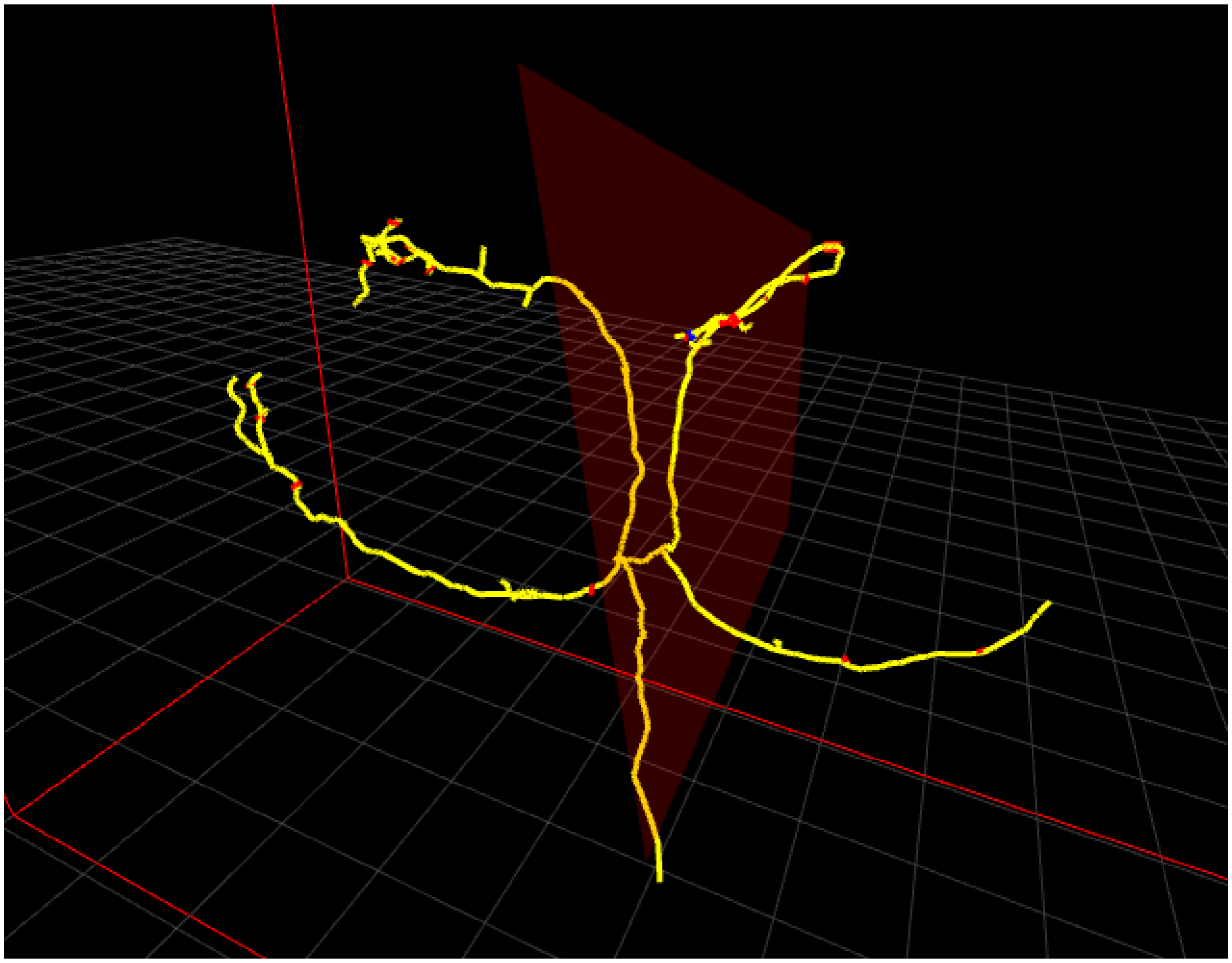


Fig. 2: CATMAID comes with a 3D viewer

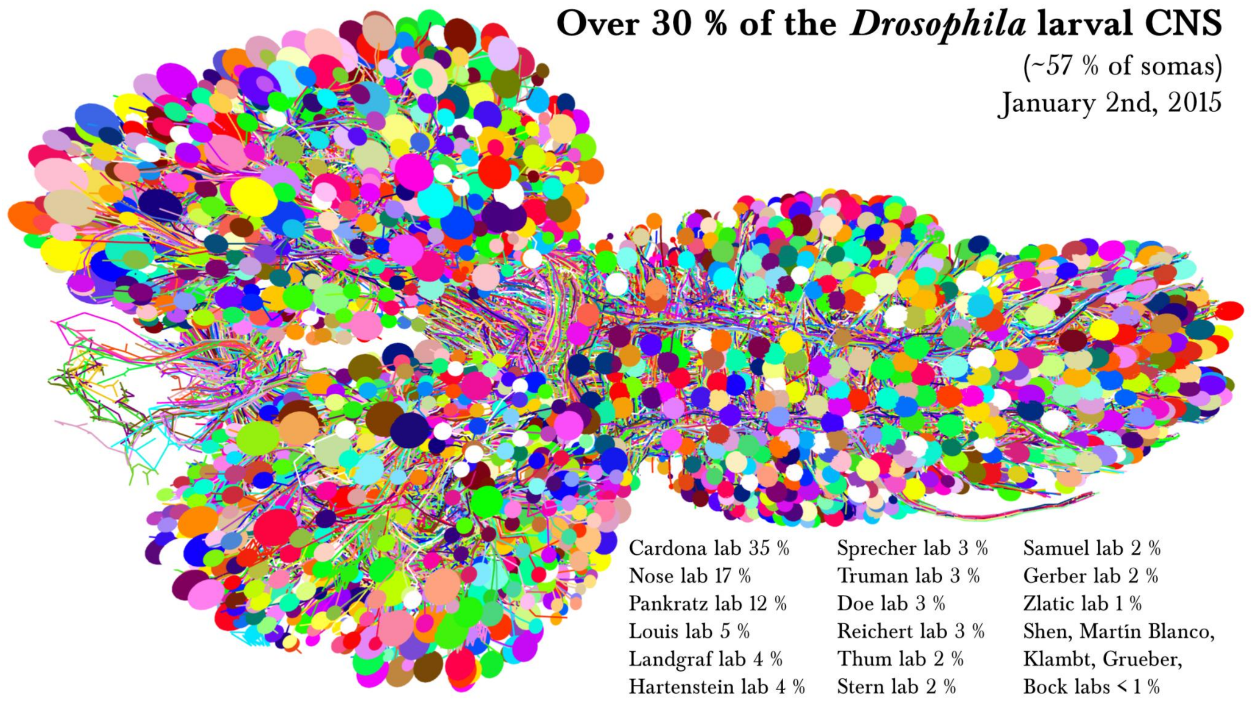
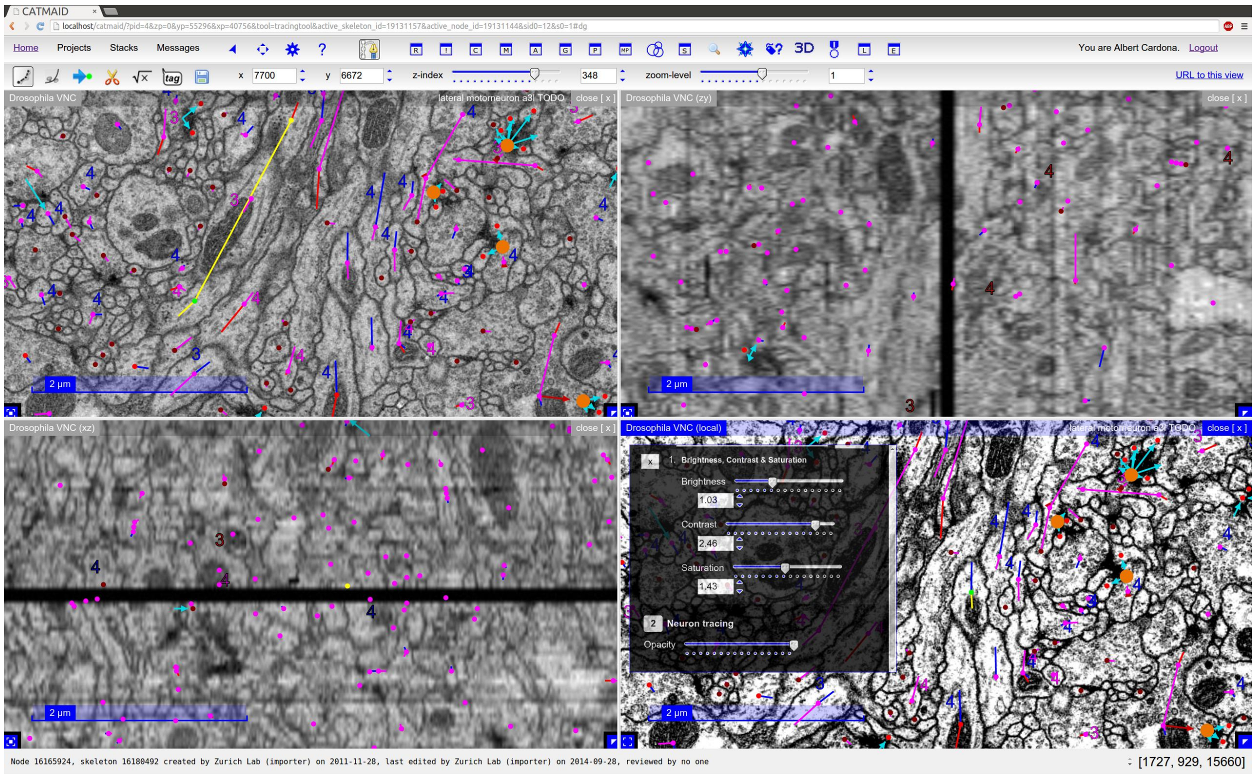


Fig. 3: 3D View of *Drosophila* Larval CNS *Tracing by Albert Cardona's lab at HHMI Janelia Research Campus and collaborators.*



```
Node 16165924, skeleton 16180492 created by Zurich Lab (importer) on 2011-11-28, last edited by Zurich Lab (importer) on 2014-09-28, reviewed by no one
```

[1727, 929, 15660]

Fig. 4: Orthogonal views of the same image stack are supported by CATMAID

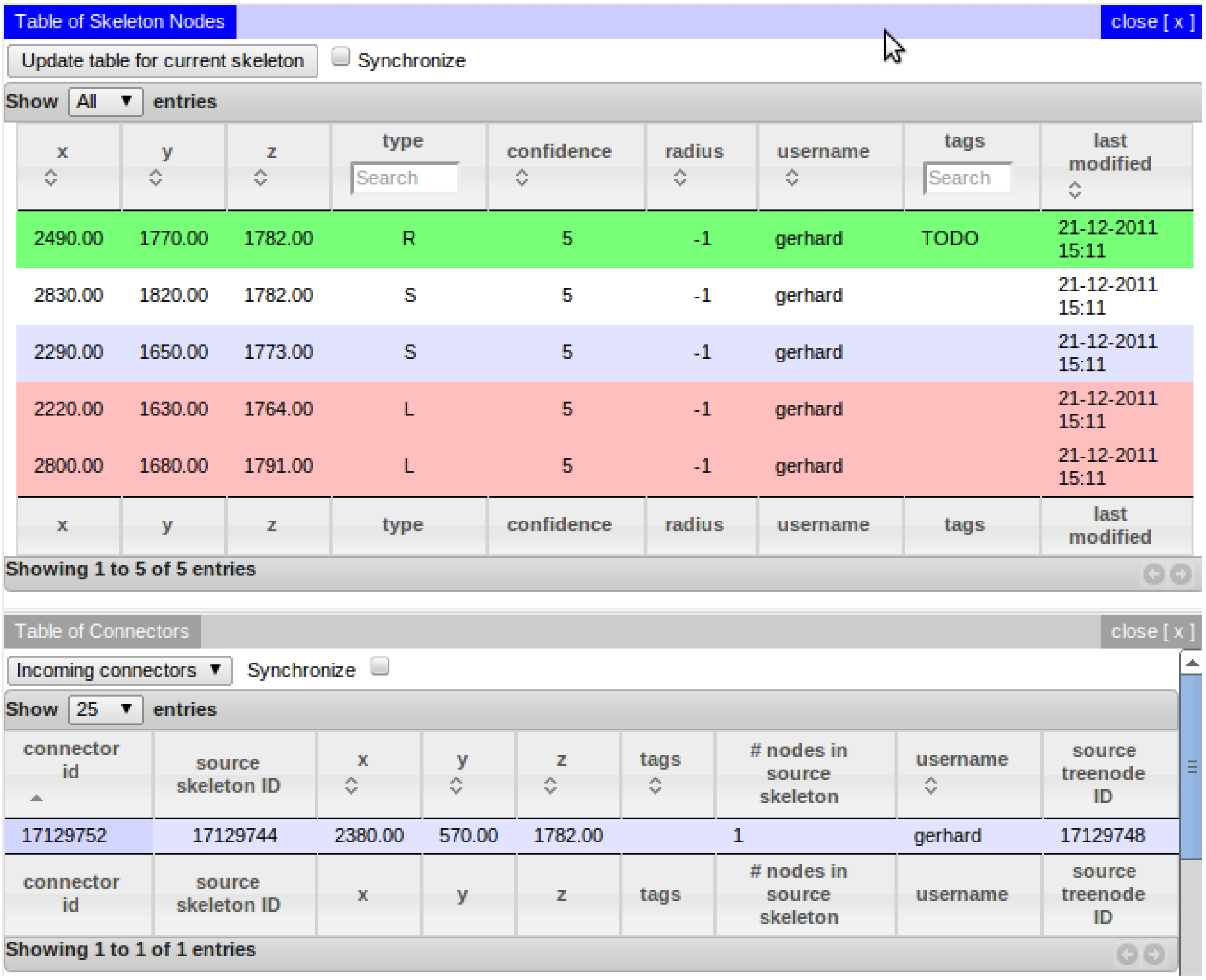


Table of Skeleton Nodes — close [x]

Update table for current skeleton — Synchronize

Show All entries

x	y	z	type	confidence	radius	username	tags	last modified
2490.00	1770.00	1782.00	R	5	-1	gerhard	TODO	21-12-2011 15:11
2830.00	1820.00	1782.00	S	5	-1	gerhard		21-12-2011 15:11
2290.00	1650.00	1773.00	S	5	-1	gerhard		21-12-2011 15:11
2220.00	1630.00	1764.00	L	5	-1	gerhard		21-12-2011 15:11
2800.00	1680.00	1791.00	L	5	-1	gerhard		21-12-2011 15:11

Showing 1 to 5 of 5 entries

Table of Connectors — close [x]

Incoming connectors — Synchronize

Show 25 entries

connector id	source skeleton ID	x	y	z	tags	# nodes in source skeleton	username	source treenode ID
17129752	17129744	2380.00	570.00	1782.00		1	gerhard	17129748

Showing 1 to 1 of 1 entries

Fig. 5: CATMAID’s Skeleton Node and Connector Table

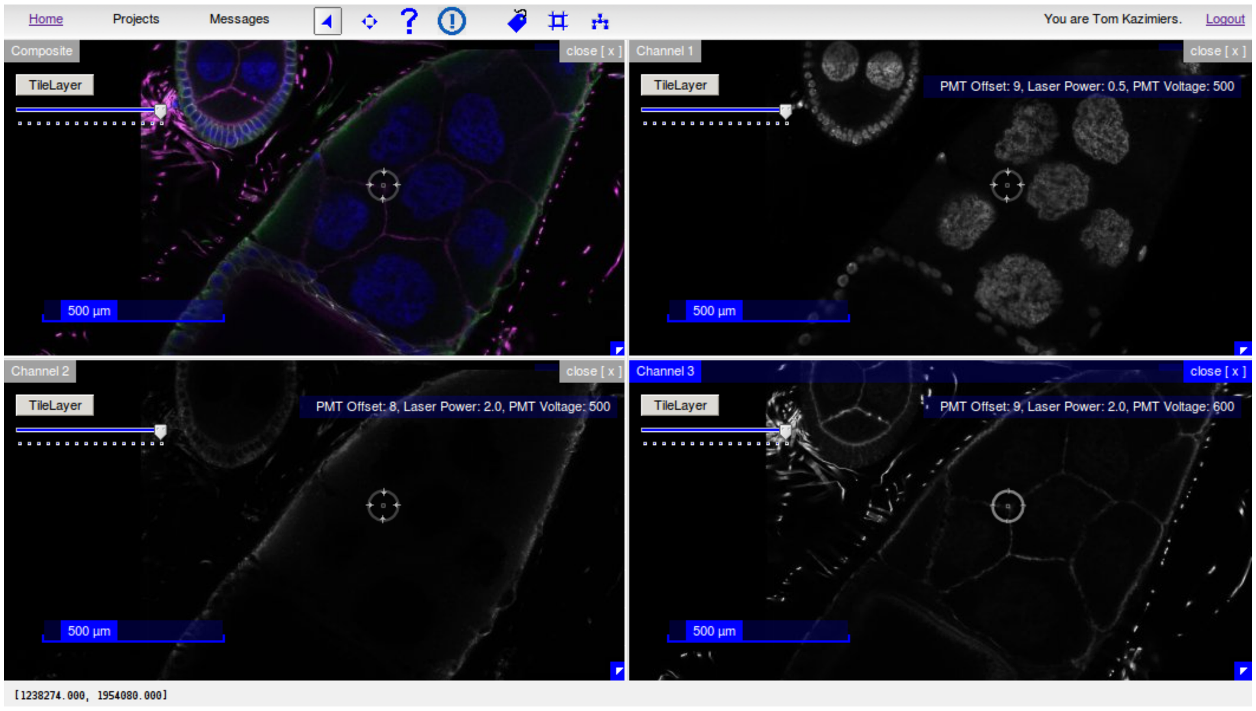
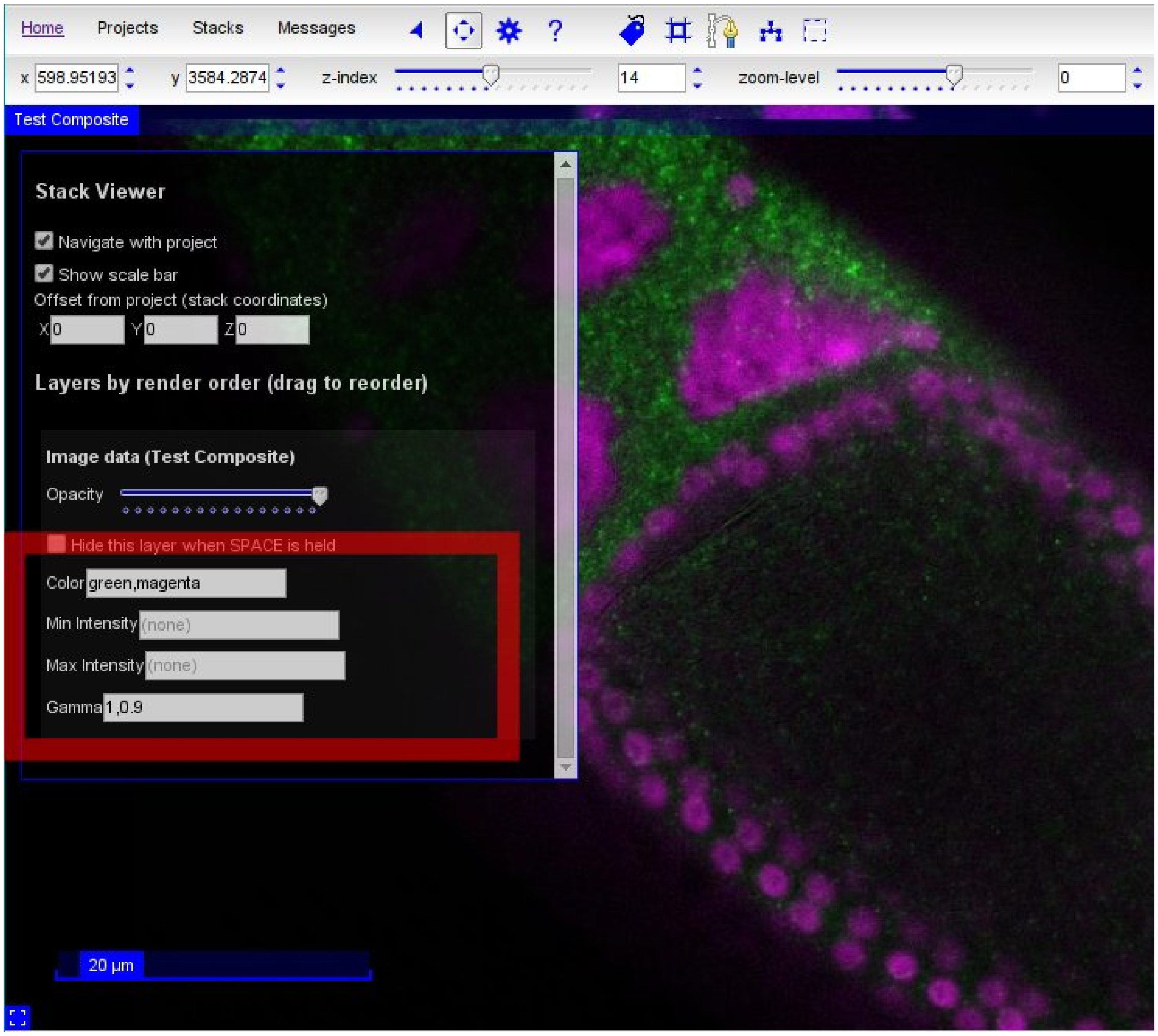

Home
Projects
Messages
You are Tom Kazimiers.
Logout
Composite
close [x]
TileLayer
500 µm
Channel 1
PMT Offset: 9, Laser Power: 0.5, PMT Voltage: 500
Channel 2
PMT Offset: 8, Laser Power: 2.0, PMT Voltage: 500
Channel 3
PMT Offset: 9, Laser Power: 2.0, PMT Voltage: 600
[1238274.000, 1954080.000]

Fig. 6: CATMAID's Selector tool: Mirrored mouse cursor in all open stacks
Some tile sources in CATMAID support dynamic settings

If there is more than one stack in the current project available, a menu, labeled "Stacks", is shown as well. All the available stacks of the project are listed there and can be selected for output. By default only the current stack is marked. For all the selected stacks the same region of interest is used. The resulting cropped stack will contain all the selected stacks and its dimensions are XYCZ ordered. That means, that the current slice is appended to the output for every stack before the next Z step is made.

To start the cropping job, one needs to click on the tickmark on the right. A confirmation message should appear. You will get a notification in the message view once the sub-stack has been cropped. By default, cropped sub-stacks will remain for two weeks on the server. This can be adjusted by configuring a periodic task that manages the clean-up. See the section about *creating periodic tasks* for an example how to do this.

The output image is a TIFF file with potentially multiple pages, with each one being an RGB image. The file contains some meta data: the EXIF tags `XResolution` and `YResolution` of every image contain the created image's X and Y resolution, respectively – in pixel per nanometer (if the image resolution in the data base is nano meter based). The `ImageDescription` tag contains ImageJ specific meta data. It passes information about the number of images, the channels and whether to use hyperstacks to ImageJ.



Stack Viewer

☑ Navigate with project

☑ Show scale bar

Offset from project (stack coordinates)

X 0 Y 0 Z 0

Layers by render order (drag to reorder)

Image data (Test Composite)

Opacity

☐ Hide this layer when SPACE is held

Color: green,magenta

Min Intensity: (none)

Max Intensity: (none)

Gamma: 1,0.9

_static/screenshots/catmaid-nblast-landmarks-pointcloud-k

Fig. 7: Landmark based transformation of neurons on the right side of the Drosophila larva brain to its left side and NBLAST matching for identification.

Ontology Tools

The documentation of the ontology tools can be found on a separate page.

1.1.4 Instructions for Tracing Neurons

Quick Start

There are step-by-step instructions below, but as a quick-reference, once you're in tracing mode, the following mouse operations and keyboard shortcuts should mostly be what you need:

- **click on a node**: make that node active
- **ctrl-click in space**: deselect the active node
- **ctrl-shift-click on a node**: delete that node
- **shift-click on a treenode**: join two skeletons (if there was an active treenode)
- **shift-click in space**: create a synapse (if there was an active treenode)
- **shift-click in space**: create a post-synaptic treenode (if there was an active synapse)
- **mouse wheel**: move up and down layers in the stack
- **+** and **-**: zoom in and out

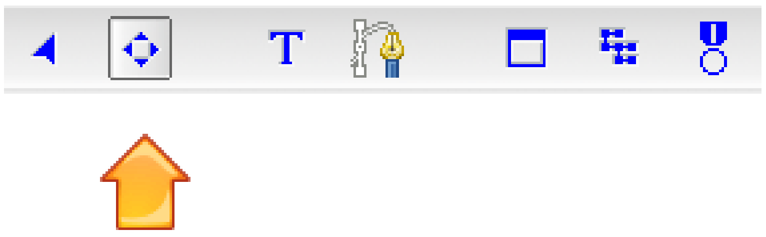
(If you're using Mac OS, then in every instance you need to use instead of `ctrl`.) For help on other keyboard shortcuts, click on the "?" button in the toolbar.

Basic Tracing

Before you can start tracing, you need to log in. You can do this in the boxes in the top right hand corner of the page.

Basic Concepts

There are two modes that you'll use in these instructions. The "navigate" mode is for moving fast around the stack, and is the default mode - it's shown in the toolbar with this icon:

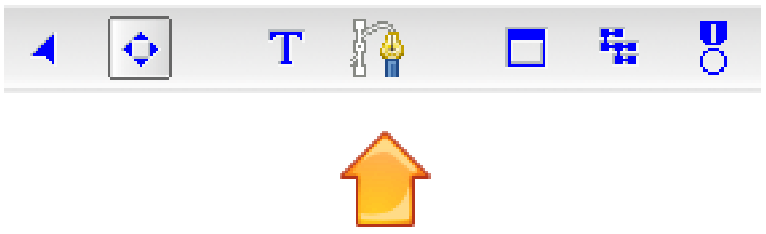


To move around in this mode you can:

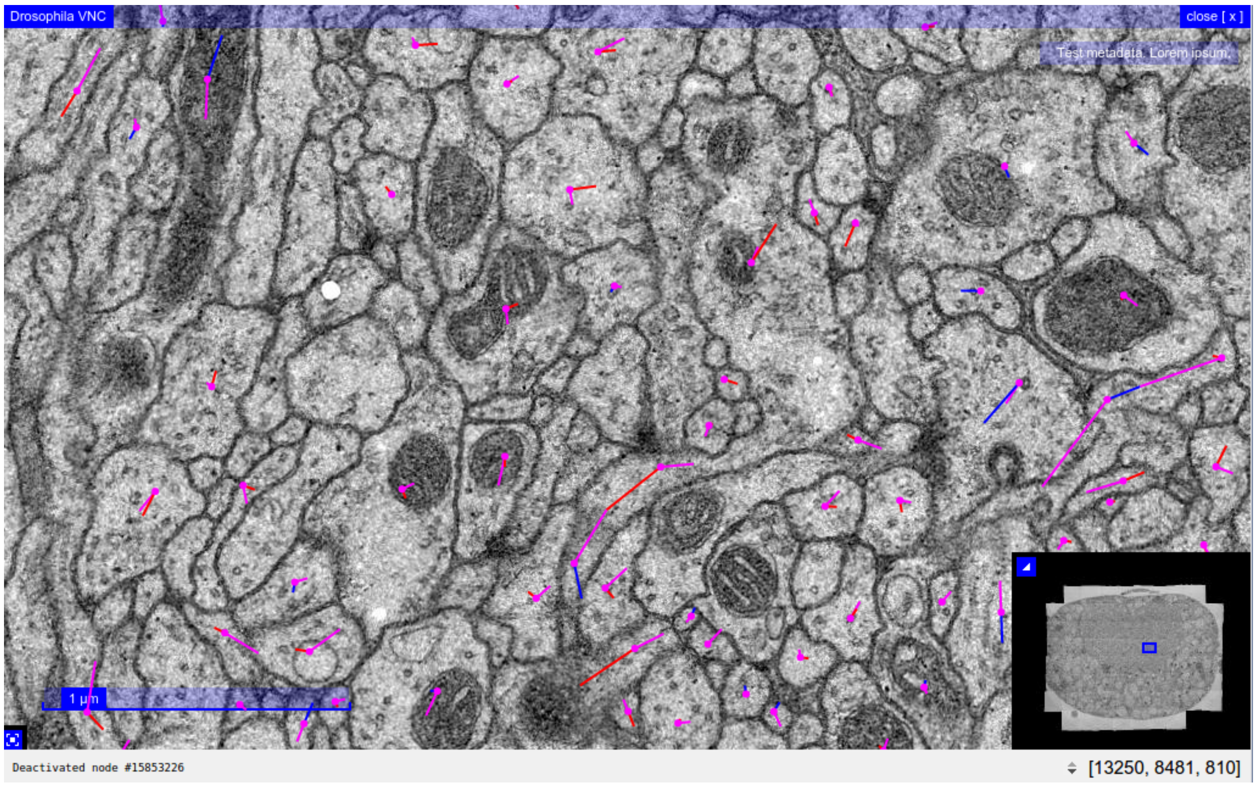
- Click and drag with the mouse to pan around the dataset, as in Google maps
- Use the mouse wheel to scroll through sections
- Press ',' and '.' to move one slice up or down the stack
- Press '<' and '>' to move ten slices up or down the stack
- Press '-' to zoom out
- Press '+' to zoom in
- Click on a section of the overview in the bottom right hand corner to jump to that area

In this mode you won't see any of the tracing annotations, however.

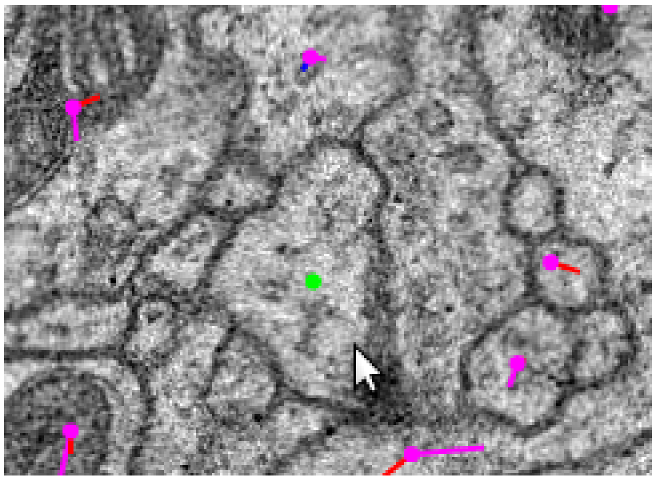
First, zoom in on an area that you'd like to start tracing in and click on the icon to change to tracing mode:



If there are any treenodes in view, you'll now see them as magenta dots:



To place the first node in a new neuron's skeleton, just left-click within a membrane that doesn't already have node in it. You should see a green node appear, like this:



The green node is always the **active node**.

The Active Node

The active node is important, since any new node you create by clicking will be linked to that node. To make a different node the active node, just left-click on it.

(If you want to create a completely new skeleton, there must be no active node - to deselect it, hold down `Control` (or on Mac OS) and left-click somewhere other than on a node, or press `D`.)

Try clicking elsewhere within this layer to create a line of nodes. Try changing the active node to one of the middle nodes, and continue clicking to create a branch.

You can move any node by clicking on it and dragging it around.

Deleting Nodes

To delete a node, hold down `Control` (or on Mac OS) and `Shift`, and left-click on the node, or activate the node and press `Delete`.

Navigating between Nodes

While you can navigate through nodes in a skeleton by changing slices and clicking on nodes, this is cumbersome. Instead, there are many key shortcuts to navigate through a skeleton based on its topology and other annotations.

The simplest navigation shortcuts are moving closer to or further from the root node. Pressing `[` will move to the parent of the active node, closer to the root node. Pressing `]` will move to the child of the active node, further from the root node. If the active node is a branch point, one of the children will be selected at random.

To discover more ways to navigate nodes, like finding leaf nodes without children, click the question mark icon in the toolbar to see a list of keyboard shortcuts.

Splitting Skeletons

To split a skeleton into two, select the node at which you want to split the skeleton, and then click on the "Split Skeleton" icon in the toolbar, which looks like this:

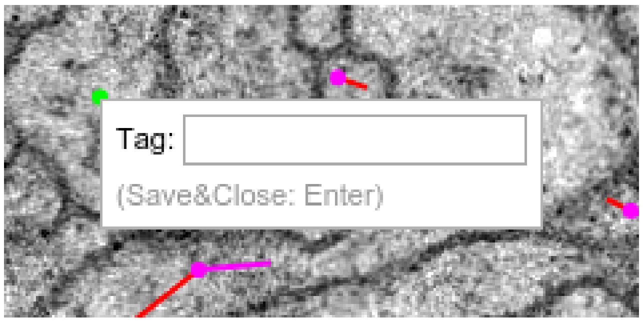
A dialog will appear with a 3D rendering of the skeleton, so you can visualize what each skeleton will look like once the split is made. If you are satisfied with the split, press the 'Ok' button. You will then end up with two skeletons. All parts of the skeleton from the root node up to and including the split point will still be in the original skeleton, but the nodes downstream from that point will be in a new skeleton and neuron.

Joining Skeletons

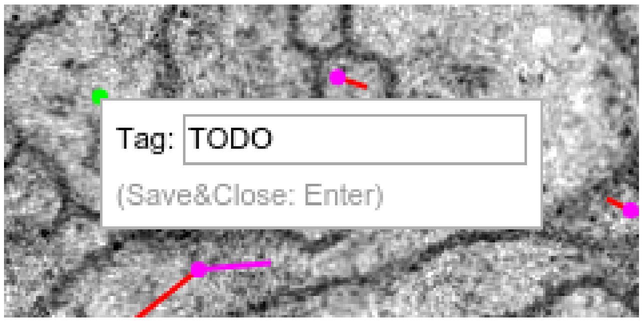
To join one skeleton to another so that they form a single skeleton, activate the node in one skeleton where you would like the join to occur, then hold down `Shift` and left-click on a node in the other skeleton. (You may not join any part of a skeleton to itself.)

Tagging Nodes

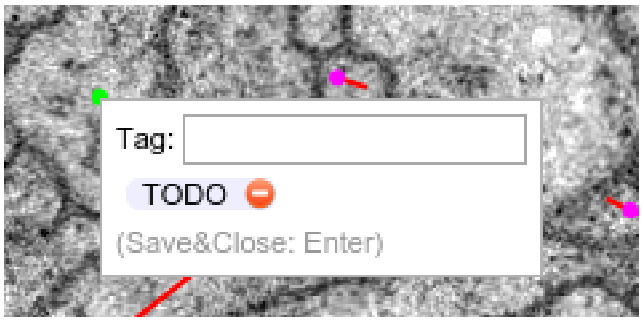
To add a tag to the active node, press `T`. You should see something like the following:



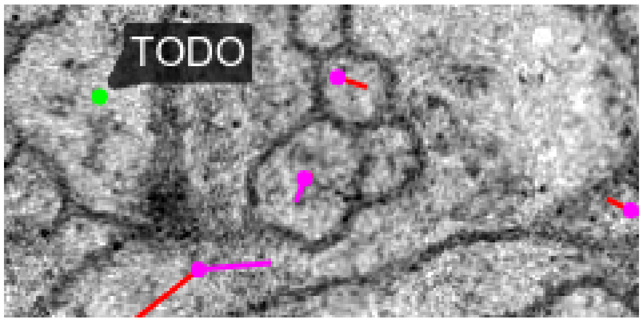
Then you can type a tag:



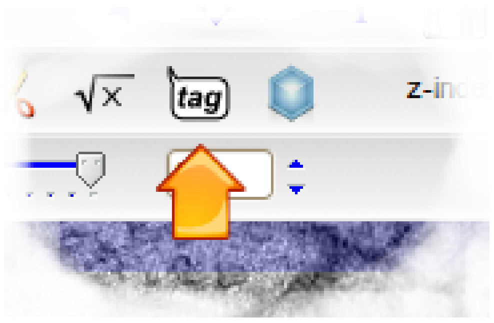
… and press `Enter`:



To actually save your tag to the database, press `Enter` again:



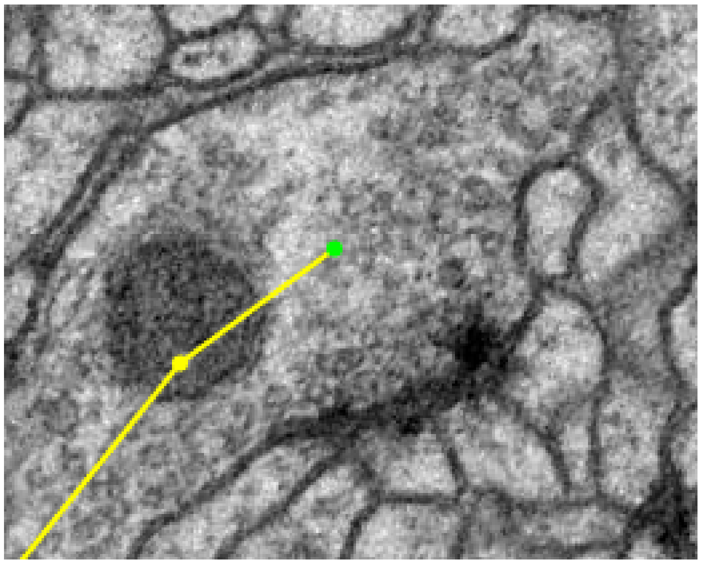
At any time, you can toggle the visibility of the tags with this button in the toolbar:



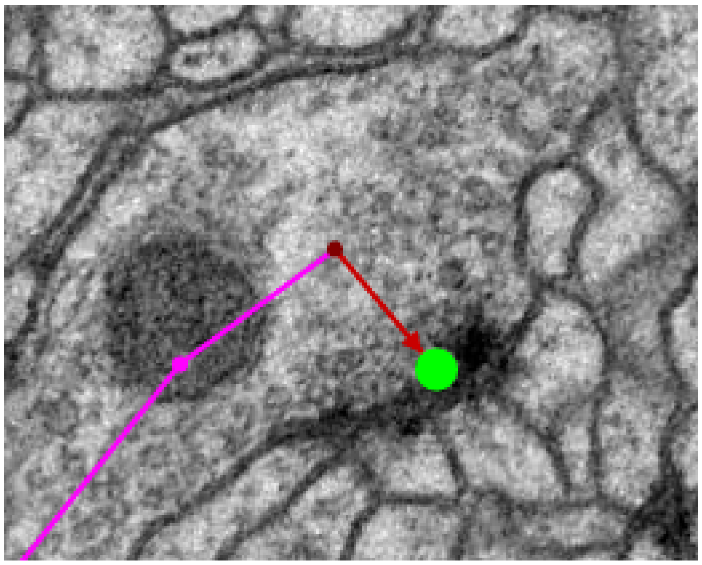
Creating Synapses

There are two possible ways of creating synapses. The recommended way unless you have many to annotate is the following:

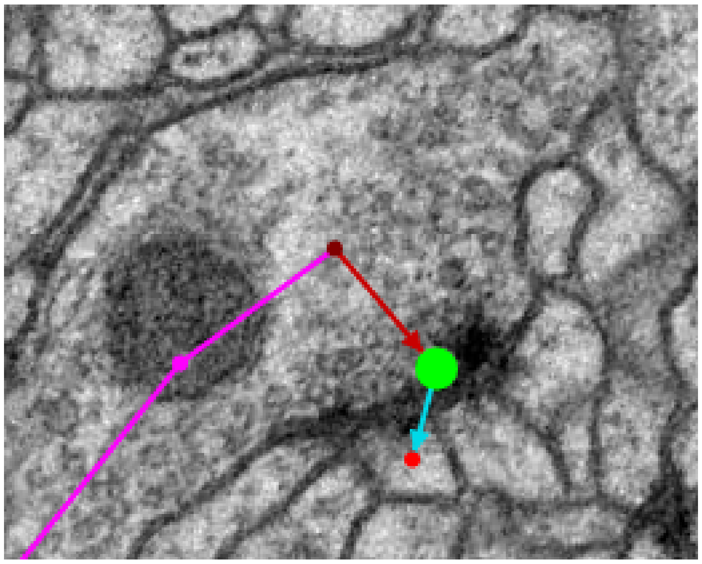
Suppose you have traced up to the active node in this screenshot (the green node) and want to annotate the nearby synapse:



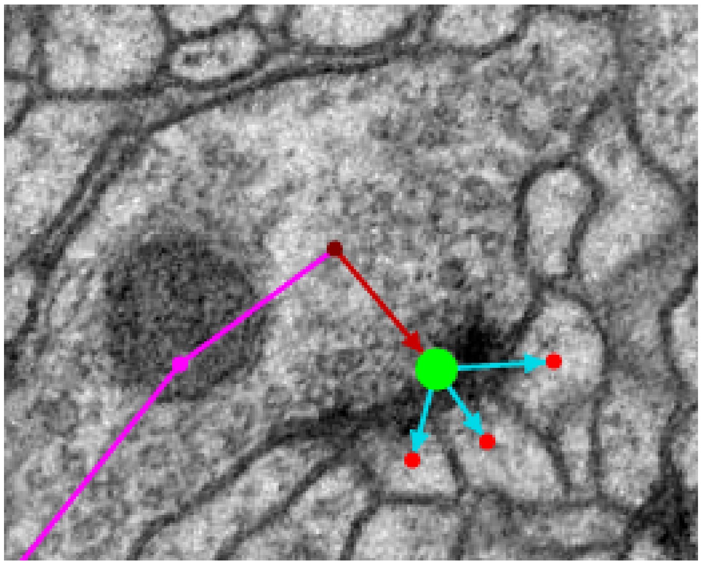
If you shift-click on the synaptic cleft (or where you think it should be) you'll see a larger green circle appear, which is a connector:



The red arrow leading to the large connector indicates that your original node is presynaptic to the connector. Then you can add a postsynaptic partner by shift-clicking in the center of a membrane that appears to be postsynaptic:



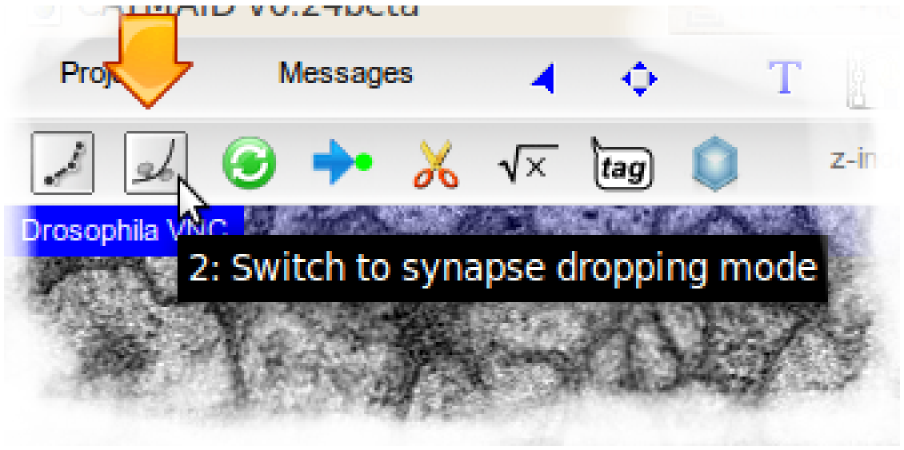
By repeating the last step you can mark multiple postsynaptic partners:



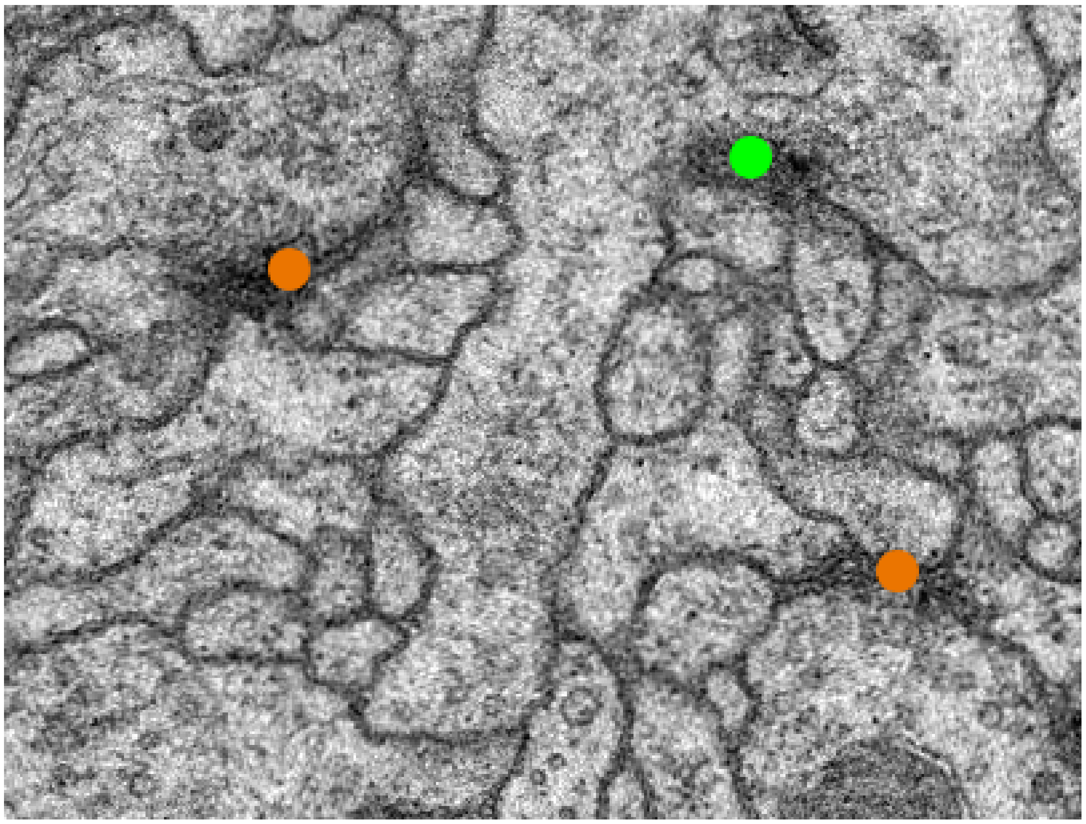
When you have finished, click on the origin node again to make it active and you can carry on tracing.

Dropping Multiple Synapses

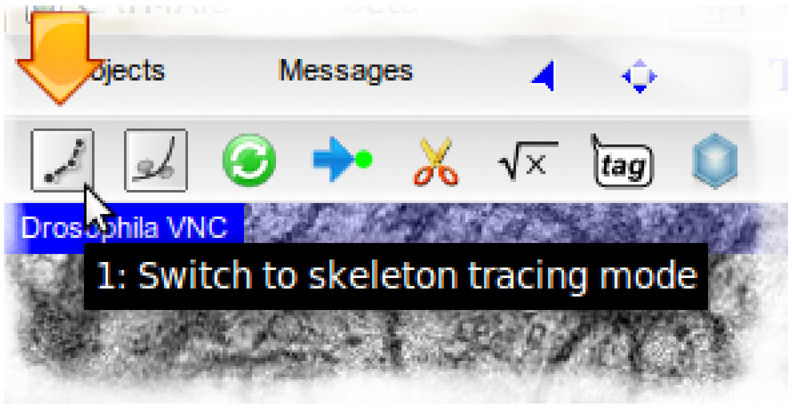
The alternative approach to creating synapse is to switch to synapse-dropping mode with this button in the toolbar:



Once you've selected that mode, any click in space on in the interface will create a synapse, so you can add many in one go, without needing to start by linking them from a treenode:

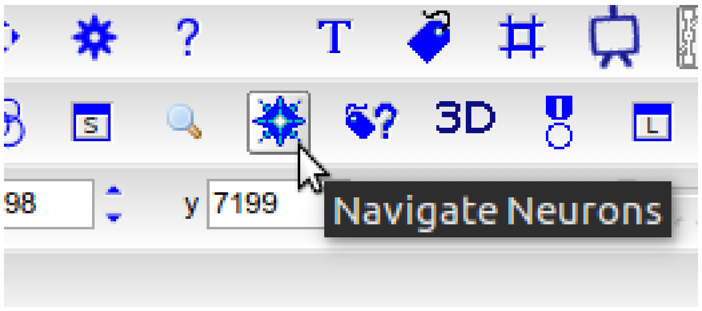


In order to carry on tracing, you need to switch back to the normal tracing mode with this button:

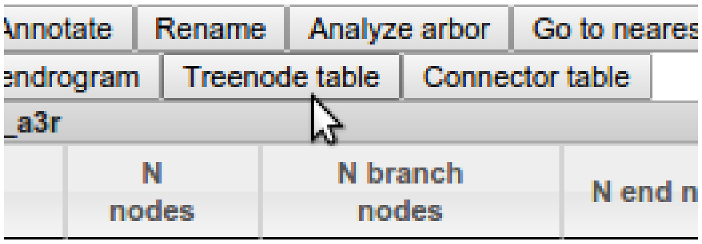


Exploring Skeletons and Neurons

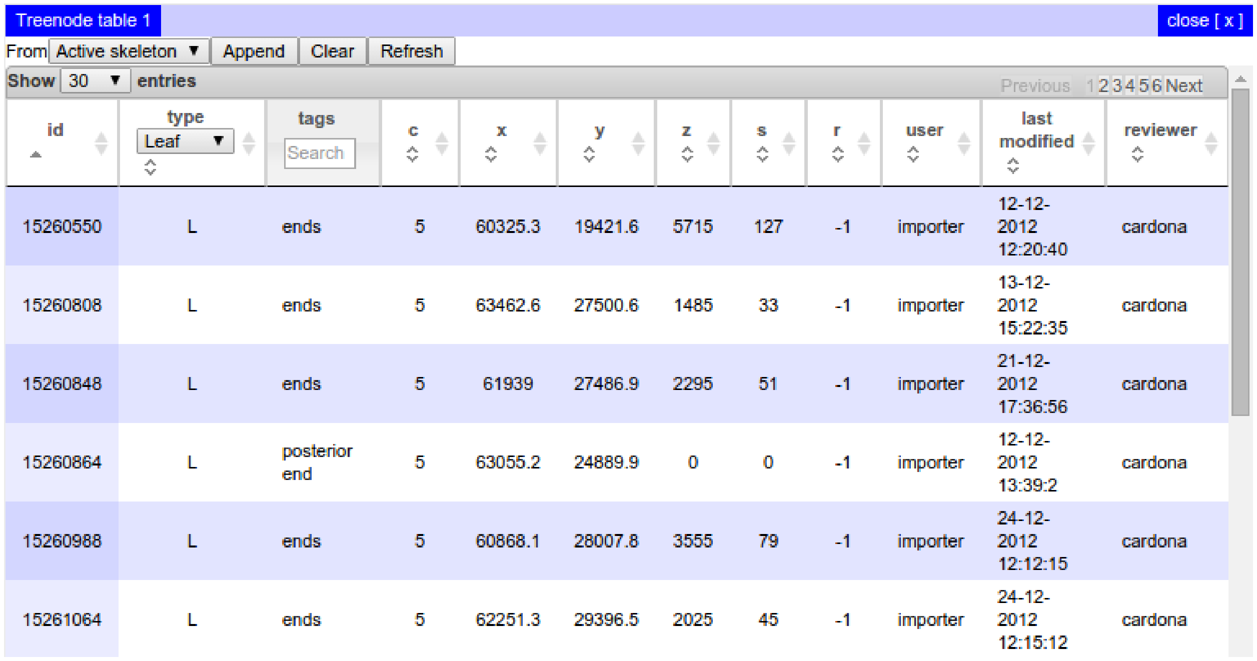
You may need to find a particular node of a skeleton, for example if it was tagged with TODO and you need to go back to it. The easiest way to locate particular nodes in a skeleton is to select a node in that skeleton and click on the Neuron Navigator button:



A window will open containing a new Neuron Navigator, which is a *widget* for exploring neurons and their annotations. CATMAID has many widgets to help catalog, explore, and analyze your data. In the section of the Neuron Navigator for the active neuron, click the Treenode Table button:

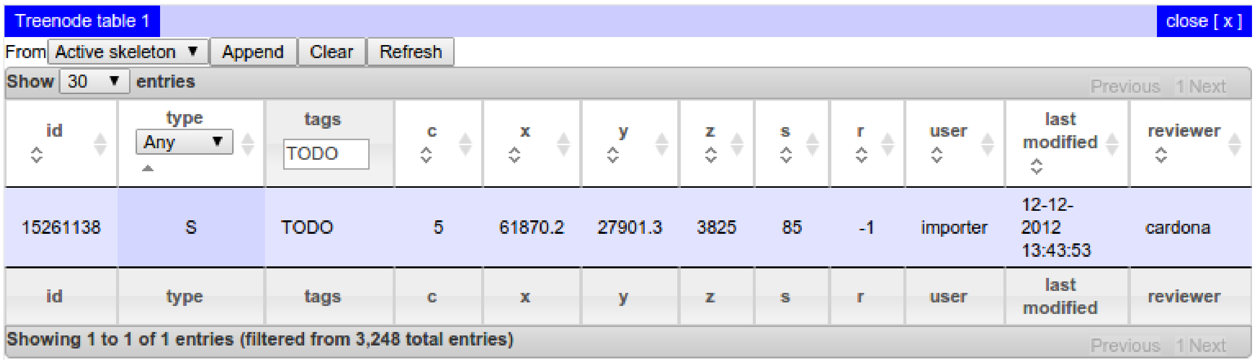


This will bring up a table of all the nodes in that skeleton:



id	type	tags	c	x	y	z	s	r	user	last modified	reviewer
15260550	L	ends	5	60325.3	19421.6	5715	127	-1	importer	12-12-2012 12:20:40	cardona
15260808	L	ends	5	63462.6	27500.6	1485	33	-1	importer	13-12-2012 15:22:35	cardona
15260848	L	ends	5	61939	27486.9	2295	51	-1	importer	21-12-2012 17:36:56	cardona
15260864	L	posterior end	5	63055.2	24889.9	0	0	-1	importer	12-12-2012 13:39:2	cardona
15260988	L	ends	5	60868.1	28007.8	3555	79	-1	importer	24-12-2012 12:12:15	cardona
15261064	L	ends	5	62251.3	29396.5	2025	45	-1	importer	24-12-2012 12:15:12	cardona

You can sort this table by clicking in the column headings. To go to a particular node, just double click on the row. To filter by node type, select an option from the dropdown in the table header. Find the TODO-tagged node you are looking for by typing "TODO" in the text input in the tags header and press `Enter`:



id	type (Any)	tags (TODO)	c	x	y	z	s	r	user	last modified	reviewer
15261138	S	TODO	5	61870.2	27901.3	3825	85	-1	importer	12-12-2012 13:43:53	cardona
id	type	tags	c	x	y	z	s	r	user	last modified	reviewer

Showing 1 to 1 of 1 entries (filtered from 3,248 total entries)

1.1.5 Training tracers

Like with most things, to get good at tracing neurons, some training is needed. The *introduction for tracing neurons* helps with first steps, but managing a whole team and being confident in the quality of the results, will require some supervision. There are of course many ways on how to go about this, and an ad-hoc approach is surely fine for smaller groups. Below we show one example of a systematic approach that was developed by Ruchi Parekh and trainers in the Connectome Annotation Team (CAT) at Janelia Research Campus, where it worked for groups of tracers with 20 and more people.

Systematic training and evaluation

This approach trains new users in using CATMAID and was developed with the FAFB dataset in mind along with manually reconstructed neurons that have been published for it. New trainees follow first the *introductory training* to get familiar with both the dataset and software, which is described in the next section. A small collection of training *videos* highlights some basic workflows in CATMAID.

Once the introductory training is completed, trainees will do the *training pipeline*, explained in the second part. There, tracers were asked to reconstruct particular neurons in increasing difficulty, highlighting different aspects of CATMAID and the dataset.

For any questions on the training protocol, please email Ruchi Parekh (`parekhr@janelia.hhmi.org`).

Introductory Training Protocol

This training protocol is meant to introduce manual neuron reconstruction with CATMAID, using the FAFB dataset. This dataset consists of the EM data of the brain of a female adult Drosophila melanogaster. at a resolution of 4 x 4 x 40 nm/vx. An example on how such a brain is dissected and prepared, Janelia's FlyLight team has some helpful videos. It might also be worthwile to check out the standard fly brain nomenclature. A publicly accessible CATMAID server can be found at spaces.itanna.org, which allows creating personal and shared training projects for the FAFBv14 dataset. Virtual Fly Brain also hosts a read-only version of the FAFB dataset. For those interested in a broader context on the training neurons and in further understanding of the mushroom body, have a look here and here.

When introducing new trainees to CATMAID, the following rough outline seemed like a useful path to take:

- Open FAFB project from front page
- Browse EM data
- Open Tracing Tool, select a few neurons by both clicking and G key
- Open 3D Viewer and add a neuron, demo space manipulation

- Activate node in 3D to see in EM
- Link from 3D Viewer to EM gray space (F10 function)
- Volume display of neuropils/whole brains (Volumes & Geometry tab in 3D)
- Provide a general intro to hot keys, help page (F1 or ? icon) and context help (upper right corner)

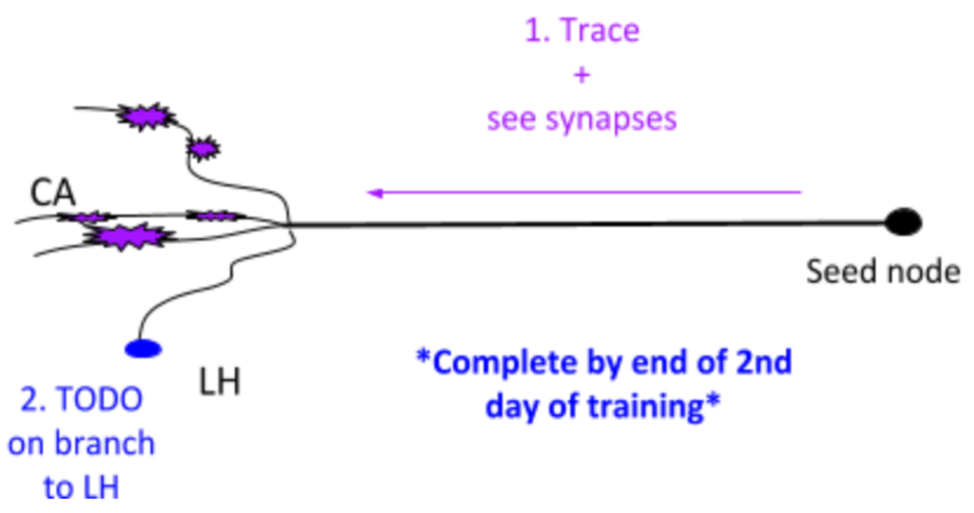
Trainers create seed nodes for new trainees, named after their initials (see video linked above). Assuming this has been done, the following steps are taken in order to provide an introduction that provides trainees with the skills required to actually step through the training pipeline.

Protocol

Each trainee will look at three intro neurons, before they can move to the training pipeline above: **1. PN -> 2. KC -> 3. PN -> 4. Pipeline**.

1. Select *Neuron Search* widget (toolbar icon: tag and question mark)
2. Enter neuron number (users initials) from trainer
3. Click on neuron seed number in window, append in 3D widget using 'Active skeleton' Start with Projection Neuron (PN)
4. Start at seed node (will be the active node) and place a 'TODO' tag - go to step 1 'Projection Neuron (PN)' section of protocol
5. Once finished with step 1 PN, trainee will be assigned a Kenyon Cell (KC) - Repeat above steps - Go to step 2 'Kenyon Cell (KC)' section of protocol
6. Once finished with step 2 KC, trainee will return to finish PN - Repeat above steps - Go to step 3 'Projection neuron (PN)' section of protocol
7. Once finished with step 3 PN, trainee will move into the pipeline

1. PN - Projection Neuron



Besides 1. and 2. in the image above, the goals are also

3. Review own work

4. Review with trainer

At end of PN - you should know

Neuroanatomy

- Recognize a branching point
- Understand the 3D of branching structures
- Recognize a synapse - Understand 4 major indicators
- Recognize differences in morphology of a bouton and fingers of a claw
- Preliminary understanding of 'stringiness' (it's *not* a characteristic of the neuron itself- it's a term we've given to describe the morphological 'look' of the neuron)

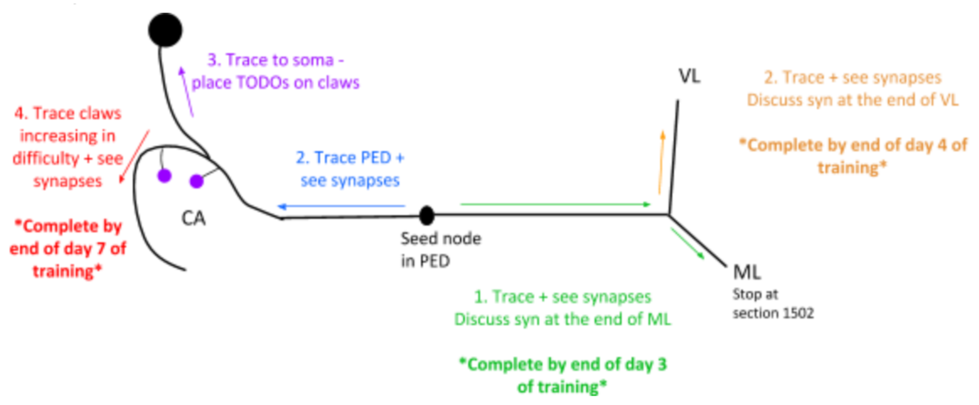
Tracing

- Place nodes in center
- Create connecting skeletons
- Create "clean" skeleton (i.e. a biologically realistic neuronal skeleton)
- Check your neuron going "forward" and "backward"
- Know how to go from an active node without forming parallel lines
- To tackle small jumps and crappy section

Software

- +/- to zoom in and out of EM stack
- Searching for neuron
- 3D viewer - zoom, rotate, activate a node, pre, post and meta tags, taking off bounding box and floor, coloring neuron etc
- F10 function
- Tags - L, K, N
- Untag - shift + whatever tag you need to untag (ex: shift+L gets rid of TODO)
- "keyboard shortcut help"

2. KC - Kenyon Cell


3. Trace to soma - place TODOs on claws
VL
2. Trace + see synapses Discuss syn at the end of VL
Complete by end of day 4 of training
4. Trace claws increasing in difficulty + see synapses
Complete by end of day 7 of training
2. Trace PED + see synapses
CA
Seed node in PED
ML
Stop at section 1502
1. Trace + see synapses Discuss syn at the end of ML
Complete by end of day 3 of training

Besides 1. to 4. in the image above, the goal is also

5. Review own work
6. Review with Trainer

At end of KC - you should know

All "you should know"s from PN + the following:

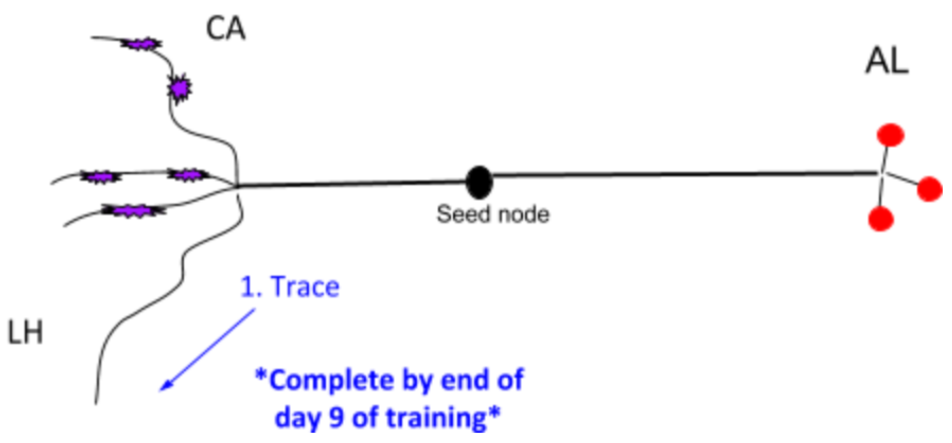
Neuroanatomy

- See further examples of synapses
  - See and recognize structural differences in synapses in KC and PN (especially in the calyx vs lobes)
  - Discuss how some indicators are more prevalent in different brain regions (vesicle clumping in PN boutons vs KC lobes)
  - Discuss different types of syn (e.g. en passant and en face)
- Differences in pedunculus, lobe, and calyx EM anatomy and 3D morphology (refer to Illustration/explanation provided by trainer)

Tracing

- Continue to understand how each node helps shape the 3D morphology of the neuron
- Know how to correctly tag soma
- Know how to properly use uncertain tags

3. PN - Projection Neuron



Besides 1. in the image above, the goal is also

2. Review own work
3. Review 1-1 with Trainer

At end of PN - you should know

All "you should know"s from PN + KC + the following:

Neuroanatomy

- Synapses in detail (bouton to claw)

- Invagination
- Cistern/ER
- Vacuole
- Vesicles
- Synaptic cleft
- Mitochondria
- Microtubules - follow for continuations
- Understand the above neurobiology in general context with the understanding that each may look different in different brain regions

Tracing

- Solid understanding of difference between PN trunk/KC pedunculus (tract) synapse and bouton/calyx (dendritic) synapse

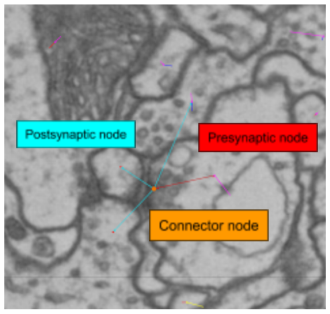
Software

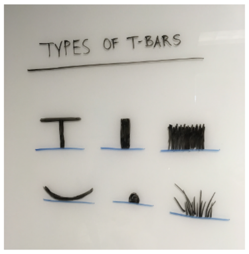
- Good sense of the hot keys and how to navigate them without a trainer

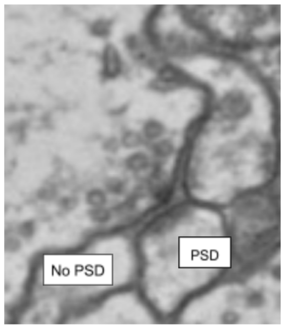
Synapse information

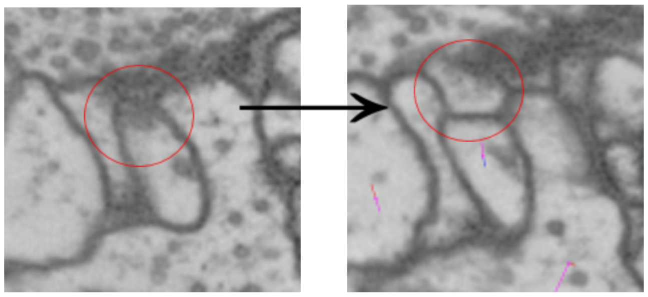
Identify a synapse:

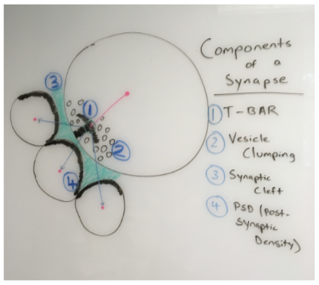
- T-bars - presynaptic protein where vesicles bind and neurotransmitter is released
- Vesicles - small neurotransmitter filled spheres
  - Important to identify clumping around T-bars
  - Scroll between sections to determine how long T-bars and vesicles exist
- Synaptic cleft - extracellular space where neurotransmitter diffuses to postsynaptic partners.
  - The cleft is located between the T-bar and the postsynaptic cells
- Postsynaptic density (PSD) - postsynaptic receptor proteins that signify uptake of neurotransmitter











Note the movement across the first two images of the second row.

Neurons used in training

- 4PN and 4KC neurons used in training. (Neurons used in training in 2019)
- Uniglomerular mALT VC3l adPN 23135 BH
- Uniglomerular mALT VM5d adPN 49866 JMR
- Uniglomerular mALT VC1 lPN 22133 BH
- Uniglomerular mALT VC3l adPN 46801 JMR
- KCaBs 515202 NM
- KCaBc 514395 NM
- KCaBs 31268 IJA
- KCaBs 7675 EW AJ

Training materials

CAT Trainer, Henrique Ludwig, has created short tutorials explaining some common tasks within the FAFB training environment, available as a playlist or here individually:

Training pipeline

We found that going through four levels of difficulty over the course of multiple weeks gave tracers the skills they needed. We started with asking trainees to look at 22 neurons, eventually settled on 18 though, where the easiest level now includes only a single neuron. A shared spreadsheet was used to keep track of the progress of all trainees. This was also used as a means of communication between trainees and trainers. An example of such a spreadsheet is shown as table below can be downloaded as Open Document Spreadsheet or Excel format.

Table 1: CAT Training Pipeline

Neuron name	CAT L1-5 "initials"	CAT L2-1	CAT L2-2	CAT L2-3	CAT L2-4	CAT L2-5	CAT L3-1	CAT L3-2	CAT L3-3	CAT L3-4	CAT L3-5	CAT L4-1A	CAT L4-1B	CAT L4-2A	CAT L4-2B	CAT L4-3A ***	CAT L4-3B ***	CAT L4-3C ***	*** If soma and soma tract found, then stop tracing
Brain region	Commissure	LO	LH	LP	MB	PB-EB-gall	Connectives	Nodules	PB	EB	GNG	–	MB	PB-EB-NO	PB-EB-NO	LP	LP	LP	
Cell type	Giant fiber input interneuron	LC4 neuron	Projection neuron	VSN	KC - CA	PB-EB-gall	CNN	PB-EB-NO	PB-EB-NO	PB-EB-NO	VCN	Giant fiber branch	PN	PB-EB-NO	PB-EB-NO	LPLC	LPLC	HSE	
What to trace	Main trunk skeleton only	Trace arborization only (no synapses)	Trace arborization only (no synapses)	Trunk + soma tract + soma + synapses	Skeleton + synapses + soma tract + soma	Skeleton + synapses	Skeleton (no synapses)	Skeleton + synapses	Skeleton + soma tract + soma + synapses	Skeleton + synapses (tracing reviewed at 1 week mark)	Skeleton (no synapses) **	Skeleton + synapses	Skeleton + synapses	Skeleton (no synapses)	Skeleton (no synapses)	Skeleton (no synapses) + soma + soma tract	Skeleton (no synapses) + soma + soma tract	Skeleton (no synapses) + soma + soma tract	
Where to trace (Z)	Start on 3470 towards 3469. Stop on 3256.	Start on 4771 towards 4772	Start on 4622 towards 4621	Start on 5368 towards 5369 and stop at end node tagged with "DO NOT PROCEED". There will be a small soma tract be-	Start on 4991 towards 4992	Start on 1635 towards 1636	Start on 5397 towards 5398	Start on 2869 towards 2868	Start on 4423 towards 4424	Start on 2889 towards 2888	Start on 5588 seed node towards 5055 end node (only stop at 5055 if there is an end node with tag "DO NOT PROCEED"	Start on 3795 towards 3796	Start on 4874 towards 4875	Start on 3569 seed node towards 3570. Stop on "DO NOT PROCEED" end node 3698	Start on 2364 seed node towards 2365. Stop on "DO NOT PROCEED" end node 2559	Start on 4291 towards 4292. Stop on "DO NOT PROCEED" end node 4925. Find and trace soma tract between start and end	Start on 5638 towards 5637. Stop on 4682. Find and trace soma tract between start and end sections	Start on 6271 towards 6272. Stop on tagged as "DO NOT PROCEED" end node 5830 (only stop on section 5830 if this	Review

Generally, the training pipeline follows following steps:

1. Trainee - Select “your” neuron name from above CAT LX-X list and search in CATMAID
2. Trainee - Trace neuron
3. Trainee - On completion - update progress sheet
4. Trainee - Pick next neuron from list
5. Trainer - Review completed neuron
6. Trainer - Update progress sheet

Example neuron names in the training dataset are:

- CAT L1-1 SC
- CAT L2-4 LR
- CAT L1-3 AW
- CAT L5-1 TP
- CAT L4-4 CP
- CAT L3-1 SA

The status of each training neuron for each trainee follows the following color code:

Table 2: Status colors

Trainee working on tracing	Trainee completed tracing	Trainer completed review	Tracing on hold

While tracing, trainees where advised to follow the following general guidelines regarding the use of node tags and edge confidence values:

- use uncertain continuation/uncertain ends as necessary, however you will be evaluated based on how you use it
- L2-2 onwards - place confidence intervals as described below for uncertain continuations/ends
- L3-5 - If needed, clarify instructions with trainer

With confidence values assigned through the numeric keys 1-5 should be used like the following:

Table 3: Confidence intervals

	5	4	3	2	1	
I am	100%	75%	50%	25%	0%	sure it continues/ends
	don’t add # 5 to node	add # 4 to node	add # 3 to node	add # 2 to node	add # 1 to node	
	continue tracing	continue tracing if uncertain continuation or stop if uncertain end	stop tracing	stop tracing	stop tracing	

While also using the following rules on deciding when to mark an uncertain continuation with a tag versus a low confidence value:

And in addition:

- **Trainees should ask questions if confused about where to stop**
- **USE uncertain tags when confused about anything**

This document allows trainees to move through a defined and comparable process to learn about CATMAID and tracing, as well as the dataset itself.

A Trainer would guide new trainees according to the above plan and track their progress using the different color codes above in the trainee's "swim lane".

1.1.6 Annotations

One central way to organize neurons and other spatial data structures in CATMAID is to use text annotations. They can be added to neurons and to annotations themselves, allowing hierarchies and nomenclatures of annotations.

There are a few special annotations that CATMAID will interpret in a particular way. The most commonly one is likely the `locked` annotation. If present on a neuron, no part of the skeleton can be changed anymore. The section below will introduce more special annotations:

Annotations can be added in many places within CATMAID. Often times widgets contain an "Annotate" button. The active skeleton can be annotated using the `F3` key. This dialog shows also the present annotations and allows to add meta-annotations, which are simply annotations on annotations. In order to do that, click on the "Click here to also add a meta annotation" text link. The Neuron Navigator widget is also a useful simple widget to see the annotations of the current neuron.

Special Annotations

The annotations below have meaning in some context within CATMAID. Whether they need to be applied to neurons or annotations is noted in the description.

`locked` If present on neuron, no change in morphology is allowed anymore.

`stable` If present on neuron, during a join, the back-end will enforce that it is the winner in the join, i.e. its ID remains. If both join partners have this annotation, the join is canceled. This annotation can be configured through the Settings Widget for users with `can_administer` permissions on a project. Only the project/instance level config is respected by the back-end.

`export: annotations` If present on a "publication annotation", it is interpreted by the exporter to export annotations for the respective set of neurons, grouped by the publication annotation.

`export: no-annotations` If present on a "publication annotation", it is interpreted by the exporter to not export annotations for the respective set of neurons, grouped by the publication annotation.

`export: tags` If present on a "publication annotation", it is interpreted by the exporter to export tags for the respective set of neurons, grouped by the publication annotation.

`export: no-tags` If present on a "publication annotation", it is interpreted by the exporter to not export tags for the respective set of neurons, grouped by the publication annotation.

`export: intra-connectors-only` If present on a "publication annotation", it is interpreted by the exporter to export only the connectors linking between the members of the respective set of neurons, grouped by the publication annotation.

`export: intra-connectors-and-placeholders` If present on a "publication annotation", it is interpreted by the exporter to export only the connectors linking between the members of the respective set of neurons, grouped by the publication annotation. Additionally placeholder nodes with random IDs and data will be exported to reflect additional connections.

`export: intra-connectors-and-original-placeholders` If present on a "publication annotation", it is interpreted by the exporter to export only the connectors linking between the members of the respective set of neurons, grouped by the publication annotation. Additionally placeholder nodes with the original IDs and data will be exported to reflect additional connections.

export: no-connectors If present on a "publication annotation", it is interpreted by the exporter not to export connectors linking between the members of the set of neurons grouped by the publication annotation.

1.1.7 Widgets

Graph Widget

Pair Statistics

Review Widget

Selection Table

1.1.8 Using the CATMAID API

You may want to query data or perform analyses in ways that are not possible through the CATMAID client. Scripting the CATMAID client is sufficient for some cases, but others require direct access to annotation data, especially for use in a programming environment like Python or R. For these cases, the same HTTP API that the CATMAID client uses is exposed to other clients.

API Documentation

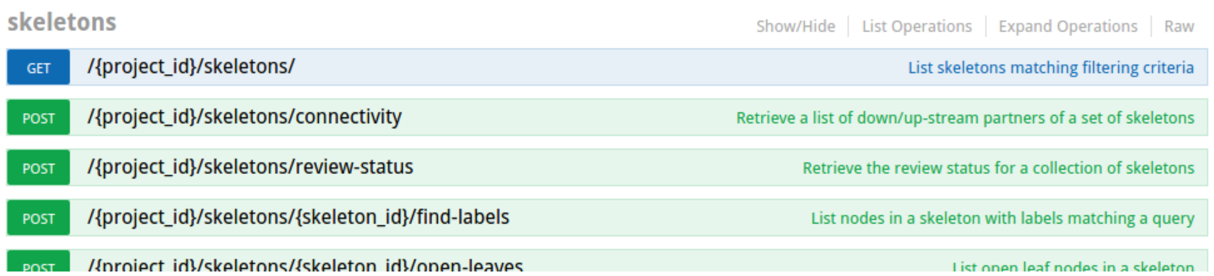
Documentation for endpoints exposed by the HTTP API is available *here online* and from the CATMAID server itself via the `/apis/` page:

```
http://localhost:8000/apis/
```

... or, for custom configurations:

```
http://<catmaid_servername>/<catmaid_subdirectory>/apis/
```

The API documentation list groups of related endpoints, like "skeletons", and specific HTTP methods ("GET", "POST", etc.) and URIs for calling each endpoint.


skeletons

Show/Hide | List Operations | Expand Operations | Raw

GET /{project_id}/skeletons/ List skeletons matching filtering criteria

POST /{project_id}/skeletons/connectivity Retrieve a list of down/up-stream partners of a set of skeletons

POST /{project_id}/skeletons/review-status Retrieve the review status for a collection of skeletons

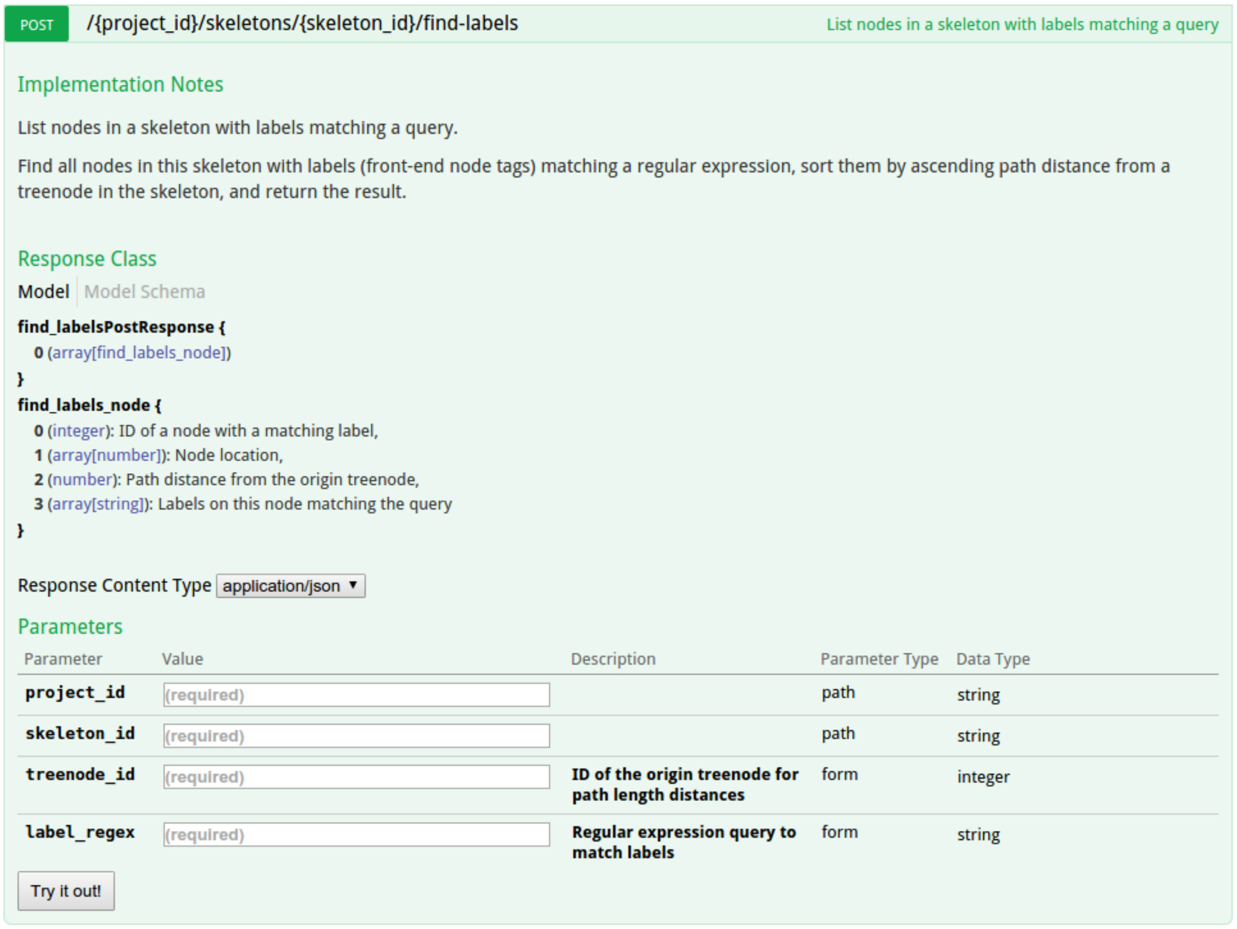
POST /{project_id}/skeletons/{skeleton_id}/find-labels List nodes in a skeleton with labels matching a query

POST /{project_id}/skeletons/{skeleton_id}/open-leaves List open leaf nodes in a skeleton

Clicking on one of these endpoints opens detailed documentation including a description of what the endpoint does, what parameters is expects and accepts, and how the response is structured. The form at the bottom of the endpoint documentation allows it to be invoked directly using values provided in the form fields.

> **Warning:** Remember that API documentation forms operate on actual data on the CATMAID server. Be careful not to unintentionally destroy or modify data.

Changes to the API are documented in the API changelog.



POST /{project_id}/skeletons/{skeleton_id}/find-labels List nodes in a skeleton with labels matching a query

Implementation Notes

List nodes in a skeleton with labels matching a query.

Find all nodes in this skeleton with labels (front-end node tags) matching a regular expression, sort them by ascending path distance from a treenode in the skeleton, and return the result.

Response Class

Model Model Schema

```
find_labelsPostResponse {
  0 (array[find_labels_node])
}
find_labels_node {
  0 (integer): ID of a node with a matching label,
  1 (array[number]): Node location,
  2 (number): Path distance from the origin treenode,
  3 (array[string]): Labels on this node matching the query
}
```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
project_id	(required)		path	string
skeleton_id	(required)		path	string
treenode_id	(required)	ID of the origin treenode for path length distances	form	integer
label_regex	(required)	Regular expression query to match labels	form	string

Try it out!

Undocumented API

Only a subset of the HTTP endpoints exposed by CATMAID are documented. While the documentation attempts to cover endpoints most likely to be useful for external analysis, undocumented endpoints can still be accessed if they are useful.

Undocumented endpoints can be discovered by looking through urls.py in the CATMAID source. This file routes URI patterns to Python functions, e.g.:

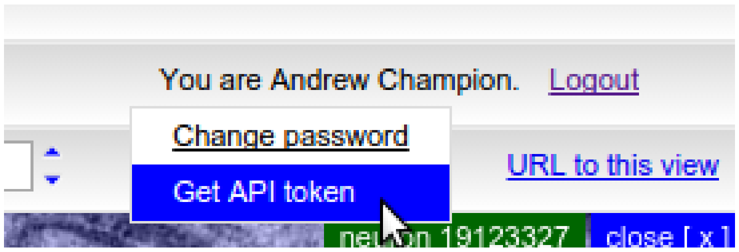
```
urlpatterns += patterns('catmaid.control.treenode',
    #...
    (r'^(?P<project_id>\d+)/treenode/delete$', 'delete_treenode'),
)
```

routes a URI like `/{project_id}/treenode/delete` to Python function `delete_treenode`, which is in the file `django/applications/catmaid/control/treenode.py`. Using an undocumented endpoint requires inspecting the Python function to see what parameters it expects.

Note that the undocumented API is considered volatile and changes to it are not included in the API changelog.

API Token

The CATMAID API authorizes requests using an API token tied to user account instead of a username and password. To obtain your API token, open the CATMAID client in your browser, hover your cursor over your name (next to the "Logout" link), and click "Get API token".



As a security measure, you will be prompted to re-enter your password, then shown your token string.

To use the API token, set the HTTP `X-Authorization` header on all of your API requests to be "Token", a space, and the token string, e.g.:

```
X-Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b
```

CATMAID uses `X-Authorization` rather than `Authorization` so that it does not conflict with server-level HTTP authorization.

Warning: Requests using your token can do anything your account can do, so **do not distribute your token or check it into source control.**

Example API Use

Command line

The perhaps easiest way to talk to the CATMAID API is to use the `curl` command line tool:

```
curl --header 'X-Authorization: Token <auth-token>' -X GET '<catmaid-url>'
```

If the CATMAID instance requires basic HTTP authentication, you can add the `-u <user>:<pass>` parameter to the `curl` command. For intance, to get all currently available node tags in a project with ID `1` from the CATMAID instance hosted at `https://example.com/catmaid/` using basic HTTP authentication with user "myuser" and password "mypass", you can do the following:

```
curl --header 'X-Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b' \
    -u myuser:mypass -X GET 'https://example.com/catmaid/1/labels'
```

Python

The lowest-friction way to access the CATMAID API using python is through the `catpy` package, which handles all of the boilerplate, includes a number of tools for working with CATMAID data, and is just a `pip install catpy` away. For more information, see **`the catpy docs<https://catpy.readthedocs.io>`_**.

If for some reason you are unable to use `catpy`, below is a minimal example of accessing the API of a CATMAID server running on `localhost` using the Requests Python package:

```
import requests
from requests.auth import HTTPBasicAuth

class CatmaidApiTokenAuth(HTTPBasicAuth):
    """Attaches HTTP X-Authorization Token headers to the given Request.
    Optionally, Basic HTTP Authentication can be used in parallel.
    """
    def __init__(self, token, username=None, password=None):
        super(CatmaidApiTokenAuth, self).__init__(username, password)
        self.token = token

    def __call__(self, r):
        r.headers['X-Authorization'] = 'Token {}'.format(self.token)
        if self.username and self.password:
            super(CatmaidApiTokenAuth, self).__call__(r)
        return r

# Replace these fake values with your own.
token = "9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b"
project_id = 1
object_ids = [42]

label_response = requests.get(
        'https://localhost/{}/labels'.format(project_id),
        auth=CatmaidApiTokenAuth(token))

annotation_response = requests.post(
        'https://localhost/{}/annotations/query'.format(project_id),
        auth=CatmaidApiTokenAuth(token),
        data={'object_ids': object_ids})
```

Other API Clients

A partial listing of libraries or programs that consume the CATMAID HTTP API:

catpy Official python 2.7 and 3.4+ client by members of the Cardona lab

RCATMAID R bindings for the CATMAID API by Gregory Jefferis.

CATMAID-to-Blender A Blender-Plugin to pull data from CATMAID by Philipp Schlegel

1.1.9 API Reference

1.1.10 Volumes

Besides skeletons CATMAID can also generate and display 3D meshes, which it calls volumes. At the moment the support is basic, but can be useful already. They are arbitrary triangle meshes or polyhedra, stored as PostGIS data for fast access. Volumes are associated to projects and can currently not be shared between them without data duplication.

The 3D Viewer's "View settings" tab allows to mark known volumes as visible. Checkboxes for individual volumes are accessible from a drop down menu. If a volume becomes visible, it takes the currently selected opacity and color from the color selector close to the drop down menu. If the "Faces" option is selected, filled triangles are displayed rather than a wireframe.

And since CATMAID's 3D viewer supports a textured Z plane, the volume's intersection with an image stack can be visually inspected as well. However, there is not yet a way to show volumes as a layer in a stack viewer.

Volumes are also accessible from the Volume Manager widget, which can be opened with the help of the 3D cube icon in the Tracing Tool's button panel. It shows all available volumes and some meta information on them. Double clicking table entries allows users to edit basic information, like its name. Additionally, the volume's bounding box is displayed on top of the image data of the active stack viewer. The mesh itself is not editable from CATMAID's web-client at this point.

The volume widget also provides means to generate new volumes based on neurons, synapses or simply as a bounding box. The sub-sections below provide some detail on this.

Besides display, CATMAID can currently use volumes only to warn if nodes are created outside of them. This is useful, for instance, if one wants to constrain neuron reconstruction to a given compartment. To select a volume for which warnings should be generated open the settings widget and open the section called "Warnings". If a volume is selected from the drop down menu, warnings will be generated for the current session.

Volume generation

In CATMAID, volumes can either be added through the volume manager or directly through the HTTP API.

Volume manager

Pressing the "Add new volume" button brings up a user interface to generate and add new volumes. The first thing to do is to select the new volume's type, available are bounding box, convex hull and alpha shape. The first is simply an axis aligned box, previewed in the tracing layer. After configuration, "Save" can be pressed to store the volume on the server. Bounding boxes are currently the only volumes that can be edited after they have been created.

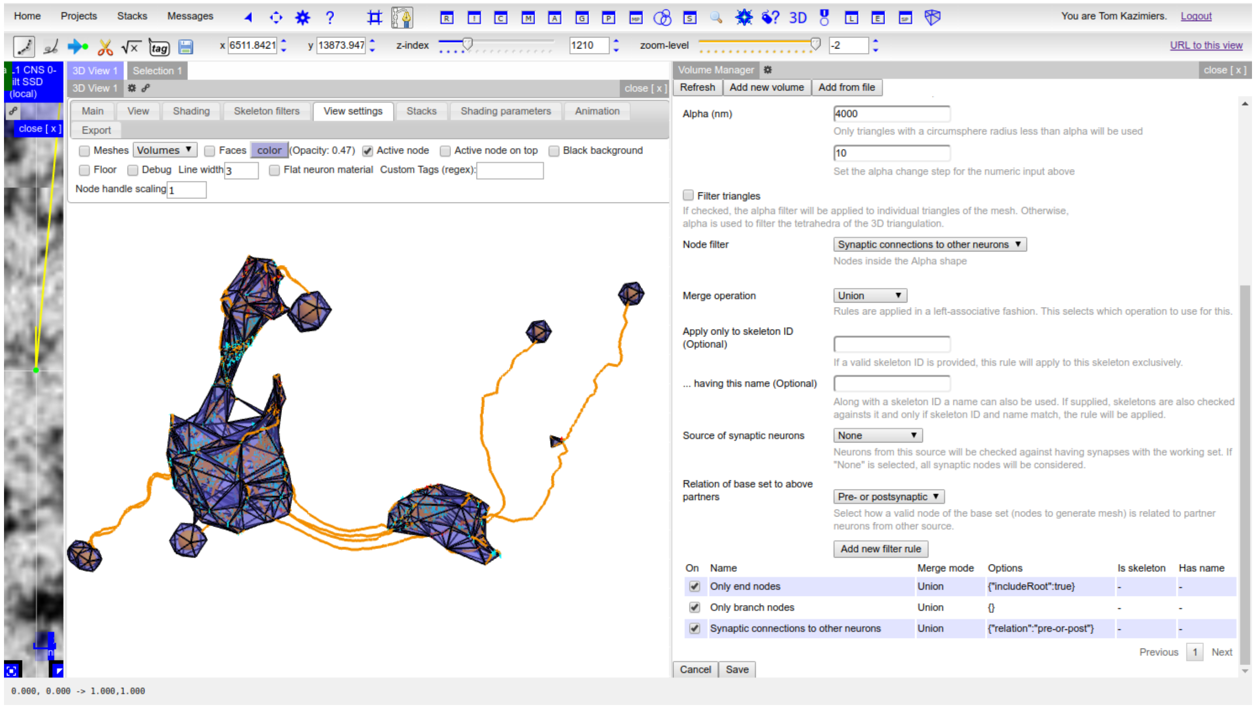
The remaining two volume types require a point set to operate on and can be previewed in the 3D viewer (it will take the first one open). For both a skeleton source can be selected on which different filters can further constrain the set of treenodes that are actually used. This es especially useful if a compartment is already reconstructed and a convex or concave hull describes it reasonably well. The skeletons' treenodes can be used create volumes. Alternatively, one can create new dummy neurons for the sole purpose pf volume generation by roughly outlining a compartment.

Skeletons can contain a lot of treenodes. Therefore, to reduce computation costs, and to provide more flexibility it is possible to apply filters to the input set of treenodes (coming from the selected skeleton source). One could for instance select only branch nodes, roots or connectors. It is possible to add multiple filters at the same time. All filters work on the nodes of the skeletons provided by the selected source. The results of all filters are combined with, optionally different, set operations. A union merges both results and takes all; an intersection only allows what is in both sets. This happens in a left associative fashion (or, here, since a table representation is used, "top associative"). Each filter operates on the same set of input nodes and only their combination creates the final result. By default, if no filter is used, all nodes from the input skeletons are used.

To add a filter, select the type, configure it through the settings below the type selection and press "Add new filter rule" and continue with the next filter.

An example: Add an "Only branch nodes" filter and then select a new type, say Synaptic connections to other neurons and select a merge operation . For synaptic connections, you now can select the partners (or choose None for any partner) and the type of relation that is required. After the second filter is added, you have a point set of branch nodes and connector nodes. Now you can go ahead and adjust the actual volume properties, e.g. the alpha value (to filter out particular tetrahedra, or optionally triangles).

Pressing the "Add new volume" button will generate the mesh. Keep in mind that volumes are materialized after they are created and currently can't be edited after creation. The screenshot below is taken during the volume generation using alpha shapes on a set of filtered skeletons. It shows a preview from the Volume Manager and as can be seen, disconnected meshes are supported as well.



x 6511.8421 y 13873.947 z-index 1210 zoom-level -2

☐ Meshes Volumes ☐ Faces color (Opacity: 0.47) ☑ Active node ☐ Active node on top ☐ Black background
☐ Floor ☐ Debug Line width 3 ☐ Flat neuron material Custom Tags (regex):
Node handle scaling 1

Alpha (nm): 4000
Only triangles with a circumsphere radius less than alpha will be used

10
Set the alpha change step for the numeric input above

☐ Filter triangles
If checked, the alpha filter will be applied to individual triangles of the mesh. Otherwise, alpha is used to filter the tetrahedra of the 3D triangulation.

Node filter: Synaptic connections to other neurons
Nodes inside the Alpha shape

Merge operation: Union
Rules are applied in a left-associative fashion. This selects which operation to use for this.

Apply only to skeleton ID (Optional):
If a valid skeleton ID is provided, this rule will apply to this skeleton exclusively.

... having this name (Optional):
Along with a skeleton ID a name can also be used. If supplied, skeletons are also checked againsts it and only if skeleton ID and name match, the rule will be applied.

Source of synaptic neurons: None
Neurons from this source will be checked against having synapses with the working set. If "None" is selected, all synaptic nodes will be considered.

Relation of base set to above partners: Pre- or postsynaptic
Select how a valid node of the base set (nodes to generate mesh) is related to partner neurons from other source.

On	Name	Merge mode	Options	Is skeleton	Has name
✓	Only end nodes	Union	{"includeRoot":true}	-	-
✓	Only branch nodes	Union	{}	-	-
✓	Synaptic connections to other neurons	Union	{"relation":"pre-or-post"}	-	-

API

If wanted, volumes can also be created outside of CATMAID and then loaded into into it through its HTTP API. As long as it is possible to represent them as a generic triangle mesh, loading them into CATMAID should be easy. The HTTP endpoint to use is:

```
/{project_id}/volumes/add
```

For more details on the API itself, consult the API documentation, which is available under `/apis` on every CATMAID instance.

To store a volume through the API, you have to create a POST request which contains the fields "type" (set to "trimesh"), "title" (a representative title) and "mesh". It is expected to be a string that encodes two lists in JSON format: [[points], [triangles]]. The list of points contains lists of three numbers, each one representing a vertex in the mesh. The array of triangles also contains three element lists as items. Each one represents a triangle based on the points in the other array, that are referenced by the triangle index values. For instance, this would constitute a valid set of fields for the creation request:

```
title: "Example mesh"
type: "trimesh"
mesh: "[[[0,0,0], [1,0,0], [0,1,0], [1,1,0]], [[0,1,2], [1,3,2]]]"
```

If everything went well, the endpoint should return a response with two JSON fields: "success" (should be true) and "volume_id", which holds he ID of the newly created volume.

If you have an API token generated for your CATMAID user, you could also use `curl`, to load the mesh from a JSON file (having the fields above):

```
curl -X POST -d @<json-file> <catmaid_url>/<project_id>/volumes/add \
  --header "Content-Type: application/json" \
  --header "X-Authorization: Token <api-token>"
```

where <catmaid_url> is the URL of your CATMAID instance, <json-file> being the file path to a JSON file representing the data to send (see above), <project_id> is the project ID (visible when creating a link to a view) and <api-token> being your API token.

1.1.11 Workflows

Combining the functionality of different tools in CATMAID can be lead to smaller and bigger workflows. This section is a collection of common tasks.

Neuron tracing in a limited area

Goal: each user should trace only in a defined subvolume in the dataset, for instance a cube.

Workflow:

1. Define a volume/mesh for each tracer in which they are supposed to trace. You do this by opening the Volume Manager and select the "Add volume" tab. By default, the box type is already selected, which is what you want. Navigate the view to to the center of the cube the user should trace in, enter a cube edge length in the "Cube at current location" field and click "Define cube at current location". You can view a preview in both the tracing layer and all 3D Viewers. It's advisable to give the volume a reasonable names there to find the volumes in list and submenus (like the 3D Viewer). Finally, press "Save", the volume should be visible in the list on the "Main tab". From there you can also list all skeletons and connectors in a volume and other actions. The third tab in the Volume Manager allows you to find all the volumes a particular set of skeleton innervates.

2. With the volume created, you can define a "tracing warning" in the Settings Widget. It's almost at the bottom of the widget, or you filter by "warning" in the top input box. In the warning section you can select a volume in which new nodes are supposed to be. If they are outside, a warning message is displayed.

1.1.12 Frequently Asked (User) Questions

Why can I only export WebM movies from the 3D viewer?

There is currently no easy way to generate other formats from within a browser. Besides, WebM is a reasonable effort to standardize and it seems to be the future play-everywhere codec. However, you can use other tools to convert the generated WebM movie to your preferred video format (e.g. to play it in PowerPoint). A convenient GUI tool to do this is Handbrake. Alternatively, if you prefer the command line, you could use `ffmpeg`:

```
ffmpeg -y -i input.webm -vcodec libx264 output.mov
```

or avconv:

```
avconv -i input.webm -vcodec copy output.mov
```

Also note that `ffmpeg` (or rather the H264 codec) expects the input to have a width and height of an even number. To crop the input movie on the fly, the following option can be added: `-filter:v "crop=<width>:<height>:0:0"`, replacing `<width>` and `<height>` with the desired width and height in pixels.

What are skeleton source subscriptions?

Many tracing related widgets allow to react to changes in skeleton lists in other widgets. Widgets supporting this got a new small chain icon in their title bar with which a subscription management user interface can be shown and hidden. Widgets that contain multiple sources, like the connectivity matrix, have one icon per source. A hover title will show which one to use for each source.

The UI allows to add subscriptions to multiple sources which can then be combined through set operations. Currently sources are combined in a strict left-associative fashion from top to bottom of the list. When "Override existing" is checked, widget local skeletons are not used when subscriptions are refreshed and will subsequently be removed. Otherwise, the local set is united with the first subscription before all other subscription sources are applied.

The scope of individual subscriptions can be adjusted: By default each subscription reacts to skeletons added, removed and updated in a source. The "Filter" selection allows to listen to only one of these events. For instance, subscribing to the active skeleton with the "Only additions" filter, allows to collect skeletons selected active skeletons without removing them again from a widget.

By default, only selected skeletons are subscribed to. This means if a skeleton becomes unselected in a source it is removed from the target widget. If the "Only selected" checkbox is unchecked, also unselected skeletons are added to a target widget. They are removed when skeletons are removed from the source and their selection state is synced.

How to make a local copy of image data available?

CATMAID represents multiple copies of image data as so-called stack mirrors. Which mirror is used can be selected in the image layer settings dialog, which can be opened through the little blue-white square button in the lower left corner of a stack viewer. Besides mirrors configured by an administrator, it is also possible to add custom mirrors. Custom mirrors are persisted in a browser cookie and will be available after reloading CATMAID. The 'Add' button in the 'Custom mirror' section of the layer settings will bring up a dialog where a new custom mirror can be added. Nearly all required fields are pre-populated from an existing mirror, only a URL has to be added.

The image data available from this URL has to match the properties in the dialog, which should normally be the case if the image data is a copy of an existing image stack. Additionally, it is recommended that this data is made available through HTTPS. As an example, a common use case is to have a copy of the image data set on an external USB SSD drive. To make this data available to CATMAID, a local webserver has to be started. An easy way to do this is to grab a copy of a simple Python server script available from the CATMAID source repository. Save a copy of this script in the root folder of the USB SDD along with a copy of the certificate, which is available from the same location and should be placed next to the `serve-directory.py` script. Next navigate with a terminal to the root of the image data and execute the Python script:

```
python serve-directory.py 8090 ./localhost.pem
```

The first argument is the port on which the server will be made available and the second argument is the downlaoded previously SSL certificate. If everything works as expected, the URL to put in CATMAID's custom mirror dialog should be:

```
https://localhost:8090/
```

If the image data is not directly available in the USB SDD's root, the relative path has to be added to the URL.

How to create the small overview images in the lower right corner?

It is possible to show a small overview image of the current section in the lower right corner. Generally, CATMAID looks for them as `small.<extension>` (e.g. `small.jpg`) using the (remote) path for the current Z coordinate and makes the file fit into 192x192px. These files can be created of course in many ways and here is one waay doing ths using `graphicsmagick` (or alternatively `imagemagick`):

It makes sense to use the highest zoom level as possible, becasue we make the image only smaller and the less data to process the quicker we have our images. Also, in this simple example, it means that we don't need to combine tiles and only have to deal with a single image.

Let's assume we have nine zoom levels and the data will occupy only one tile at this zoom level, i.e. the highest value displayed in the UI as *z-index* is 8, because zoom-levels are zero-indexed. Like said above, CATMAID wants these files to fit into 192x192px, so we need to find out how much we need to scale the zoom-level. However, at this zoom level, there will be zome extra void data, because the scaled-down dataset is less wide than the defined tile width. If we know our image data has a larger width than height, we can compute the actual width of the data at zoom-level 9 through `<dataset-width-at-zoom-0>/2**<zoom-level-to-use>)`. This can be used to obtain the scale factor required for the 192x overview image, which in turn can be used to find out by how much to scale a tile at that zoom level so that the data it contains fits into the 192x192px overview image:

```
new_tile_width = (192 / (<dataset-width-at-zoom-0>/2**<zoom-level-to-use>)) * <tile-
↪width>
```

For instance, a dataset that has a width of 135200 at zoom level zero, a tile size of 1024px and nine zoom levels:

```
744.55 = (192 / (135200/2**9)) * 1024
```

From a tile that is scaled to this width, we would then only use the top left cutout for the overview, the rest is empty data. This can be done using `convert` tool (of the `graphicsmagick` or `imagemagick` package):

```
convert /path/to/input/tile/ -resize <data-width-at-zoom>x -gravity NorthWest -extent
↪<overview-width>x<overview-height> /path/to/z/directory/small.<extension>
```

Sticking to the example above, and assuming data in a tile source type type 4 ("Backslash tile source") directory structure under `/data/tiles/` and file extension `jpg`, this command could look like the following to generate the overview for section 0:

```
convert /data/tiles/0/9/0_0.jpg -resize 745x -gravity NorthWest -extent 192x170 /data/
↪tiles/0/small.jpg
```

To run this for the whole image stack, a small Bash loop can be used:

```
for f in (ls /data/tiles/); do convert /data/tiles/f/9/0_0.jpg -resize 745x -gravity␣
↪NorthWest -extent 192x170 /data/tiles/$f/small.jpg; done
```

Note that the 170px height of the the overview image can be computed by scaling the original data so that its width fits into 192px. If the data was taller than wide, the height would be 192px and the width adjusted.

1.1.13 Exporting and Importing Data

For importing, there are currently two different tool-sets available in CATMAID. A front-end in Django's admin interface is only available for importing project and stack information. If you want to import tracing data, you have to resort to the command line.

Exporting and importing neuron tracing data

Two management commands for Django's `manage.py` tool are available in CATMAID that allow exporting and importing neuron tracing data. They are called `catmaid_export_data` and `catmaid_import_data`. To use them, you have to be in the `virtualenv` and it is probably easiest to work from the `django/projects/` directory.

Exporting data

At the moment, the export command is able to create a JSON representation of neurons, connectors, tags and annotations. To constrain the exported neurons, annotations can be used. To export data, you have to use the `catmaid_export_data` command:

```
manage.py catmaid_export_data
```

Adding the `--help` option will show an overview over all available options. When called without any option, the command will ask the user for the project to export from and will start exporting the whole project right away. Use the additional options to be more precise about what should be exported.

Without any parameter, everything is exported. The type of data to be exported can be adjusted by the `--notreenodes`, `--noconnectors`, `--noannotations` and `--notags` parameters. To constrain the exported neurons, the `--required-annotation` option can be used. For instance, to export all neurons from the project with ID `1` that are annotated with "Kenyon cells", one would have to call:

```
manage.py catmaid_export_data --source 1 --required-annotation "Kenyon cells"
```

This will create a file called `export_pid_<pid>.json`, which would be `export_pid_1.json` in our case. A different file name can be specified using the `--file` option and if the passed in string contains "{}", the braces will be replaced by the source project ID.

Users are represented by their usernames and it is not required to export user model objects as well. The importer can either map to existing users or create new ones. If wanted, though, complete user models can be exported (and imported) as well by providing the `--users` option. Be aware though that this includes the hashed user passwords.

Importing data

The JSON file generated in the previous section can be used to import data into a CATMAID project. This project can be non-empty or a new one and can be part of the source CATMAID instance or a completely different one. do this so, use the `catmaid_import_data` management command:

```
manage.py catmaid_import_data
```

You can use the `--help` switch to get an overview of the available options. Like the exporter, the importer will ask a user if it needs more information. If no project is specified, users can select an existing one or create a new empty target project interactively.

Assuming a file called `export_pid_1.json` is available and a new CATMAID project with ID `1` has been created, the following command will start the import:

```
manage.py catmaid_import_data --source export_pid_1.json --target 1
```

By default, the importer tries to map users referenced in the input data to existing users. If this is not wanted, the option `--map-users false` has to be used.

The importer looks at the `user_id`, `editor_id` and `reviewer_id` fields of imported objects, if available. CATMAID needs to know what to do with this information. Besides mapping users, CATMAID can also override all import data user information with a single user. Setting `--map-users false` and either selecting a user interactively oder with the help of the `--user <id>` option, will accomplish this.

If users are mapped and a username does not exist, the import is aborted—unless the `--create-unknown-users` option is provided. This will create new inactive user accounts with the provided usernames that are referenced from the new imported objects. The allows to not require full user profiles to be part of exported data.

By default, the importer won't use the IDs of spatial data provided from the import source and instead will create new database entries. This default is generally useful and doesn't risk replacing existing data. All relations between objects are kept. If however needed, the use of the source provided IDs can be enforced by using the `--preserve-ids` option.

Semantic data, like annotations and tags however are re-used if already available in the target project. The only exception to this are neuron and skeleton objects, which are technically semantic objects, but are expected to not be shared or reused.

Bulk loading large data sets

Importing data using the `catmaid_import_data` management command works well for thousands of neurons and connectors. It becomes however very slow and memory intensive to load millions or billions of neurons. On this scale, loading data direcrly into the database is the best strategy. It requires extra care, because most safe-guards the API and management commands provides will be bypassed.

Details on the import process are collected on the Bulk loading page.

Importing project and stack information

Image data in CATMAID is referenced by stack mirrors, which belong to a particular stack. Stacks in turn are organized in projects. The data used by a stack mirror can have one of various types of data sources. A simple and often used source is a simple folder structure of tiled image data for each stack. To be accessible, a stack mirror's image base has to give access to such a folder from the web. Of course, stacks, stack mirrors and projects can be created by hand, but there is also an importing tool available in Django's admin interface. It can be found under *Custom Views* and is named *Import projects and image stacks*.

The admin import tool can import project and stack information from various source:

- Other CATMAID instances,
- A general remote API endpoint
- A remote or local JSON file
- A textural JSON definition
- A local file and folder structure

All JSON representaitons that can be imported are of the format that is returned by CATMAID's `/projects/export` API endpoint. This very same format can be used with a managment command, `catmaid_import_projects`, on the command line.

The command line option and the file based import are explained in more detail below. The latter expects a certain data folder layout to work on and als relies on so called *project files* (a file in a simple YAML format) to identify potential projects. Its data layout is explained below as well.

How to use the importing tool will be shown in the last section.

Note: Regardless of whether you import image stacks with the help of the importer or add projects, stacks, stack mirrors, project-stack links and permissions by hand: you need to make the image data accessible through your web-server. For instance, say you are running CATMAID on the URL `example.com/catmaid` and you are using the importer and have `IMPORTER_DEFAULT_IMAGE_BASE` set to `https://example.com/catmaid/data` and `CATMAID_IMPORT_PATH` to `/home/catmaid/data/`. The webserver would now need to map `/catmaid/data` to `/home/catmaid/data/`. For Nginx this would look like that:

```
location /catmaid/data/ {
  alias /home/catmaid/data/;
}
```

Importing projects and stacks on the command line

The management command `catmaid_import_projects` can be used to import project from the command line. It understands the basic `/projects/export` API endpoint format. A simple example might look like this:

```
[{
  "project": {
    "title": "L1 CNS",
    "stacks": [{
      "title": "L1 CNS",
      "dimension": "(28128, 31840, 4841)",
      "mirrors": [{
        "fileextension": "jpg",
        "position": 3,
```

(continues on next page)

(continued from previous page)

```
        "tile_source_type": 4,
        "tile_height": 512,
        "tile_width": 512,
        "title": "Example tiles",
        "url": "https://example.com/ssd-tiles/"
      }],
      "resolution": "(3.8,3.8,50)",
      "translation": "(0,0,6050)"
    }]
  }
}]
```

This can be provided as a file with its path provided as `--input` argument to `catmaid_import_projects` or through the `stdin` stream that is piped into it. Additionally, it possible to provide permissions for the imported projects. This is done through one ore more `--permission` parameters. Each of them will take an argument in the form `type:name:permission`, where `type` can either be `user` or `group`. `name` is either a username or a groupname and `permission` can be `can_browse`, `can_annotate` and so forth.

The import can be configured with additional options which are explained when using the `--help` option.

Project Files

If the importing tool encounters a folder with a file called `project.yaml` in it, it will look at it as a potential project. If this file is not available, the folder is ignored. However, if the file is there it gets parsed and if all information is found the tool is looking for, the project can be imported. So let's assume we have a project with two stacks having one image data copy each in folder with the following layout:

```
project1/
  project.yaml
  stack1/
  stack2/
```

A project file contains the basic properties of a project and its associated stacks. It is a simple YAML file and could look like this for the example above:

```
project:
    title: "Wing Disc 1"
    stacks:
      - title: "Channel 1"
        description: "PMT Offset: 10, Laser Power: 0.5, PMT Voltage: 550"
        dimension: "(3886,3893,55)"
        resolution: "(138.0,138.0,1.0)"
        mirrors:
          - title: "Channel 2 overlay"
            folder: "stack1"
            fileextension: "jpg"
      - title: "Channel 2"
        description: "PMT Offset: 10, Laser Power: 0.7, PMT Voltage: 500"
        dimension: "(3886,3893,55)"
        resolution: "(138.0,138.0,1.0)"
        downsample_factors: ["(1,1,1)", "(2,2,1)", "(4,4,1)"]
        mirrors:
          - title: Channel 2 image data
            folder: "stack2"
            fileextension: "jpg"
```

(continues on next page)

(continued from previous page)

```
        stackgroups:
          - title: "Example group"
            relation: "has_channel"
      - title: "Remote stack"
        dimension: "(3886,3893,55)"
        resolution: "(138.0,138.0,1.0)"
        zoomlevels: 3
        downsample_factors: ["(1,1,1)", "(2,2,1)", "(4,4,1)", "(8,8,1)", "(16,16,1)"]
        translation: "(10.0, 20.0, 30.0)"
        mirrors:
          - tile_width: 512
            tile_height: 512
            tile_source_type: 2
            fileextension: "png"
            url: "https://my.other.server.net/examplestack/"
        stackgroups:
          - title: "Example group"
            relation: "has_channel"
```

As can be seen, a project has only two properties: a name and a set of stacks. A stack, however, needs more information. In general, there are two ways to specify the data source for a folder: 1. an optional `path` and a `folder`, both together are expected to be relative to the `IMPORTER_DEFAULT_IMAGE_BASE` settings or 2. a `url`, which is used as a stack mirror's image base.

The first stack in the example above is based on a folder in the same directory as the project file. The `folder` property names this image data folder for this stack, relative to the project file. The name of stack is stored in the `title` field and metadata (which is shown when a stack is displayed) can be added with the `metadata` property. A stack also needs `dimensions` and `resolution` information. Dimensions are the stacks X, Y and Z extent in *pixel*. The resolution should be in in *nanometers per pixel*, in X, Y and Z.

In addition to the `folder` information, the second and third stack above use the `downsample_factors` field to declare the number of available "zoom levels". This is done by providing a list of 3D scale factors, one for each zoom level available, with `1,1,1` being the unscaled voxel size of the data set. In a classical anisotropic TEM data set, it is common to zoom out in steps of factors of two in X and Y and keep the visual section thickness the same. Assuming there are four additional zoom levels besides the original version, this could be expressed as the downsample factors `["(1,1,1)", "(2,2,1)", "(4,4,1)", "(8,8,1)", "(16,16,1)"]`, scaling X and Y by a factor of two and Z not at all for each entry of the array.

Of course, any other configuration could be chosen as well. Likewise for isotropic data sets like FIB-SEM, the `Z` component in each tuple would be scaled as well. If no downsample_factors are given, CATMAID's default behavior is to use as many scale levels as are needed in the above anisotropic downsample factors for the original image width or height to fit on a single tile. Since different stack mirrors can have different tile sizes, the minimum of those are used. Without any mirrors, no zoom levels will be assumed.

The example above also specifies the file extension of the image files with the `fileextension` key. Both fields are required.

The last stack in the example above *doesn't* use a local stack folder, but declares the stack mirror's image base explicitly by using the `url` setting. Like done for the folder based stacks, a url based stack mirror needs the `tile_width`, `tile_height` and `tile_source_type` fields. The corresponding stack defines the `resolution` and `dimension` fields.

CATMAID can link stacks to so called stack groups. These are general data structures that relate stacks to each other, for instance to denote that they represent channels of the same data, orthogonal views or simple overlays. There is no limit on how many stack groups a stack can be part of. Each stack in a project file can reference stack groups by `title` and the type of `relation` this stack has to this stack group. At the moment, valid relations are `channel` and `view`. All stacks referencing a stack group with the same name will be linked to the same new stack group in the new project. In the example above, a single stack group named "Example group" will be created, having stack 2 and 3 as members—each representing a layer/channel. Stack groups are used by the front-end to open multiple stacks at once in a more intelligent fashion (e.g. open multi-channel stack groups as layers in the same viewer).

All specified stacks within a project are linked into a single space. By default each stack origin is mapped to the project space origin (0,0,0). An optional translation can be applied to this mapping: If a stack has a `translation` field, the stack is mapped with this offset into project space. Note that this translation is in project space coordinates (physical space, nanometers). The example above will link the last stack ("Remote stack") to the project "Wing Disc 1" with an offset of `(10.0, 20.0, 30.0)` nanometers. Both other stacks will be mapped to the project space origin.

Also, it wouldn't confuse the tool if there is more YAML data in the project file than needed. It only uses what is depicted in the sample above. But please keep in mind to *not use the tab character* in the whitespace indentation (but simple spaces) as this isn't allowed in YAML.

Ontology and classification import

The project files explained in the last section can also be used to import ontologies and classifications. While CATMAID supports arbitrary graphs to represent ontologies and classifications,only tree structures can be imported at the moment.

The `project` object supports an optional `ontology` field, which defines an ontology hierarchy with lists of lists. An optional `classification` field can be used to define a list of ontology paths that get instantiated based on the provided ontology. Classification fields require that an ontology is defined and can be used on `project` level, `stack` level and the `stackgroup` level. Consider this example:

```
project:
   title: "test"
   ontology:
     - class: 'Metazoa'
       children:
         - relation: 'has_a'
           class: 'Deuterostomia'
         - relation: 'has_a'
           class: 'Protostomia'
           children:
             - relation: 'has_a'
               class: 'Lophotrochozoa'
               children:
                 - relation: 'has_a'
                   class: 'Nematostella'
                   children:
                     - relation: 'has_a'
                       class: 'Lineus longissimus'
   stackgroups:
     - title: 'Test group'
       classification:
           - ['Metazoa', 'Protostomia', 'Lophotrochozoa', 'Nematostella', 'Lineus␣
↪longissimus']
   stacks:
     - title: "Channel 1"
       description: "PMT Offset: 10, Laser Power: 0.5, PMT Voltage: 550"
       dimension: "(1024,1024,800)"
       resolution: "(2.0,2.0,1.0)"
       zoomlevels: 1
       translation: "(10.0, 20.0, 30.0)"
       classification:
```

(continues on next page)

(continued from previous page)

```
          - ['Metazoa', 'Deuterostomia']
       mirrors:
          - title:  Channel 1
            url: "https://example.org/data/imagestack/"
            fileextension: "jpg"
     - title: "Channel 1"
       description: "PMT Offset: 10, Laser Power: 0.5, PMT Voltage: 550"
       dimension: "(1024,1024,800)"
       resolution: "(2.0,2.0,1.0)"
       zoomlevels: 1
       translation: "(10.0, 20.0, 30.0)"
       mirrors:
         - title: Channel 1
           url: "https://example.org/data/imagestack-sample-108/"
           fileextension: "jpg"
       stackgroups:
        - title: "Test group"
          relation: "has_channel"
     - title: "Channel 2"
       description: "PMT Offset: 10, Laser Power: 0.5, PMT Voltage: 550"
       dimension: "(1024,1024,800)"
       resolution: "(2.0,2.0,1.0)"
       zoomlevels: 1
       mirrors:
        - title: Channel 2
          folder: "Sample108_FIB_catmaid copy"
          fileextension: "jpg"
       stackgroups:
        - title: "Test group"
          relation: "has_channel"
```

The project level ontology definition represent an ontology with the root node “Metazoa” which has two children: “Deuterostomia” and “Protostomia”, connected through a “has_a” relation. While the first child is a leaf node and has no children, the second child has a child node as well (and so on). It is possible to have multiple roots (i.e. separate ontology graphs) and multiple children, both are lists.

Individual stacks and stackgroups are then allowed to instantiate a certain path of the ontology and be linked to the leaf node of the path. They do this by supporting a `classification` field. The example creates two classification paths and links one leaf node to the stack group and one to an individual stack.

Currently, the importer expects that those two classes are only related on the ontology level a single time. This allows for an easier file syntax with a simple list. An import will fail if the project defined ontology doesn’t contain a class used in a classification.

File and Folder Layout

The importing tool expects a certain file any folder layout to work with. It assumes that there is one data folder per CATMAID instance that is accessible from the outside world and is somehow referred to within a stack mirror’s image base (if referring to folders in the project file). As an example, let’s say a link named *data* has been placed in CATMAID’s top-level directory. This link links to your actual data storage and has a layout like the following:

```
data/
  project1/
  project2/
  project3/
```

(continues on next page)

(continued from previous page)

```
  tests/
    project4/
```

Each project folder has contents similar to the example in the previous section. Assuming your webserver has been configured to make this folder accessible, it should be possible to have the browser access it under:

```
https://<CATMAID-URL>/data
```

A typical URL to a tile of a stack could then look like this (if you use `jpeg` as the file extension):

```
https://<CATMAID-URL>/data/project1/stack1/0/0_0_0.jpeg
```

The importer uses this data directory or a folder below it as working directory. In this folder it treats every sub-directory as a potential project directory and tests if it contains a project file named `project.yaml`. If this file is found a folder remains potential project. A folder is ignored, though, when the project file is not available.

Also note, that such a data folder needs to be made accessible by the webserver. Otherwise, the CATMAID front-end won't be able to retrieve and show image data. The "Note" box at the beginning of this section has more details on this.

Importing skeletons through the API

The CATMAID API supports raw skeleton data import using SWC files. As can be seen under `/apis`, the `{project_id}/skeletons/import` URL can be used to import skeletons that are repesented as SWC. The script `scripts/remote/upload_swc.py` can be of help here. It is also possible to just use `cURL` for this:

```
curl --basic -u fly -X POST --form file=@<file-name> \
    <catmaid_url>/<project_id>/skeletons/import \
    --header "X-Authorization: Token <api-token>"
```

Using the importer admin tool

The import offers to import from local project files, remote CATMAID instances or remote project files/exports.

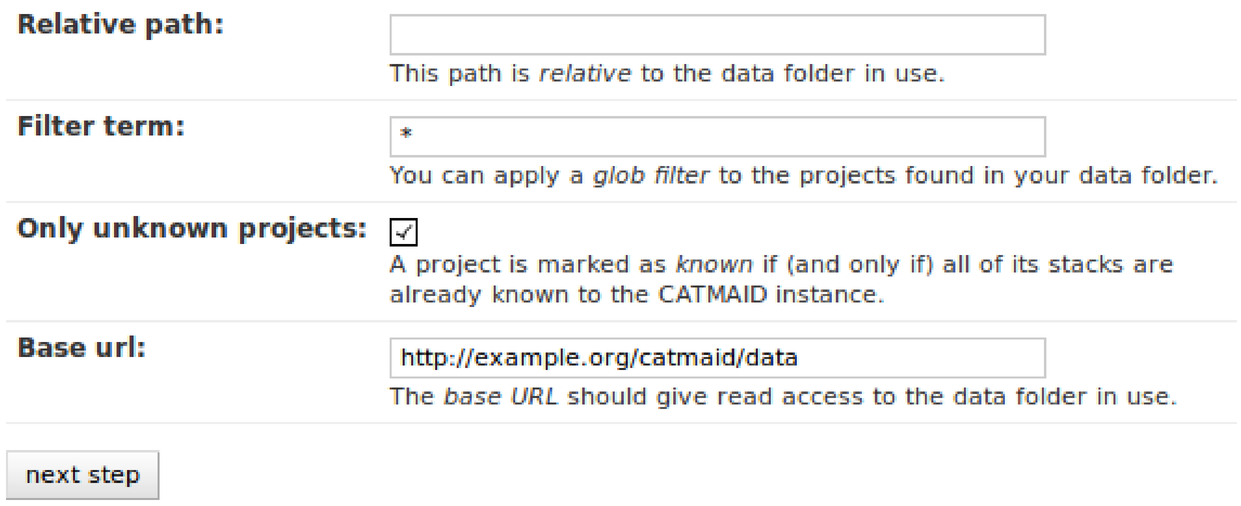
To use the importer with project files, you have to adjust your CATMAID settings file to make your data path known to CATMAID. This can be done with the `CATMAID_IMPORT_PATH` settings. Sticking to the examples from before, this setting might be:

```
CATMAID_IMPORT_PATH = <CATMAID-PATH>/data
```

For imported stack mirrors that don't provide an image URL by themselves, CATMAID can construct an image base from the the `IMPORTER_DEFAULT_IMAGE_BASE` setting plus the imported project and stack names. For the example above, this variable could be set to:

```
IMPORTER_DEFAULT_IMAGE_BASE = https://<CATMAID-URL>/data
```

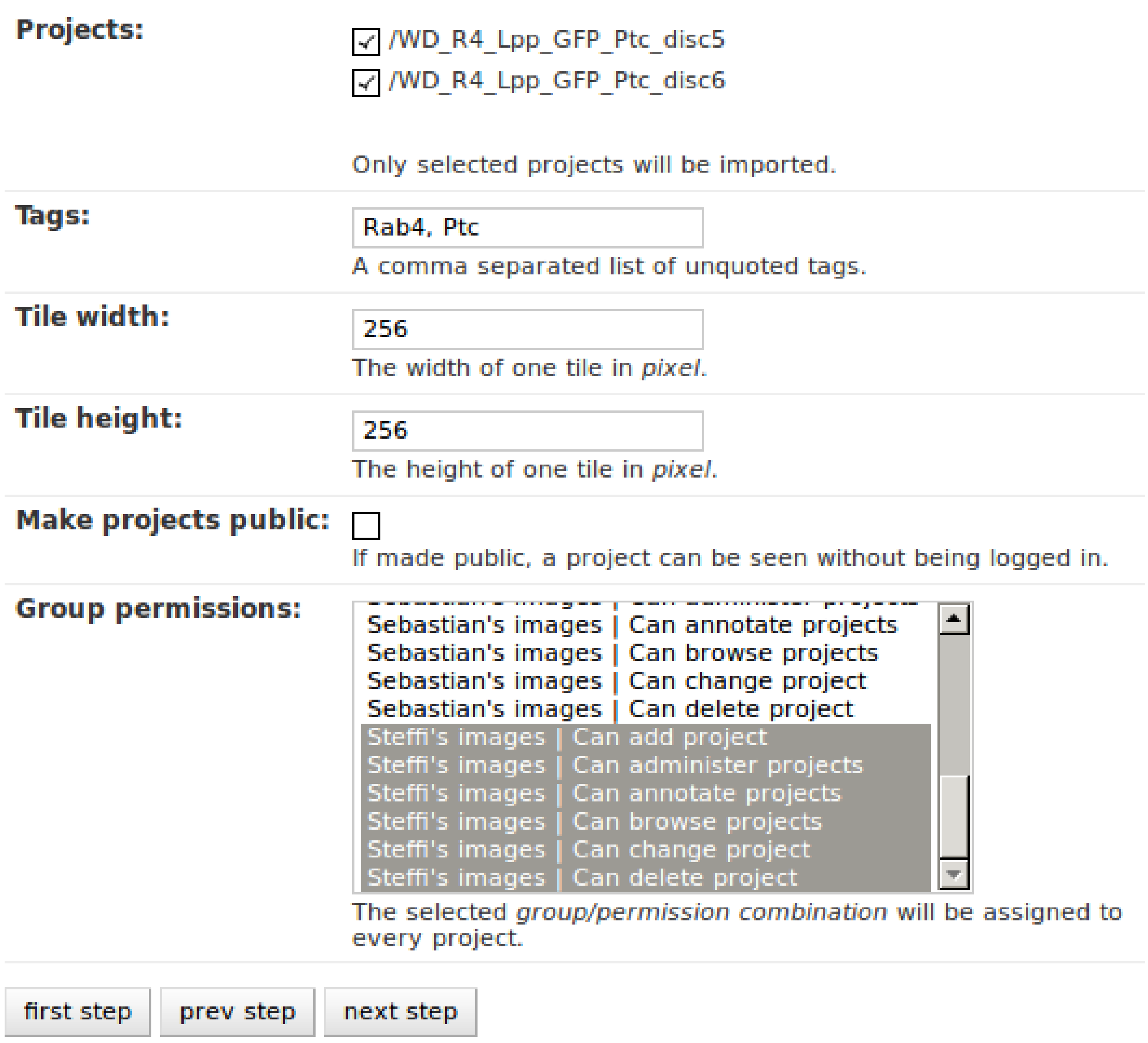
With this in place, the importer can be used through Django's admin interface. It is listed as *Image data importer* under *Custom Views*. The first step is to give the importer more detail about which folders to look in for potential projects:



Relative path:	This path is *relative* to the data folder in use.
Filter term:	* You can apply a *glob filter* to the projects found in your data folder.
Only unknown projects:	✓ A project is marked as *known* if (and only if) all of its stacks are already known to the CATMAID instance.
Base url:	http://example.org/catmaid/data The *base URL* should give read access to the data folder in use.

With these settings, you can narrow down the set of folders looked at. The relative path setting can be used to specify a sub-directory below the import path. When doing so, the working directory will be changed to `CATMAID_IMPORT_PATH` plus the *relative path*. If left empty, just the `CATMAID_IMPORT_PATH` setting will be used. Additionally, you can *filter folders* in tho working directory by specifying a filter term, which supports Unix shell-style wildcards. The next setting lets you decide how to deal with already existing (known) projects and what is considered known in the first place. A project is known can be declared to be known if the name of an imported project matches the name of an already existing one. Or, it can be considered known if if there is a project that is linked to the very same stacks like the project to be imported. A stack in turn is known if there is already a stack with the same mirror image base. The last setting on this dialog is the *Base URL*. By default it is set to the value of `IMPORTER_DEFAULT_IMAGE_BASE` (if available). This setting plus the relative path stay the same for every project to be imported in this run. It is used if imported stacks don't provide a URL explicitly. To continue, click on the *next step* button.

The importer will tell you if it doesn't find any projects based on the settings of the first step. However, if it does find potential projects, it allows you to unselect projects that shouldn't get imported and to add more details:



Projects:	☑ /WD_R4_Lpp_GFP_Ptc_disc5 ☑ /WD_R4_Lpp_GFP_Ptc_disc6 Only selected projects will be imported.
Tags:	Rab4, Ptc A comma separated list of unquoted tags.
Tile width:	256 The width of one tile in *pixel*.
Tile height:	256 The height of one tile in *pixel*.
Make projects public:	☐ If made public, a project can be seen without being logged in.
Group permissions:	Sebastian's images | Can annotate projects Sebastian's images | Can browse projects Sebastian's images | Can change project Sebastian's images | Can delete project Steffi's images | Can add project Steffi's images | Can administer projects Steffi's images | Can annotate projects Steffi's images | Can browse projects Steffi's images | Can change project Steffi's images | Can delete project The selected *group/permission combination* will be assigned to every project.

Besides deciding which projects to actually import, you can also add tags which will be attached to the new projects. If the tile size differs from the standard, it can be adjusted here. If you want your projects to be accessible publicly, you can mark the corresponding check-box.

When the *Check classification links* option is selected, the importer tries to suggest existing classification graphs to be linked to the new project(s). These suggestions are optional and based on the tags you entered before. If existing projects have the same tags or a super set of it, their linked classification graphs will be suggested.

The last adjustment to make are permissions. With the help of a list box you can select one or more group/permission combinations that the new projects will be assigned. If all is how you want it, you can proceed to the next dialog.

The third and last step is a confirmation where all the information is shown the importer found about the projects and stacks to be imported. To change things in this import, simply go back to a step before, using the buttons at the bottom of the page. If all the project and stack properties as well as the tags and permissions are correct, the actual import can start.

In the end the importer will tell you which projects have been imported and, if there were problems, which ones not.

1.1.14 Using Data Views

Introduction

Providing a good entry point to the different data sets served by a CATMAID instance is important. Depending on number and type of the projects and stacks, different views on this data might help you with keeping an overview or to spot the data set you are interested in. CATMAID's data views try to help you with that. With them you can define and tweak the front page of CATMAID.

In fact, you can have as many front pages as you want. The ones available are accessible through a menu, activated by hovering the *Home* link in the menu. The one used when the CATMAID site is opened, differs only in one aspect from the others: it is marked as default. All data views are configured instances of a particular *data view type*. They define the basic structure of the page, but allow to pass parameters to adjust some parts.

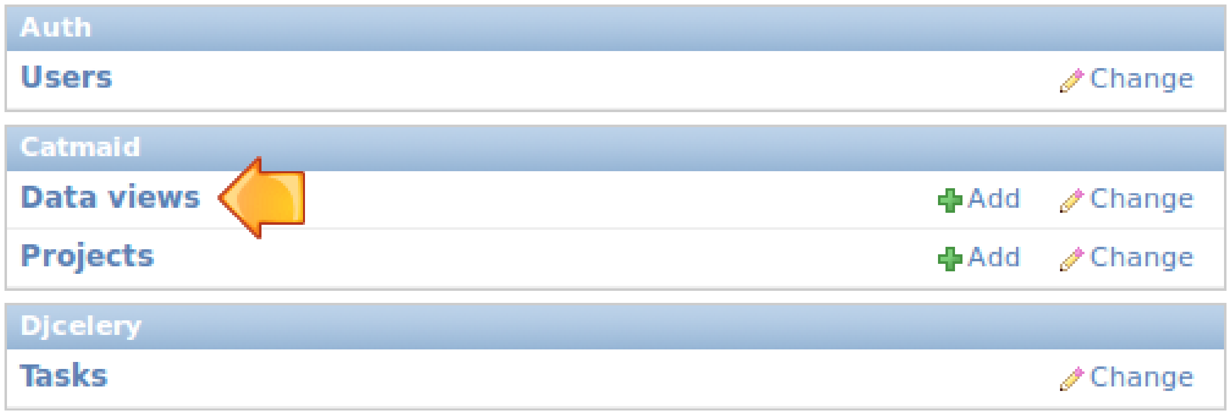
Currently, CATMAID comes with two pre-defined data views: *Project list* and *Project table with images*. The latter is used as the default, i.e. it is shown when CATMAID is loaded. Both views have different view types. They are *Project list* and *Project table*. There is also two more data view types that are not used by the pre-defined views: *Simple project list*. It allows the display of a project list used before the advent of data views. Then there is also the *Project tag table* which allows the construction of a table based on tags. The details are described below.

Different use cases might require different data view types. How to extend CATMAID and create own data view types is explained in section *Creating New Data View Types*.

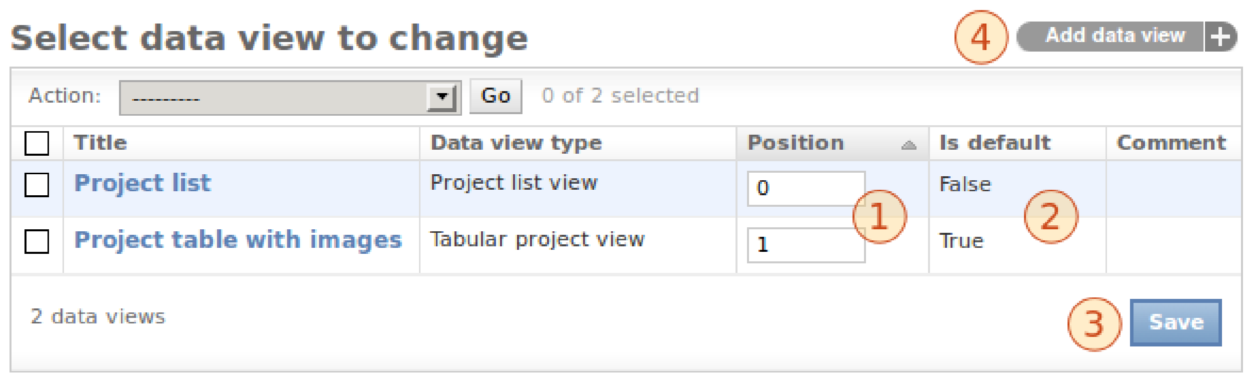
Configuring Data Views

To add new or modify the present views, the Django admin interface is used. By default, you'll find it at *http://<CATMAID-URL>/admin* or a similar URL. It is not necessarily the case, that every CATMAID user can administer data views. This depends on the permissions granted and in case you can't log-in or don't see the relevant parts within the admin interface, talk to your CATMAID administrator.

In the admin interface you will be presented different configuration options for CATMAID. Amongst them, the data view management:



Depending on your CATMAID setup, the admin interface might look a bit different. Opening the data view configuration will get you a list of all available data views. By default this looks like this:



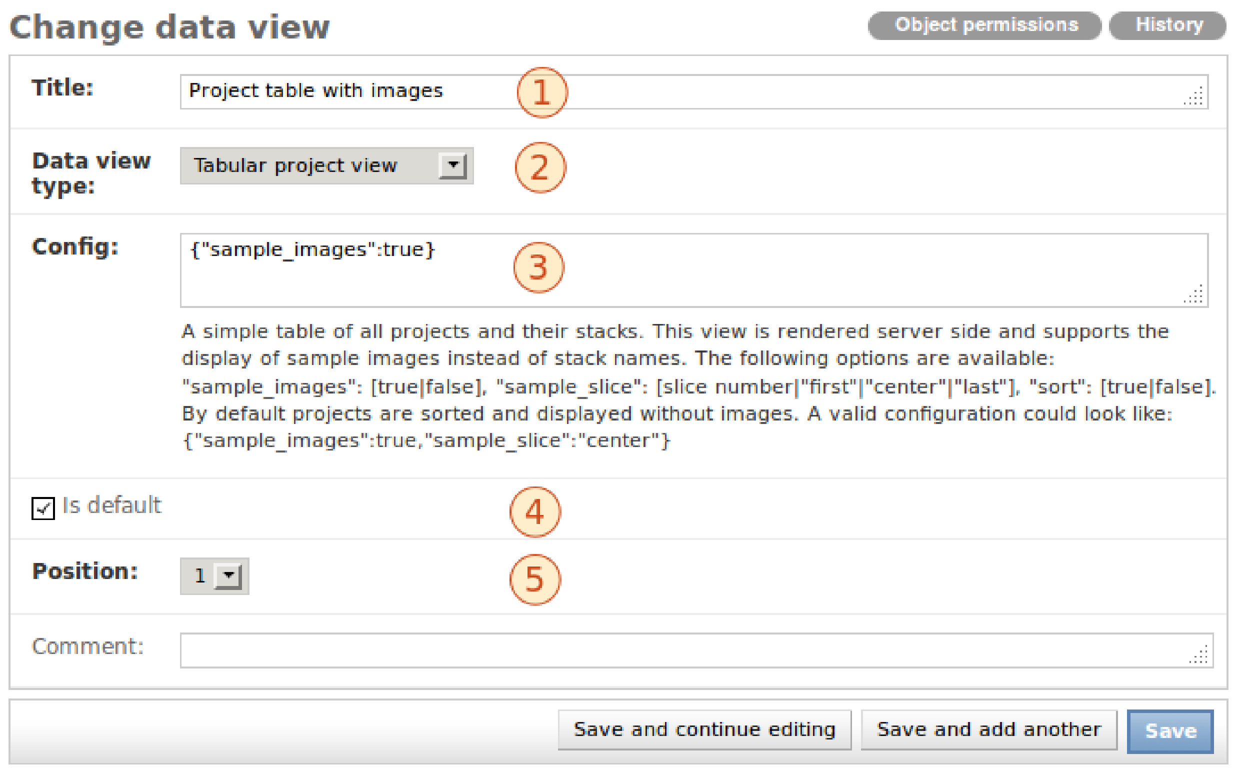
Select data view to change

Action: --------- 0 of 2 selected

Title	Data view type	Position	Is default	Comment
Project list	Project list view	0	False	
Project table with images	Tabular project view	1	True	

2 data views

Next to the name and the type of each view, the position is visible (1). With the help of this number you can define the order of the views in the drop down menu and within this admin view list. The positioning can be changed right there in the list. If you edited the position numbers, click *Save* (3) to, well, save your changes. Also indicated is whether a view is default or not (2). If you want to change this setting, please open up the configuration of one particular data view (see below). Creating a new data view can be accomplished by the button in the upper right (4). The same interface is used for adding new views and editing existing ones. Editing the default view looks like this:



Change data view

Title:	Project table with images
Data view type:	Tabular project view
Config:	{"sample_images":true}
Is default	✓
Position:	1
Comment:	

A simple table of all projects and their stacks. This view is rendered server side and supports the display of sample images instead of stack names. The following options are available: "sample_images": [true|false], "sample_slice": [slice number|"first"|"center"|"last"], "sort": [true|false]. By default projects are sorted and displayed without images. A valid configuration could look like: {"sample_images":true,"sample_slice":"center"}

There are quite some configuration options here: One can easily adjust the title (1), mark a view as the default (4) or define its position (5). Setting a data view as the default view, will mark all other views as non-default at the same time.

If the data view type is changed (2), the help text under the configuration box (3) is updated accordingly. Every data view type has its own set of options and the help text will tell you about those. In the case of the figure above, a *Tabular project view* type is used. A detailed description of the options available can be found below. The format used to define them is called JSON and in order to only use the default options you have to set the configuration to `{}`. If you enter text that cannot be parsed as JSON you will get an error when saving. Saving invalid text is thereby prevented. Typos in option names, however, will not be identified.

If wanted, a comment can be added as well. This will be displayed to the right of the data view title in the menu.

Different Data View Types

Like said above, a data view type defines the general structure of a concrete data view, but gives some configuration options. For instance, most of the types offer to show sample images. If an option is not present in a data views configuration, its default value is used.

Options in help texts usually have the following notation:

```
"<name>":[option1|option2]
```

An option name has to be quoted and the value has to be delimited by a colon. However, also values might need quoting if they are strings other than `true` or `false` (and no numbers). Alternative values are separated by a logical *or*/pipe symbol (`|`) and only one of them can be used. Let's have a look at the *sort* option as an example: All data view types, except the *Simple Project List* type, support the option

```
"sort":[true|false].
```

When set to `true` (default), all projects are sorted naturally and when set to false they are displayed like they come from the database. If we don't want to have the projects sorted, we could use this configuration:

```
{"sort":false}
```

When multiple options are used, each option is separated by a comma from another one – within the same pair of curly braces:

```
{"name1":opt1, "name2:opt2, ...}
```

Like already mentioned, there are options that are supported by all the different data view types. The only exception to this is the *Simple Project List* type, which doesn't support such general options. All the other types respect the following general options:

Name	Options	Default
`sort`	`true, false`	`true`
`filter_tags`	`A list of tags, e.g. ["TagA", "TagB"]`	`[]`
`catalogue_link`	`true, false`	`true`

With the help of the `sort` option one can make sure the list of projects a data view deals with is sorted (or that it is not). The `filter_tags` option allows to define a list of tags that have to be assigned to all projects looked at. If neuron annotations are used in a project, a so called neuron catalogue can be displayed. By default, a link to this overview is displayed for every project that has such annotations. This can be disabled, by setting the `catalogue_link` option to `false`. If a data view should get an unsorted list of projects that are tagged with `Test` as well as `CNS` and catalogue links should not be displayed, the configuration would look like:

```
{"sort":false, "filter_tags":["Test", "CNS"], "catalogue_link":false}
```

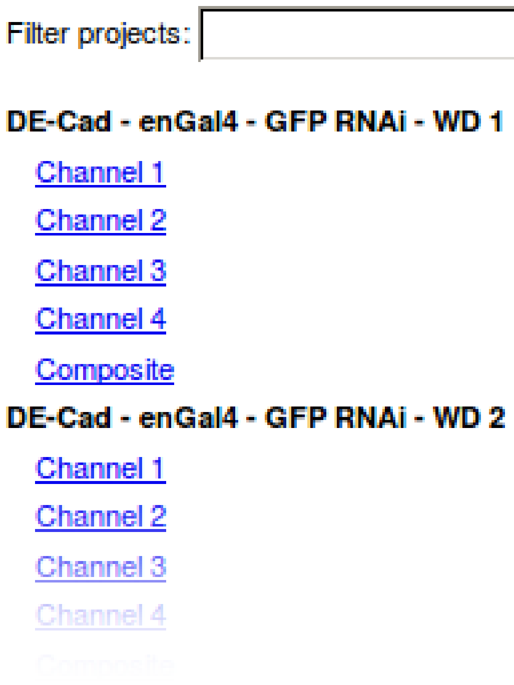
The remainder of this section will briefly discuss the different data view types available.

Simple Project List

All data views except this one are processed or the server-side. This project list is done with the help of JavaScript in the browser of a user. It supports an interactive live filters for both projects and stacks/stack groups, which can optionally pre-set with the data view configuration. The `filter` option decides if the filter input boxes are visible.

Name	Options	Default
`filter`	`true, false`	`true`
`projectFilterTerm`	`Plain string or regular expression`	`""`
`stackFilterTerm`	`Plain string or regular expression`	`""`
`header`	`true, false`	`true`
`message`	`String`	`def. message`
`with_stacks`	`true, false`	`true`
`with_stackgroups`	`true, false`	`true`
`show_empty_projects`	`true, false`	`false`
`show_controls`	`true, false`	`false`
`favorites_first`	`true, false`	`true`
`sample_images`	`true, false`	`false`
`sample_mirror_index`	`position of mirror to use`	`0`
`sample_slice`	`slice index, "first", "center", "last"`	`"center"`
`initial_location`	`3-Element array, e.g. [10, 10, 10]`	`""`
`initial_zoom`	`zoom level index`	`""`
`initial_tool`	`Can be "tracingtool`	`""`
`initial_layout`	`A layout out string`	`""`

An example:


Filter projects:

DE-Cad - enGal4 - GFP RNAi - WD 1

Channel 1

Channel 2

Channel 3

Channel 4

Composite

DE-Cad - enGal4 - GFP RNAi - WD 2

Channel 1

Channel 2

Channel 3

Channel 4

Project List

The appearance of *Project List* is very similar to the one of *Simple Project List*. Currently, it doesn't support live filtering of projects, though. When a project has no stacks, it won't be visible with this view type. It is processed server-side and offers some configuration options:

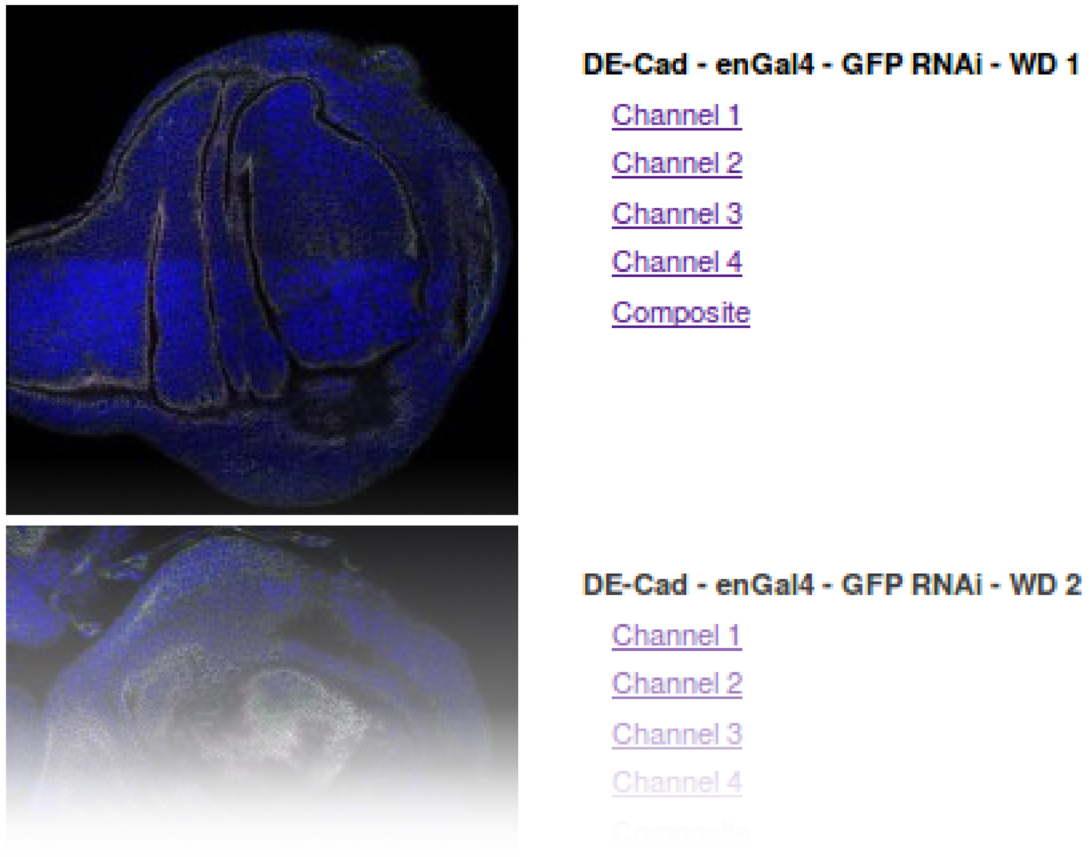
Name	Options	Default
`sort`	`true, false`	`true`
`filter_tags`	`A list of tags, e.g. ["TagA", "TagB"]`	`[]`
`catalogue_link`	`true, false`	`true`
`sample_images`	`true, false`	`false`
`sample_mirror_index`	`position of mirror to use`	`0`
`sample_stack`	`stack index, "first", "last"`	`0`
`sample_slice`	`slice index, "first", "center", "last"`	`"center"`
`sample_width`	`width in pixel, e.g. 50 or 75`	`orig. width`
`sample_height`	`height in pixel, e.g. 50 or 75`	`orig. height`
`show_mirrors`	`true, false`	`false`

Please refer to the introduction to this section for an explanation of the `sort`, `filter_tags` and `catalogue_link` options. The `sample_images` option defines whether there should be a sample image displayed to the left of each project entry. If an image is displayed, the `sample_stack` option defines from which stack of the current project the image should be taken. The `stack index` value of it represents a single integer (e.g. 0). If this index is out of bounds, no image is displayed. With the option `sample_slice` one defines which slice of the sample stack should be used as sample image. Again, `slice index` is an integer number and no image is shown if this is not within bounds. The option values `"first"` and `"last"` refer to the first and last slice of the stack, `"center'` refers to the middle slice is the stack. To make the sample images appear smaller or larger, the `sample_width` and `sample_height` option can be used. The assigned number is treated as pixel width and height, respectively. If a sample image is used, a particular mirror can be selected using the `sample_mirror_index` field. The mirror with the selected position will be used.

So if you were to configure a *Project List* data view with a sample image which should be the middle slice of every last stack in a project, you would do this:

```
{"sample_images":true, "sample_stack":"last", "sample_slice":"center"}
```

The result could look like the following:



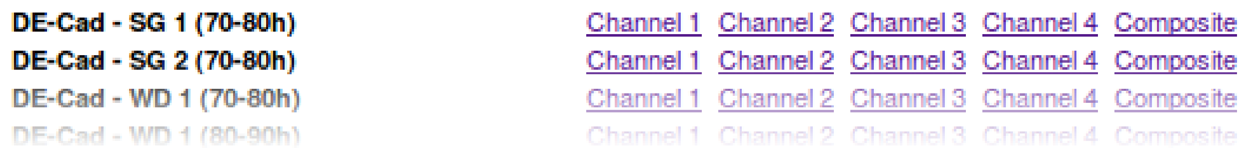
Project table

While the *Project List* presents all the stacks of a project as links below each other, the *Project table* will list all stacks in the same row as the project name. Especially when there are many projects, this helps to get an overview of all available image data. It allows the display of images as well, but in a different way than the *Project List* type. Instead of providing one image per project, this data view type will show one image per stack – replacing the stack name. Like the *Project List* type, it won't show a project, if it has no stacks associated. The following options are supported:

Name	Options	Default
`sort`	`true, false`	`true`
`filter_tags`	`A list of tags, e.g. ["TagA", "TagB"]`	`[]`
`catalogue_link`	`true, false`	`true`
`sample_images`	`true, false`	`false`
`sample_mirror_index`	`position of mirror to use`	`0`
`sample_slice`	`slice index, "first", "center", "last"`	`"center"`
`sample_width`	`width in pixel, e.g. 50 or 75`	`orig. width`
`sample_height`	`height in pixel, e.g. 50 or 75`	`orig. height`
`show_mirrors`	`true, false`	`false`

Have a look to this section's introduction for an explanation of the `sort`, `filter_tags` and `catalogue_link` options. If images should be displayed, the `sample_images` option should be set to `true`. Since there is one image per stack, an option like `sample_stack` in the *Project list* type, isn't needed. However, with the help of the `sample_slice` option a slice can be selected. Again, this can be done with an integer index or one of the string arguments (mind the quotes!). If the numeric index is out of range, no image will be displayed. Like in the *Project List* view type, the option values `"first"`, `"center"` and `"last"` refer to the first, middle and last slice of each stack. To adjust the (pixel) size of sample image, the options `sample_width` and `sample_height` can be used. If only one of the two is used, the images are scaled proportionally. If a sample image is used, a particular mirror can be selected using the `sample_mirror_index` field. The mirror with the selected position will be used.

If you wanted to display a text-only table, you would actually not need to define anything, but `{}` (because of the defaults) to get something like:

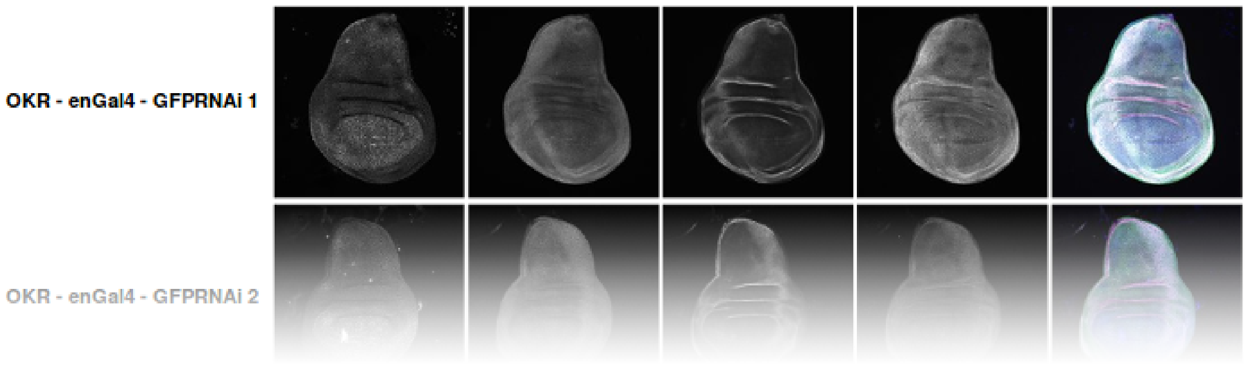


DE-Cad - SG 1 (70-80h)	Channel 1	Channel 2	Channel 3	Channel 4	Composite
DE-Cad - SG 2 (70-80h)	Channel 1	Channel 2	Channel 3	Channel 4	Composite
DE-Cad - WD 1 (70-80h)	Channel 1	Channel 2	Channel 3	Channel 4	Composite
DE-Cad - WD 1 (80-90h)	Channel 1	Channel 2	Channel 3	Channel 4	Composite

However, to get an image table with the center slice of each stack where every sample image has a width of 100px, you would need to define

```
{"sample_images":true, "sample_slice":"center", "sample_width":100}
```

and you would get for example this:



OKR - enGal4 - GFPRNAi 1

OKR - enGal4 - GFPRNAi 2

Project tag table

In CATMAID, projects and stacks can be tagged. This can be done through the admin interface or the tagging tool (see *Tagging*). Based on a *Project tag table* data view type, a data view can create a table where each cell is associated with a *row tag* and *column tag*. Which tag is linked to which row and which tag is linked to which columns can be configured.

To understand the purpose of this data view type, let's look at an example: You have light microscopy image stacks of different tissues. For every tissue there are image stacks for multiple proteins you are interested in. To have the tissue and the protein associated with CATMAID projects, one could just use tags: Every project would be tagged with a tissue name and protein name. To organize the data with the help of the *Project tag table* data view type, one could then assign the tissue tags to the columns and the protein tags to the rows of the resulting table. Each cell of the table refers then to one column (tissue) tag and one row (protein) tag. Each project will appear in a table cell that refers to tags the project itself is tagged with.

However, there are more options than the tags themselves that can be configured:

Name	Options	Default
`sort`	`true, false`	`true`
`filter_tags`	`A list of tags, e.g. ["TagA", "TagB"]`	`[]`
`catalogue_link`	`true, false`	`true`
`row_tags`	`A list of tags, e.g. ["TagA", "TagB"]`	`[]`
`col_tags`	`A list of tags, e.g. ["TagA", "TagB"]`	`[]`
`linked_stacks`	`stack index, "first", "last", "all"`	`"all"`
`force_stack_list`	`true, false`	`false`
`highlight_tags`	`A list of tags, e.g. ["TagA", "TagB"]`	`[]`
`sample_images`	`true, false`	`false`
`sample_mirror_index`	`position of mirror to use`	`0`
`sample_slice`	`slice index, "first", "center", "last"`	`"center"`
`sample_width`	`width in pixel, e.g. 50 or 75`	`orig. width`
`sample_height`	`height in pixel, e.g. 50 or 75`	`orig. height`

Have a look at this section's introduction for an explanation of the `sort`, `filter_tags` and `catalogue_link` options. The tags to use for the rows and columns can be set with the `row_tags` and the `col_tags` keywords. To control which stacks of a project appear in a table cell, the `linked_stacks` and the `force_stack_list` option can be used: They define if a list of stacks is displayed per project (like in the *Project list* type) and which stacks should make it into this list. By default, a complete stack list is displayed for each project. In some situations, however, it is not preferable to list all stacks associated with a project and so you can limit this to either the `first` stack, the `last` or one of a specific index. Is there only one stack (selected or at all), then the link to open it is rendered solely as the project title and no list is displayed. If this is not wanted, the `force_stack_list` option can be set to true to get a list with one entry.

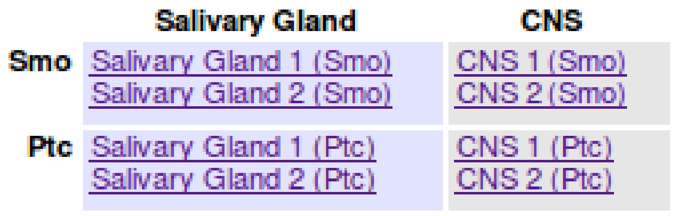
It may be desirable to highlight a group of projects which have the same tags. This can be achieved by providing `highlight_tags`. All projects tagged with at least one of those tags will be displayed in bold.

Like with the other data types, one can opt for showing images instead of stack names. To do so, employ the `sample_images` option. These images will then form links to the actual stack display. With the help of the `sample_slice` keyword, the displayed slice can be selected. Again, one can choose the `"first"`, `"last"` or `"center"` slice of the stack. In case the default size of these sample is too big or too small, the `sample_width` and `sample_height` options can be used. It takes a numerical pixel value and scales the result images accordingly. If a sample image is used, a particular mirror can be selected using the `sample_mirror_index` field. The mirror with the selected position will be used.

As an example, consider the situation described above: We have image stacks of several tissues and with multiple protein markers. The imaged tissues are *CNS* and *salivary gland*. For both of them there are stacks labeled with markers called *Smo* and *Ptc*. The stacks have been labeled accordingly. Also, we only want to consider images tagged as "Valid". Additionally, we don't went to see all the stacks, but only the project name that links to the last stack of each project. A configuration might look like this:

```
{"filter_tags":["Valid"], "row_tags":["Smo", "Ptc"],
"col_tags":["CNS", "Salivary Gland"], "linked_stacks":"last" }
```

With this, the rendered result could look like the following:



	Salivary Gland	CNS
Smo	Salivary Gland 1 (Smo) Salivary Gland 2 (Smo)	CNS 1 (Smo) CNS 2 (Smo)
Ptc	Salivary Gland 1 (Ptc) Salivary Gland 2 (Ptc)	CNS 1 (Ptc) CNS 2 (Ptc)

1.1.15 Similar Software Tools

Interactive skeleton tracing and segmentation

- TrakEM2
- Knossos
- NeuTu
- Reconstruct
- Raveler (*)
- Viking (*)
- Program Elegance (*)
- SSECRETT and Neurotrace (*)
- NeuroStruct (*)
- ESPina (*)

These are all desktop-based tools. (*) are not directly available for download.

Web-based image viewer

- Mouse Brain Architecture Viewer
- Biological Image Viewer Kitware
- iConnectome
- Virtual Fly Brain
- WebImage Browser
- IIPIImage
- Djatoka
- OpenLayers
- JCB Data Viewer

iPad Apps

- WholeSlide

Segmentation Software

- Ilastik (and preview integration with CATMAID)
- Fiji

1.2 Administrator Documentation

Note: These instructions describe installing CATMAID from source for long-term use. If you want to quickly try CATMAID for evaluation or demonstration purposes, consider using the *docker image*.

1.2.1 Basic Installation Instructions

These installation instructions have been tested on Ubuntu 20.04 LTS (Focal Fossa), so may need some minor changes for other Debian-based distributions. For installation on Mac OS X, first read these additional instructions.

Introduction

The most fundamental dependencies of CATMAID are:

1. PostgreSQL 12 and PostGIS 2.5
2. CPython 3.6, 3.7, 3.8, 3.9 or PyPy3.7 (CPython 3.8 is recommended)

To get the required PostgreSQL version for Debian-based systems, such as Ubuntu, you have to add the officical Postgres repository as an extra Apt repository (if you haven't done so already):

```
PG_URL="http://apt.postgresql.org/pub/repos/apt/"
APT_LINE="deb ${PG_URL} $(lsb_release -cs)-pgdg main"
echo "${APT_LINE}" | sudo tee "/etc/apt/sources.list.d/pgdg.list"
sudo apt-get install wget ca-certificates
PG_KEY_URL="https://www.postgresql.org/media/keys/ACCC4CF8.asc"
wget --quiet -O - ${PG_KEY_URL} | sudo apt-key add -
sudo apt-get update
```

While other Python versions are supported, we recommend the use of Python 3.8. To be able to install it on Ubuntu 16.04 and earlier, the following needs to be done:

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt-get update
```

The Django version we use also requires a GDAL version of at least 2.0. The installed version can be checked using `gdalinfo --version`. Should version 2 or newer not be available on your system, use the following PPA:

```
sudo add-apt-repository ppa:ubuntugis/ppa
sudo apt-get update
```

And then you can install these dependencies with:

```
sudo apt-get install python3.8 postgresql-12 postgresql-12-postgis-2.5 gdal-bin
```

CATMAID is based on the Django web framework. If you just wish to work on developing CATMAID, you can use Django's built-in lightweight development server. However, for production you will need to configure your web-server to talk to the CATMAID web application using WSGI. We have successfully tested using either Apache with mod_wsgi, or Nginx with gevent, gunicorn or uSWGI. Particularly Nginx with uWSGI have been tested extensively. Setting up one of these web servers is described in later sections.

1. Clone the repository

The git repository is hosted at https://github.com/catmaid/CATMAID - clone this repository somewhere outside your web root, e.g. in `/home/alice`, so that the source code is in `/home/alice/catmaid`:

```
git clone https://github.com/catmaid/CATMAID.git catmaid
```

2. Install required Python packages

We recommend the use of Python 3.8 or newer for production use. With a few limitations PyPy3 can be used as well (no cropping, no back-end plotting, no synapse clustering, no ontology clustering).

We strongly recommend that you install all Python package dependencies into a virtualenv, so that they are isolated from the system-wide installed packages and can be upgraded easily. Some of these Python packages depend on system-wide libraries that you will need to install in advance, however. You can do this with the following command on Ubuntu (you may need additional PPAs, such as ubuntugis and postgresql):

```
sudo apt-get install postgresql-12 postgresql-12-postgis-2.5 gcc gfortran \
                     git python3.8-dev postgresql-common libpq-dev \
                     libhdf5-serial-dev libboost-dev virtualenvwrapper \
                     libboost-python-dev libxml2-dev libxslt1-dev \
                     libjpeg-dev libtiff-dev libblas-dev liblapack-dev \
                     pkg-config binutils libproj-dev gdal-bin
```

Virtualenv Wrapper needs to source your environment. Start a new terminal or if you are using the bash:

```
source ~/.bashrc
```

Please test if `virtualenvwrapper` is set up correctly, by executing:

```
mkvirtualenv --version
```

If it gives you a version, everything is fine. Otherwise, e.g. if the command `mkvirtualenv` is not found, add the following line to your `~/.bashrc` file and call `source ~/.bashrc` again:

```
source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
```

To create a new virtualenv for CATMAID's Python dependencies, you can do:

```
mkvirtualenv --no-site-packages -p /usr/bin/python3.8 catmaid
```

That will create a virtualenv in `~/.virtualenvs/catmaid/`, and while your virtualenv is activated, Python libraries will be imported from (and installed to) there. After creating the virtualenv as above, it will be activated for you, but in new shells, for example, you will need to activate it by running:

```
workon catmaid
```

Note: Many distributions ship with an outdated version of Pip. This is the tool we use to install Python packages within the virtualenv, so let's update it first:

```
python -m pip install -U pip
```

Note: It is possible to use PyPy as Python implementation, which can improve performance of back-end heavy end-points. Most functionality is available, except for the following: Ontology clustering, Cropping, Synapse clustering, HDF 5 tiles and User analytics. To use PyPy, a new virtualenv using the PyPy executable has to be created:

```
mkvirtualenv --no-site-packages -p /usr/bin/pypy catmaid
```

Note: If you are using Python 3.6 or newer on Ubuntu 14.04 and 16.04, never uninstall Python 3.5, because it might break some parts of the system.

Install all of the required Python packages with:

```
cd /home/alice/catmaid/django
pip install -r requirements.txt
```

If that worked correctly, then the second-last line of output will begin `Successfully installed`, and list the Python packages that have just been installed.

If you set up a production environment, please also install the `requirements-production.txt` file:

```
pip install -r requirements-production.txt
```

This will rebuild some dependencies from source. If asked whether existing packages should be replaced by the new version, answer with yes.

3. Install and configure PostgreSQL

If you are comfortable with creating a new PostgreSQL database for CATMAID, then you should do that and continue to the next section. If you decide to do so, please make sure to also install the `postgis` extension and the `pg_trgm` extension for the new CATMAID database. The advice here is a suggested approach for people who are unsure what to do.

If you are uncomfortable with using the PostgreSQL interactive terminal from the command line, you may wish to install an alternative interface, such as pgAdmin (`sudo apt-get install pgadmin3`) or phpPgAdmin (`sudo apt-get install phppgadmin`).

We suppose for the examples below that you want to create a database called `catmaid` and a database user called `catmaid_user`. Firstly, we need to reconfigure PostgreSQL to allow password-based authentication for that user to that database. To do that, edit the file `/etc/postgresql/12/main/pg_hba.conf` and add this line as the *first* rule in that file:

```
local catmaid catmaid_user md5
```

After saving that file, you need to restart PostgreSQL with:

```
sudo service postgresql reload
```

You can generate the commands for creating the database and database user with the `scripts/createuser.sh` helper script. This takes the database name, the database user and the user's password as arguments and outputs some commands that can be interpreted by the PostgreSQL shell. These can be piped directly to `psql`, so you could create the database and the user with, for example:

```
scripts/createuser.sh catmaid catmaid_user p4ssw0rd | sudo -u postgres psql
```

Besides creating the database and the database user, it will also enable a required Postgres extension, called `postgis`. You should now be able to access the database and see that it is currently empty except for PostGIS relations, e.g.:

```
psql -U catmaid_user catmaid
Password for user catmaid_user:
psql (12.9 (Ubuntu 12.9-2.pgdg20.04+1))
Type "help" for help.

catmaid=> \d
         List of relations
 Schema |       Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | geography_columns | view  | postgres
 public | geometry_columns  | view  | postgres
 public | raster_columns    | view  | postgres
 public | raster_overviews  | view  | postgres
 public | spatial_ref_sys   | table | postgres
```

4. Create the Django settings files

Now you should change into `/home/alice/catmaid/django/` and run:

```
cp configuration.py.example configuration.py
```

You should now edit `configuration.py` and fill in all the details requested. Then you should run:

```
./create_configuration.py
```

This will output some suggested Nginx and Apache configuration in the terminal, and generate the files `django.wsgi` and `settings.py` in `/home/alice/catmaid/django/projects/mysite`. An explanation of all possible settings in the *settings.py* file can be found *here*.

5. Create the database tables

The commands in the following sections are all based on the Django site's admin script `manage.py`, which would be in `/home/alice/catmaid/django/projects`, so these instructions assume that you've changed into that directory:

```
cd /home/alice/catmaid/django/projects
```

Now create all required tables and bring the database schema up to date for applications that mange changes to their tables with South:

```
./manage.py migrate
```

6. Prepare the static files

The static files (mostly Javascript, CSS and image files) that CATMAID requires need to be collected together into `/home/alice/catmaid/django/static` before they will be available. To do this, you need to run:

```
./manage.py collectstatic -l
```

(The `-l` means to create symbolic links to the original location of the files rather than copy them.)

7. Create an administrative user

In order to be able to log in to the CATMAID admin interface, you will need to create a "superuser" account to log in with. You can do this with:

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

8. Optionally add some example projects

If you want to have some example projects to try in your new CATMAID instance, you can create a couple with the following command:

```
./manage.py catmaid_insert_example_projects --user=1
```

(The superuser you just created should have the user ID `1`.)

9. Try running the Django development server

You can run the Django development server with:

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

You should then be able to visit your instance of catmaid at http://localhost:8000. Note though that in its default configuration CATMAID will prevent static files from being served with the `runserver` command and while the website should load it may not look like expected. To temporarily allow this to test without enabling debug mode, set `SERVE_STATIC = True` in `settings.py`. For a production setup, the webserver should take care of serving static files.

10. Setting up a production webserver

You have various options for setting up CATMAID with a production webserver - you can choose from (at least) the following two:

1. *Nginx and uWSGI* (or alternatively with Gevent or Gunicorn)
2. Apache and mod_wsgi, in which case see apache

We prefer to use Nginx because of a more straight-forward configuration, smaller memory footprint and better performance with available WSGI servers. In production setups we made good experience with uWSGI running behind Nginx, which is described in more detail in the *Nginx and uWSGI* section.

Note if the domain you are serving your image data from is different from where CATMAID is running, CORS headers have to be sent by the image server or some aspects of the web front-end won't work as expected. For more details, have a look *here*. The same is true for CATMAID back-ends that should be accessed by clients originating not from the same domain. Check the *CORS setup* section for more details.

In general you want to fine-tune your setup to improve performance. Please have a look at our *collection of advice* for the various infrastructure parts (e.g. webserver, database, file system). This can really make a difference. An explanation of all possible settings in the *settings.py* file can be found *here*.

11. Using the admin interface

You should be able to log in to the CATMAID admin interface and complete administration tasks by adding `/admin/` after the root URL of your CATMAID instance. For example, with the development server, this would be:

```
http://localhost:8000/admin/
```

. . . or, to use the variables used in the `configuration.py` (see step 4), the URL would be:

```
http://<catmaid_servername>/<catmaid_subdirectory>/admin/
```

12. Creating tiles for new CATMAID stacks

You can generate the image tiles for a stack with the `scripts/tiles/tile_stack` script or by exporting from TrakEM2 with its "Export > Flat Images" option and selecting the "Export for web" checkbox. Make the folder with the image pyramid web-accessible and use the URL as `image_base` URL for your stack.

13. Making tools visible

CATMAID offers a growing set of *tools*. To not overload the user-interface, all tools which go beyond navigation are hidden by default. Which tools are visible is stored in a *user profile* for each user. You can adjust these settings at the bottom of the page while editing a user in the admin interface.

1.2.2 Configuration options

A CATMAID instance can be configured mainly through the `settings.py` file, which is located in the `django/projects/mysite` directory. Along with `settings_base.py` (which is not supposed to be edited) the instance configuration is defined. Settings defined in `settings_base.py` can be overridden in `settings.py`. Below is an explanation of all available settings.

`CELERY_WORKER_CONCURRENCY` Controls how many asynchronous Celery workers are allowed to run. This controls how many asynchronous tasks can be processed in parallel.

`NODE_LIST_MAXIMUM_COUNT` The maximum number of nodes that should be retrieved for a bounding box query as it is used to render tracing data. If set to `None`, no limit will be applied which can be slighly faster if node limiting isn't necessary in most cases.

`NODE_PROVIDERS` This variable takes a list of node provider names, which are iterated during a node query as long as no result data is found. The next provider, if any, is only used if the current node provider either doesn't match the request or decides it can't provide a useful answer. An entry can either by a single node provider name (e.g. "postgis2d") or a tuple (name, options) to pass in additional options for a node provider. Possible node provider names are: postgis2d, postgis2dblurry, postgis3d and postgis3dblurry. In addition to these, cache table can be configured, which allows the use of the following node proviers: cached_json, cached_json_text, cached_msgpack.

`CREATE_DEFAULT_DATAVIEWS` This setting specifies whether or not two default data views will be created during the initial migration of the database and is `True` by default. It is typically only useful if the `DVID` or `JaneliaRender` middleware are in use and doesn't have any effect after the initial migration.

`MAX_PARALLEL_ASYNC_WORKERS` Control how many co-processes can be spawned from an async (Celery) worker. This means if `MAX_PARALLEL_ASYNC_WORKERS` is set to `3` and assuming `CELERY_WORKER_CONCURRENCY` is set to `2`, asyncronous processing in CATMAID can be expected to use a maximum of `6` processes.

`DATA_UPLOAD_MAX_MEMORY_SIZE` This option controls the maximum allowed request size that the client application is allowed to send, in bytes. By default this is set to 10 MB. If a request exceeds this limit, error code 400 is returned.

`REQUIRE_EXTRA_TOKEN_PERMISSIONS` To write to CATMAID through its API using an API token, users need to have a dedicated "API write" permission, called "Can annotate project using API token" in the admin UI. To allow users with regular annotate permission to write to the backend using the API, this variable can be set to *False*. The default value is *True*.

`SPATIAL_UPDATE_NOTIFICATIONS` If enabled, each spatial update (e.g placing, updating or deleting treenodes, connectors, connector links) will trigger a PostgreSQL event named "catmaid.spatial-update". This allows cache update workers to update caches quickly after a change. Disabled by default.

`CLIENT_SETTINGS` Can be a JSON string or dictionary that keeps default values for the whole instance for the client settings. Keys of the dictionary are the client-settings values, e.g. "neuron-name-service". The values are the settings values in the format the front-end expects. For instance, to show by default all annotations that are labeled with "neuron name" as textual representation of neurons, this line could be used:

CLIENT_SETTINGS = '{"neuron-name-service": {"component_list": [{"id": "skeletonid", "name": "Skeleton ID"}, {"id": "neuronname", "name": "Neuron name"}, {"id": "all-meta", "name": "All annotations annotated with "neuron name"", "option": "neuron name"}]}}'

By default, no settings are set and this value is *None*.

`FORCE_CLIENT_SETTING` By default, existing client settings won't be replaced if they exist already. To force a replace, set this variable to *True*.

`CMTK_TEMPLATE_SPACES` A list that defines folders with additional CMTK template spaces that can be used with e.g. elmr or the nat.virtualflybrains R packages. Empty by default.

`STATIC_EXTENSION_ROOT` The absolute local path where the static extension files are kept. These can be loaded by the front-end to include custom out-of-source extensions in the front-end (e.g. a custom widget). Defaults to `<catmaid-paith>/django/staticext`.

`STATIC_EXTENSION_URL` The URL under which custom front-end code can be made available. It is expected to map to `STATIC_EXTENSION_ROOT` and is by default set to to `<catmaid-subdir>/staticext/`.

`STATIC_EXTENSION_FILES` A list of file names that are allowed to be loaded by the front-end through `STATIC_EXTENSION_URL` and `STATIC_EXTENSION_ROOT`. Empty by default.

`NEW_USER_CREATE_USER_GROUP` Whether or not a matching user group should be created for every newly created user. This can make user management easier if a lot of shared data access is configured.

`USER_REGISTRATION_CONFIRM_TERMS` Whether or not a user registration form requires users to accept a set of terms and conditions. Disabled by default.

`USER_REGISTRATION_CONFIRM_TERMS_TEXT` The terms and conditions users need to accept upon registering, if this confirmation is required.

`PROJECT_TOKEN_USER_VISIBILITY` If enabled, logged in users will only see other users when they share a project token. Disabled by default.

1.2.3 Administering a CATMAID Instance

This section presents information on how to update a running CATMAID instance and how to backup/restore its database. These administrative tasks might be needed from time to time. Newer versions of CATMAID (obviously) often include bug fixes and new features.

Often times, administrative tasks require CATMAID to be restarted. Generally, it is advisable to announce this to active users. The first section has information on how to see all active connections if `uwsgi` is used as application server.

List active connections

If `uwsgi` is used as WSGI server, you can get an idea how many active connections there are to a CATMAID back-end with the `uwsgitop` tool. It has to be installed separately into the `virtualenv`:

```
pip install uwsgitop
```

Then make sure `uwsgi` is configured to export statistics through a socket file. This can be done by adding the following two lines to the `uwsgi.ini` file for the CATMAID setup:

```
stats = <path-to-run-dir>/uwsgi-stats.socket
memory-report = true
```

Note that `<path-to-run-dir>` needs to be writable by the user running `uwsgi`. Once `uwsgi` is restarted and exports the socket file, it can be queried using `uwsgitop` (make sure to have the `virtualenv` active):

```
uwsgitop <path-to-run-dir>/uwsgi-stats.socket
```

It will list cumulative and current statistics for each worker process, which is useful to gauge the impact of a restart.

Updating to a newer version

Before updating to a newer version, please make sure that CATMAID is not currently running and that you have a backup of your current database state.

You may be asked to upgrade the database in the release notes for the versions between your current and your target version. If this is the case, follow those instructions first (after doing a database backup and stopping CATMAID).

Updating a CATMAID instance involves several steps: First move to your CATMAID instance's root directory. This is also the root of CATMAID's Git repository:

```
cd <LOCAL-CATMAID-PATH>
```

Next, get the source code version you are interested in. The following example will update to the current master branch of Git's "origin" remote:

```
git pull origin master
```

Note that this will merge into your local branch. So if you have local commits that you want to keep you might want to rebase those on origin's master branch using the "–rebase" option. If you are familiar with git branches, you may prefer to switch to the maintenance/RELEASE branch for your target version (if you do this, you'll switch branches again the next time you do an upgrade).

Have a careful look at the required steps to update your current version to the target version. Both `CHANGELOG.md` and `UPDATE.md` provide this information, the latter in a more condensed form. It is also available here. If there are extra steps required, apply them as directed. You should read these for every version after your current version up to your target version.

Then move into the Django sub-directory:

```
cd django
```

Activate the *virtualenv*:

```
source env/bin/activate
```

Update Python packages:

```
pip install -r requirements.txt
```

Should a database upgrade be required (e.g. Postgres 11 to 12 or PostGIS 2.5 to 3), make sure to upgrade PostGIS first, by installing the new version for the current database and connect to every (!) database in the cluster and update the `postgis` extension like this. This can be skipped if no database upgrade is required:

```
$ sudo -u postgres psql
> \c catmaid
> ALTER EXTENSION postgis UPDATE;
> \c <next-database>
> ALTER EXT...
> ...
```

Once done, install the new database version and upgrade the database cluster using `pg_upgrade`. Using the `--link` option can save time on large databases and has been robust in our experience. This step can also be skipped, if no database upgrade is needed.

Synchronize the Django environment with the database:

```
./projects/manage.py migrate
```

Collect new and changed static files:

```
./projects/manage.py collectstatic -l
```

Finally, start the CATMAID process, and visit it in your browser to ensure it is functioning. When done, if you have other work to do on the system, you can close the virtualenv as follows:

```
deactivate
```

Note: Updating PostGIS on your host system could cause CATMAID to stop working. See *here* for how to fix this.

Note: Updating from a CATMAID release before 2015.12.21 (with applied database migrations) requires to update to release 2015.12.21 first, apply all database migrations and then continue with the release you actually want. With the newer version, you have to then fake the initial migration: `manage.py migrate catmaid --fake 0001_initial`.

Backup and restore of the database

There is a dedicated documentation page on this *here <backup>*.

Adding custom code

CATMAID supports adding custom code to its front end. This can be used to create custom tools separate from upstream development, which can make administration easier: To do so, collect your custom JavaScript files in a folder and add their filenames to the `settings.py` array variable `STATIC_EXTENSION_FILES`, for instance:

```
STATIC_EXTENSION_FILES += ('test.js', )
```

Next you will have to instruct your web-server to make this folder available through the URL defined in `STATIC_EXTENSION_URL`, which defaults to "/staticext/"). CATMAID will then try to load those files after its own files.

Performance tuning

There are various application involved to make CATMAID work: A web-server/load balancer, a WSGI server to run the Python back-end and a PostgreSQL database server. The configuration of all of them can be optimized to experience better performance. The following list of suggestions is not exhaustive and if you have suggestions we are happy to hear about them.

Operating system and infrastructure

- In conjunction with the shared memory setting of PostgreSQL (see below), one should increase the kernel's shared memory limit. It defines how much memory can be used as a shared resource by different processes. A rule of thumb is that one should use about 25% of the system's RAM, but if the machine is equipped with plenty of RAM one should be fine for most setups with 4GB (or even less). You can check this kernel setting with `sysctl kernel.shmmax`. The default for most distributions is in the range of kilobytes and megabytes.
- The partition that is hosting the image tiles should be mounted with the `noatime` option. This makes sure no access time is written every time an image file is read. Alternatively, you can use `chattr` to set this option for individual files and folders.
- If LDAP is used to authenticate users and to check permissions on the server CATMAID is running or the image data is loaded from, LDAP queries should be cached locally. Otherwise, an LDAP request will be made every time a file is accessed.
- If the your server has a lot of memory, the Linux kernel defaults for the threshold for writing dirty memory pages to disk are too high (10% of the available memory for start writing out, 20% for absolute maximum before I/O blocks until write-out is done). To avoid large write-out spikes, it is advisable to have the kernel start writing out dirty pages after a lower threshold, e.g. 256MB: `vm.dirty_background_bytes = 268435456`. Also, the threshold for the absolute maximum dirty memory threshold before I/O blocks until the write-out is finished should be lowered, to e.g. 1GB: `vm.dirty_bytes = 107374182`.
- The kernel should also be discouraged from swapping cached data by setting `vm.swappiness = 10`.

Webserver

- The access log should be turned off and only critical errors should be written to the log. CATMAID can produce a lot of requests and writing every single one to disk, especially if multiple users use CATMAID, can be a real performance hit.
- Make use of the HTTP/2 protocol. Modern browsers and webservers support it and it only requires you to set up SSL/TLS as an additional step before activating it. Through multiplexing, compression and prioritization, it makes much better use of single connections. Requests can be answered more quickly and CATMAID will feel more responsive.
- A cache server like Varnish can be beneficial on the machine that serves the image data. If multiple users load the same image data, it will reduce the number of times image data has to be loaded from the hard drive.
- Have the webserver transfer data with GZIP. Make sure this includes JSON data with the content-type `application/json` and binary data with the content-type `application/octet-stream`. In nginx, you can include both by adding `application/json` and `application/octet-stream` to the `gzip_types` setting.
- The CATMAID web-client can send large requests to the server. Increasing the web-server's request buffer can prevent writing such requests temporarily to disk. A buffer of 512kB should be plenty. In Nginx, this can be done with `client_body_buffer_size 512k;`
- Request responses generated by CATMAID can be large as well. Increasing the webserver's buffers to match common response sizes can increase performance quite a bit if the buffer is large enough for the webserver to avoid writing CATMAID's response temporarily to a file and clients have access to a fast connection. For Nginx this means increasing both `proxy_buffer_size` and `proxy_buffers`. The former is used for the response headers only and can be (much) lower: `proxy_buffer_size 64k;`. The latter however defines how many buffers of what size can be used for a single connection. For instance, if the uncompressed (!) response of a typical spatial query for neurons is 1.5-2MB in size, allowing a 2MB proxy buffer per connection would help performance. If you have enough memory available, you could set this with `proxy_buffers 512 4k;` (512 4k pages equals 2MB). Make sure there is enough memory available: for 100 active connections this proxy buffer setting would require already 2GB.
- The webserver should mark image tiles to not expire so that they can be cached by a client. If the image data is public, one could let the webserver also set the `Cache-Control: public` header for the images.
- To not require clients to ask every minute for particular updates (like new messages) use an ASGI server like we describe *here*. This reduces some basline level of requests.

Database management system

- PostgresSQL's shared memory setting should match what is allowed by the kernel. So if you set your kernel to allow 4GB (see above), Postgres should use make use of it. This can be adjusted with the setting `shared_buffers` in `postgresql.conf`.
- Keeping statistics of the CATMAID tables up to date is very important. These statistics are used by the query planer to decide about the optimal realization of a query. This can be done manually by calling `VACUUM ANALYZE` while being connected to the CATMAID database in a psql shell. It is also possible (and advisable) to automate this with by setting `autovacuum = on` in `postgresql.conf`.
- According to the Django manual, Django expects the following parameters for its database connections: `client_encoding: 'UTF8'`, `default_transaction_isolation: 'read committed'` and `timezone: 'UTC'` when `USE_TZ` is True, value of `TIME_ZONE` otherwise (`USE_TZ` is CATMAID's default). All of these settings can be configured in `postgresql.conf` or more conveniently per database user with ALTER ROLE. If these parameters are not the default, Django will do some additional queries to set

these parameters for each new connection. Having those defaults set will improve the database performance slightly.

- We found that making JIT compilation of queries less likely, helps with many spatial queries, where cost estimates aren't very reliable in many cases. This can be done by raising the default value for the `jit_above_cost` (100,000) variable in the `postgresql.conf` file to a value of 1,000,000 or even higher:

```
jit_above_cost = 1000000
```

CATMAID

- Make sure CATMAID is not running in debug mode by checking `settings.py` in `django/projects/mysite`: It should contain `DEBUG = False`. If you get a *Bad Request (400)* response, make sure you have set your `ALLOWED_HOSTS` setting in the `settings.py` file correct.
- Set Django's `CONN_MAX_AGE` option in the database settings of your `settings.py` file, if you don't use a greenlet based threading model for your WSGI server's workers (see here for an explanation). This setting controls how long (in seconds) a database connection can be re-used. In the default configuration, this is set to `0`, which causes every request to use a new database connection. To test if this setting can be used in your environment, set it to a value like `60` and monitor the number of database connections (e.g. with `SELECT count(*) FROM pg_stat_activity;`). If this number matches your number of WSGI workers (plus your own `psql` connection), everything is fine. If the number increases over time, you should set `CONN_MAX_AGE` back to `0`, because new connections are apparently not closed anymore (which can happen with greenlet based threading).
- If database connection pooling is used (see `CONN_MAX_AGE` above), it can help spatial query performance to use prepared statements. These are created for each database connection and pose an overhead without connection pooling. To enable prepared statement add `PREPARED_STATEMENTS = True` to the `settings.py` file.
- Depending on the number of nodes per section, using a different spatial query type can help performance. By default CATMAID uses the so called `postgis3d` node provider as query strategy. This can be changed to the alternative `postgis2d` node provider by adding `NODE_PROVIDER = 'postgis2d'` to the `settings.py` file. It is also possible to cache larger field of views on tracing data and only update this cache periodically. This can improve performance dramatically. Read more about it *here*.
- If there are too many nodes to be displayed with usable performance, the number of returned nodes can be limited. This can be done by setting `NODE_LIST_MAXIMUM_COUNT = <number>` in the `settings.py` file to a maximum number of nodes to be queried (e.g. 20000). If however a node limit is not really needed and most requests don't hit it, setting `NODE_LIST_MAXIMUM_COUNT` to `None` can slightly improve performance, too.
- If neuron reconstruction statistics are slow to compute, consider running the management command `manage.py catmaid_refresh_node_statistics` to populate an optional statistics summary table. Consider running this command regularly over, e.g. over night using Celery or a cron job.
- If large client requests result in status 400 errors, you might need to raise the `DATA_UPLOAD_MAX_MEMORY_SIZE` setting, which is the maximum allowed request body size in bytes. It defaults to 10 MB (83886080).
- Consider using node grid cache for large tracing data set, which can speed up loading and supports level-of-detail as well as dynamic updates based on database events. Automatic cache updates require `SPATIAL_UPDATE_NOTIFICATIONS` to be set to true in `settings.py` (default). If caching is not an option, make sure to set `SPATIAL_UPDATE_NOTIFICATIONS = False` if you deal with large skeletons (>50k nodes) to make operations like joins faster.

Making CATMAID available through SSL

By default the connection between the CATMAID server and a browser is unencrypted. This means data can be read and manipulated on the way between both sides. To protect sensitive data like passwords and to improve security as a whole, it is recommended to use SSL/TLS to encrypt this communication. Below you will find notes on how to do this with Nginx.

The webserver is the first place where the configuration has to be changed. Given that you created a certificate and key file, you would add the following to your Nginx server configuration:

```
server {
    listen 443;
    ...

    ssl on;
    ssl_certificate /etc/nginx/ssl/server.crt;
    ssl_certificate_key /etc/nginx/ssl/server.key;
    ssl_prefer_server_ciphers on;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers
↪"EECDH+ECDSA+AESGCM:EECDH+aRSA+AESGCM:EECDH+ECDSA+SHA256:EECDH+aRSA+SHA256:EECDH+ECDSA+SHA384:EECD
↪aNULL:!eNULL:!MEDIUM:!LOW:!3DES:!MD5:!EXP:!PSK:!SRP:!DSS:!RC4:!SEED";

    ...
}
```

If you refer to certificates and keys in Nginx that it didn't know before, you have to restart it (instead of reloading the configuration). The reason is that the Nginx process drops privileges after loading and root permissions are required to read the certificates and keys.

A good resource to test your configuration and to disable weak ciphers is Qualys SSL Labs.

Django's `settings.py` has to be updated as well to make sure it will only hand out session cookies and CSRF tokens on a secure connection:

```
# This CATMAID instance is served through SSL/TLS. Therefore, send session
# cookies only over HTTPS and don't add CSRF tokens for non-HTTPS connections.
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
# Assume a secure connection, if the X-FORWARDED-PROTO header is set to
# 'https'. This implies that one has to make sure this head is only set to
# 'https' if the connection is actually secure.
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')
```

Please make also make sure that you override the `X-Forwarded-Proto` header passed to Django. It should only contain "https" if the connection is actually secure. Consult the Django documentation to read more about this.

With this you should be able to provide a secure connection to your CATMAID server.

1.2.4 Setting Up Nginx for CATMAID

We made good experience using Nginx together with uWSGI and this setup will get be explained in more detail below. CATMAID will of course work with other web-servers and UWSGI servers as well. Further down some information on alternative setups using Gevent or Gunicorn are briefly discussed.

The installation instructions provided here, assume that you have set up the database and Django as described in the *standard installation* instructions.

Setup based on Nginx and uWSGI

uWSGI is a versatile WSGI server written in C, and can serve as the middle layer between Nginx and CATMAID. It works well with connection pooling and communicates efficiently with Nginx.

1. Install nginx, on Ubunto this would be:

   ```
   sudo apt-get install nginx
   ```

2. Being in CATMAID's `virtualenv`, install uwsgi:

   ```
   pip install uwsgi
   ```

3. Create a new configuration file for `uwsgi` called `catmaid-uwsgi.ini` in CATMAID's `django/projects/mysite/` folder:

   ```
   ; uWSGI instance configuration for CATMAID
   [uwsgi]
   virtualenv = <path-to-virtual-env>
   chdir = <catmaid-path>/django
   socket = /var/run/catmaid/uwsgi.socket
   mount = /<catmaid-relative-url>=<catmaid-path>/django/projects/mysite/django.wsgi
   manage-script-name = true
   workers = 2
   threads = 2
   disable-logging = true
   ```

   Important: there should *not* be a trailing slash after the mountpoint `/<catmaid-relative-url>` above. Note also that each thread of each worker will typically have one database connection open. This means Postgres will try to allocate a total of about workers * threads * work_mem (see `postgresql.conf`). Make sure you have enough memory available here.

4. Make sure that the `socket` directory from your `.ini` file (`/var/run/catmaid/` above) exists and is readable and writable by the user that will run `uwsgi`. You now should be able to start uWSGI manually, running it as the current user:

   ```
   uwsgi --ini <catmaid-path>/django/projects/mysite/catmaid-uwsgi.ini
   ```

   Also note that Nginx needs to be able to access the created `socket` file to communicate with uWSGI. Either you run `uwsgi` as the user running Nginx (typically `www-data`) or you give the Nginx user access on the file, e.g. by using a `SetGID` sticky bit on the `socket` folder so that all files created in it have automatically the default group of the Nginx running user assigned (typically `www-data`).

5. Here is a sample nginx configuration file, where `<catmaid-relative-url> = /catmaid` (replace this with `/` if you don't run in a subdirectory). At the end of this chapter you will find a more complete example configuration:

```
server {
  listen 80;
  server_name <CATMAID-HOST>;

  # Give access to Django's static files
  location /catmaid/static/ {
    alias <CATMAID-PATH>/django/static/;
  }

  # Route all CATMAID Django WSGI requests to uWSGI
  location /catmaid/ {
    include uwsgi_params;
    uwsgi_pass unix:///var/run/catmaid/uwsgi.socket;
  }
}
```

Note: To serve static files, Nginx needs execute permission on every directory in the path to those files (`<CATMAID-PATH>/django/static` in example above). To check this, the `namei` command can be very helpful, because it can list permissions for each path component when called like this: `namei -l <CATMAID-PATH>/django/static`.

Also, it is easy to miss, but important that the the relative URL in the `mount` line of the uWSGI configuration in step 3 has to be exactly the same as the uWSGI location block in the Nginx configuration in step 5, including whether there is an ending slash character.

CORS

Serving image data works the same way as serving CATMAID static data. If the image data is going to be accessed from websites that aren't served from the same server, CORS has to be set up. CORS stands for Cross Origin Resource Sharing and manages security of web-clients accessing a particular resource. The same has to be done if a CATMAID API should be accessible by other CATMAID servers.

In its simplest form, this can be done by adding the following line to an Nginx `location` block. It's only this simple though if no basic HTTP authentication is in use:

```
Access-Control-Allow-Origin *
```

Without this header, only a CATMAID instance served from the *same* domain name as the image data will be able to access it. If the image data or CATMAID server should be accessed by CATMAID instances served on other domains, this header is required.

In order to support basic HTTP authentication and allow caching of preflight requests, consider the following more complete example, which can be added to any `location` block:

```
# Require HTTP Bsic Auth, except for OPTIONS requests. This is needed
# for CORS preflight requests. If no HTTP basic auth is wanted, this block
# can be skipped.
limit_except OPTIONS {
        auth_basic "Restricted";
        auth_basic_user_file /path/to/logins;
}

# Allow any origin (be more restrictive if wanted)
add_header 'Access-Control-Allow-Origin' '*' always;
```

(continues on next page)

(continued from previous page)

```
# Credentials can be cookies, authorization headers or TLS client certificates
add_header 'Access-Control-Allow-Credentials' 'true' always;
# What methods should be allowed when accessing the resource in response to a␣
↪preflight request
add_header 'Access-Control-Allow-Methods' 'GET, POST, PATCH, PUT, DELETE, OPTIONS'␣
↪always;
# Access-Control-Allow-Headers response header is used in response to a preflight␣
↪request to indicate which HTTP headers can be used during the actual request.
add_header 'Access-Control-Allow-Headers' 'DNT,X-CustomHeader,Keep-Alive,User-Agent,X-
↪Requested-With,If-Modified-Since,Cache-Control,Content-Type,X-Authorization,
↪Authorization' always;

# Preflighted requests. Headers from above are repeated, because of the
# new context being created due to the return statement (causing above
# headers to not be visible).
if ($request_method = 'OPTIONS' ) {
        # We need to re-add these headers, because the return statement in the if-
↪block causes this to be a different context.
        add_header 'Access-Control-Allow-Origin' '*' always;
        add_header 'Access-Control-Allow-Credentials' 'true' always;
        add_header 'Access-Control-Allow-Methods' 'GET, POST, PATCH, PUT, DELETE,␣
↪OPTIONS' always;
        add_header 'Access-Control-Allow-Headers' 'DNT,X-CustomHeader,Keep-Alive,User-
↪Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,X-Authorization,
↪Authorization' always;
        # Tell client that this pre-flight info is valid for 20 days
        add_header 'Access-Control-Max-Age' 1728000;
        add_header 'Content-Type' 'text/plain charset=UTF-8';
        add_header 'Content-Length' 0;
        return 204;
}
```

Note that this allows any other client to access the API, if added to the regular WSGI `location` block. It also makes sure preflight requests are cached.

Image data

Image data often is supposed to be accessed from many different clients, some of which aren't originating from the same domain name the images are hosted. In order to make this as seamless as possible, CORS needs to be set up (see previous section). A typical tile data location block could look like the example below. There a tile is looked up and if not found, a black default tile is returned instead:

```
location ~ /tiles/dataset/(.*)$ {
  # Try to open path to tile, fallback to black.jpg for non-existent tiles.
  try_files /data/dataset/tiles/$1 /data/dataset/tiles/black.jpg =404;

  expires max;
  add_header Cache-Control public;
  # CORS header to allow cross-site access to the tile data
  add_header Access-Control-Allow-Origin *;
  # Logging usually not needed
  access_log off;
}
```

Besides adding the CORS header, caching is also set to be explicitly allowed, which might be helpful for data that doesn't change often.

Of course, like with other static files, Nginx must be able able read those files and it needs execute permissions on every directory in the path to the image data.

Process management with Supervisor

Depending on your setup, you might use custom scripts to run a WSGI server, Celery or other server components. In this case, process management has to be taken care of as well, so that these scripts are run after a e.g. a server restart. One way to do this is using `supervisord`. We found it to be reliable, flexible and easy to configure with multiple custom scripts. For each program or program group a new configuration file has to be created:

```
/etc/supervisor/conf.d/<name>.conf
```

Such a configuration file can contain information about individual programs and groups of them (to manage them together). Below you will find an example of a typical setup with a uWSGI start script and a Celery start script, both grouped under the name "catmaid":

```
[program:catmaid-app]
command = /home/catmaid/catmaid-server/django/env/bin/uwsgi --ini /opt/catmaid/django/
↪projects/mysite/catmaid-uwsgi.ini
user = www-data
stdout_logfile = /home/catmaid/catmaid-server/django/projects/mysite/uwsgi.log
redirect_stderr = true
stopsignal = INT

[program:catmaid-celery]
command = /home/catmaid/catmaid-server/django/projects/mysite/run-celery.sh
user = www-data
stdout_logfile = /home/catmaid/catmaid-server/django/projects/mysite/celery.log
redirect_stderr = true

[group:catmaid]
programs=catmaid-app,catmaid-celery
```

This of course expects a CATMAID instance installed in the folder `/opt/catmaid/`. The `stopsignal = INT` directive is needed for `uwsgi`, because it interprets Supervisor's default `SIGTERM` as "brutal reload" instead of stop. An example for a working `run-celery.sh` script can be found **:ref:`here <celery_supervisord>`__**. With the configuration and the scripts in place, `supervisord` can be instructed to reload its configuration and start the catmaid group:

```
$ sudo supervisorctl reread
$ sudo supervisorctl update
$ sudo supervisorctl start catmaid:
```

For changed configuration files also both `reread` and `update` are required.

Maintenance mode

A simple way to display a maintenance mode page in case of an unreachable WSGI server can be configured with the help of Nginx. First, a simple HTML error page is made available as named location block. The CATMAID repo includes an example. The main CATMAID entry location block then references the maintenance location in the case of an unreachable upstream server:

```
location / {
  # Handle error pages
```

(continues on next page)

(continued from previous page)

```
  location @maintenance {
    root /home/catmaid/catmaid-server/docs/html;
    rewrite ^(.*)$ /maintenance.html break;
  }

  location /tracing/fafb/v14/ {
    error_page 502 503 504 @maintenance;
    include uwsgi_params;
    uwsgi_pass catmaid-fafb-v14;
    expires 0;
    # Add optional CORS header
  }
}
```

1.2.5 Example configurations

This shows a more complete example configuration that we have used in a similar form in production, including support for WebSockets (`/channels/` endpoint). The CORS config above is made available as `/etc/nginx/snippets/cors.conf` and included in the CATMAID config:

```
server {
  listen 443 ssl http2;

  server_name <CATMAID-HOST>;

  ssl_certificate <CERT-PATH>;
  ssl_certificate_key <CERT-KEY-PATH>;

  # Force browsers to keep using https instead of http
  add_header Strict-Transport-Security "max-age=604800";

  location ~ /tiles/dataset/(.*)$ {
    # Try to open path to tile, fallback to black.jpg for non-existent tiles.
    try_files /data/dataset/tiles/$1 /data/dataset/tiles/black.jpg =404;

    expires max;
    add_header Cache-Control public;
    # CORS header to allow cross-site access to the tile data
    add_header Access-Control-Allow-Origin *;
    # Logging usually not needed
    access_log off;
  }

  location /path/to/catmaid/static/ {
    alias /home/catmaid/catmaid-server/django/static/;
  }
  location /path/to/catmaid/files/ {
    alias /home/catmaid/catmaid-server/django/files/;
  }
  location /path/to/catmaid/channels/ {
    proxy_pass http://catmaid-fafb-v14-asgi/channels/;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
```

(continues on next page)

(continued from previous page)

```
    proxy_redirect     off;
    proxy_set_header   Host $host;
    proxy_set_header   X-Real-IP $remote_addr;
    proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Host $server_name;
  }
  location /path/to/catmaid/ {
    error_page 502 503 504 @maintenance;

    # Use default uWSGI params and unix socket
    include uwsgi_params;
    uwsgi_pass unix:///var/run/catmaid/uwsgi.socket;

    # No caching
    expires 0;
    add_header Cache-Control no-cache;

    # We need to allow larger requests for our setup (long neuron lists).
    client_max_body_size 20m;

    # Include open CORS config
    include snippets/cors.conf;
  }

  # Handle error pages
  location @maintenance {
    root /home/catmaid/public_html;
    rewrite ^(.*)$ /502.html break;
  }
}
```

In order to makue sure, the in-memory filesystem folder `/run/catmaid/uwsgi.socket` is available after a reboot, the following can be added to `/etc/rc.local`:

```
if [ ! -d /var/run/catmaid ]; then
  mkdir /var/run/catmaid/
  chown www-data:www-data /var/run/catmaid/
  chmod 755 /var/run/catmaid/
fi

# Make sure to exit rc.local with success code
exit 0
```

The following uWSGI configuration allows zero-downtime updates and includes a statistics socket for `uwsgi-top`:

```
;uWSGI instance configuration for CATMAID
[uwsgi]
chdir = /home/catmaid/catmaid-server/django/
virtualenv = /home/catmaid/catmaid-server/django/env
pidfile = /var/run/catmaid/uwsgi.pid
chmod-socket = 666
socket = /var/run/catmaid/uwsgi.socket
mount = /path/to/catmaid=/home/catmaid/catmaid-server/django/projects/mysite/django.
↪wsgi
manage-script-name = true
uid = www-data
```

(continues on next page)

(continued from previous page)

```
gid = www-data
#plugins = python
workers = 8
threads = 2
disable-logging = true
master = true

# POST buffering
post-buffering = 8192

# Stats
stats = /var/run/catmaid/uwsgi-stats.socket
memory-report = true

# During deploy, old and new master share the same socket. With vacuum=true,
# old master would delete it during shutdown.
vacuum = false

# CATMAID in started in lazy-apps mode, i.e. each worker has a full copy of
# the code in memory. Workers are managed by a master process (no emperor).
master = true
lazy-apps = true

# Use this file as a flag to indicate the uwsgi process is ready to accept
# connections.  The file can be looked up during deploy, but it has no meaning
# afterwards. Even there, it is not strictly necessary. It's only a safety
# check.
hook-accepting1-once = write:/var/run/catmaid/catmaid.ready ok
hook-as-user-atexit = unlink:/var/run/catmaid/catmaid.ready

# Create two FIFO slots that we can switch between during runtime. A switch is
# done by sending [0,10] to the current FIFO, by default the first (0) is
# selected.
master-fifo = /var/run/catmaid/new_instance.fifo
master-fifo = /var/run/catmaid/running_instance.fifo

# If there is a running instance, terminate it as soon as the first worker is
# ready to accept connections.
if-exists = /var/run/catmaid/running_instance.fifo
  hook-accepting1-once = writefifo:/var/run/catmaid/running_instance.fifo q
endif =

# On start-up, switch from the initial new_instance fifo queue to the
# new_instance queue, by providing the new fifo's index (1) and update the PID
# file (P).
hook-accepting1-once = writefifo:/var/run/catmaid/new_instance.fifo 1P
```

Like in the initial example, the Supervisor config ties all programs together, this time including the ASGI server Daphne:

```
[program:catmaid-server-uwsgi]
directory = /home/catmaid/catmaid-server/django/projects/
command = /home/catmaid/catmaid-server/django/env/bin/uwsgi --ini /etc/uwsgi/apps-
↪available/catmaid-server.ini
user = www-data
stdout_logfile = /var/log/catmaid/catmaid-server.log
```

(continues on next page)

(continued from previous page)

```
redirect_stderr = true
stopsignal = INT

[program:catmaid-server-daphne]
directory = /home/catmaid/catmaid-server/django/projects/
command = /home/catmaid/catmaid-server/django/env/bin/daphne --unix-socket=/var/run/
↪catmaid/daphne.sock --access-log - --proxy-headers mysite.asgi:application
user = www-data
stdout_logfile = /var/log/catmaid/daphne-server.log
redirect_stderr = true

[program:catmaid-server-celery]
directory = /home/catmaid/catmaid-server/django/projects/
command = /home/catmaid/catmaid-server/django/env/bin/celery -A mysite worker -l info␣
↪--pidfile=/var/run/catmaid/celery.pid
user = www-data
stdout_logfile = /var/log/catmaid/celery.log
redirect_stderr = true

[program:catmaid-server-celery-beat]
directory = /home/catmaid/catmaid-server/django/projects/
command = /home/catmaid/catmaid-server/django/env/bin/celery -A mysite beat -l info --
↪pidfile=/var/run/catmaid/celery-beat.pid --schedule=/var/run/catmaid/celery-beat-
↪schedule-catmaid-server
user = www-data
stdout_logfile = /var/log/catmaid/celery-beat.log
redirect_stderr = true

[group:catmaid-server]
programs=catmaid-server-uwsgi,catmaid-server-daphne,catmaid-server-celery,catmaid-
↪server-celery-beat
```

1.2.6 Backup and restore the database

Making a backup of CATMAID's Postgres database as well as restoring it, is currently a manual process. Making backups can and should be automated through the operating system though. Also keep in mind that a certain database state is related to a certain source code version. This means after restoring a backup, a database migration might still be needed. To avoid this, reflecting the commit name in the backup name, might be a good idea. In most situations this shouldn't be a concern though.

There are different ways of backing up CATMAID databases. Below three different options are shown that increase with complexity, but also come with additional benefits like less space requirements or point in time recovery (PITR). The first method is easy to set up and if nothing else, this should be set up as a basic backup strategy.

Modifying the database directly

To avoid database triggers firing during direct database modifications during a backup restore, it can be useful to disable database triggers. The following SQL can be used to disable triggers temporarily:

```
SET session_replication_role = replica;

/* Do your edits */

SET session_replication_role = DEFAULT;
```

Generally though, triggers are wanted and you should have a very good reason to disable them.

Backup of complete databases using `pg_dump`

To backup a CATMAID database (here named `catmaid`) execute:

```
pg_dump -Fc --clean -U <CATMAID-USER> catmaid -f "catmaid-`date +%Y%m%d%H%M`.gz.dump"
```

This produces a file that includes a time stamp in its name (note the backticks!) and that can be used to restore an entire CATMAID instance. The file itself uses a Postgres specific format to improve loading speed and restoration options.

To restore the dumped database into a database named `catmaid` (which would have to be created as described in the basic install instructions, including the database user referenced in the backup):

```
pg_restore -U <CATMAID-USER> -d catmaid catmaid_dump.sql
```

Both commands will ask for the password. Alternatively, you can use the scripts `scripts/database/backup-database.py` and `scripts/database/revert-database.py`, which do the same thing. Those don't ask for a password, but require a `.pgpass` file (see PostgreSQL documentation).

Note that the `pg_dump` command above will not include any Postgres user information. Like explained in the installation instructions, this would have to be created first, before the `pg_restore` command is called. Alternatively, users and other global database objects can be backed up as well in a separate file:

```
sudo -u postgres pg_dumpall --globals-only | gzip -9 > "globals.gz.dump"
```

Which in turn could be restored in a completely new database like this:

```
zcat globals.gz.dump | sudo -u postgres psql
```

Afterwards the above `pg_restore` command can be executed without further action.

Excluding materialized views from backup

Some tables in CATMAID contain data that is procomputed from other tables. These "materialized views" can be omitted from backups and recreated after a backup restore. This reduces the size of backups, but increases the time to reload backups.

If e.g. `-T treenode_edge` is used with `pg_dump`, the `treenode_edge` table is not part of the backup. Without any `-T` option, all tables are exported and no additional steps are required after a restore.

The following tables can be ommitted from a backup (`-T` option with `pg_dump`), because they can be recreated after a backup is restored: `treenode_edge`, `treenode_connector_edge`, `connector_geom`, `catmaid_stats_summary`, `node_query_cache`, `catmaid_skeleton_summary`.

If one or more of these tables isn't part of a backup, it is required to backup the schema separately by using `pg_dump --schema-only`. When restoring, the schema has to be restored first, because the tables not included in the backup need to be created regardless. This command is followed by a `pg_restore --data-only --disable-triggers` of the data dump.

If the `-T` option was used, the following command has to be executed additionally to complete the import:

```
manage.py catmaid_rebuild_edge_table
```

The script `scripts/database/backup-min-database.sh` can be used to export all databases without including the tables mention above. To restore such a backup, four steps are needed. Assuming the database name is `catmaid` (otherwise change the `-d catmaid` parameters), they are:

1. Import the schema, which includes all tables. Make sure the relevant database user exists already, or use the "globals" export file. The target database name is part of the filename and matches the original database:

   ```
   $ sudo zcat catmaid.schema.gz.dump | sudo -u postgres psql -p 5432
   ```

2. Import the data into the new database:

   ```
   $ sudo -u postgres pg_restore -p 5432 -d catmaid --data-only --disable-triggers \
   ```

   -S postgres –jobs=4 /path/to/backups/catmaid.all.gz.dump

3. Analyze the database, for faster restoration of materialzied views:

   ```
   $ sudo -u postgres psql -p 5432 -d catmaid -c "\timing on" -c "ANALYZE;"
   ```

4. Recreate all materializations:

   ```
   $ manage.py catmaid_rebuild_all_materializations
   ```

Automatic periodic backups

A cron job can be used to automate the backup process. Since this will be run as the `root` user, no password will be needed. The root user's crontab file can be edited with:

```
sudo crontab -e
```

The actual crontab file is not meant to be edited directly, but only through the `crontab` tool. To run the above backup command every night at 3am, the following line would have to be added:

```
0 3 * * * sudo -u postgres pg_dump --clean catmaid -f "/opt/backup/psql/catmaid_
↪$(date +\%Y\%m\%d\%H\%M).sql"
```

This creates a new file in the folder `/opt/backup/psql` at 3am every night. It will fail if the folder isn't available or writable. The file name includes the date and time the command is run and will look like `catmaid_201509101007.sql`. Because `cron` treats `%` characters differently, they have to be escaped when calling `date`). The first five columns represent the date and time pattern when the command (`sudo -u postgres ...`) should be run. It consists of *minute*, *hour*, *day of month*, *month* and *day of week* with asterisks meaning *any*. For more information see the manual pages of `cron` and `crontab`. Because this command is run as *root* and the actual `pg_dump` call is executed as *postgres* user with the help of `sudo`, no database password is required. If your actual backup command gets more complicated than this, it is recommended to create a script file and call this from cron.

1.2.7 Node providers

CATMAID uses so called *node providers* query tracing data from the database. Depending on the query there are different query strategies that can be useful. By default CATMAID uses a node provider called `postgis3d`, which uses a 3D bounding box query on a PostGIS representation of the tracing data. This works well with very dense data and seems to be a good average. The `NODE_PROVIDERS` setting in `settings.py` gives access to the behavior and allows to configure different node providers along with constraints that would mark them valid or invalid for a particular request. This allows to e.g. use a caching node provider that doesn't update immediately for large field of views. For smaller field of views a regular `postgis3d` node provider might be used. These are the available node providers:

`postgis2d` Nodes are queried using a PostGIS 2D index. This seems fast for sparser data and large field of views.

`postgis2dblurry` Like `postgis2d`, but one intersection test less, which includes more false positives for a given bounding box. It is however also faster.

`postgis3d` Nodes are queried using a PostGIS 3D index. This seems fast for denser data and smaller field of views.

`postgis3dblurry` Like `postgis3d`, but one intersection test less, which includes more false positives for a given bounding box. It is however also faster.

`cached_json` A cached version of the data for a given section using the `node_query_cache` table. It is stored as JSON database object.

`cached_json_text` A cached version of the data for a given section using the `node_query_cache` table. It is stored as JSON text string.

`cached_msgpack` A cached version of the data for a given section using the `node_query_cache` table. It is stored as msgpack encoded binary database object.

Cached node queries

The caches for use with the last three entries can be populated using the following management command:

```
manage.py catmaid_update_cache_tables
```

It allows to populate cache data for the formats `json`, `json_text` and `msgpack` based on a set of parameters. It can be run for all projects or a subset. A typical call could look like this:

```
manage.py catmaid_update_cache_tables --project_id 1 --type msgpack --orientation xy -
↪-step 40 --node-limit 0
```

The `--step` parameter sets the section thickness, i.e. Z resolution in `xy` orientation. A `--node-limit` of 0 will remove any existing node limits. The type `msgpack` turned out to be the fastest one in our tests so far.

It makes sense to automate this process to run once every night. This can be done with a cron-job or with predefined *Celery* tasks, which can be added to `settings.py` like this:

```
CELERY_BEAT_SCHEDULE['update-node-query-cache'] = {
  'task': 'update_node_query_cache',
  'schedule': crontab(hour=0, minute=30)
}
```

This would require Celery Beat to run. If it does, it would update all caches defined in `NODE_PROVIDERS` every night at 00:30.

Using multiple node providers

It is possible to define multiple node providers that are valid in different or the same situation. An example could look like this:

```
NODE_PROVIDERS = [
    ('cached_msgpack', {
        'enabled': True,
        'min_width': 200000,
        'min_heigth': 120000,
        'orientation': 'xy'
    }),
    ('postgis3d', {
        'project_id': 2
    }),

    # Fallback
    'postgis2d'
]
```

For an incoming request, CATMAID will first find all valid node providers, depending on e.g. the project ID, or bounding box of the query. It will then iterate this list and return results from the first node provider that returns results. The following options are available for all node providers:

`enabled` Whether the node provider can be used at all.

`project_id`: For which project this node provider can be used.

`orientation` For which orientation this node provider can be used.

`min_width` Which minimum width the query bounding box must have for this node provider (in project coordinates).

`min_height` Which minimum height the query bounding box must have for this node provider (in project coordinates).

`min_depth` Which minimum depth the query bounding box must have for this node provider (in project coordinates).

`max_width` Which maximum width the query bounding box can have for this node provider (in project coordinates).

`max_height` Which maximum height the query bounding box can have for this node provider (in project coordinates).

`max_depth` Which maximum depth the query bounding box can have for this node provider (in project coordinates).

1.2.8 Permissions and access control

There are multiple levels of user permissions available in CATMAID. All of them are configured from within the admin interface, reachable by appending `/admin` to your regular CATMAID URL and opening it in a browser.

Permissions can be given to either users or groups. Users can be members of multiple groups and using them makes user and permission management often a little bit easier.

Project access and visibility

Users can access projects through either the top menu bar, a front page (data view), a deep link or directly through the API. Which projects users can see and access by these means is determined by whether they have `can-browse` permission on those projects.

To view and change project permissions, open a project in CATMAID's admin view and click "Object permissions" in the upper right corner. There it is possible to add either individual users or groups to the various permissions that exists for projects (including `can-browse`).

If users or groups have `can-annotate` permissions they are allowed to create new data (e.g. neuron reconstructions or ontology classifications) in a project. This permission also makes it possible to edit data that was created by other users. However, for this another test comes into play as well and not every user can edit the data of all other users. See next section for more details.

The `can-administer` permission provides access to additional group management and user analysis tools, mainly useful for group managers. It will add additional tools to the statistics widget in the web-client.

The anonymous user is with respect to project visibility a special case. If a project should be publicly visible, the anonymous user needs to have `can-browse` permissions on the project, but another permission is needed as well: in the anonymous user's own user settings the general "catmaid | can browse projects" setting has to be assigned. With this, the anonymous user acts just like a regular user and can be assigned project specific `can-browse` and `can-annotate` permissions.

In total there are seven different permissions CATMAID understands with respect to projects:

Permission	Description
`can_administer`	Edit various project wide settings and view user statistics.
`can_annotate`	Write to a project by creating e.g. tracing data or annotations.
`can_browse`	Read data from the project, needed see the project and look at it.
`can_import`	New data can be imported into a project, either from files or remote sources.
`can_queue_compute_task`	Tasks like large project data exports can be started.
`can_annotate_with_token`	Write operations are allowed through the API.
`can_fork`	Personal copies of projects (only stacks) can be created.

By default, no permission is assigned. All of them can be assigned to users or groups like described above though.

Inactive users

Users who are marked as inactive won't be able to log into CATMAID. Once reactivated, these user accounts are fully functional again. By default, users have to be marked inactive manually. It is however also possible to automatically deactivate user accounts after a specified time. If *Celery and Celery Beat* are set up, a periodic task will check every night if user accounts need deactivation.

Which user accounts get deactivated is configured based on which user groups they are in. Group Inactivity Periods are objects that associate a user group with a maximum inactivity time as well as an optional message and (internal) comment. During activity check, the last login of a user is looked up and compared against the specified maximum inactivity time. If the last login exceeds this interval, the user account is made inactive. These Group Inactivity Periods can be managed in the respective admin view.

For each Group Inactivity Period contact, users can be specified, who are displayed to users who have been inactivated due to this mechanism try to login. A message displayed to them explains the situation and how long the inactivity period is they violated.

Editing data of other users

Users with `can-annotate` permission on a project can create new data and by default, users can't edit data of each other. However, they are allowed to to so if they are member of a group with a name matching the data creator's login name. So user A can edit user B's data if user A is member of a group named B. Superusers can edit data of everyone.

Additionally, there is the special case of imported data. When skeleton data is imported from a remote source, CATMAID will try to keep the original creator and editor information in tact. With the rules above, this can possibly have the effect that the user who imported a skeleton can't edit it or even remove it, if it was imported by accident. This happens if the importing users has no permission over data of the users who originally created the imported skeleton. To avoid this, there is the following extra rule: a user who imported a node or skeleton has full permissions over it, just as if it was their own. No permissions are granted this way on the imported data of other users. However, if user A has imported data and user B has permission on that user's data (by being in user A's user group), user B will also have permissions over the imported data.

1.2.9 Installation of the Celery task queue

Some tasks of CATMAID can be somewhat time consuming and don't fit into the regular request-response cycle, e.g. cropping or statistics aggregation. These tasks are run asynchronously by the task queue Celery. CATMAID doesn't talk directly to Celery, but uses a so called message broker, which Celery talks to to get new tasks. Different brokers are supported by Celery and a popular choice is RabbitMQ. Like most of the CATMAID server configuration, Celery can be configured through the `settings.py` file.

Note that Celery or the message broker don't need to be running to run CATMAID in general. This only prevents certain functionality (e.g. cropping) from working. If however a message broker is around, CATMAID will accept tasks, which will get executed once Celery is started.

Below you find information on how to setup Celery and RabbitMQ in the context of CATMAID. Since the message broker is the part that CATMAID talks to, it is configured first.

Prerequisites

The *basic CATMAID setup* needs to be completed before configuring Celery and RabbitMQ (or any other message broker). This should already install Celery as a dependency.

RabbitMQ as message broker

It is the so called message broker that takes tasks and tells Celery to execute them. There are several options available and we focus on RabbitMQ, which has proven fast and reliable for our use cases and can be configured to provide access through Django's admin interface. First, the RabbitMQ server has to be installed:

```
sudo apt-get install rabbitmq-server
```

This should also start the server automatically. RabbitMQ comes with a plugin infrastructure and one particular useful plugin is one that adds support for management commands. Based on this one is able to get information on Celery workers through the broker from within Django's admin interface. To enable it, call:

```
sudo rabbitmq-plugins enable rabbitmq_management
```

If the `rabbitmq-plugins` binary is not available in your environment, check the typical installation directory and run the tool from there:

```
/usr/lib/rabbitmq/lib/rabbitmq-server-<RABBITMQ-VERSION>/sbin/
```

After enabling or disabling plugins, RabbitMQ has to be restarted:

```
sudo service rabbitmq-server restart
```

To display a list of all available plugin and whether they are enabled, call:

```
sudo rabbitmq-plugins list
```

This also enables a web-interface will be available on port `15672`. The default user and password combination is guest/guest. Of course it is advisable to change these login credentials.

This should be all to have RabbitMQ running. Next, we need to add a user and virtual host on the RabbmitMQ server, which adds to security and makes it easier to run multiple isolated Celery servers with a single RabbmitMQ instance:

```
sudo rabbitmqctl add_user catmaid_user catmaid_pass
sudo rabbitmqctl add_vhost catmaid
sudo rabbitmqctl set_permissions -p catmaid catmaid_user ".*" ".*" ".*"
```

Now we can configure Celery to talk to this message broker.

Note: To troubleshoot whether messages get received and consumed, it is useful to enable the management plugin like shown above and then download from the local RabbitMQ server the rabbitmqadmin tool. This allows to interact with the server and its queues from the command line:

```
wget http://localhost:15672/cli/rabbitmqadmin
chmod +x rabbitmqadmin
./rabbitmqadmin list queues vhost name node messages message_stats.publish_details.
↪rate
./rabbitmqadmin -f long -d 3 list queues
./rabbitmqadmin get queue=<queue-name> requeue=true
```

Celery configuration

The configuration of celery and the message broker happens in the `settings.py` file. To tell Celery where to expect which broker, the `CELERY_BROKER_URL` is used. If the default RabbitMQ port was not changed (5672) and together with the previously created user and virtual host, the broker URL looks like this:

```
CELERY_BROKER_URL = 'amqp://catmaid_user:catmaid_pass@localhost:5672/catmaid'
```

If the defaults don't work for you, you can read more about the format here.

To specify how many concurrent tasks Celery should execute, you can use the `CELERY_WORKER_CONCURRENCY` variable. It defaults to the number of CPU cores, but if you would want to limit it to e.g. a single process, set:

```
CELERY_WORKER_CONCURRENCY = 1
```

There are many more configuration options, but these two are the two central ones in our context. You can find a list of all options along with their description in the Celery documentation. Note that for CATMAID all options have to have the prefix `CELERY_` and have to be upper case. Also, CATMAID currently doesn't need a result back-end.

In a production environment you'll want to run the worker in the background as a daemon, but for testing you should be able to to start the Celery worker like this:

```
celery -A mysite worker -l info
```

To run Celery as a daemon, you have to integrate in your process management system. The section discussing *Supervisord* for process management includes an example on how to do this for Celery and an actual start script for Celery is shown *below*. Also, make sure that this Celery daemon process has the permissions to write to the temporary directory (`TMP_DIR`).

Message broker access from admin panel

To collect worker events, one has to start Celery workers with the `-E` flag, e.g.:

```
celery -A mysite worker -l info -E
```

All tasks will then be manageable from with Django's admin interface.

Periodic Tasks

The Celery infrastructure can also be used to execute tasks periodically. To do so, both a *Celery worker* (see above) and the *Celery beat scheduler* have to be started. The scheduler can be run like this:

```
celery -A mysite beat -l info
```

The Celery documentation has to say a lot mor about this, but in general periodic tasks are taken from the `CELERY_BEAT_SCHEDULE` setting. CATMAID includes two default tasks that are configured to run every night, if enabled:

```
At 23:30 Cleanup cropped image stacks
At 23:45 Update project statistics
```

Like said earlier, to actually execute these tasks, both a Celery worker and a Celery beat scheduler have to be running. If you in fact use these tasks, you may also want to disable the automatic removal of cropped images with every download by setting:

```
# Disable automatic clean-up of the cropping tool
CROP_AUTO_CLEAN = False
```

Both tasks above are defined in CATMAID's `settings_base.py` file. New tasks can be added by adding new entries to the `CELERY_BEAT_SCHEDULE` dictionary in the `settings.py` file. For instance, to print the number of available CATMAID projects once a minute, the following could be added to `settings.py`:

```
from celery import shared_task
from celery.schedules import crontab

@shared_task(name='print_project_count')
def print_project_count():
  from catmaid.models import Project
  n_projects = Project.objects.count()
  return 'Number of available projects: {}'.format(n_projects)

CELERY_BEAT_SCHEDULE['print-project-count'] = {
  'task': 'print_project_count',
  'schedule': crontab(minute='*/1')
}
```

To specify when and how often the task should be run, `datetime.timedelta` can be used as well . Other tasks can be defined in a similar fashion.

Besides defining the tasks themselves, the scheduler also requires write permissions to the `projects/mysite` directory. By default it will create there a file called `celerybeat-schedule` to keep track of task execution. To adjust this file name and path of this file, use the `--schedule` option for Celery beat.

Supervisord

Supervisord is a process management tool which makes setting up processes very easy. This documentation talks *here* in detail about it. A script that can be used with the example provided there would look like this (`run-celery.sh` in the example):

```
#!/bin/bash

# Virtualenv location
ENVDIR=/path/to/catmaid/django/env
# Django project directory
DJANGODIR=/path/to/catmaid/django/projects
# Which settings file should Django use
DJANGO_SETTINGS_MODULE=mysite.settings

echo "Starting celery as `whoami`"

# Activate the virtual environment
cd $DJANGODIR
source $ENVDIR/bin/activate
export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
export PYTHONPATH=$DJANGODIR:$PYTHONPATH

# Run Celery
exec celery -A mysite worker -l info
```

1.2.10 WebSockets and ASGI

WebSockets allows CATMAID instances to push data to the client even if the client didn't request it. This can be useful to inform the client about updates on the server. An example is the notification about new messages, created e.g. by cropping a sub-volume out of the image data.

Without WebSockets, the CATMAID web front-end will ask the server for updates every minute. Avoiding these requests especially with many clients lowers the resource requirements of CATMAID. ASGI is the protocol used for this bi-directional communication and a separate ASGI server (the asynchronous sibling of WSGI). A common choice for this server is Daphne. It is used below for an example setup.

Note that `manage.py runserver` supports ASGI out of the box. This however is not meant to be used for production setups.

All dependencies for this setup are optional and can be installed with:

```
pip install -r django/requirements-async.txt
```

Setting up an ASGI server

Daphne is already installed as part of CATMAID's dependencies. To run it use the following command:

```
daphne -b 127.0.0.1 -p 8001 mysite.asgi:application
```

This will start a new Daphne server, listening on port `8001` on the localhost network interface. Additionally, worker instance can be started for support with a higher websocket connection volume or low latency async tasks. See the example of the Supervisor/Nginx/Daphne setup in the `Channels` documentation here.

Route ASGI requests to ASGI server

To make the ASGI server available to the client, the public facing webserver has to know about it. Of course it would be possible to replace existing *WSGI* setups altogether and use only Daphne for both ASGI and WSGI. There are many situations where this is impractical and could cause problems. Therefore we recommend to only route ASGI requests to the ASGI server and let everything else be handled by the regular WSGI server.

To make this easier, CATMAID makes all ASGI endpoints available under:

```
<CATMAID-URL>/channels/
```

With this we can tell Nginx (or similar in other webservers) to route all URLs starting with `/channels/` to the ASGI server. This is accomplished by the following `location` block:

```
location /channels/ {
    proxy_pass http://127.0.0.1:8001/channels/;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";

    proxy_redirect     off;
    proxy_set_header   Host $host;
    proxy_set_header   X-Real-IP $remote_addr;
    proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Host $server_name;
}
```

Be sure to use your sub-directory structure as a prefix, if you use any. Also, if you see errors like "`(104: Connection reset by peer) while reading response header from upstream`" make sure to increase the available file handles on the OS level by setting:

```
sysctl -w fs.file-max=65536
```

Make sure to persist this in `/etc/sysctl.conf`. Additionally, allow Nginx more fie handles by setting this on the most outer level in `/etc/nginx.conf`:

```
worker_rlimit_nofile 10000;
```

Use RabbitMQ as back-end

The WebSockets layer needs a back-end to share data between different processes. This happens though a separate database, typically Redis or RabbitMQ. While `channels_redis` is available and well supported, we so far only used `channels_rabbitmq`, because RabbitMQ is also used in other parts of our infrastructure (e.g. *Celery*). If you install RabbitMQ make sure to install at least version 5.8. With previous version we experienced connectivity issues. To have a similar setup, install the `channels_rabbitmq` layer package into the virtualenv:

```
pip install -U channels_rabbitmq
```

If you have installed packages from the dependency list `requirements-async.txt`, the package doesn't need to be installede separately.

Additionally, the folllowing has to be added to `settings.py`:

```
CHANNEL_LAYERS = {
   "default": {
       "BACKEND": "channels_rabbitmq.core.RabbitmqChannelLayer",
       "CONFIG": {
           "host": "amqp://<user>:<pass>@localhost:5672/asgi"
       },
   },
}
```

Adjust the RabbitMQ line above according to your RabbitMQ setup. Details on how to configue a user, password and vhost (asgi) can be found in the CATMAID documentation about *Celery*. You can find more information on this channel layer here.

Process management with Supervisord

Supervisord is used as an example for a process management configuration in other parts of this documentation and so we use it here to show how the above ASGI configuration can be managed alongside the existing Supervisord configuration. This assumes a Supervisor process group named "catmaid" is defined in the following file:

```
/etc/supervisord/conf.d/catmaid.conf
```

Add the following lines to this file, between the last `[program:<name>]` section and the `[group:catmaid]` section:

```
[program:catmaid-daphe]
directory = /opt/catmaid/django/projects/
command = /opt/catmaid/django/env/bin/daphne -b 127.0.0.1 -p 8001 mysite.
↪asgi:application
user = www-data
stdout_logfile = /opt/catmaid/django/projects/mysite/daphne.log
redirect_stderr = true
```

It is also possible to have additional workers help with the work, should there be many ASGI requests. The details for `Supervisord` and `Nginx` can be found in the example setup in the `Channels` documentation here.

1.2.11 Tracing data caching

For setups with a large amount of tracing data it can be helpful to setup intersection caching, especially for larger fields of view (FOV). Such caches store precomputed results to spatial queries for the respective container. There currently two types of node caches (as they are called) available: `node-query-cache` and `node-query-grid-cache`. The former stores intersection results for whole sections and the latter one allows to specify a regular grid for which results will be precomputed. They are generally populated using the Django management command `catmaid_update_cache_tables`. It allows to specify the type of cache using the `--cache` option, which can be `section` or `grid`.

Caches can be automatically updated on data changes as long as the variable `SPATIAL_UPDATE_NOTIFICATIONS` is set to `True` (disabled by default). If caches are not used on a CATMAID instance with a large amount of tracing data, it might make sense to disable this setting to improve the speed of operations like skeleton joins.

Both cache types support level of detail (LOD) configurations. This allows to access cached data in terms of a set of an ordered set of buckets. Being able to access only a part of all caches consistently, allows for quicker browsing of larger data sets and maintaining more detail on lower zoom levels.

All caches can store the data in either simple *JSON text*, as *JSON database object* or a binary *msgpack* representation. The `msgpack` version seems to be the fastest in most situations.

Node Query Cache

For a given orientation this cache stores the intersections with a full section, i.e. all nodes intersecting the plane at a given depth (Z for XY, etc.). To enable this cache for look-up, add a section like this to your `NODE_PROVIDERS` array in the `settings.py` file:

```
('cached_msgpack', {
      'enabled': True,
      'orientation': 'xy',
      'step': 40
}),
```

This will make the back-end look for a msgpack format section cache. If nothing is found, the next available node provider will be checked. Besides `cached_msgpack` there is also `cached_json` and `cached_json_text`. The regular node provider options like `min_x`, `max_z`, etc. are supported as well. Also this entry indicates the cache is only defined for the XY orientation and assumes a section thickness of `40 nm`. These values are also used as defaults for the refresh of the cache.

This cache is fast for large field of views that cover most of a section or has to be limited clearly for limit the node count. Otherwise it can cause too many nodes to be loaded.

To populate the cache, the `catmaid_update_cache_tables` command can be used for instance like this:

```
./manage.py catmaid_update_cache_tables --type msgpack --orientation xy \
    --step 40 --node-limit 0 --min-z 80000 --max-z 8400 --noclean \
    --n-largest-skeletons-limit 1000
```

This will ask interactively for the project you want to the create the cache for. With this done a whole section query for each Z in XY orientation is created. The distance between to sections is set to be `40 nm`. Also, only a range of Z values is computed, which is sometimes useful for testing different configurations. The `--noclean` option ensures CATMAID isn't removing existing cached data. Additionally, only the 1000 largest skeletons are displayed.

There are more options available, which can be read on using:

```
./manage.py catmaid_update_cache_tables --help
```

Node Query Grid Cache

The `grid` cache option for the `catmaid_update_cache_tables` populates a grid made out of cells, each with the same height, width and depth. For each of these cells a separate spatial query is cached, which also allows independent updates. In its simplest form, the cache can be enabled for lookup by adding an entry like the following to the `NODE_PROVIDERS` array in `settings.py`:

```
('cached_msgpack_grid', {
      'enabled': True,
      'orientation': 'xy',
}),
```

This will make node queries look for grid caches for the XY orientation. Like with the section cache, there are options like `min_x`, `max_x`, etc. can be used to limit for which volume the cache should be defined.

To create this cache, the `catmaid_update_cache_tables` management command can be used like this:

```
./manage.py catmaid_update_cache_tables --project=1 --cache grid \
    --type msgpack --cell-width 20000 --cell-height 20000 --cell-depth 40
```

As a result a uniform msgpack encoded grid cache with cells with the dimensions 20um x 20um x 40 nm (w x h x d).

The optional settings parameter `DEFAULT_CACHE_GRID_CELL_WIDTH`, `DEFAULT_CACHE_GRID_CELL_HEIGHT` and `DEFAULT_CACHE_GRID_CELL_DEPTH` allow to define defaults for the above management command.

To speed up the computation, it is possible to provide the parameters `--jobs` and `--depth-step`. With `--jobs` the number of parallel processes can be specified that can be used, which allows parallel cache filling. By default only a single process is used. With `--depth-steps` it is possible to reevaluate the number of cells to look at N `n` times during the run. For instance, using `--depth-steps=2` will do a bounding box query when the process is through with half of the depth dimension (Z for XY orientation). By default only a single bounding box query will be made. Updating the bounding box every 100 sections or so can lead to large improvements in cache cell update times. By default, 10 cache cells are executed per process in a parallel run. This can be adjusted using the `--chunk-size` parameter.

Updating caches

Functionality to update cells automatically is available as well. CATMAID uses Postgres' `NOTIFY/LISTEN` feature, which allows for asynchronous event following the *pubsub* model. To lower the impact on regular tracing operations (especially joining), an insert/update/delete trigger for treenode and connector will execute a conditional trigger function, which is set on CATMAID startup.

These events ("catmaid.spatial-update" and "catmaid.dirty-cache") are disabled by default, because they add slightly to the query time, even if not used. To enable these database events and allow automatic cache updates, set the `settings.py` variable *SPATIAL_UPDATE_NOTIFICATIONS = True*. Once enabled, these events can be consumed by third party clients as well.

Cache updates work by running two additional worker processes in the form of management commands: *catmaid_spatial_update_worker* and *catmaid_cache_update_worker*. The former is responsible for listening to the "catmaid.spatial-update" Postgres event and adds entries to the table *dirty_node_grid_cache_cell* for each intersected cache cell in an enabled grid cache. Upon inserts and updates this table issues the "catmaid-dirty" cache event, which the second management command will listen to. It's its responsibility to update the respective cache cells and remove entries from the dirty table. If single worker processes aren't enough, more workers need to be started.

When treenodes are created, moved or deleted the database emits the event "catmaid.spatial-update" along with the start and end node coordinates. The same happens with changed connectors and connector links. Other processes can use this to asynchronously react to those events without writing to another table or blocking trigger processing in other ways.

Alternatively, it is possible to monitor the `catmaid_transaction_info` table and see which entries caused spatial changes and recompute selectively.

Level of detail

The node query result for either a whole section or a single grid cell is not stores as a single big entry in the cache. Instead it is stored in level of detail (LOD) buckets, each one only allowing a maximum amount of nodes except for the last one, which takes all remaining nodes. This allows requests that make use of this cache declare they are only interested in e.g. 5 nodes per grid cell. With small enough grid size dimensions this allows for a uniform control of reasonable node distributions for each zoom level in the front-end.

To configure LOD relevant parameters during cache constructions the options `--lod-levels`, `--lod-bucket-size` and `--lod-strategy` can be used with the `catmaid_update_cache_tables` management command. The options are optional and have defined defaults.

The first option defines how many LOD levels there should be. By default only one level is defined, which effectively means there are no levels of detail.

The second option defines how many nodes are allowed in every bucket (except the last one). The default here is a bucket size of `500`.

The last option allows to select between the strategies `linear`, `quadratic` and `exponential`. Each one defines a way how the bucket size of every bucket will be computed based on the last one. In linear mode, each bucket has the same size, the one defined with `--lod-bucket-size`. In quadratic mode, the first bucket has the passed in size, the following are computed by multiplying the initial bucket size with `lod-level ** 2`, i.e. the second bucket allows for the square of the initial bucket size. This mode is also the default. In exponential mode, the initial bucket size is multiplied with `2 ** lod-level`, i.e. buckets grow faster.

To create a usable LOD configuration for a grid cache, the command line could look like this:

```
./manage.py catmaid_update_cache_tables --project=1 --cache grid \
    --type msgpack --cell-width 20000 --cell-height 20000 --cell-depth 40 \
    --lod-levels 50 --lod-bucket-size 5 --lod-strategy quadratic
```

This will start with the first level of detail with a bucket of size 5, then 25, 125 and so on up to 12,500 in bucket 50.

The front-end allows to set a "Level of detail" (LOD) value in the tracing layer settings. By default, this is set to "max", which causes all LOD levels to be included. Setting this to 1, will include only the first level. The font-end also allows to map zoom levels to particular LOD levels. This allows flexible zooming behavior with adaptive display limits using cached data.

1.2.12 History and provenance tracking

CATMAID keeps track of all changes to its database tables. If a database row is changed, all old values will be stored in a so called history table together with a time range representing the datas validity. This time period is represented by the half-open interval `[start, end)` for which a row is valid starting from time point `start` and is valid until (*but not including!*) `end`. Keeping track of changes is managed entirely by the database. Besides disabling or enabling history tracking, the only thing Django can change, is providing a label for the current transaction, which is useful to give some semantics to a set of database changes. Currently, all CATMAID tables except `treenode_edge` and a few others are versioned, which can typically be regenerated. CATMAID also keeps track of changes in most non-CATMAID tables, that is the tables used by Django and Django applications we use, except for asynchronous task related Celery and Kombu.

History tables

Each versioned table has a so called history table associated, indicated by the `__history` suffix (e.g. `project` and `project__history`). A convenient view that includes live and history data, is available with the `__with_history` suffix. This is simply a union between both tables. A double underscore is used to minimize collisions with existing names. This history table is populated automatically through database triggers: whenever data in a live table is updated or deleted, the history table will be be updated. It contains a complete copy of every historic version of each row and specifies a time period for its validity. This time period is called "system time" and is represented through the additional `sys_period` column in each history table. This time range spans typically the time of the last edition (or creation) to the time of change. If a live table doesn't store such a start time stamp, a separate 1:1 tracking table, which keeps track of editions, is created and managed. Such tracking tables are named like the original table plus the suffix `__tracking`.

CATMAID's history system has one requirement for tables it keeps track of: a single column primary key has to be used. Extending it to support multi-column primary keys is possible, not needed at the moment.

By default, all tables of CATMAID itself plus the user table (a Django table) are set up to track history. To enable this for other tables (e.g. if new tables are added), the database function `create_history_table( live_table )` can be used. This will create the history table and sets up all required triggers. Likewise, there is a `drop_history_table( live_table )` function, which makes sure a history table and triggers are removed cleanly if this is wanted. The table `catmaid_history_table` keeps track of all currently active history tables.

Transaction log

Each endpoint of the CATMAID API that changes data is supposed to leave a log entry in the transaction log. This way, database changes can be associated with a particular back-end operation. Like explained in the *contributor documentation*, data changing endpoints in `urls.py` are wrapped in a `record_view` decorator, which is parameterized with a label. This label is used by the back-end to find affected tables of a change.

Disabling history tracking

While history tracking is important and in most situations desirable, there are a few situations where it would beneficial to disable it (e.g. some database maintenance tasks, potentially more performance). To do this the setting `HISTORY_TRACKING` can be set to `False`, i.e. add the following line to the `settings.py` file:

```
HISTORY_TRACKING = False
```

With the next restart of CATMAID, history tracking will be disabled. Likewise, it can be enabled again by setting `HISTORY_TRACKING = True` (or removing the line). If the history system is enabled after it was disabled (i.e. database triggers have to be created), all tracking tables are updated to match the live data again.

Schema migration

In case there are schema changes to any of the tracked live tables, the history tables have to be changed as well and triggers have to be regenerated. Every column change of a table has to be reflected in the history triggers and tables. Ideally, this would be implemented with DDL triggers in Postgres, which is currentl only possible using a custom C extension. Because this would make CATMAID harder to install, this history table update involves some manual work when creating database migrations that change table columns. Before the actual migration can happen, the history system has to be disabled:

```
SELECT disable_history_tracking_for_table('<table-name>'::regclass,
       get_history_table_name('<tablel-name>'::regclass));
SELECT drop_history_view_for_table('<table-name>'::regclass);
```

Depending on how the columns are changed, different scenarios are expected to happen:

- If a *column is added*, a new history table column is added.
- If a *column is removed*, the equivalent history table column is removed as well.
- If a *column is renamed*, no copy is performed and the renaming is applied directly to the history table.
- If the *data type of a column changes*, the history table column data should be updated to the new data type as well, if possible. If the original data needs to be preserved, the original history column is renamed (append first free "_n" suffix) and the new column is added. If no information loss is present (e.g. float to double), the original history column should however just be changed without backup to save storage space.

After both the live table and the history table have been updated, history tracking has to be enabled again:

```
SELECT create_history_view_for_table('<table-name>'::regclass);
SELECT enable_history_tracking_for_table('<table-name>'::regclass,
        get_history_table_name('<table-name>'::regclass), FALSE);
```

In addition to column changes on existing tables, operations on whole tables are handled like this:

- If a *table is removed*, the history table and history triggers need to be removed as well:

  ```
  SELECT disable_history_tracking_for_table('<table-name>'::regclass,
          get_history_table_name('<tablel-name>'::regclass));
  SELECT drop_history_view_for_table('<table-name>'::regclass);
  SELECT drop_history_table('<table-name>'::regclass);
  ```

- If a *table is added* and its history should be tracked, history tracking has to be enabled for it. To do this, call `SELECT create_history_table( <tablename>::regclass, <timecolumn>, <txidcolumn> );`, with `<timecolumn>` being an edit reference time and `<txidcolumn>` being a column tracking a row's transaction ID. For most CATMAID tables those parameters are `'edition_time'` and `'txid'`, respectively. If both `<timecolumn>` and `<txid>` are `NULL`, a tracking table will be created automatically. Only providing one of the two is currently not supported. To let CATMAID know if you expect this table to have a history table, add the table to the appropriate list in the `HistoryTableTest` class. This way you can also mark a table as not versioned.
- If a live *table is renamed*, the history table is not renamed automatically, use the function `history_table_name(<tablename>::regclass)` to create the new name.

1.2.13 Data replication

CATMAID is designed for collaborative work, possibly from many different places around the world. The latency that users experience to load data increases with the distance they are away from both the server hosting a CATMAID instance and the source of the image data. In order to speed up read access to the data, both image data and the CATMAID database can be replicated (mirrored) to other locations, that would be closer to the users. Below, both image stack mirrors and database replication is discussed in more detail.

Stack mirrors

Multiple image mirrors can be configured for each stack in CATMAID through the admin interface. All stack mirrors are available to the front-end when loading a particular stack. If its **canary location** is reachable, the stack is used. Otherwise the next one is tried and so on. This makes it possible to have also network internal mirrors for faster access, parallel to slower public hosts.

Database replication

The CATMAID front-end can read data from different servers. This is mainly useful for large amounts of data, e.g. neuron reconstructions, meshes or large connectivity matrices. For writing and reading of most small information, CATMAID will still talk to the primary server though. The replication server (replica) will mirror data from the primary server and update automatically using the physical replication capabilities of Postgres. Still, the replication server needs to run a regular CATMAID instance (possibly through Docker), which effectively is read-only.

Physical replication in Postgres performs a byte-by-byte copy of the database, based on the Write Ahead Log (WAL). Therefore it is important that both the primary server and all replicas run the same Postgres version. CATMAID requires Postgres 12 at the moment.

Reachability and secure communication

Replica servers need to be able to log in to the primary Postgres server. This means there needs to be an open port that makes the primary server reachable from the replicas. This could mean opening a port in the server or network firewall, depending on the setup. If the firewall can be configured to only allow incoming requests from the particular replica IP, then is typically a good idea to do for the sake of security. One other aspect is, that it often reduces server load if non-standard ports are used (to lower random login requests), especially if popular applications like Postgres are involved. For instance, Postgres' default port is 5432, so a pragmatic unassigned public port might be 7432. To allow Postgres to only listen on the local loopback interface and route incoming traffic on port 7432, the following `stream` block can be used in an Nginx configuration on the primary server, after the respective streaming module was loaded:

```
# Enable TCP streaming
load_module modules/ngx_stream_module.so;

stream {
  upstream postgres_db {
    server 127.0.0.1:5432;
  }

  server {
    # Forward this port to the internal Postgres database. This is
    # used for the replication user.
    listen 7432;
    proxy_pass postgres_db;
  }
}
```

In case Postgres should be reachable directly, make sure the `listen_addresses` setting in `postgresql.conf` is set to your network interface correctly. With a port forwarding like above, this is not needed.

With Postgres being reachable from the outside, the primary server can now be configured for replication. Given that data is transmitted to replication servers, it is a good idea to encrypt this traffic. For this a self-signed certificate can be used, but if the server has already a certificate, this can be used as well. The private key and certificate need to readable by Postgres, which is typically run by the `postgres` user. Then SSL can be enabled in the `postgresql.conf` file:

```
ssl = on
ssl_cert_file = '/etc/ssl/postgresql/cert/server.crt'
ssl_key_file = '/etc/ssl/postgresql/private/server.key'
```

Let the point to the respective files on your system and make sure Postgres can read them.

Configure the primary

If replicas can connect to Postgres, they need to login. Use a dedicated replication user for this. Login to Postgres (`sudo -u postgres psql`) and create a new user, enable proper password encryption and set a strong password:

```
CREATE ROLE replication_user WITH REPLICATION LOGIN;
SET password_encryption = 'scram-sha-256';
\password replication_user
```

Next, Postgres needs to be told to use the WAL for replication. Edit `postgresql.conf` and apply the following settings:

```
wal_level = replica
max_wal_senders = 3 # max number of walsender processes
wal_keep_segments = 64 # in logfile segments, 16MB each; 0 disables
```

The `replica` value for the `wal_level` is the default. The number of kept WAL segments should be enough for most setups, but on write heavy setups, it is advisable to also enable *WAL archiving*, using the placeholders `%p` for the full path of the archive file and `%f` for the filename only:

```
archive_mode = on
archive_command = 'rsync -a %p postgres@replica:/opt/postgresql_wal/%f'
```

This copies the WAL segments to a place that has to be accessible by all replicas. If e.g. `rsync` is used like above, make sure to setup SSH access by public key. On most setup, WAL archiving is likely not needed though.

The last step on the primary server is to allow the replication user to login. Add the following line to your `pg_hba.conf` file:

```
hostssl     replication     replication_user     xxx.xxx.xxx.xxx/yy      scram-sha-256
```

Replace `xxx.xxx.xxx.xxx/yy` with the IP of the replica or the subnet of multiple replicas are used.

Finally, Postgres on the primary server has to be restarted.

Configure the replica

With the primary server ready, the replica has to be configured as well. First, the replica Postgres has to be stopped. Edit `postgresql.conf` and give the same (or similar, depending on hardware) configuration as on the primary. This way, this server can act as a failover server of the primary goes down. Also, add the following line:

```
hot_standby = on
```

Next the Postgres data directory (`data_directory` in `postgresql.conf`) will be prepared for the replication. If it contains your `postgresql.conf`, `pg_hba.conf` and/or certificates, make a backup of those files. Now this is done, *delete* all files in this data directory. Warning: this will remove all databases in this Postgres instance!

Now the data of the primary server has to be copied over using `pg_basebackup`. This has to be done as the `postgres` user:

```
sudo -u postgres pg_basebackup -h my.primary.db.xyz -p 7432 \
    -P --checkpoint=fast -U replication_user  -D /var/lib/postgresql/12/main/
```

Assuming `/var/lib/postgresql/12/main/` is our data directory and `my.primry.db.xyz` is the primary database server, listening on port `7432`, this command should ask you for the password of the replication user on the primary and print progress information. This will take a while, depending on your database size, because all the data from the primary server is copied over.

Of course the replica shouldn't allow general write-operations to its replicated databases. Instead it should follow the primary. This is done by adding the following settings to your `postgresql.conf` file:

```
primary_conninfo      = 'host=my.primary.db.xyz port=7432 user=replication_user␣
↪password=<password>'
promote_trigger_file = '/var/lib/postgresql/12/main/primary.now'
#restore_command = 'cp /opt/postgresql_wal/%f "%p"
```

To promote a standby/replica cluster to a primary, create the file `primary.now` in the Postgres data directory (`/var/lib/postgresql/12/main/` in this example). This file needs to be owned by the `postgres` user and the `postgres` group.

This configuration makes Postgres start as a standby (read-only) server. It will automatically contact the primary server to stay up-to-date until it is promoted to a primary using a file like `primary.now` or using tools like `pg_promote`.

Note: In Postgres 11, the setup was done by creating a file named `recovery.conf` in the Postgres data directory with the following content, instead of the `postgresql.conf` changes above):

primary_conninfo = 'host=my.primary.db.xyz port=7432 user=replication_user password=<password>' trigger_file = '/var/lib/postgresql/11/main/primary.now' #restore_command = 'cp /opt/postgresql_wal/%f "%p" standby_mode = 'on'

This file needs to be owned by the `postgres` user and the `postgres` group.

Also, in Postgres 11, if the `trigger_file` is created on the file system, the replica is promoted to a primary.

Now the replica can be started. A line similar to the following should show up in the log:

```
started streaming WAL from primary at FD6/EB000000 on timeline 1
```

On the primary server, replicas should be visible in a query like this:

```
select * from pg_stat_activity  where usename = 'replication_user';
```

This provides a basic replication setup. It might be useful to also look at PITR (backups) of Postgres. This would cause a copy of each WAL file can be created (and also used on the replica). This can be done using the `archive_command` setting on the primary or the replica. The `restore_command` can then be used to actually restore the backup into a data directory. The *database backup documentation page* has more details on this. This will for instance also allow to bring a replica server "up to speed" (get to the current database version) after a longer downtime with the WAL being already removed in the primary.

Testing the setup

In order to get an idea what the replication status of both primary and replica are, the following commands can be used as a starting point.

On the Primary:

```
SELECT * FROM pg_stat_replication;
```

And on the replica:

```
SELECT * FROM pg_stat_wal_receiver;
```

Configure CATMAID

To load all neuron reconstruction data from a replica instead of the primary server, the replica has to be set up as a fully working read-only CATMAID instance. Being read-only doesn't need to be configured separately, because the replica database doesn't allow writes anyway.

With this done, the main CATMAID instance can now be configured to use the replica server for read-only data. To do so, two different aspects have to be configured in the *Settings Widget*. First, the remote CATMAID server has to be added as "Other CATMAID instance". Once this is done, the *Read-only mirror index* setting in the *Tracing Overlay* section can be set to the 1-based index of the entry in the list of other CATMAID instances. With this, the tracing layer will use the respective remote server to load tracing data except for the active node. Empty values, -1 or 0 will disable the use of a database replica.

While the replica's CATMAID front-end isn't meant to be accessed directly, if one wanted to do that, it seems possible by using `memcached` for the session storage and configuring CATMAID the following additional settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.dummy.EmailBackend'

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
        'OPTIONS': {
                # Allow maximum object size of 10MB
                'server_max_value_length': 1024 * 1024 * 10,
        }
    }
}

SESSION_ENGINE = 'django.contrib.sessions.backends.cache'

INSTALLED_APPS += tuple(['nolastlogin'])
NO_UPDATE_LAST_LOGIN = True
```

Should that not work for some reason, a file based session cache might be an option, too:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.file'
```

1.2.14 Postgres administration

This is a collection of some common and uncommen database administration tasks.

Create a read-only database user

This is useful for some independent analysis tasks. Assuming the CATMAID database is called `catmaid` and the read-only user should be called `catmaid_read_only`, login to Postgres and run the following:

```
CREATE ROLE catmaid_read_only WITH LOGIN PASSWORD 'a-very-strong-password'␣
↪NOSUPERUSER INHERIT NOCREATEDB NOCREATEROLE NOREPLICATION VALID UNTIL 'infinity';
\c catmaid
GRANT CONNECT ON DATABASE catmaid TO catmaid_read_only;
GRANT USAGE ON SCHEMA public TO catmaid_read_only;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO catmaid_read_only;
GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO catmaid_read_only;
GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA public to catmaid_read_only;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO catmaid_read_only;
```

Copy data from other (CATMAID) database

Postgres makes it possible to work with other databases directly using its foreign data wrappers. This alles for instance to copy user accounts or tracing data from one CATMAID intance to another on the database level. In order to use data from other databases, one has to create a foreign data wrapper first:

```
CREATE EXTENSION postgres_fdw;
CREATE SCHEMA remote_catmaid_fdw;
CREATE SERVER remote_catmaid FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (dbname '<remote-catmaid-db>', port '5432', host 'localhost');
CREATE USER MAPPING for CURRENT_USER SERVER remote_catmaid
  OPTIONS (user '<catmaid-user>', password  '<catmaid-pass>');
```

With this individual tables (or alternatively all) can be imported into the local database:

```
IMPORT FOREIGN SCHEMA public LIMIT TO (auth_user, auth_group,
  auth_user_group, auth_user_groups, catmaid_userprofile,
  guardian_userobjectpermission, guardian_groupobjectpermission)
FROM SERVER remote_catmaid INTO remote_catmaid_fdw;
```

Now it's possible to select data from the remote source:

```
SELECT * from remote_catmaid_fdw.auth_user;
```

Example of copying user accounts across instances

To check if local users have the same ID/name combinations as remote users one could now do:

```
SELECT rau.*, au.* FROM remote_catmaid_fdw.auth_user rau
JOIN auth_user au
  ON au.id = rau.id
WHERE au.username <> rau.username;

SELECT rag.*, ag.* FROM remote_catmaid_fdw.auth_group rag
```

(continues on next page)

(continued from previous page)

```
JOIN auth_group ag
  ON ag.id = rag.id
WHERE ag.name <> rag.name;

SELECT raug.*, aug.* FROM remote_catmaid_fdw.auth_user_groups raug
JOIN auth_user_groups aug
  ON aug.id = raug.id
WHERE aug.user_id <> raug.user_id OR aug.group_id <> raug.group_id;

SELECT rcup.*, cup.* FROM remote_catmaid_fdw.catmaid_userprofile rcup
JOIN catmaid_userprofile cup
  ON cup.id = rcup.id
WHERE cup.user_id <> rcup.user_id;

SELECT rguop.*, guop.* FROM remote_catmaid_fdw.guardian_userobjectpermission rguop
JOIN guardian_userobjectpermission guop
  ON guop.id = rguop.id
WHERE guop.user_id <> rguop.user_id
  OR guop.object_pk <> rguop.object_pk;

SELECT rggop.*, ggop.* FROM remote_catmaid_fdw.guardian_groupobjectpermission rggop
JOIN guardian_groupobjectpermission ggop
  ON ggop.id = rggop.id
WHERE ggop.group_id <> rggop.group_id
  OR ggop.object_pk <> rggop.object_pk;
```

If these and similar queries return empty, remote data can be imported without conflicts. If results are returned, a pragmatic and reasonably safe option would be to move their IDs into negative number space, e.g:

```
UPDATE auth_user_groups SET id = -id WHERE id IN (
  SELECT aug.id FROM remote_catmaid_fdw.auth_user_groups raug
  JOIN auth_user_groups aug
  ON aug.id = raug.id
  WHERE aug.user_id <> raug.user_id OR aug.group_id <> raug.group_id);
```

To see which users, groups, userprofiles and permissions would be imported, use:

```
SELECT rau.* FROM remote_catmaid_fdw.auth_user rau
LEFT JOIN auth_user au ON au.id = rau.id
WHERE au.id IS NULL;

SELECT * FROM remote_catmaid_fdw.auth_group rag
LEFT JOIN auth_group ag ON ag.id = rag.id
WHERE ag.id IS NULL;

SELECT * FROM remote_catmaid_fdw.auth_user_groups raug
LEFT JOIN auth_user_groups aug
  ON aug.id = raug.id
WHERE aug.id IS NULL
  OR aug.user_id <> raug.user_id
  or aug.group_id <> raug.group_id;

SELECT * FROM remote_catmaid_fdw.catmaid_userprofile rcup
LEFT JOIN catmaid_userprofile cup
  ON cup.id = rcup.id
WHERE cup.id IS NULL;
```

(continues on next page)

(continued from previous page)

```
SELECT * FROM remote_catmaid_fdw.guardian_userobjectpermission rguop
LEFT JOIN guardian_userobjectpermission guop
  ON guop.id = rguop.id
WHERE guop.id IS NULL;

SELECT * FROM remote_catmaid_fdw.guardian_groupobjectpermission rggop
LEFT JOIN guardian_groupobjectpermission ggop
  ON ggop.id = rggop.id
WHERE ggop.id IS NULL;
```

If this matches the expectation, this can now be imported:

```
INSERT INTO auth_user
SELECT rau.* FROM remote_catmaid_fdw.auth_user rau
LEFT JOIN auth_user au ON au.id = rau.id
WHERE au.id IS NULL;

INSERT INTO auth_group
SELECT rag.* FROM remote_catmaid_fdw.auth_group rag
LEFT JOIN auth_group ag ON ag.id = rag.id
WHERE ag.id IS NULL;

INSERT INTO auth_user_groups
SELECT raug.* FROM remote_catmaid_fdw.auth_user_groups raug
LEFT JOIN auth_user_groups aug
  ON aug.id = raug.id
WHERE aug.id IS NULL;

INSERT INTO catmaid_userprofile
SELECT rcup.* FROM remote_catmaid_fdw.catmaid_userprofile rcup
LEFT JOIN catmaid_userprofile cup
  ON cup.id = rcup.id
WHERE cup.id IS NULL;

INSERT INTO guardian_userobjectpermission
SELECT rguop.* FROM remote_catmaid_fdw.guardian_userobjectpermission rguop
LEFT JOIN guardian_userobjectpermission guop
  ON guop.id = rguop.id
WHERE guop.id IS NULL;

INSERT INTO guardian_groupobjectpermission
SELECT rggop.* FROM remote_catmaid_fdw.guardian_groupobjectpermission rggop
LEFT JOIN guardian_groupobjectpermission ggop
  ON ggop.id = rggop.id
WHERE ggop.id IS NULL;
```

In case such imports are performed, it is important to reset the ID sequence coutners for all modified tables if they haven't been set manually to something else already:

```
SELECT setval('auth_user_id_seq', coalesce(max("id"), 1), max("id") IS NOT null) FROM␣
↪auth_user;
SELECT setval('auth_group_id_seq', coalesce(max("id"), 1), max("id") IS NOT null)␣
↪FROM auth_group;
SELECT setval('auth_user_groups_id_seq', coalesce(max("id"), 1), max("id") IS NOT␣
↪null) FROM auth_user_groups;
```

(continues on next page)

(continued from previous page)

```
SELECT setval('catmaid_userprofile_id_seq', coalesce(max("id"), 1), max("id") IS NOT␣
↪null) FROM catmaid_userprofile;
SELECT setval('guardian_userobjectpermission_id_seq', coalesce(max("id"), 1), max("id
↪") IS NOT null) FROM guardian_userobjectpermission;
SELECT setval('guardian_groupobjectpermission_id_seq', coalesce(max("id"), 1), max("id
↪") IS NOT null) FROM guardian_groupobjectpermission;
```

Alterantively, if such a sync operation is happening repeatedly, it can be convenient to set the ID sequences of the target database to a different range, e.g. to start new IDs only with enough headroom to the repeated imports.

1.2.15 Docker

With the help of Docker and Docker-compose it is possible to run CATMAID without much manual setup involved. With Docker alone, CATMAID will be available as demo locally, but no added data is persisted after a restart. With Docker-compose however, it is possible to keep added data. In both variants, a superuser is created by default with the username “admin” and the password “admin”.

CATMAID demo with Docker

If you want to try CATMAID before performing a *complete installation*, a Docker image is available containing a running basic CATMAID installation. Docker is a system for distributing programs, dependencies, and system configuration in *containers* that work like lightweight virtual machines.

After installing Docker, download and run the CATMAID image:

```
docker run -p 8000:80 --name catmaid catmaid/catmaid-standalone
```

Navigate your browser to http://localhost:8000 and you should see the CATMAID landing page. You can log in as a superuser with username “admin” and password “admin”. The Docker image contains a few example CATMAID projects and stacks, but you can add your own through the admin page.

Warning: Make sure you change the default password of the admin user.

Warning: Any users, projects, stacks or annotations you add to the running Docker container will by default be lost when you next run it. To save these changes, you must commit them with docker. However, this is not a best practice for using Docker, and we currently do not recommend the CATMAID Docker image for production use.

Persistence with Docker compose

Using *Docker-compose* is an alternative to the demo mode described above. With Docker-compose, the database, the webserver and CATMAID run in different containers. The database container stores the database outside of the container so it is kept over restarts. To run this setup, first install install Docker-compose:

```
sudo sh -c "curl -L https://github.com/docker/compose/releases/download/1.24.1/docker-
↪compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose"
sudo chmod +x /usr/local/bin/docker-compose
sudo sh -c "curl -L https://raw.githubusercontent.com/docker/compose/1.24.1/contrib/
↪completion/bash/docker-compose > /etc/bash_completion.d/docker-compose"
```

Next clone the `catmaid-compose` repo to a convenient location. Note that by default the database will be stored in this location, too:

```
git clone https://github.com/catmaid/catmaid-docker.git
cd catmaid-docker
```

The database (and static files) will be saved outside of the containers in the folder `volumes`. This allows to optionally create a symlink with this name to a different location for the database.

Run containers:

```
docker-compose up
```

Navigate your browser to http://localhost:8000 and you should see the CATMAID landing page. You can log in as a superuser with username “admin” and password “admin”. The Docker image contains a few example projects, which are added by default. To disable these, set `CM_EXAMPLE_PROJECTS=false` in the `environment` section of the `app` service (in `docker-compose.yaml`) before starting the containers for the first time. This is also the place where database details can be configured.

Additionally, the environment option `CM_IMPORTED_SKELETON_FILE_MAXIMUM_SIZE` can be used to set the maximum allowed import file size in bytes.

Warning: Make sure you change the default password of the admin user.

Start on boot

This is easiest done with `systemd`. Create a new service file, e.g. `/etc/systemd/system/catmaid.service`:

```
[Unit]
Description=CATMAID
Requires=docker.service
After=docker.service

[Service]
Restart=always
ExecStart=/usr/bin/docker-compose -f /home/catmaid/catmaid/docker-compose.yml up
ExecStop=/usr/bin/docker-compose -f /home/catmaid/catmaid/docker-compose.yml stop

[Install]
WantedBy=multi-user.target
```

This still requires manual rebuilds during updates.

Updating docker images

Docker images are not updated automatically. Which images are currently locally available can be checked with:

```
docker images
```

Which images containers are currently running can be seen with:

```
docker ps
```

Depending on whether a standalone docker image or a docker-compose setup is used, updating is done slighly differently.

Standalone docker

If you want to persist changes from the currently running container, you can export the database first:

```
docker exec -u postgres catmaid /usr/bin/pg_dumpall --clean -U postgres > backup.pgsql
```

And if you want to make sure you can go back to the old version, you could commit a new docker images with the current state:

```
docker commit catmaid catmaid:old
```

Before updating the images, make sure to stop the containers using `docker stop catmaid` (if you didn't used `--name` with `docker run`, use the container ID instead of "catmaid").

First update the CATMAID base image:

```
docker pull catmaid/catmaid
```

Then, to update `catmaid-standalone` (regular Docker) use:

```
docker pull catmaid/catmaid-standalone
```

If no previous state should be persisted, the docker container can be started normally again:

```
docker run -p 8000:80 --name catmaid catmaid/catmaid-standalone
```

If you however want to start the new container from a previously saved database dump, set the `DB_FIXTURE` variable to `true` and pipe the backup file to the `docker run` command:

```
cat backup.pgsql | docker run -p 8000:80 -i -e DB_FIXTURE=true --name catmaid catmaid/
↪catmaid-standalone
```

The database will then be initialized with the data from the `pg_dumpall` image in the file `backup.pgsql`, created above. The Docker image will automatically apply all missing database migrations.

Docker-compose

Before updating the docker images, the database should be backed up. The easiest way to do this and also be able to quickly restore in case something goes wrong, is to perform a file based copy of the `volumes` folder after stopping the database. To stop CATMAID and the database, call the following command from the `catmaid-docker` directory (containing the `docker-compose.yml` file):

```
docker-compose stop
```

And then copy the complete `volumes` folder:

```
sudo cp -r volumes volumes.backup
```

Next update your local copy of the `docker-compose` repository:

```
git pull origin master
```

Then update your docker images:

```
docker-compose pull
```

Finally the docker containers have to be built and started again:

```
docker-compose up --build -d
```

In case a newly pulled docker image introduces a new Postgres version, CATMAID's docker-compose start-up script will detect this and abort the container execution with a warning. This warning says that an automatic update of the data files can be performed, but this will only be done if `DB_UPDATE=true` is set in the `docker-compose.yml` file. If you don't see such a warning, the update should be successful. If you see this warning, a few additional steps are required. First `DB_UPDATE=true` has to be added as environment variable of the `db` app in the `docker-compose.yml` file. The docker-compose setup needs then to be rebuilt and run:

```
docker-compose up --build -d
```

After a successful upgrade, the `DB_UPDATE` variable should be set to `false` again, to not accidentally upgrade the data files without ensuring a back-up has been made.

In order to check logs of the running containers, `docker-compose logs` can be useful.

Starting docker-compose as systemd service

If placed in the `/etc/systemd/system` folder, the file `catmaid.service` could look like this and allow the container management through systemd:

```
[Unit]
Description=CATMAID
Requires=docker.service
After=docker.service

[Service]
Restart=always
ExecStart=/usr/bin/docker-compose -f /path/to/catmaid-docker-compose/docker-compose.
↪yml up
ExecStop=/usr/bin/docker-compose -f /path/to/catmaid-docker-compose/docker-compose.
↪yml stop
```

(continues on next page)

(continued from previous page)

```
[Install]
WantedBy=multi-user.target
```

Backup and restore of Docker Postgres databases

In order to create simple `pg_dumpall` backups from the host, the following line executed as a `cron` job, can be used to create backups regularly:

```
docker exec -t <container-name> pg_dumpall -c -U catmaid > /data/backup/catmaid/
↪postgresql/$(TZ='America/New_York' date +%Y-%m-%d-%H-%M)-catmaid-db.sql
```

The `<container-name>` is `catmaid` for the regular `docker` setup, and `catmaid_db_1` for the `docker-compose` setup. Also not that this expectes the folder `/data/backup/catmaid/postgresql/` to be writable.

In order to restore such a backup, a line similar to this can be used:

```
cat <backup.sql> | docker exec -i <container-name>-db-container psql -U catmaid
```

Notes on shared memory in Docker

Due to the low default allowed shared memory in Docker containers (64MB), bigger instances might run into an error similar to this:

```
Traceback (most recent call last):
[...]
psycopg2.OperationalError: could not resize shared memory segment
"/PostgreSQL.909036009" to 70019784 bytes: No space left on device
```

To fix this, the allowed shared memory (which is what Postgres makes heavy use of) can be increased. When running `docker` directly, add the `--shm-size=2g` option. If `docker-compose` is in use, add `shm_size: '2gb'` to the build context of the database (db container). It requires at least v3.5 of the docker-compose file format:

```
version: '3.5'

services:
  build:
    context: db
      shm_size: '2gb' # build
    shm_size: '2gb' # run-time
```

Note that the `db` identifier moves from the `build:` line to the new `context:` line.

For more available shared memory, increase the example of `2gb`.

Building a CATMAID Docker image

Make sure to have a clean working directory, because everything in the CATMAID directory tree will be included in the Docker image. The easiest way to make sure of this is to clone the CATMAID repository into a new folder and move to the branch to build an image for. Then, build the image with the correct tag, login to DockerHub and push it:

```
git clone git@github.com:catmaid/CATMAID.git catmaid-docker
cd catmaid-docker
git checkout -b origin/dev
docker build -t catmaid/catmaid:latest .
docker login
docker push catmaid/catmaid:latest
```

Parameterizing Docker containers

Both the standalone Docker container and the docker-compose setup can be parameterized with various options. Some of them have already been discussed above. Generally, Docker parameters are provided as environment variables. For the regular Docker setup this happens by adding `-e KEY=VALUE` parameters to the `docker run` call. For `docker-compose`, the respective entries have to be added to the `docker-compose.yaml` file. The available settings can broadly be categorized in infrastructure settings (database, webserver) and CATMAID settings.

The following infrastructure settings are available:

DB_HOST The dabase hostname. Default: localhost

DB_PORT The port the database is listening on. Default: 5432

DB_NAME The name of the CATMAID database. Default: catmaid

DB_USER The user as who to connect to the databae. Default: catmaid_user

DB_PASS The password of the database user. Default: catmaid_password. Please change this!

DB_CONNECTIONS The maximum number of allowed database connections. Default: 50

DB_TUNE Whether the contaienr should try to tune the database on initial startup. Default: true

DB_FORCE_TUNE Whether the next start of the container should include a database tuning update. Default: false

DB_FIXTURE Whether or not to expect raw SQL as input on stdin. This can be piped directly to the database. Assuming there is simple database dump with text SQL commands in the file backup.sql, the following command can be used to load it into the container database: `cat backup.sql | docker run -i -e DB_FIXTURE=true --name catmaid catmaid/catmaid-standalone`. Default: false.

INSTANCE_MEMORY The amount of memory, the docker instance should have available. This is the basis for tweaking some database parameters. By default, this is estimated automatically, but can be overridden in terms of megabtes of memory, i.e. a value of 4096 means 4GB.

The following CATMAID settings are available. If anything, the administration password should be changed to something more secure (`CM_INITIAL_ADMIN_PASS`).

CM_INITIAL_ADMIN_USER This admin user is created during initial setup. Default: admin

CM_INITIAL_ADMIN_PASS This initial password of the admin user defined in CM_INITIAL_ADMIN_USER. This should be changed to something more secure! Default: admin

CM_INITIAL_ADMIN_EMAIL This initial email address of the admin user defined in CM_INITIAL_ADMIN_USER. Default: admin@localhost.local

CM_INITIAL_ADMIN_FIRST_NAME The first name of the admin user defined in CM_INITIAL_ADMIN_USER. Default: Super

CM_INITIAL_ADMIN_LAST_NAME The last name of the admin user defined in CM_INITIAL_ADMIN_USER. Default: User

CM_DEBUG Whether or not to run CATMAID in debug mode. Default: false

CM_EXAMPLE_PROJECTS Whether or not to setup example projects. Default: true

CM_INITIAL_PROJECTS A set of project and stack definitions that the container will set up initiall. The expected format is JSON as it is returned by the `/projects/export` API endpoint. This can be a multiline environment variable, but Docker is somewhat picky about how this is provided.

Consider the following JSON representation of a Drosophila larva L1 project, stored in the file `larva-l1-project.json`:

```
[{
  "project": {
    "title": "L1 CNS",
    "stacks": [{
      "title": "L1 CNS",
      "dimension": "(28128, 31840, 4841)",
      "mirrors": [{
        "fileextension": "jpg",
        "position": 3,
        "tile_source_type": 4,
        "tile_height": 512,
        "tile_width": 512,
        "title": "Example tiles",
        "url": "https://example.com/ssd-tiles/"
      }],
      "resolution": "(3.8,3.8,50)",
      "translation": "(0,0,6050)"
    }]
  }
}]
```

This can now be used in the CM_INITIAL_PROJECTS environment variable like this as a `docker run` parameter:

```
-e CM_INITIAL_PROJECTS="$(cat larva-l1-project.json)"
```

Alterantively, such a JSON block could be included also directly into the call on the command line:

```
docker run ... -e CM_INITIAL_PROJECTS='[{
  "project": {
    ...
  }
}]' -e ...
```

CM_INITIAL_PROJECTS_IMPORT_PARAMS The parameter string provided to the `catmaid_import_projects` management command by the importer to import the projects and stacks provided in `CM_INITIAL_PROJECTS`. This can for instance be give the anonymous user read permissions on the imported data:

```
CM_INITIAL_PROJECTS_IMPORT_PARAMS="--permission user:AnonymousUser:can_browse"
```

CM_IMPORTED_SKELETON_FILE_MAXIMUM_SIZE The maximum allowed file size for skeletons that are imported through the API into the container. In Bytes.

CM_HOST The network interface in the container, the CATMAID application server should be listening on. Default: 0.0.0.0 (all interfaces).

CM_PORT The network port in the container, the CATMAID application server should be listening on. Default: 8000

CM_FORCE_CONFIG_UPDATE Whether the CATMAID configurating should be updated on container start. Normally, the settings are updated on initial container start. Default: false

CM_WRITEABLE_PATH Where CATMAID can expect to be able to write data. This can be useful to make this folder accessible through a Docker volume. Default: "/tmp".

CM_NODE_LIMIT The maximum number of reconstruction nodes that should be loaded by a single field of view query. Default: 10000

CM_NODE_PROVIDERS How the back-end node providers should be configured. Default: "['postgis2d']

CM_SUBDIRECTORY The subdirectory relative to the domain root that CATMAID is running in, e.g. "/catmaid". By default, no subdirectory is used ("").

CM_CSRF_TRUSTED_ORIGINS Which servers to trust to bypass CSRF checks. None by default (""). The format is expected to be a Python like list, e.g. '["example.com"].

CM_CLIENT_SETTINGS A JSON string representing a set of client settings that are used as default instance level client settings. Already defined settings take precedence. By default no client settings are provided ("").

This is an example that will set the neuron name rendering to prefer a name set by an annotation that is meta-annotated with "Neuron name":

```
CLIENT_SETTINGS: '{"neuron-name-service": {"component_list": [{"id": "skeletonid",
↪ "name": "Skeleton ID"}, {"id": "neuronname", "name": "Neuron name"}, {"id":
↪"all-meta", "name": "All annotations annotated with \"neuron name\"", "option":
↪"neuron name"}]}}'
```

CM_SERVER_SETTINGS A valid Python string that is added to the container's settings.py file. All specific settings above override these more general settings. For instance, an alternative way to set the node query limit and enabling the cropping tool by default would be:

CM_SERVER_SETTINGS="NODE_LIST_MAXIMUM_COUNT=50000nPROFILE_SHOW_CROPPING_TOOL=True"

CM_FORCE_CLIENT_SETTINGS Normally, the above client settings are only used if there is none already defined for a user. To enforce the use of the CM_CLIENT_SETTINGS settings, this can be set to true. Default: false

CM_RUN_ASGI Whether or not to run the ASGI server Daphne inside the container. This enables support for WebSockets, which add some convenience features on the front-end. This is enabled by default, set `CM_RUN_ASGI=false` to disable.

CM_RUN_CELERY For asynchronous tasks, CATMAID uses Celery. By default a Celery instance is also run inside the Docker container. Since Celery isn't neccessarily required for normal operation (only some operations like cropping or NBLAST won't work) and Celery can also be run in a spearate container, running Celery within the CATMAID container can be disabled by setting this to "false". By default, Celery is started. If Celery is enabled, an asynchronous task scheduler will schedule some maintenance tasks every night (e.g. cleaning up cropped image, updating statistics, etc.).

CM_CELERY_BROKER_URL If Celery is not run within this Container but somewhere else, this variable can be used to let CATMAID know about where to find Celery.

CM_CELERY_WORKER_CONCURRENCY By defeault Celery runs with one worker in the container. This can be adjusted here by setting it to a higher number.

CM_CELERY_TIMEZONE There are a handful of maintenance tasks that are executed by CATMAID every night. By default this happens around midnight in UTC time. The time zone which is used here, can be configured with this variable. Use e.g. 'America/New_York' for US east coast time.

TIMEZONE The timezone this server runs in. By default CATMAID tries to guess. Otherwise see https://en.wikipedia.org/wiki/List_of_tz_zones_by_name.

CORS_OPEN Whether or not the webserver configuration in the container will allow open CORS access, including optional authentication. Defaults to true.

1.2.16 User accounts

User Profiles

To manage per-user settings, in CATMAID each user has a profile attached. These profiles can be adjusted in the admin interface by going to a users settings page. Currently, these settings control access to CATMAID's different tools.

Those settings also have default values which are user for new users and which can be set for all users at once. To adjust those defaults to your use case, add and change the following lines to your Django configuration (likely *settings.py*):

```
PROFILE_SHOW_TEXT_LABEL_TOOL = False
PROFILE_SHOW_TAGGING_TOOL = False
PROFILE_SHOW_CROPPING_TOOL = False
PROFILE_SHOW_SEGMENTATION_TOOL = False
PROFILE_SHOW_TRACING_TOOL = False
PROFILE_SHOW_TEXT_LABEL_TOOL = False
PROFILE_SHOW_ONTOLOGY_TOOL = False
PROFILE_SHOW_ROI_TOOL = False
```

As you can see, by default all tools are invisible. Change `False` to `True` if you want to make a tool available on the toolbar for new users. If you want to use these default settings for existing users, please use this management command:

```
./manage.py catmaid_set_user_profiles_to_default
```

If you add the `--update-anon-user` option, the user profile of the anonymous user will get updated to the current default settings as well. By default, the anonymous user is not updated.

User registration

If `USER_REGISTRATION_ALLOWED = True` in `settings.py`, CATMAID will allow new users to register. The registration page is reachable through the "Register" link in the upper right corner of the CATMAID website.

Optionally, new users can be required to confirm their email address. This requires a working E-Mail setup (`EMAIL_HOST`, `EMAIL_PORT`, `EMAIL_HOST_USER`, `EMAIL_HOST_PASSWORD`, `DEFAULT_FROM_EMAIL`). If this is working can be tested from the Python shell:

```
from django.core.mail import send_mail
send_mail('Test-Mail', 'Test-Message', '<valid-sender>', ['<recipient>'], fail_
↪silently=False)
```

If this works, then confirmation emails can be enabled by setting:

```
USER_REGISTRATION_EMAIL_CONFIRMATION_REQUIRED = True
```

The text of the welcome message can be adjusted using the `USER_REGISTRATION_EMAIL_CONFIRMATION_EMAIL_TEXT` setting. Look at the default in `settings_base.py` for an example.

This confirmation email also requires admins to defined the correct domain for their CATMAID instance in the admin interface by changing the default `example.com Site` entry. Make sure to provide a fully qualified domain like `em.catmaid.org`.

Once users have their email address confirmed, or right after registration, if no confirmation is required, a welcome email can be sent to users. To enable this email, set `USER_REGISTRATION_EMAIL_WELCOME_EMAIL = True`. Its text can be modified as well using the `USER_REGISTRATION_EMAIL_WELCOME_EMAIL_TEXT` setting. Like with the confirmation email, have a alook at the default for an example.

Additionally, it is possible to add a `Reply-To` header in those emails, by setting `USER_REGISTRATION_EMAIL_REPLY_TO` to an email address.

1.2.17 Additional back-ends

CATMAID's front-end can retrieve project and stack information from other services. To do this, the back-end has to be configured slightly differently to re-route API endpoints related to information retrieval for both projects and stacks. In this mode, CATMAID is only useful for image viewing at the moment. There are currently two web services supported, the Janelia Render Service and DVID. Only one of these options can be used at a time.

Janelia Render Service

Add the following to CATMAID's *settings.py* file and adjust URL, default resolution as well as tile dimension to your setup:

```
# Janelia rendering service
MIDDLEWARE += ('catmaid.middleware.JaneliaRenderMiddleware',)
JANELIA_RENDER_SERVICE_URL = 'http://renderer.int.janelia.org:8080/render-ws/v1'
JANELIA_RENDER_DEFAULT_STACK_RESOLUTION = (4,4,35)
JANELIA_RENDER_STACK_TILE_WIDTH = 1024
JANELIA_RENDER_STACK_TILE_HEIGHT = 1024
```

To also see all projects on the front-end (rather than only in the menus), the *simple project list view* has to be be used as a data view. This can be done by either adding it to the list of data views or removing all existing data views, because it is the default fallback. Both is possible from CATMAID's admin view. Projects will be organized by owners.

DVID

A setup similar to the Janelia Render Serivce can be used with DVID. To do so add the following to your *settings.py* file and adjust to your setup:

```
MIDDLEWARE += ('catmaid.middleware.DVIDMiddleware',)
DVID_URL = 'http://emdata2.int.janelia.org:7000'
DVID_FORMAT = 'jpg:80'
DVID_SHOW_NONDISPLAYABLE_REPOS = True
```

1.2.18 R setup for NBLAST and related tools

CATMAID can make use of R to provide support for NBLAST and NRRD export of neurons. To talk to R, CATMAID uses the `rpy2` library, which is an optional dependency. If it isn't found, CATMAID will print a warning that NBLAST support is disabled to the log. Before `rpy2` can be installed, a recent version of R needs to be installed first. The instruction below focus on Ubuntu 16.04, but should be similar for other operating systems.

Installing a recent version of R

Add the R 4.0 repository along with its public keys:

```
sudo gpg --keyserver hkp://keyserver.ubuntu.com:80 --recv-key␣
↪E298A3A825C0D65DFD57CBB651716619E084DAB9
sudo gpg -a --export E298A3A825C0D65DFD57CBB651716619E084DAB9 | sudo apt-key add -
sudo add-apt-repository 'deb https://cloud.r-project.org/bin/linux/ubuntu focal-
↪cran40/'
```

Install R and its development packages as well as two additional dependencies that are required to install all needed R packages:

```
sudo apt-get update
sudo apt-get install r-base r-base-dev mesa-common-dev libglu1-mesa-dev \
                     libssl-dev libssh2-1-dev libcurl4-openssl-dev cmtk
```

After a few moments an R version usable by `rpy2` should be installed and the next steps focus on the CATMAID setup.

Installing CATMAID's R dependencies

With the `virtualenv` activated, install `rpy2`:

```
pip install rpy2
```

With recent R versions, this should succeed without problems. Next, the R packages providing the NBLAST implementation need to be installed. It is improtant to to make these installed R packages accessible to both the active user and the user running the WSGI server or Celery. This can be done by installing the libraries either system wide (`sudo -i R`) or, preferred, to create a shared folder that can be used with R both from the command line and from CATMAID:

```
mkdir <catmaid-path>/django/projects/r_libs
```

The actual location doesn't matter as long as it is accessible by the current user and by the user running CATMAID and Celery. To make R recognize this library folder from the command line, the environment variable `R_LIBS_USER` needs to be set to the new folder:

```
export R_LIBS_USER=<catmaid-path>/django/projects/r_libs
```

If R is started now, it will also use this directory in its list of library paths. To make CATMAID aware of this as well, add the following the `settings.py`:

```
R_LIBS_USER = '<catmaid-path>/django/projects/r_libs/'
os.environ['R_LIBS_USER'] = R_LIBS_USER
```

With this in place, the NBLAST R dependencies can be installed. Since this includes quite a few packages in their development version, one optional step is recommended before we do this though: setting the `GITHUB_PAT` environment variable to a valid Personal Access Token for your GitHub account (if you have one). Such a token can be generated in the "Developer settings" section of the User settings view. With a PAT generated, make it available in the current session:

```
export GITHUB_PAT='<a-random-string-of-letters-and-digits'
```

Like said above, this is optional, but might speed up the installation a bit. With this done, it should be enough to call the following management command to install all needed R dependencies into the new R library folder:

```
manage.py catmaid_setup_nblast_environment
```

If this was successful, CATMAID should now able to talk to R.

Note: If a system-wide installation is used, the installation has to be done manually:

```
$ sudo -i R  # Or just R if R_LIBS_USER is used
> install.packages(c("doMC", "R6", "rgl", "plyr"))
> devtools::install_github(c("natverse/nat", "natverse/nat.nblast",
>         "natverse/rcatmaid", "natverse/fafbseg", "natverse/elmr",
>         "natverse/nat.templatebrains", "natverse/nat.flybrains",
>         "natverse/nat.jrcbrains"))
>
> nat.jrcbrains::download_saalfeldlab_registrations()
> nat.flybrains::download_jefferislab_registrations()
```

If each command was successful, every user on the system should now be able to run CATMAID related R code.

Using NBLAST and other R based tools

The front-end will provide support for NBLAST mainly through the "Neuron Similarity" widget. Everything should work with the default configuration.

R can execute some functionality in parallel (e.g. computing NBLAST scores). The number of parallel processes is set to one by default, but can optionally be configured using the `MAX_PARALLEL_ASYNC_WORKERS` setting, e.g.:

```
MAX_PARALLEL_ASYNC_WORKERS = 4
```

The NRRD access functions need slightly more setup at the moment. In order to use them, a small set of additional variables need to be set in the `settings.py` configuration file. The complete URL of the CATMAID instance is needed:

```
CATMAID_FULL_URL = "https://example.com/catmaid/"
```

And if basic HTTP authentication is in use, the following needs to be set as well:

```
CATMAID_HTTP_AUTH_USER = "<http-auth-user>"
CATMAID_HTTP_AUTH_PASS = "<http-auth-password>"
```

This is not ideal and likely to change in the future, but for now this is needed. For some operations CATMAID has R connect to itself through HTTP.

1.2.19 Configuring NBLAST

After setting up the *R environment* for NBLAST the front-end tools like the Similarity Widget should work. By default these tools will read the most recent skeleton data from the database. For bigger datasets this can introduce some performance problems. Typically, in bigger datasets, only a small portion of skeleton does actually change and a cache can be used for some data.

Creating skeleton caches

Caches are stored in R's binary RDS files in the `cache` directory in the `MEDIA_ROOT` path. At the moment, caches are created either manually, e.g. through the management shell or through a cron job:

```
from catmaid.control.nat.r import create_dps_data_cache
project_id = 1
create_dps_data_cache(project_id, 'skeleton', tangent_neighbors=5, detail=10, min_
↪nodes=100, progress=True)
```

This would create the cache file *r-dps-cache-project-1-skeleton-10.rda*`, following the pattern `r-dps-cache-project-<project-id>-<type>-<detail>.rda`.

Caches can be created for the types `skeleton` and `pointcloud`. The `tangend_neighbords` settings defines how many neighbor points should be used to compute a tangent vector, the default is 20, but 5 often yields good results as well and is faster. The `detail` setting defins the branching level below which skeletons will be pruned. Using the `min_nodes` setting, only skeletons with the respective minimum number of nodes are included. By default, no progress is shown, which can be changed using the `progress` setting.

1.2.20 Setting up OAuth2 providers for login

CATMAID can use other OAuth2 services like GitHub or Orcid.org to sign users in. Internally a regular Django account is created that is linked to the selected remote service. These accounts respect the `NEW_USER_DEFAULT_GROUPS` setting and default tool configuration.

Below there are instructions to set this up with GitHub and Orcid.org. A lot more services are supported, more details can be found on the django-allauth documentation.

All OAuth2 providers have a parameter called "redirect URLs". When a user tries to login to such a service through CATMAID, a URL to a login API on this CATMAID server is sent along the login service. The login service has a white list of URLs that are passed in (the redirection URLs) and allows only a successful login if the passed in URL is on this white list.

GitHub

1. Add GitHub support to installed applications in `settings.py`:

   ```
   INSTALLED_APPS += ('allauth.socialaccount.providers.github',)
   ```

2. Create GitHub OAuth application:

   https://github.com/settings/applications/new

   Make sure that the "Authorization callback URL" machtes your setup. Generally, the form is:

   ```
   https://<catmaid-path>/accounts/github/login/callback/
   ```

   For a local development setup, use this URL:

   ```
   http://localhost:8000/accounts/github/login/callback/
   ```

   This creates a *Client ID* and a *Client Secret*, which are needed in a later step.

3. Create a Site object in the CATMAID admin view with `ID = 1`, or if you have configured your own `SIDE_ID` in `settings.py` use this one.

4. Create *Social application* using the GitHub provider in CATMAID's admin view. Use *Client ID* and *Client Secret* created in step 2.
5. The front-end should now display a menu for when hovering the mouse over the Login button, showing the new entry "Login with GitHub". Clicking it should ask your GitHub user account if CATMAID has permission to use it.

Orcid.org

To use Orcid.org as login service, their Public API is sufficient, which is available to everyone to everyone with an Orcid.org account. The details on how to create a new application and obtain the *Client ID* and *Client Secret* can be found on orcid.org. Everyone with an Orcid ID can create new applications from the developer tools (available in the user account drop down menu). Remember to define the redirect URLs or domains properly. With this done, CATMAID can be configured like this:

1. Add ORCID support to installed applications in `settings.py`:

```
INSTALLED_APPS += ('allauth.socialaccount.providers.orcid',)
```

2. Create a Site object in the CATMAID admin view with `ID = 1`, or if you have configured your own `SIDE_ID` in `settings.py` use this one. Multiple login services can use the same site.

4. Create *Social application* using the Orcid.org provider in CATMAID's admin view. Use *Client ID* and *Client Secret* assigned in the Orcid.org sign-up process.
5. The front-end should now display a menu for when hovering the mouse over the Login button, showing the new entry "Login with Orcid.org". Clicking it should ask your Orcid user account if CATMAID has permission to use it.

Orcid.org sandbox

Obtaining an Orcid.org sandbox account is easier and a prerequisite of a production account. More details on the process can be found on orcid.org. This process will also result in a *Client ID* and a *Client Secret*.

The rest of the Orcid-Sandbox configuration is just like the production Orcid.org setup, but needs additionally the following entry in `settings.py`:

```
SOCIALACCOUNT_PROVIDERS = {
    'orcid': {
        # Base domain of the API. Default value: 'orcid.org', for the production API
        'BASE_DOMAIN':'sandbox.orcid.org',  # for the sandbox API
        # Member API or Public API? Default: False (for the public API)
        'MEMBER_API': True,  # for the member API
    },
}
```

1.2.21 Frequently Asked Questions

My CATMAID instance is working in debug mode, but can't be reached in production. What is the problem?

If you see return code 400 (Bad Request), check the `ALLOWED_HOSTS` setting in your Django configuration file:

```
django/projects/mysite/settings.py
```

Since Django 1.5 this setting is present and should contain a list of all host/domain names that your CATMAID instance is reachable under. Access will be blocked if target host isn't found in this list. For more detail have a look at the Django documentation.

Be aware that if you use Nginx and make a WSGI server available through an *upstream* definition, the host that Django sees is the upstream's name. So this is what you want to add to `ALLOWED_HOSTS`. Alternatively, you can add a `X-Forwarded-Host` header when calling the upstream in a Nginx location block to forward the original host to Django:

```
proxy_set_header X-Forwarded-Host $host;
```

If you then instruct Django to use this header by setting `USE_X_FORWARDED_HOST = True` in `settings.py` (see doc), you can add the original host name to `ALLOWED_HOSTS`.

I have more than one CATMAID instance running on the same (sub-)domain, but in different folders. When I open different instances in the same browser at the same time, one session is always logged out. Why?

Django uses the cookie name 'sessionid' for its session information in a cookie called 'csrftoken' for CSRF information. This is fine if only one instance is running on a certain domain. If, however, multiple instances run on the same domain, this naming scheme fails. The same cookie name is then used for both instances, which leads to the logout in all but one instances if the multiple instances are opened in the same browser.

To fix this, either regenerate your settings.py file with a recent CATMAID version or manually set different names for the relevant cookies in all your `settings.py` files by overriding the variables `SESSION_COOKIE_NAME` and `CSRF_COOKIE_NAME`. Recent CATMAID versions do this automatically, based on the specified sub-folder.

I get an error 500 response and in debug mode I see the error "libhdf5.so.8: cannot open shared object file: No such file or directory". This might have started after the system update.

Apparently, your system's HDF5 library was changed. Therefore, the Python bindings that are used by CATMAID have to be updated. Given you are within the `virtualenv` and in CATMAID's `django` directory, the following should fix it:

```
grep h5py requirements.txt | xargs pip install -I
```

This will reinstall (and recompile) the HDF5 python bindings with the version specified in our dependency file (`requirements.txt`).

CATMAID stopped working after PostGIS update

Updating PostGIS on your host system could cause CATMAID to stop working. You might see errors like:

```
django.core.exceptions.ImproperlyConfigured: Cannot determine PostGIS
version for database "catmaid". GeoDjango requires at least PostGIS
version 1.3. Was the database created from a spatial database template?
```

This can happen when old PostGIS library files are removed and PostGIS can't find what it expects. To fix this, log into the CATMAID Postgres database and update the PostGIS extension:

```
sudo -u postgres psql -d <CATMAID-DB-NAME>
ALTER EXTENSION postgis UPDATE;
```

No image data due to lack of Cross-Origin Resource Sharing (CORS) headers

You might get an error like this if you don't serve images with CORS header fields:

```
Image from origin 'http://images.catmaid-host.org' has been blocked from
loading by Cross-Origin Resource Sharing policy: No
'Access-Control-Allow-Origin' header is present on the requested resource.
Origin 'http://other.catmaid-host.org' is therefore not allowed access.
"""
```

This can be fixed by sending an access policy header along with the images, coming in the form of this header field:

```
Access-Control-Allow-Origin *
```

An example setup for Nginx can be found *here*.

Nginx won't serve static files

Besides checking the Nginx configuration itself, make sure the files are readable by the user running Nginx (e.g. `www-data`). Also, to serve static files, Nginx needs execute permission on every directory in the path to those files. To check this, the `namei` command can be very helpful, because it can list permissions for each path component when called like this: `namei -l /path/to/static/files`.

1.2.22 Installing Catmaid with an RDS database backend

Catmaid can be installed (usually in AWS) using Amazon RDS as its backend, with a few small changes to the install/upgrade procedure.

Things to note

- On RDS you don't get full admin over your database. The admin account for your database has sufficient privileges for most things you will need to do
- You can use AWS security groups to restrict access to your database to the EC2 instance your Catmaid is in (if you are in EC2) or to the IP your Catmaid has (if you are not)

The process

1. On a Catmaid host that's not running a local PostgresSQL server, you only need to install the Postgresql client libraries and commandline tools. The PostgreSQL server components should not be installed.
2. When making the RDS instance, a m4.large (or equivalent) should be sufficient for most use. If you ever need more performance you can easily up the stats on the host with a modify operation. Unless your Catmaid is outside of EC2, you should allocate the host without a public IP.
3. If your Catmaid host is in EC2, put it in a security group called "catmaid" (do not attach a policy directly to this SG - it is just for grouping), and allocate your RDS host in a security group called "catmaid-db". In the security group settings for catmaid-db, write an inbound policy allowing inbound 5432/TCP from the catmaid security group.
4. If your Catmaid host is not in EC2, you only need a catmaid-db security group, with an inbound policy allowing inbound 5432/TCP from the IP of your Catmaid server.
5. The choice between production and testing setup for the Catmaid RDS is up to you; the multi-zone settings in AWS's production setup get you higher uptime, but the difference is small and having your RDS more reliable than your (single-homed) Catmaid serves no actual purpose.
6. The admin user might be named catmaid; the database made in the instance should be named catmaid (it is fine to let the AWS console make this)

After you have the RDS instance up, connect to the catmaid database using the postgres command from your catmaid server, verify connectivity, and then setup GIS as follows:

```
CREATE EXTENSION postgis;
CREATE EXTENSION fuzzystrmatch;
CREATE EXTENSION postgis_tiger_geocoder;
CREATE EXTENSION postgis_topology;

ALTER SCHEMA tiger OWNER TO catmaid;
ALTER SCHEMA tiger_data OWNER TO catmaid;
ALTER SCHEMA topology OWNER TO catmaid;

CREATE FUNCTION exec(text) returns text language plpgsql volatile AS $f$ BEGIN␣
↪EXECUTE $1; RETURN $1; END; $f$;
SELECT exec('ALTER TABLE ' || quote_ident(s.nspname) || '.' || quote_ident(s.relname)␣
↪|| ' OWNER TO rds_superuser;')
  FROM (
    SELECT nspname, relname
    FROM pg_class c JOIN pg_namespace n ON (c.relnamespace = n.oid)
    WHERE nspname in ('tiger','topology') AND
    relkind IN ('r','S','v') ORDER BY relkind = 'S')
s;
```

This is necessary to make the permissions appropriate for GIS support.

Next, to migrate to RDS (or between RDS instances during an upgrade), one must use pg_dump (in text format, not custom format) during an upgrade, and use the psql command to load the data into the new (RDS) instance.

To tell Catmaid how to connect to the database, set the following two fields in configuration.py:

```
catmaid_database_host = 'my-catmaid-rds.c99999hh4gfj.us-east-2.rds.amazonaws.com'
catmaid_database_port = '5432'
```

(adjusting the former for the RDS endpoint your instance has). Do NOT replace the RDS endpoint hostname with its IP (in the long term the IP is not guaranteed to be stable).

1.3 Developer Documentation

1.3.1 Developer setup with Vagrant

CATMAID has a lot of dependencies and is only supported on Ubuntu. To ease the setup for developers, CATMAID has a Vagrant configuration, which allows you to develop inside a virtual ("guest") machine.

Unlike docker, data will be persisted between restarts, and you can interact with the container over SSH.

The repository directory is mounted into the container so all changes are reflected on both the host and the guest machine.

Quickstart (linux)

- Install VirtualBox and vagrant
- Make a fresh clone of the CATMAID repository and `cd` into it
- `vagrant up`
- **`vagrant ssh`, then once inside**
  - Go to code directory: `cd /CATMAID`
  - Finish setup: `scripts/vagrant/optional.sh`
  - Start the dev server: `django/projects/manage.py runserver [::]:8888`
- Develop, inside or outside the container!

Vagrant

Vagrant is an easy-to-configure abstraction layer over a number of virtual machine providers (e.g. VirtualBox, VMWare, Hyper-V) implemented in ruby 2.6. Once installed, you need only `vagrant up` inside the repository to create a fully-fledged Ubuntu VM with PostgreSQL running, fully populated Python and Node environments, and a base R environment.

Install vagrant and the VirtualBox VM provider. You may need to enable virtualization in the BIOS. Additionally, install the `vagrant-disksize` plugin, with `vagrant plugin install vagrant-disksize`. Note also that VirtualBox may not be compatible with Docker under windows.

`vagrant up` Start up the container. If the image (box) doesn't exist, it will be downloaded. If the container doesn't exist, it will be provisioned (i.e. dependencies will be installed).

`vagrant ssh` SSH into a running container.

`vagrant ssh-config` Print out configuration for SSHing into the container with the regular *ssh* command (see below).

`vagrant suspend` Hibernate the container, saving its current state.

`vagrant resume` Resume a ``suspend``ed container.

`vagrant destroy` Delete the container.

See the vagrant documentation for more details.

Tools like VScode need to be able to SSH into the container themselves. They get their configuration from your user's SSH configuration (`~/.ssh/config` on Linux). Copy and paste (or `echo`) the output of `vagrant ssh-config` (ignoring any warnings from ruby gems) into this file. The hostname of the container will be `catmaid-vm`.

Warning: `suspend` the container before shutting down the host!

Snapshots

You can save the state of a running VM and restore it later. This is helpful if you install additional tools or data into your VM, and is quicker to load than a full re-provision.

`vagrant snapshot save <name>` Save the VM's current state with the given name.

`vagrant snapshot list` Show the available snapshots.

`vagrant snapshot restore <name>` Restore the named snapshot.

Note that destroying a VM also destroys all snapshots of it.

More information is available in the vagrant documentation.

Setup

Guest machine configuration

By default, the guest machine is quite small: it may not be able to hold much data and have poor performance. You can customise the machine's "hardware" by setting environment variables on the host machine when the guest is first set up.

The guest's disk size is 60GB by default (but does not take up that much space until it's full). To increase this, set the `CATMAID_VM_DISK` environment variable to something like `"100GB"`. The guest's RAM size, in MB, will be 2048: change it by setting `CATMAID_VM_RAM_MB`. It will use 2 CPUs, which can be set with `CATMAID_VM_CPUS`.

Consider using direnv or dotenv, or adding these variables to your shell rc or profile script, to ensure you always have them set when interacting with Vagrant.

Provisioning

The first time the VM is started, it is "provisioned" - i.e. CATMAID's dependencies are installed. Subsequent startups will be much faster.

Some red messages during provisioning are expected: every line prepended with a + is just showing what command is being run.

This provisioning gets you up to step 3 in the basic installation instructions (setting up the OS-level dependencies and python environment). The database and CATMAID configuration are done separately, in case you prefer your own configuration to the recommendations in the installation instructions.

To finish off the installation according to the instructions, SSH into the VM (`vagrant ssh`) and `bash /CATMAID/scripts/vagrant/optional.sh`. If the `DB_NAME`, `DB_USER`, `DB_PASSWORD`, or `TIMEZONE` environment variables are set (on the guest), they will override the defaults (when the machine is provisioned, the host's timezone will be added to `~/timezone`, which is used as the default timezone here). This creates your local settings, applies database migrations, collects static files as symlinks, creates a CATMAID superuser (you will need to input your the username, email, and password), inserts example projects (N.B. the data for these projects is probably not accessible), and sets CATMAID's writable directory to `/CATMAID/data`.

Virtual machine layout

The container runs Ubuntu Linux 18.04. In Linux, `/` is the root directory, and `~` is the home directory of the user (called `vagrant` in the container: so `~` is `/home/vagrant`).

- The CATMAID repository is in `/CATMAID`. This is the exact same directory as lives on the host.
- The Node environment is stored in `~/catmaid-npm-overlay/node_modules`, and overlaid onto `/CATMAID/node_modules`. This prevents it interfering with the host's node environment and vice versa.
- The Python environment is stored in `~/catmaid-env`, and is automatically activated when you SSH in.
- R packages are in `~/R`.

Some guest ports are forwarded to the host machine so that you can access the database, test with the dev server, and look at the generated sphinx docs.

Service	Guest port	Host port	Notes
PostgreSQL	5555	5555	Not the default port 5432
Django dev server	8888	8888	`django/projects/manage.py runserver [::]:8888`
Docs server	8889	8889	`cd sphinx-doc && make serve`

If *optional.sh* was used to configure the VM, and no parameters were given using environment variables:

- The CATMAID database is called "catmaid".
- The database user is called "catmaid_user".
- The database user passwrod is "p4ssw0rd".
- The CATMAID time zone is the same as the host machine, if the host machine has a POSIX terminal, or defaults to `America/New_York` (but the guest machine is UTC).

Development

Because the development server will technically be accessed from outside of the machine it's running on, you will need to start it with `django/projects/manage.py [::]:8888`

From inside the container, connect to the database with `psql -U catmaid_user catmaid`. From the host, add the options `-h localhost -p 5555`.

VSCode's Remote - SSH extension allows you to develop in the container directly. The connection details are picked up from your `~/.ssh/config` file.

PyCharm Professional has support for remote interpreters built in.

You can also install your own development toolchain inside the container - it's just ubuntu! Alternatively, you can make your edits using the host machine, and just use the VM to test, lint, run the database, etc.

Making commits

By default, the git user is not globally configured inside the VM, and cannot make commits. You have a few options:

- Interact with git only from the host machine
- Configure git globally inside the VM
- Configure your user locally in the repository (allowing its use from either the host or the guest)

1.3.2 Contributing to CATMAID

CATMAID is open source software and welcomes contributions. This document provides a brief overview of the structure of CATMAID and guidelines contributing developers follow to help keep the codebase easy to understand and easy to extend.

If you are considering contributing a feature to CATMAID, you can get guidance from other active developers through the CATMAID mailing list and GitHub repository. Always check the list of open issues as there may be valuable discussion relevant to your plans.

Before developing any features you should follow the *basic installation instructions* to set up your development environment.

Architecture Overview

CATMAID is a distributed client-server application. The backend HTTP API, hosted by the server, retrieves and stores information about projects, image stacks, and annotations. The client frontend, which runs in the browser, provides an interface and suite of analysis tools which interact with the backend's HTTP API. The frontend also has its own APIs which allow new tools to be quickly constructed or expert users to perform novel analysis using the browser console.

The backend is written primarily in Python 3.8 using the Django web framework. Annotations and metadata about stacks are stored in a PostgreSQL database. Most endpoints in the backend API expect and return JSON.

The frontend is written primarily in Javascript and makes use of a several external libraries. Most interfaces are built dynamically through Javascript; few HTML templates are used.

A core philosophy of this architecture is to keep the backend API fast and minimal. The primary purpose of the backend is to mediate the database. Complex analysis and data processing is performed on the client whenever possible. This allows large scale collaboration with constrained server resources. Distributing computation this way also exploits CATMAID's implementation choices, as modern Javascript VMs are typically much faster than Python.

CATMAID is not an image host. Rather, the CATMAID backend provides resource, spatial, and semantic metadata about image stacks hosted elsewhere, while the CATMAID frontend is capable of rendering and navigating these image stacks. More information about the types of image hosts CATMAID supports is available in the *tile source conventions documentation*.

Project Organization

Code you are likely to be interested in is under the `django` folder in the repository root. The sections below outline basic folder, file, and module structure for the backend and frontend, as well as primers on a few common data structures.

Backend

All of the relevant backend code is in the `django/applications/catmaid` folder. Within this folder, `models.py` defines the database schema and logical objects on which the back API operates, while `urls.py` maps URI endpoints in the API to Python methods. Both are useful starting points when locating particular functionality or determining where to add new functionality. In case an endpoint changes data, a transaction log entry is added. This way semantic information can be linked to individual database changes.

Most of the API routes to the `catmaid.control` module and folder. Within this module API functions are organized into logical units like skeleton or connector, which are grouped into corresponding Python modules. These often contain utility functions not exposed by the API that may be useful, so when developing a new API endpoint be sure to check related modules for reusable utilities.

Back-end errors should always be signaled to the front-end with the help of Exceptions. Regardless whether an argument is missing, permissions are lacking or something went wrong otherwise. A dedicated middleware will catch them and return them in an expected format to the front-end.

Frontend

If developing frontend functionality, a good strategy is to start by running scripts in the browser console to quickly prototype and become familiar with client APIs. The scripting wiki provides an introduction to these APIs and snippets for common scripting tasks.

Javascript source files should be placed in the `django/applications/catmaid/static/js` folder. External libraries are located in the `django/applications/catmaid/static/libs` folder, although there is also a special CATMAID library for shared, stable components. Javascript and CSS assets from these locations are managed by django-pipeline. When you add a Javascript file to the `static/js` folder and then run:

```
./manage.py collectstatic -l
```

from the project folder, pipeline detects these assets, compiles and compresses them (if configured to do so), then passes them to Django to be linked from the configured static server directory. Assets for this pipeline are configured in `django/projects/mysite/pipelinefiles.py`. Source files placed in `static/js` will be detected automatically, but any external libraries added to the `static/libs` folder must also be added to `pipelinefiles.py`.

Within the `static/js` folder and within the CATMAID frontend there is a distinction between *tools* and *widgets*. A tool contains a suite of annotations, interfaces and analyses. A widget, meanwhile, provides a single specific interface. Most likely you are familiar with a single tool in CATMAID, the tracing tool, but many widgets within the tracing tool, such as the 3D viewer, connectivity widget, and selection table.

Widgets are generally prototyped objects that extend `InstanceRegistry`, which provides an easy means to track open instances of a particular widget. Rather than construct their own DOM, most widgets' DOM is built by a corresponding method in `WindowMaker`. `WindowMaker` binds events from the DOM it constructs to relevant handlers in the widget object.

Code Style and Conventions

Over the history of its development, CATMAID has accumulated a mixture of many coding styles. To improve the consistency and clarity of code going forward, as well as to prevent some common technical pitfalls, the core developers now follow some simple guidelines for new code. These guidelines are relaxed and permissive.

If modifying existing code, feel free to imitate the style of the surrounding code if it conflicts with these guidelines.

Python

CATMAID does not currently adhere to a specific Python style convention like PEP8. However, code should still follow common Python conventions and idioms including:

- 4 spaces (not tabs) for indentation
- Maximum line length of 79 characters for comments
- Maximum line length of 120 characters for code
- PEP8 naming conventions

All new code should include docstrings that follow PEP257 and use Google's argument formatting.

For consistency, python code will be linted using `flake8`: the configuration can be found in the repository. Contributions which do not conform with those rules will fail CI checks. Several rules are ignored due to a large volume of non-compliant legacy code. New code should try to conform even to these rules where appropriate, unless doing so would reduce the consistency of the codebase.

Type annotations should be used where possible. Currently, these are checked on CI using `mypy` for informational purposes only.

HTTP API

Documentation for endpoints exposed by the HTTP API is available from the CATMAID server itself via the `/apis/` page:

```
http://localhost:8000/apis/
```

. . . or, for custom configurations:

```
http://<catmaid_servername>/<catmaid_subdirectory>/apis/
```

Functions that are exposed as HTTP API endpoints should declare what HTTP methods they accept using the `@api_view` decorator. Endpoints' docstrings should define what parameters they accept and the strucuture of their response in Swagger spec using django-rest-swagger's YAML hooks:

```
@api_view(['GET', 'POST'])
def api_endpoint(request):
    """Short endpoint description.

    Longer description of the endpoint's purpose, expectations and behavior.

    This endpoint returns an array of objects, so the model of the objects
    in the array must be specified in a separate ``model`` stanza.
    ---
    parameters:
        - name: resource_id
```

(continues on next page)

(continued from previous page)

```
          description: ID of a resource.
          required: true
          type: integer
          paramType: form
    models:
      api_endpoint_inner_type:
        id: api_endpoint_inner_type
        properties:
          name:
            description: Name of some example type that this endpoint
            type: string
            required: true
    type:
    - type: array
      items:
        $ref: api_endpoint_inner_type
      required: true
    """
    #...
```

API URLs should prefer plural resource names and use hyphens rather than underscores. Non-terminal endpoint paths that represent resources should have a trailing slash, e.g., `GET http://localhost/{project_id}/skeletons/`, but not terminal operations on that resource collection like `GET http://localhost/{project_id}/skeletons/review-status`.

Parameters that are not resource identifiers should be passed as query or form parameters, not in the URL path. If an endpoint accepts an array of parameters, it should support receiving the array encoded as JSON; form array parameters may be accepted, but a JSON array in a single form parameter must be accepted for ease of use.

Prefer descriptive, consistent names for parameters. For example, an endpoint receiving a list of skeleton identifiers should prefer a parameter named `skeleton_ids` over `skids` or `ids`; a few bytes in the header are not going to have a performance impact relative to the packaging of HTTP and transport, much less when HTTP/2 and modern compression-aware browsers are involved. However, abbreviated property names or array-packed values are acceptable for the responses of performance-critical endpoints.

Date and time response values should be in UTC and formatted as ISO 8601.

Endpoints containing write operations should be decorated with a `record_view` decorator in `urls.py`, which expects a label as argument. This label should follow the pattern `resource.action` and just like URI itself, the `resource` is expected to be in its plural form. Make sure to follow this convention for new endpoints.

Javascript

New code in CATMAID is styled similar to the Google Javascript style guide, with notable exceptions that:

- CATMAID does not use any Google libraries
- CATMAID does not use any requirements/dependency libraries
- CATMAID uses CamelCase namespace naming

New javascript files should place all code inside an IIFE to namespace it inside the `CATMAID` object and use ES5 strict mode:

```
(function (CATMAID) {

  "use strict";
```

(continues on next page)

(continued from previous page)

```
  var variableNotExposedOutsideFile;

  var ClassExposedOutsideFile = function () {
    //...
  };

  CATMAID.ClassExposedOutsideFile = ClassExposedOutsideFile;

})(CATMAID);
```

This prevents unintentional leaking of variables into the global scope and possible naming conflicts with other libraries.

CATMAID makes full use of ES5 language features and allows the following ES6 features:

- Promises
- Maps and Sets (IE11-supported `get`, `has`, `set`, `delete` and `forEach` only)
- `const` and `let` declarations (in strict mode contexts only)

All features must work correctly in recent versions of Chrome and Firefox, while core browsing features must work in IE11. Requiring polyfills for IE is acceptable.

Git

Try to follow the seven rules of great git commit messages:

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

That said, always prefer clarity over dogma. The core CATMAID contributors break #2 frequently to keep messages descriptive (apologies to our VAX users). If a commit focuses on a particular component or widget, prefix the commit message with its name, such as “Selection table:” or “SVG overlay:”.

Granular commits are preferred. Squashes and rollups are avoided, and rebasing branches then fast-forwarding is preferred over merge commits when merging, except for large feature branches.

Development occurs on the `dev` branch, which is merged to `master` when a release is made. It is usually best to develop new features by branching from `dev`, although critical fixes or extensions to particular releases can be based on `master` or the appropriate release tag.

Never rewrite history of `master`, `dev`, or any other branch used by others.

Linting and Testing

As part of the continuous integration build, several automated processes are performed to help verify the correctness and quality of CATMAID:

- *Unit and integration tests for Django backend*
- Linting (static analysis) of the python code with flake8
- Type checking of the python code with mypy
- Linting (static analysis) of the javascript code with JSHint
- Linting of CSS with csslint
- Unit tests of javascript code with QUnit

If you enable Travis-CI for your fork of CATMAID on GitHub, Travis will run all of these checks automatically. However, Travis builds take a long time, and you may want feedback before committing and pushing changes. Luckily all of these checks are easy to run locally.

Django tests are run through Django's admin commands:

```
cd /<path_to_catmaid_install>/django/projects
./manage.py test catmaid.tests
```

flake8 and mypy are installed along with other python development dependencies. Run them with:

```
flake8 django
mypy django
```

JSHint can be installed from NPM or your platform's package manager and should use CATMAID's config settings:

```
cd /<path_to_catmaid_install>
jshint --config=.travis.jshintrc --exclude-path=.travis.jshintignore django/
↪applications
```

If you do not want to configure your own JSHint settings, you can set these as defaults:

```
ln -s .travis.jshintrc .jshintrc
ln -s .travis.jshintignore .jshintignore
jshint django/applications
```

CSS linting is performed by running *csslint* from the static CSS directory:

```
cd django/applications/catmaid/static/css
csslint .
```

QUnit tests can be run from the browser while your Django server is running. For example, with the default configuration this would be:

```
http://localhost:8000/tests
```

. . . or, for custom configurations:

```
http://<catmaid_servername>/<catmaid_subdirectory>/tests
```

Alternatively, the front-end tests can be run in a terminal (as it is done in our CI setup). To do so, first a few dependencies have to be installed and then *karma* is used to execute the tests from the CATMAID root directory:

```
cd /<path_to_catmaid_install>
npm install --only=dev
karma start karma.conf.js
```

Documentation

In addition to the backend, HTTP API, and frontend documentation mentioned above, CATMAID provides a general documentation manual for users, administrators, and developers (including this page) and in-client documentation for keyboard shortcuts and widget help.

General Documentation

General documentation is part of the CATMAID repository under the `sphinx-doc` folder. This documentation is written in Sphinx ReStructured Text. Documentation from commits pushed to the official CATMAID repository are built by Read the Docs and hosted at catmaid.org.

CATMAID's documentation can be built in various formats by navigating to the `sphinx-doc` directory and using make. The default (i.e. `make` with no arguments) is HTML, which builds the documentation at sphinx-doc/build/html/index.html.

Every build target can automatically be built when the source files change, by using `make watch-<target>`. Build the HTML docs, watch for changes, and serve the documentation at `http://localhost:8889` using `make serve`.

In-Client Documentation

Documentation is provided from within the web client through tool-scoped mouse and keyboard shortcut documentation (accessed by pressing `F1`) and per-widget help accessible through the question mark icon in the title bar of some widgets.

If you find that widget help documentation is missing, incomplete, confusing, or incorrect, you can contribute better documentation by creating an issue on GitHub or editing the `helpText` property of the widget and creating a pull request.

Other Policies

Security

The disclosure policy of the CATMAID developers for vulnerabilities is that arbitrary SQL execution by anonymous users or users with "browse" permissions must be notified to the mailing list simultaneous with patch publication. Vulnerabilities only exploitable by users with "annotate" permissions will be noted in the release changelog but will not be sent to the mailing list.

1.3.3 Tile Source Conventions

CATMAID does not serve image data itself. Instead, it stores a few critical pieces of information about the image volume and specifies a set of conventions for retrieving image data from a separate image volume host. These conventions are *tile source types*, which define parameters and URL formats both the CATMAID backend and frontend can use to access image data for rendering and analysis.

This document provides information on the type source types specified by CATMAID, the parameters which they use to retrieve image data, how this is stored in the CATMAID backend, and how to create a new tile source type in the CATMAID backend and frontend.

Also note that CATMAID will clamp the display of tiled data by default to zero (i.e. no request is made for tiles at coordinate [-1, 0] in the plane). This behavior can be changed by adding the boolean field `clamp` to the stack's `metadata` field with a value of `false`. The `metadata` field expects a JSON format, so the content would look like `{"clamp": false}`.

Tile Source Parameters

Each tile source type uses a subset of the following parameters to determine its source regardless of the specific tiles being retrieved. Though specific APIs may use different names for these parameters, the names provided here are used for consistency in defining tile source types below:

`sourceBaseURL` (string) This defines a base URL for the source server and volume, including trailing slash.

`dimension` (integer array [x, y, z]) Dimension of the image volume in pixels at its original scale level.

`resolution` (double array [x, y, z]) Resolution of the image volume in nanometers at its original scale level.

`numZoomLevels` (integer) The number of zoom levels (downsampling octaves) which the source supports.

`fileExtension` (string) Filename extension for image tiles served by the source.

`tileWidth` (integer)

`tileHeight` (integer) Dimensions of image tiles in pixels. For pre-tiled sources, this should be the dimensions of the source tiles, while for dynamic sources this should be the preferred dimension for retrieval and rendering.

`orientation` (integer|string) Orientation of the stack relation to project space. Semantically this value is either "XY", "XZ", "ZY", but is usually passed as an integer in {0, 1, 2} enumerating these, respectively.

Currently only DVID tile source types (6, 8) must know orientation directly. Other tile source types encode orientation in the `sourceBaseURL`.

`tileSourceType` (integer) A integer enumerating the types of tile sources listed below.

Tile Parameters

Tile sources may additionally make use of a subset of the following parameters when determining how to retrieve specific tile requests:

`row` (integer) The row of the tile in the image grid for this z-slice, e.g., the floor of the tile y origin divided by `tileHeight`.

`col` (integer) The column of the tile in the image grid for this z-slice, e.g., the floor of the tile x origin divided by `tileWidth`.

`zoomLevel` (integer) The zoom level of tiles to retrieve. Zoom level 0 is the original scale, zoom level 1 is the first downsampled octave, etc.

pixelPosition (integer array [x, y, z]) The *stack space* position in pixels of a location in the plane of the tile. Usually this position is shared for many tile requests, e.g., the center of a stack viewer, and is used only to generate a path using the `z` location in the stack.

Tile Source Types

Tile source types are listed by the enumeration integer ID referenced by `tileSourceType`:

1. File-based image stack

URL format:

```
<sourceBaseUrl><pixelPosition.z>/<row>_<col>_<zoomLevel>.<fileExtension>
```

2. Request query-based image stack

URL format:

```
<sourceBaseURL>?x=<col * tileWidth>
               &y=<row * tileHeight>
               &z=<pixelPosition.z>
               &width=<tileWidth>
               &height=<tileHeight>
               &scale=<2^-zoomLevel>
               &row=y
               &col=x
```

3. HDF5 via CATMAID backend

Note: To use this tile source, the `h5py` Python library has to be installed.

This is the only tile source type which the CATMAID backend serves directly. Django retrieves tile requests from an image volume stored in an HDF5 file. This is a convenience source intended for quick exploration of small volumes only and does not scale to large volumes or many users.

As an exceptional source, this uses the following tile source parameters that should not be used by any other source:

catmaidURL (string) Base URL of the CATMAID instance serving the volume, including trailing slash.

projectId (integer) ID of the CATMAID project.

stackId (integer) ID of the CATMAID stack.

URL format:

```
<catmaidURL><projectId>/stack/<stackID>/tile?x=<col * tileWidth>
                                             &y=<row * tileHeight>
                                             &z=<pixelPosition.z>
                                             &width=<tileWidth>
```

(continues on next page)

(continued from previous page)

```
                                                         &height=<tileHeight>
                                                         &scale=<2^-zoomLevel>
                                                         &row=y
                                                         &col=x
                                                         &file_extension=<fileExtension>
                                                         &basename=<sourceBaseURL>
                                                         &type=all
```

4. File-based image stack with zoom level directories

A variation on tile source type 1 where the zoom level is also a directory.

URL format:

```
<sourceBaseUrl><pixelPosition.z>/<zoomLevel>/<row>_<col>.<fileExtension>
```

5. Directory-based image stack

Like tile source types 1 and 4, but with all components being directories.

URL format:

```
<sourceBaseUrl><zoomLevel>/<pixelPosition.z>/<row>/<col>.<fileExtension>
```

6. DVID `imageblk` voxels

This type supports loading tiles from voxel data instances in DVID using `imageblk` (`uint8blk`, `rgba8blk`) datatypes.

For DVID `imageblk` tile sources, `sourceBaseURL` should reference the data instance REST resource with orientation information directly, that is:

```
<api URL>/node/<UUID>/<data name>/raw/<dims>/
```

`fileExtension` may also specify a quality parameter, e.g., `jpg:80`.

Because orientations are just permutations of coordinates in the voxel volume, each orientation has a slightly different URL format.

XY format:

```
<sourceBaseUrl><tileWidth>_<tileHeight>/<col * tileWidth>_<row * tileHeight>_
↪<pixelPosition.z>/<fileExtension>
```

XZ format:

```
<sourceBaseUrl><tileWidth>_<tileHeight>/<col * tileWidth>_<pixelPosition.z>_
↪<row * tileHeight>/<fileExtension>
```

ZY format (actually YZ, see note):

```
<sourceBaseUrl><tileWidth>_<tileHeight>/<pixelPosition.z>_<row * tileHeight>_
↪<col * tileWidth>/<fileExtension>
```

Note: Because DVID prefers YZ axis ordering over ZY, note that tiles for that orientation must be transposed to be consistent with other tile source types.

7. Render service

This tile source type retrieves image tiles from the dynamic render service used at Janelia Research Campus.

URL format:

```
<sourceBaseURL>largeDataTileSource/<tileWidth>/<tileHeight>/<zoomLevel>/
↪<pixelPosition.z>/<row>/<col>.<fileExtension>
```

8. DVID `imagetile` tiles

This type supports loading tiles from tile data instances in DVID using `imagetile` datatypes.

For DVID `imagetile` tile sources, `sourceBaseURL` should reference the data instance REST resource directly, that is:

```
<api URL>/node/<UUID>/<data name>/tile/
```

Because orientations are just permutations of coordinates in the voxel volume, each orientation has a slightly different URL format.

XY format:

```
<sourceBaseUrl>xy/<zoomLevel>/<col>_<row>_<pixelPosition.z>
```

XZ format:

```
<sourceBaseUrl>xz/<zoomLevel>/<col>_<pixelPosition.z>_<row>
```

ZY format (actually YZ, see note):

```
<sourceBaseUrl>yz/<zoomLevel>/<pixelPosition.z>_<row>_col
```

Note: Because DVID prefers YZ axis ordering over ZY, note that tiles for that orientation must be transposed to be consistent with other tile source types.

9. FlixServer tiles

This type supports loading tiles from a FlixServer instance. This tile server generates images dynamically and supports CATMAID's default tile source URL format:

```
<sourceBaseUrl><pixelPosition.z>/<row>_<col>_<zoomLevel>.<fileExtension>
```

Additional GET parameters can be used to adjust color, dynamic range and gamma mapping:

```
color = (red,green|blue|cyan|magenta|yellow|white) for coloring
min/max = [0, 2^16] for specifying the dynamic range
gamma = [0, 2^16] for the exponent of a non-linear mapping
```

For multi-channel images, a comma separated list can be used as parameter value (e.g. color=cyan,magenta).

10. H2N5 tiles

This type supports loading tiles from the H2N5 server, which dynamically slices tiles from N5 tensor files.

The URL format for this source is:

```
<sourceBaseUrl>.<fileExtension>
```

However, unlike other sources, `sourceBaseUrl` is not a valid URL on its own. It contains several substitution strings the CATMAID replaces on each tile request. This is necessary to support n-dimensional volumes. The substitution strings are:

`%SCALE_DATASET%` (optional) Where to insert the dataset name for different scale levels. Currently these are `s0`, `s1`, etc., but in the future may be read from the N5 attributes of the parent dataset of where this substitution string appears.

`%AXIS_0%` Position in the coordinates part of the H2N5 URL to replace with `col * tileWidth`

`%AXIS_1%` Position in the coordinates part of the H2N5 URL to replace with `row * tileHeight`

`%AXIS_2%` Position in the coordinates part of the H2N5 URL to replace with `pixelPosition.z`

While `%AXIS_0%` and `%AXIS_1%` could be inferred by parsing the URL for the slicing dimensions, note that `%AXIS_2%` could not.

11. N5 image blocks

This type supports loading **image blocks** from N5 served over HTTP.

Unlike other sources, `sourceBaseUrl` is not a valid URL on its own. It contains several substitution strings the CATMAID replaces on each tile request. This is necessary to support n-dimensional volumes. The substitution strings are:

`%SCALE_DATASET%` (optional) Where to insert the dataset name for different scale levels. Currently these are `s0`, `s1`, etc., but in the future may be read from the N5 attributes of the parent dataset of where this substitution string appears.

Further, `sourceBaseUrl` has special path components. It should end in a path component specifying the ordered dimensions from which image data should be sliced, separated by underscores. For example, `2_1_0` will slices N5 dimensions 2, 1, and 0 as the x, y, and z stack dimensions, respectively.

This tile source will also look for a path component ending in `.n5` to use as the N5 root directory. If it does not find such a component, it will assume the origin is the root N5 directory.

For example:

```
https://catmaid.org/path/root.n5/group/dataset/%SCALE_DATASET%/raw/0_1_2
```

12. JHU/APL Boss tiles

This type supports loading tiles from a JHU/APL Boss instance, using a manually entered API token for authorization. The API token entered through the front-end layer control UI is passed in an `Authorization` header with all tile requests.

Currently only XY tiles are fully supported. `tileWidth` and `tileHeight` must be equal.

`sourceBaseURL` should reference the REST resource directly:

```
<instance URL>/v1/tile/<collection>/<experiment>/<channel>/
```

XY URL format:

```
<sourceBaseURL>xy/<tileWidth>/<zoomLevel>/<col>/<row>/<pixelPosition.z>
```

13. CloudVolume tiles

Using this type, the back-end is instructed to go get a tile through the cloudvolume library. The URL can be anything that CloudVolume can take, e.g. a Neuroglancer Premcomputed image dataset like public Google hosted FAFB dataset:

```
precomputed://gs://neuroglancer-fafb-data/fafb_v14/fafb_v14_clahe
```

Currently, only HTTPS mode is supported (i.e. no explicit credentials).

14. Neuroglander precomputed image blocks

This type supports loading **image blocks** from Neuroglancer Precomputed format served over HTTP.

Like with N5 image blocks and unlike other sources, `sourceBaseUrl` is not a valid URL on its own. It contains several substitution strings the CATMAID replaces on each tile request. This is necessary to support n-dimensional volumes. The substitution strings are:

`%SCALE_DATASET%` (optional) Where to insert the dataset name for different scale levels. Currently these are `s0`, `s1`, etc., but in the future may be read from the N5 attributes of the parent dataset of where this substitution string appears.

Further, and also like the N5 source, `sourceBaseUrl` has special path components. It should end in a path component specifying the ordered dimensions from which image data should be sliced, separated by underscores. For example, `2_1_0` will slices chunk dimensions 2, 1, and 0 as the x, y, and z stack dimensions, respectively.

This tile source will also look for a path component ending in `/info` to use as the dataset root directory. If it does not find such a component, it will assume the origin is the root directory.

For example:

```
gs://neuroglancer/pinky100_v0/son_of_alignment_v15_rechunked/%SCALE_DATASET%/
↪0_1_2
```

Also note that Neuroglancer datasets can have a property called `voxelOffset`, which by default is `(0, 0, 0)`. If set to other values the dataset will be artificially resized at the origin accordingly. To Match voxel coordinates between CATMAID and Neuroglancer, the respective CATMAID stack needs to have the `voxelsOffset` metadata field set with the the value and "Apply primary stack voxel offset" has to be checked in the Settings Widget (default). For instance, if the zero-zoom level voxel offset is `(-3072,`

-3072, 0), then the metadata of the CATMAID stack should look like this: {"voxelOffset": [-3072, -3072, 0]}.

Backend Representation

Tile source parameters are stored in the Stack Django model. To create a new tile source type, one needs only to establish a convention for the integer that enumerates that type (after communication with the developers) and begin using it in stacks.

To support cropping, the backend also implements tile sources. To support cropping for a new tile source type, implement a method in catmaid.control.cropping.CropJob like other get_tile_path_<tileSourceType> methods that returns the correct URL, then make sure it is called from the CropJob.initialize method.

Frontend Retrieval

The front end implements tile source URL generation in django/applications/catmaid/static/js/tile-source.js. To define a new tile source type, follow the convention of the existing tiles sources by creating a function that returns an object with the appropriate getTileURL, getOverviewURL, and getOverviewLayer methods. The overview URL should locate a thumbnail of the current stack z-section. Then map the tileSourceType enumeration of your tile source type to your implementation in CATMAID.TileSources.

1.3.4 Creating widgets

Individual tools and views on the data are displayed in individual windows, so called *widgets*. This section will go over the steps required to create a new widget and how widgets can interact.

To make CATMAID of a new widget, the new window has to be registered:

```
CATMAID.widget(name, options);
```

The name is an arbitrary unique string that is used to refer to the new widget. With the help of the options object, more detail about what the widget should do, can be specified. For instance, general event handlers for e.g. initialization and destruction can declared. See below for more information.

Registering widgets this way allows for a very modular and extensible design. New UI elements and tools can be added without any change in CATMAID itself.

If the widget registration was successful, the new widget can be instantiated by calling:

```
CATMAID.createWidget(name);
```

Widget options

Details on how a widget should be realized, is specified in the options object during widget registration. The following fields are *required*:

Field	Value
getName	Function, returning human readable name of the widget.
getContentID	Function, returning the expected container ID used for controls
createContent	Function, called with a DOM container where the widget content can be placed in

Additionally, the following fields can be used:

Field	Default	Value
`helpText`	`undefined`	String, describing the usage of this widget.
`class`	`undefined`	String, containing a CSS class name that the widget window should get assigned
`controlsID`	`undefined`	String, containing the expected container ID used for controls
`createControls`	`undefined`	Function, called with a DOM container where controls can be placed in
`destroy`	`undefined`	Function, called when the widget should be destroyed. Own handlers should be unregistered in here.

Example widget

In this section, the information above is used to create an example widget. It registers itself with CATMAID and can be opened. Like it is explained in *Contributing to CATMAID*, the widget is defined with an IIFE:

```
(function(CATMAID) {
  CATMAID.widget('test-widget', {
    getName: function() { return "Test widget"; },
    getContentID: function() { return "test-widget"; },
    createContent: function(container) {
      // Add some example text
    }
  });
});
```

This widget has one button to refresh

1.3.5 Creating New Data View Types

There are currently only rather general data view types available in CATMAID. If these don't suit your needs, you can just add your own. A data view can then configure your new data view type and be used as a CATMAID front page.

To add a new data view type, you basically need to define two things:

- A Django template that defines the internals of your view type and
- an entry in the `data_view_type` table to link data views to it.

In the following both steps will be discussed. Details about how to use data views can be found in the section *Using Data Views*.

A new Django template

Data view types are written in the Django template language. To make a new template accessible to CATMAID, put it in the `django/templates/catmaid` directory. There reside the templates for the already available data view types as well. Obviously, it is good practice to choose a template name that has something to do with what it does. E.g., the *Project List* data view template is called `project_list_data_view.html`.

Data view templates get passed a context in which they live. There you have access to the variables `data_view`, `projects` and `config`. With `data_view` the template gets access to the current `DataView` model for which it is the data view type. A list of all projects is available in the `projects` variable. If you have a look at the existing templates you can see that this list is usually walked (e.g. with a `for` tag) to render each project and its stacks. Also, you need not to deal with the `sort` option yourself. This is already dealt with in the Django view.

Therefore, `projects` is already sorted when requested. In the `config` variable one gets access to the already parsed configuration for the data view to render. This will keep the options defined for a data view.

A template can now do whatever it wants with these variables. The available templates start with these lines:

```
{% load data_view_config %}
{% include "catmaid/common_data_view_header.html" %}
```

The first thing is to load custom template tags and filters to make e.g. option handling easier. The file lives in the folder `django/applications/catmaid/templatetags/` and you might want to have a look at it.

Next, another template (`common_data_view_header.html`) is included. This just prints a simple CATMAID header text.

The available templates then start with option parsing, e.g:

```
{% with opt1=config|get_or_none:"opt1"|default_if_none:0 %}
{% with opt2=config|get_or_none:"opt2"|default_if_none:"center" %}
...
{% endwith %}
{% endwith %}
```

There is made use of Django's template filters and a custom one (`get_or_none`) to get a configuration option or a default. Let's take the first line as an example: within this `with`-block a new variable variable is assigned: `opt1`. It's value is created as follows: Use `config` (see above) as the input for the `get_or_none` filter, parametrized with `"opt1"`. This filter checks if its argument (`opt1`) exists in the input dictionary (`config`) and returns its value if that is the case. If is not found, `None` is returned. This result is then passed to the `default_if_none` filter which in turn is parametrized with the option's default value (`0` in that case). It checks if the input is none and if, this is the case, returns the parameter, otherwise the input.

Note: By default, Django doesn't support the Python keywords None, True and False in templates. Therefore, you might use 0 as False and 1 as True.

Within those `with`-blocks you can then write your actual presentation logic. Have a look at the existing templates to get an idea how this could be done.

A new table entry

If your template is ready, you can make it known to CATMAID and thereby usable by data views. To do so you need to add an entry to the table `data_view_type`. This is currently not available from within the Django admin interface, so you have to do it manually.

A data view type needs basically three things there:

- Name it, give it a `title`, e.g. "Filtering data view type".
- It needs a so called `code_type`. This is the template name without file extension, e.g. "filtering_data_view".
- Also, provide a descriptive help text that explains what this view type does, what options it has and maybe provide an example. Put this into the `comment` field.

As an example, in the following the entry in the `data_view_type` table of the *Project List* data view type is shown:

`title`:

```
Project list view
```

`code_type`:

```
project_list_data_view
```

comment:

```
A simple adjustable list of all projects and their stacks.
This view is rendered server side and supports the display
of sample images. The following options are available:
"sample_images": [true|false], "sample_stack": ["first"|"last"],
"sample_slice": [slice number|"first"|"center"|"last"]. By
default projects are sorted. Use "sort":false to turn this
off. Thus, a valid sample configuration could look like:
{"sample_images":true,"sample_stack":"last","sample_slice":"center"}
```

Having done this, a data view should then be able to use your data_view_type (also from within Django admin).

1.3.6 Deep Link URL Format

CATMAID supports deep link URLs that allow users to share persistent references to particular stack locations and views. These URLs also support opening tools, multiple stack viewers, and limited specialized behaviors such as activating nodes in the tracing tool. Which stack mirror is used for a referenced stack is up to the client instance.

CATMAID deep link URLs are standard HTTP URLs with query strings, for example:

```
https://localhost/cat/?pid=1&zp=0&yp=0&xp=0&tool=navigator
```

All URLs begin with the protocol, hostname and path (including trailing slash) of the CATMAID instance *configured during installation*.

Required Query Parameters

`pid` (integer) ID of the project to open.

`zp` (integer)

`yp` (integer)

`xp` (integer) Coordinates in project space (nm) to center in the stack viewer.

`tool` (string) Name of the tool to open. Must be one of `navigator`, `tracingtool`, or `classification_editor`.

Optional Query Parameters

`active_node_id` (integer) Specifies the ID of a node to activate in the tracing tool.

`sid<n>` (integer) Opens the stack with this ID in a stack viewer. Multiple stacks can be opened by passing incremental indexes, i.e., `sid0`, `sid1`, etc. The index must start at 0.

`s<n>` (integer) Zoom level for the corresponding stack ID above. That is, `s0` specifies the initial zoom level for the stack viewer viewing the stack ID passed in `sid0`.

`composite` (integer) If `1`, load all stacks as layers in a single stack viewer rather than separate stack viewers.

`sg` (integer) Open a stack group. If present, individual stacks are ignored.

`sgs` (integer) Initial zoom level for a loaded stack group.

`current_dataview` (integer) ID of a data view to switch to.

`layout` (string) An optional layout specification in the format of layouts described in the Settings Widget. E.g. `layout=h(XY, { type: "neuron-search", id: "neuron-search-1"}, 0.6)` to show a Neuron Search widget in a 1/3 wide column on the right. Depending on the widget referenced, additional options are available. For instance, the Neuron Search can be instructed to search for a particular annotation right away: `layout=h(XY, { type: "neuron-search", id: "neuron-search-1", options: {"annotation-name": "papers"}}, 0.6)`. Available options are collected below. Additionally, individual layout elements can have a `"skeletons"` parameter (like the `"options"` parameter), which can take a list of skeleton IDs or, alternatively objects having an `id` and a `color` field. The color can be defined in terms of common CSS color representations like "red" or "rgb(0.2,1,0.5)". These skeletons are added to the newly created widgets, if possible.

`token` (string) An optional project token to give access to a particular project. If this is used, the link becomes essentially an invitation link. With admin permissions, this tokens can be generated in the Project Management widget.

Legacy Query Parameters

These parameters may be found in old URLs but are no longer generated and may not be properly processed.

`x` (integer)

`y` (integer)

`z` (integer) Coordinates in project space to center the stack viewer. Used in an obsolete query format where no project or stack ID were specified and defaulted to `1`.

`s` (integer) Zoom level for the stack viewer.

`active_skeleton_id` (integer) Specifies the ID of a skeleton to activate in the tracing tool.

`account` (string) Username with which to login to CATMAID.

`password` (string) Password with which to login to CATMAID.

URL widget options

`Neuron Search` Supports `annotation-name` to search for a particular annotation and `with-subannotations` to include sub-annotations for this search.

1.3.7 Database Migrations

Historically, keeping your database in sync with what's expected by the code has been error-prone and annoying, but CATMAID uses a system for schema and data migrations called South which simplifies and structures this task. In contrast to past versions of CATMAID, South migrations *won't* get applied automatically. So when updating the code base, the database has to be updated as well. This page describes what you need to know about these migrations.

General things about South

South has been built to bring migrations to Django projects. Since CATMAID's backend is Django based, it can make use of it. South is stable, tries to be simple and is database-independent. CATMAID, however, depends on some PostgreSQL specific features and database-independence is therefore not really important here.

Every migration is kept in its own file. They are stored in a *migrations* folder within a Django application's directory. In the case of CATMAID this is:

```
django/applications/catmaid/migrations
```

In there you find files like these:

```
0005_add_ontology_visibility_setting_to_profile.py
0006_add_restriction_tables.py
0007_add_treenode_parent_index.py
```

Each migration file groups logically connected changes to the database schema and data. As can be seen above, migrations are ordered by the first characters of their file name. Obviously, the order is important and when creating migrations you want to make sure to not create ambiguous file names.

A migration file contains of a class with a `forward()` and a `backward()` method as well as a dictionary called `models` which contains all available models which were around when the migration was created (i.e. it contains the changes of the migration; read more about it here).

The main way to interact with South is with the help of `manage.py` commands. South adds multiple commands to it, the most often used will probably be `schemamigration` to create new migrations and `migrate` to run migrations (see below). To use `manage.py`, you need to be in the *virtualenv* environment (activate it with `workon catmaid`). The commands in other parts of this page assume you are in the *virtualenv* and the folder where the `manage.py` file lives.

Of course, the migration files need to be added to the source code management.

Checking for new migrations and applying them

To check if there are new migrations that need to be applied, run:

```
./manage.py showmigrations
```

CATMAID utilizes two other applications that use South as well: guardian and djcelery. If you want to refer only to CATMAID, do:

```
./manage.py showmigrations catmaid
```

If there is no migration that has not been applied, you will see a list like the following:

```
(*) 0005_add_ontology_visibility_setting_to_profile
(*) 0006_add_restriction_tables
(*) 0007_add_treenode_parent_index
```

The `(*)` marks indicate that a migration has been applied. In turn `( )` would mean it hasn't. So if migration 0007 wouldn't be applied yet, it would read:

```
(*) 0005_add_ontology_visibility_setting_to_profile
(*) 0006_add_restriction_tables
( ) 0007_add_treenode_parent_index
```

And if you would then want to apply migration 0007, you would need to either run:

```
manage.py migrate catmaid 0007
```

to only apply this particular migration. Alternatively, you can apply *all* not yet applied migrations with:

```
manage.py migrate catmaid
```

I want to make a change to the database

Usually, changes to the database are needed because there have been changes to CATMAID's models in the file (like adding a new field to a class):

```
django/applications/catmaid/models.py
```

If this is the case, you can let South create a migration for this change by running:

```
manage.py schemamigration catmaid [title] --auto
```

This will create a new file in CATMAID's migration folder. It will make sure it has the next free ID and `[title]` is the remainder of the file name. If `[title]` isn't provided, South will come up with an own name. The parameter `--auto` instructs South to inspect the models module of CATMAID and to create a migration based on the changes with respect to the last migration.

If you in turn wanted to create an empty migration to do changes the database that are not based on the models module, run:

```
manage.py schemamigration catmaid [title] --empty
```

These commands, however, do only create the migration file. They don't apply it. This has to be done manually afterwards.

I want to use raw SQL in a migration

Using raw SQL in a migration is is perfectly possible. Instead of using the object relational mapper, you can execute SQL statements directly within the `forward()` and `backward()` methods within the migrations file. To do so you would first need to create an empty migration by running:

```
manage.py schemamigration catmaid [title] --empty
```

Edit the new file and pass your SQL statements as string arguments to the `db.execute()` method. For instance:

```
db.execute("DROP INDEX IF EXISTS treenode_parent_id_index")
```

If you actually add or delete tables or fields, make sure that the `models` dictionary is consistent with it (e.g. doesn't state a model has the field you just deleted manually).

Merge branches with migrations into branches with newer migrations

Of course, it can happen that one works on a branch where new migrations are added while another branch (e.g. upstream's master) got new migrations added, too. This might introduce problems when you want to merge one branch into the other.

For example, let's say the most recent migration on *master* starts with `0007`. You create a new topic branch based on this and you add a new migration with a name starting with `0008_add_column`. After some time you want to merge this branch back into *master*, which meanwhile also got a new migration with a name stating with `0008_add_table`.

If you just merge your branch, both migration files will be present next to each other. South loads migrations in ASCII sort order, so in principal both are at the correct position. This isn't really a problem *if* those migrations don't modify the same models. You can then simply run `migrate` with the `--merge` option to apply those out of order migrations.

Though, this works in most situations, it is not very pretty. As an alternative, you might want to consider the following: Re-create the migration(s) to have the correct ID, based on the upstream commits. This however needs some manual work. So before merging a branch, check whether there are conflicting IDs and, if so, do the following in the topic branch (referring to the example above):

1. Roll back the migrations to the last non-conflicting state, here `0007`:

   ```
   manage.py migrate catmaid 0007
   ```

2. Delete all conflicting migrations in the topic branch. If custom migration code has been added (like raw SQL), make sure to keep it around.
3. Merge the branch with the newer migrations into your topic branch (e.g. upstream/master).
4. Re-create your migrations (the new files will get correct IDs):

   ```
   manage.py schemamigration catmaid [title] --auto
   ```

   Note that this will create *one* migration containing all the database changes you made. Of course, you can also create migrations for single models if you want.

   If you have custom migration code, create new empty migrations and add your custom migration code to them:

   ```
   manage.py schemamigration catmaid [title] --empty
   ```

5. Migrate your database to make sure everything works and if so, create a new commit to add the new migrations
6. Merge the topic branch into the target branch

Also note that the South documentation has an own section on team workflows. You can find it here.

I just want to drop the database and start from scratch

If you're *really* sure that you don't need any of the data in your catmaid database, you can just drop the database and start again:

Drop the database:

```
sudo -u postgres dropdb catmaid
```

Run the commands generated by the *createuser.sh* script to make sure that the database, the database user, various functions and the plpgsql language are all created. The parameters to that script are the database name, the database user and the password for that database user:

```
scripts/createuser.sh catmaid catmaid_user p4ssw0rd | sudo -u postgres psql
```

(You may get errors saying that the user role has already been created, and that the functions already exist. You can safely ignore these.)

Now visit your CATMAID web page and the schema of the database will be updated. If you want to add back the example projects, you need to run the script *scripts/database/insert-example-projects.py*.

1.3.8 Django Unit Tests for CATMAID

If you want to be able to run the unit tests, you will need to allow the catmaid database user (`catmaid_user` by default) to create new databases.

Start a postgres shell with:

```
sudo -u postgres psql
```

You can change the role with:

```
postgres=# ALTER USER catmaid_user CREATEDB;
ALTER ROLE
```

. . . and you should also add this line at the top of `/etc/postgresql/XversionX/main/pg_hba.conf`:

```
local test_catmaid catmaid_user md5
```

. . . and then restart PostgreSQL:

```
sudo /etc/init.d/postgresql restart
```

Running tests

You can run the tests with:

```
./manage.py test
```

If you see an error:

```
DatabaseError: must be owner of extension plpgsql
```

Fix it with:

```
sudo -u postgres psql
ALTER ROLE catmaid_user WITH superuser;
```

Loading the test fixtures (schema and data) into a new CATMAID database

Directly editing the data in the test fixtures (`django/applications/catmaid/fixtures/catmaid_testdata.json`) would be very error-prone. A better idea is to create a new CATMAID instance pointing to a database that only contains the fixture data. Then you should only very carefully make changes to the CATMAID instance where they are required to support a new test you want to add.

To create such an instance, follow the usual installation instructions for CATMAID up to the point where you would use the `createuser.sh` script, and make sure that you specify a new database name, e.g. `catmaid_fixture`:

```
scripts/createuser.sh catmaid_fixture catmaid_user p4ssw0rd | sudo -u postgres psql
```

Then, rather than loading the usual example data, just import the fixture data:

```
psql -U catmaid_user catmaid_fixture < django/applications/catmaid/fixtures/catmaid_
↪testdata.json
```

If you want to start again and reload the fixture data into the test database, you can do:

```
sudo -u postgres dropdb catmaid_fixture
scripts/createuser.sh catmaid_fixture catmaid_user p4ssw0rd | sudo -u postgres psql
psql -U catmaid_user catmaid_fixture < django/applications/catmaid/fixtures/catmaid_
↪testdata.json
```

Dumping changes in a CATMAID instance back to the test fixtures

Suppose that to create the data for your test, you needed to add a skeleton and a neuron in your CATMAID instance. You should then dump the database back to the fixture files afterwards:

```
cd ~/catmaid
scripts/database/dump-database.sh catmaid_fixture > django/applications/catmaid/
↪fixtures/catmaid_testdata.json
```

Then run `git diff` to check that the additions to the fixtures make sense. (It's a good idea to check how these changes to the fixtures affect which tests pass.)

Test coverage

If you install `coverage.py` (`pip install coverage` in your virtualenv) you can generate test coverage statistics with:

```
coverage run ./manage.py test catmaid
```

Then run:

```
coverage html
```

. . . to generate HTML output in `htmlcov/index.html`.

GUI tests

To make sure some typical workflows work with development and release versions, we maintain a small set of Selenium GUI tests. Those are defined as Django unit tests and live under `django/applications/catmaid/tests/gui/`. To run these tests locally the `GUI_TESTS_ENABLED` setting has to be set to True or otherwise all GUI tests are ignored. Our CI test setup activates GUI tests automatically and also sets `GUI_TESTS_REMOTE` to True, which allows it to run Selenium tests on the virtual machines of saucelabs.com automatically.

1.3.9 Models, State and Commands

Models

To talk to CATMAID's back-end, its *API* is used. To make this more convenient and provide extra functionality, some of these APIs are abstracted into front-end models, which are defined in the JavaScript files in the `models` sub-folder of the CATMAID library:

```
django/applications/catmaid/static/lib/catmaid/models/
```

The majority of the common front-end operations can be found in there. A typical function, like node creation, has a signature like this:

```
CATMAID.Neurons.create: function(state, projectId, x, y, z, parentId, radius,
    confidence, useNeuron, neuronName)
```

All back-end parameters are available plus a state object. This state is required as a safety measure to not accidentally change data that was already updated by someone else. The next section goes into more detail about that.

State

In a collaborative environment, clients can never be sure if the information they see is the most recent one. Therefore, some CATMAID APIs support state checks to prevent changes by a client that was not aware of changes done by another client. Such a state is sent along with the request created by our front-end models and consists of information about the node of interest and its neighborhood.

To represent the (local) state the client sees the world in, different state implementations can be used. The tracing layer, for instance, has its own implementation and undo/redo utilizes a much sparser representation. States provide access to nodes, their state information and special serialization methods. State information on various parts of a local node neighborhood can be represented in parallel. This allows for flexibility and granular access control. Information on individual nodes, their parents, children and links can be stored. Connectors are supported as well.

A complete node *neighborhood state* consists of the *node*, *children*, *parent* and *links*. A node state represents a node ID along with an edition time, a parent state encapsulates this information about a parent of a node. Then there is also a *no chack state*, which causes the back-end to disable state checking for a request.

Different actions require different states, below you find a list of stateful endpoints and what they expect. This list isn't complete yet, some functions don't support state checks, yet.

Operation	Required state
Delete node	Neighborhood state for node to node
Create node	Parent state for node append, else none
Insert node	Node state and children of edge
Move node	Node state
Edit node radius	Node state
Edit node confidence	Node state
Create connector	For initial links partner node states, else none
Delete connector	Connector neighborhood state
Update connector confidence	Connector node state
Update connector links	Connector and link state
Create/update/remove annotation	Node state
Create/update/remove tag	Node state
Change neuron name	Neuron state
Link connector	Node and connector state
Unlink connector	Node and connector state

Undo

Some of the user user actions are reversible, they can be undone and redone. Undoing a command is as simple as pressing `Ctrl + Z`. Alternatively, the history dialog accessible through the `F9` key can be used, where redo can be selected as well. Actions that can be undone are listed below and CATMAID wraps these in so called *commands*. These maintain information about the applied changes and their inverse. This is a list of currently available commands and what their inverse operation is:

Operation	Inverse
Delete node	Create node, along with connectors
Create node	Delete node
Insert node	Delete node
Move node	Move node back
Edit node radius	Set original radius
Edit node confidence	Set original confidence
Create connector	Delete connector
Delete connector	Create connector and links
Update connector confidence	Set original confidence
Update connector links	Restore original links
Create/update/remove annotation	Delete/reset/create annotation
Create/update/remove tag	Delete/reset/create tag
Change neuron name	Set original name
Link connector	Unklink connector
Split skeleton	*Block undo*
Join skeletons	*Block undo*

Splitting and joining skeletons results at the moment in undo being blocked for this point in history. That is, commands executed before splitting or joining, can't be undone for now.

Commands are typically defined in the same file as the model functions they wrap.

Warning: The CATMAID AMI is not currently maintained and provides an outdated version of CATMAID. Consider *installing CATMAID from scratch* or *using a Docker image* instead.

If you are interested in updating the CATMAID AMI, please let us know on the CATMAID mailing list.

1.3.10 Creating a CATMAID Instance on EC2

If you use Amazon's EC2 service to host web services in the cloud, you may find it easiest to create a new EC2 instance with a running CATMAID from our new AMI (Amazon Machine Image). You can launch an EC2 instance (in the `eu-west-1` zone, directly from this link or if you want to find the AMI by hand in the public images in `eu-west-1`, its AMI ID is: `ami-ec545e98`.

This AMI is based on Canonical's Ubuntu 12.04 (precise) i386 image, backed by EBS. It will run in a "micro" instance (one of which is available in Amazon's free tier for a year if you sign up for the first time) but for a production server you would want a faster (and more expensive) virtual machine to run CATMAID on. (The CPU throttling on Micro instances will make performance very unpredictable.)

Launching the EC2 Instance

When you launch the EC2 instance, make sure you choose a security group (or security groups) that include at least SSH (TCP port 22) and HTTP (TCP port 80).

Logging In

You should log in over SSH with the username `ubuntu` and specifying as your identity file the private key that you downloaded from Amazon after creating your keypair; for example:

```
ssh -i ~/.ssh/my-aws.pem ubuntu@whatever.compute.amazonaws.com
```

The `ubuntu` user can gain root privileges using `sudo`, which doesn't require a password. The CATMAID code is owned by (and runs as) the `catmaid` user, so if you wish to change the code, you should switch to that user:

```
sudo su - catmaid
```

The source code is in `/home/catmaid/catmaid/`.

Server Configuration

The AMI is configured to use Nginx + Gunicorn; by default it is configured to use 4 synchronous worker threads, which can be changed in `/etc/init/gunicorn-catmaid.conf`. You can restart Gunicorn with:

```
sudo initctl restart gunicorn-catmaid
```

1.3.11 Creating CATMAID extensions

In the past, anyone wanting to extend CATMAID for their specific use case would need to fork the main repository, making it difficult to take advantage of future improvements to mainline CATMAID, and decreasing utility of the extension to people who may want to make use of it later. Therefore, we have designed the extension system to allow third parties to create external modules which interface with mainline CATMAID without having to change it.

If all that is needed is a modification or extension of front-end code (i.e. no new back-end functionality or database schema changes), it might be enough to configure static front-end extensions. What these are is explained at the end of this section.

Overview

CATMAID extensions are python modules which work as Django apps. When that module is made available to the python environment, Django can pick up any database models, API endpoints, and static files associated with the app, and code in the app can interact with code in mainline CATMAID. This modular approach allows much greater interoperability between different versions of CATMAID and the extension.

In the documentation below, we use a fictional extension called `myextension`.

Installing an Extension

1. Install the app into your python environment, either by using `pip install` from PyPI, or cloning the repo and using `pip` to install from the local `setup.py`
2. Run `python manage.py migrate` to update the database as necessary. WARNING: it is possible for a migration to irreversibly change or delete data in your existing database.
3. Run `python manage.py collectstatic` to pick up static files including stylesheets and frontend widgets.

API endpoints should be available at `BASE_URL/ext/myextension/...`

Note: N.B. CATMAID will only recognise extensions it knows about - i.e. those listed in `KNOWN_EXTENSIONS` in `CATMAID/django/projects/pipelinefiles.py`. Check this if it doesn't seem to be working.

Creating an extension

To quickstart development, you may find this cookiecutter template valuable: clbarnes/CATMAID-ext-cookiecutter. To do it yourself:

1. Decide on a name! We'll use `myextension` here.
2. Make a branch of CATMAID, adding `"myextension"` to `KNOWN_EXTENSIONS` in `CATMAID/django/projects/pipelinefiles.py`. Make a pull request for this to be included in mainline CATMAID - until then, just use this branch for testing. This should be the only required change.
3. Create a directory which will hold the module and repository-related cruft (we recommend naming it something obvious like `CATMAID-myextension`), navigate to it, and then create an empty django app with `django-admin startapp myextension`
4. Add an appropriate `README`, `LICENSE`, `setup.py`, `MANIFEST.in` and so on as laid out in Django's documentation, in `CATMAID-myextension`. The manifest and setup files are particularly important.
5. If your extension includes javascript and/or stylesheets, create `myextension/pipelinefiles.py` to make Django Pipeline aware of them. See synapsesuggestor and CATMAID for how they interoperate.
6. Develop away! For testing purposes, you will need to *install* the extension in your CATMAID environment - it's convenient to use `pip install -e` to install the module in editable mode and `python manage.py collectstatic -l`.

Examples

- CATMAID-synapsesuggestor
- `CATMAID-autoproofreader <https://github.com/pattonw/CATMAID-autoproofreader>'_
- `CircuitMap <https://github.com/BrainCircuitsIO/circuitmap>'_

Community Standards

- See the *contributing* page.
- Don't pick an extension name which may clash with other python modules.
- Don't write any migrations which will change data or database tables in the underlying CATMAID installation.
- Be aware of CATMAID's namespace - don't add dependencies or tables which could cause name collisions.
- As per Django's guidelines, namespace all static files, templates and so on to your app - e.g. static files should be in a directory called `myextension/static/myextension/<files>`

Simple front-end extensions

The CATMAID front-end can load extra JavaScript files that are not part of the regular source tree. This allows adding e.g. a new widget or custom analysis code without creating a full extension like explained above. This is sometimes easier and can be configured through three variables in the `settings.py` file.

Which additional files are loaded is defined in the `STATIC_EXTENSION_FILES` variable, which is expected to be a list of file names. They are expected to be relative to the path defined in `STATIC_EXTENSION_ROOT`. It is the responsibility of the webserver in use (e.g. Nginx) is expected to map the relative URL path in `STATIC_EXTENSION_URL` to the `STATIC_EXTENSION_ROOT` path, which allows the front-end to load the specified files from the static extension URL.

1.3.12 CATMAID Releases

New CATMAID releases are named after the date they are created, e.g. `2021-12-20`. Most of the process to create a new release is automated: a new release can be created with the script `scripts/dev/make-release.py`. This will ask the user a few questions related to the release (contributors, name, etc.) and will then create a new release branch, update the changelog and related files, create a new API doc version and update all version references.

This release branch can then be pushed to the main repository to have CI test it. Once it passes all tests it can be merged into the master/main branch.

As part of the release script the release commit is also tagged with the release name (e.g. `2021.12.20`). This tag jas to be pushed manually to GitHub and the `stable` tag has to be updated as well:

```
git push origin <tag-name>
git tag -f stable && git push origin +stable
```

As a last step, a new development cycle needs to be started by running the `scripts/dev/start-dev-cycle.py` script.

INDEX