**Running the VDJbase backend under PyCharm**

Requirements:

* Pycharm
* MySQL
* A Python virtual environment (I’ve assumed Conda)
* Git
* Db.sqlite3 databases for each dataset the backend should support
* A sample gene\_order.py

Steps:

1. Clone the code

git clone https://github.com/williamdlees/digby\_backend.git digby\_backend\_test

1. Build the Python environment

cd digby\_backend\_test  
conda create --name digby\_backend\_test python=3.7  
conda activate digby\_backend\_test

pip install -r requirements.txt

1. Create some directories and unzip sample files (from wherever you have stored them)

mkdir exports

mkdir exports\vdjbase\_metadata

mkdir static\output

cd static/study\_data/VDJbase

mkdir samples

mkdir samples\Human

mkdir samples\Human\Human\_IGH

unzip **samples.zip** -d samples\Human\Human\_IGH

mkdir samples\Human\New\_Pipeline\_IGH

unzip **new\_pipeline\_samples.zip** -d samples\Human\New\_Pipeline\_IGH

cd ../../..

1. Create a new database in your local MySQL with whatever name you like
2. Review the configuration entries in sample\_secret.cfg. Modify SQLALCHEMY\_DATABASE\_URI to suit your installation and save to secret.cfg. This file is deliberately excluded from github, so you may wish to store a copy somewhere.
3. Initialise the MySQL database (make sure that your Python venv is activated when running these commands):

(digby\_backend\_test) d:\Research\digby\_backend\_test>flask db init

Creating directory d:\Research\digby\_backend\_test\migrations ... done

Creating directory d:\Research\digby\_backend\_test\migrations\versions ... done

Generating d:\Research\digby\_backend\_test\migrations\alembic.ini ... done

Generating d:\Research\digby\_backend\_test\migrations\env.py ... done

Generating d:\Research\digby\_backend\_test\migrations\README ... done

Generating d:\Research\digby\_backend\_test\migrations\script.py.mako ... done

Please edit configuration/connection/logging settings in 'd:\\Research\\digby\_backend\_test\\migrations\\alembic.ini' before proceeding.

(digby\_backend\_test) d:\Research\digby\_backend\_test>flask db migrate

migrating app

<SQLAlchemy engine=mysql+mysqldb://root:\*\*\*@localhost/digby\_test?charset=utf8>

INFO [alembic.runtime.migration] Context impl MySQLImpl.

INFO [alembic.runtime.migration] Will assume non-transactional DDL.

INFO [alembic.autogenerate.compare] Detected added table 'role'

INFO [alembic.autogenerate.compare] Detected added table 'species'

INFO [alembic.autogenerate.compare] Detected added table 'study'

INFO [alembic.autogenerate.compare] Detected added table 'user'

INFO [alembic.autogenerate.compare] Detected added table 'ref\_seq'

INFO [alembic.autogenerate.compare] Detected added table 'roles\_users'

INFO [alembic.autogenerate.compare] Detected added table 'sequence'

INFO [alembic.autogenerate.compare] Detected added table 'feature'

INFO [alembic.autogenerate.compare] Detected added table 'sample'

INFO [alembic.autogenerate.compare] Detected added table 'sample\_sequence'

INFO [alembic.autogenerate.compare] Detected added table 'sequence\_feature'

Generating d:\Research\digby\_backend\_test\migrations\versions\2690f5c3ebdd\_.py ... done

(digby\_backend\_test) d:\Research\digby\_backend\_test>flask db upgrade

migrating app

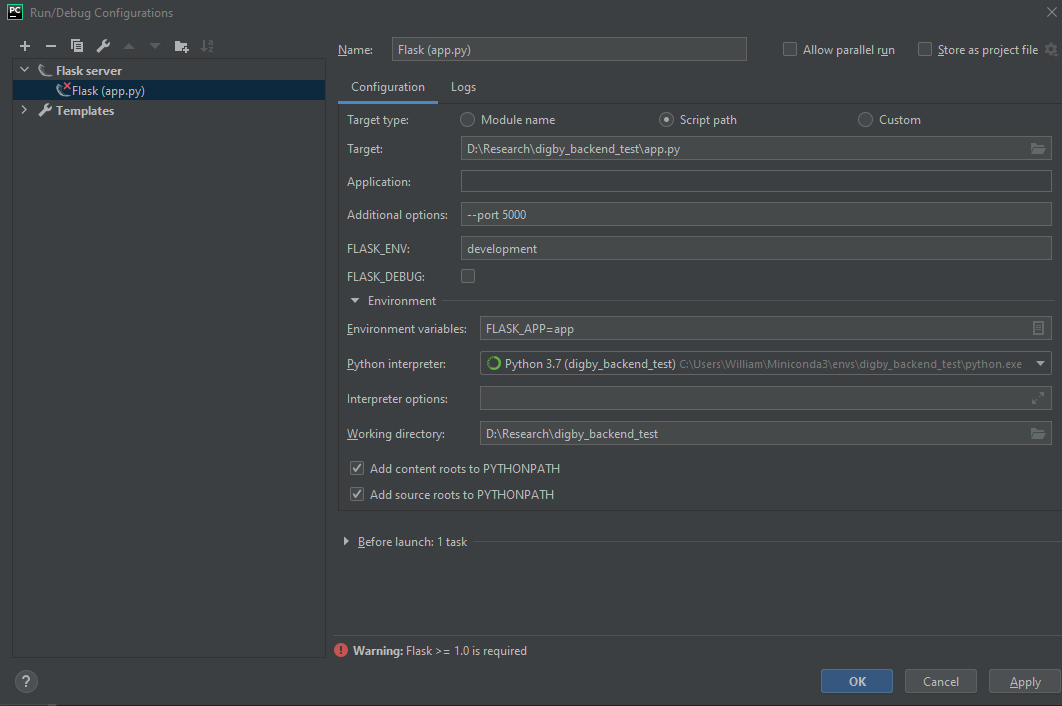
<SQLAlchemy engine=mysql+mysqldb://root:\*\*\*@localhost/digby\_test?charset=utf8>

INFO [alembic.runtime.migration] Context impl MySQLImpl.

INFO [alembic.runtime.migration] Will assume non-transactional DDL.

INFO [alembic.runtime.migration] Running upgrade -> 2690f5c3ebdd, empty message

1. Start Pycharm. Open the directory that you cloned the source to. Go to Run/Edit Configurations and create a new configuration that looks like this.



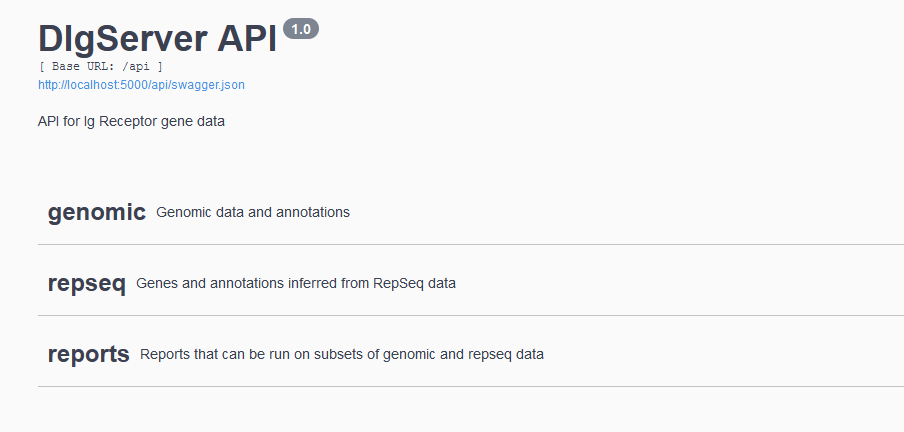
Notes:

* It should use the virtual environment that you set up with Conda
* The working directory should be the directory that you cloned the source to

1. Testing

Run the configuration in PyCharm.

In the browser, go to localhost:5000/api. You should see a screen like this:



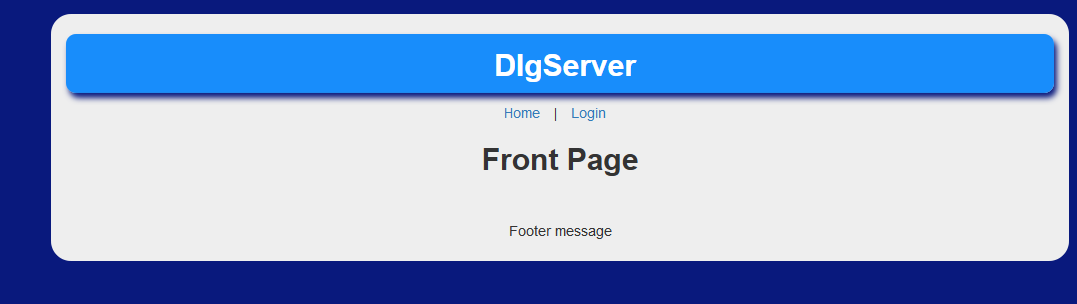
From here you can explore and test the API. For example, click on ‘repseq’ and then on the item marked ‘Returns information and statistics on the dataset’. Click ‘Try it out’ and enter Human as the species, and Human\_IGH as the dataset. In the server response you will see a list of samples and some other details. This is the API that is used to populate the ‘browse AIRR-Seq data’ page.

Some of the API calls can be complex to construct, but you can inspect the calls made by the front end: connect to ar.lees.org.uk/digby, press F12, and switch to the Network tab of the browser debug console. Now when you switch a tab in VDJbase or page the output, you will see the call that is sent to the backend. You can right-click on it and select ‘copy as cURL’ to save a copy that you can execute from the command line: just change the host name to localhost to run against your local version, and get rid of any extraneous options beyond the URL..

1. Backend Administration

First you need to create an admin user. To do this, browse to localhost:5000/create\_user. When there are no admin users in the SQL database, this creates a user with username ‘admin@vdjbase.org’ and password ‘admin’. You should see a message ‘User created’.

Now go to localhost:5000. You should see a screen like this:



Login with the newly created admin account.

There’s no user interface for maintenance functions at the moment, although you can go to Admin to modify the details of admin accounts if you like. To carry out other maintenance functions, you enter various URLs when logged in.

1. Importing RepSeq data from the pipeline

*A note on databases within VDJbase*

The MySQL database is used for genomic data and for admin account management. It isn’t used for RepSeq data.

RepSeq data is held in one or more sqlite databases, using the same schema as the old VDJbase. These databases, and the associated sample files, are held in the following file structure:

static/study\_data/VDJbase/db/<species>/<dataset>/db.sqlite3, db\_description.txt  
static/study\_data/VDJbase/samples/<species>/<dataset>/…

During the installation process, you created two human datasets, Human\_IGH and New\_Pipeline\_IGH.

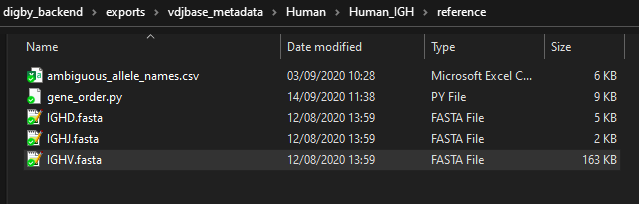
*Exporting Rep-Seq metadata*

It’s easiest to start with exports. To export, browse to localhost:5000/export\_vdjbase\_metadata . After a while, you should see the message ‘Export Complete’. The software builds a directory structure under exports/vdjbase\_metadata, which contains a projects.yml file for each dataset, reflecting the data in each sqlite database.

*Importing RepSeq data*

Repseq data for upload to VDJbase should be in the form of a zip, containing two directories, ‘reference’ and ‘samples’. Current data sets are maintained in the digby\_docker repo, in the datasets directory. Please refer to them for examples.

The ‘reference’ subdirectory should contain:  
- gene\_order.py – a file that specifies the order in which genes in this dataset should be displayed, both locus order and alpha order. There is a sample which this can be based on at db/sample\_gene\_order.py.  
- ambiguous\_allele\_names.csv – a file that lists alleles associated with an ambiguous name. There is a sample at sample\_ambiguous\_allele\_names.csv.  
- germline reference files for the genes referred to in the samples. V genes should be IMGT gapped. There can be any number of files with any names, as long as the extension is .fasta.



The samples subdirectory should contain the samples, using any directory structure and naming convention that has been used so far this year. It may also contain igsnper files, but it doesn’t have to (igsnper files are experimental at this stage).

The dataset directory *may* contain a projects.yml file, containing metadata for all the samples in the dataset. If this is present, it will be used. If not, the metadata will be taken from individual yml files in each sample directory (the import process will, in this case, construct a consolidated.yml file, and it can be useful to look in this for problems if the import fails).

RepSeq data is managed through the menu on the home page of the admin server on port 5000. From here, click on ‘Manage AIRR-seq Datasets’ to take you to a page where you can delete existing datasets and upload new ones. Note that uploading a dataset does not make it go live. It processes the data and builds production files for the dataset, but these must then be made live via the second link on the home page, ‘Copy AIRR-seq data to Live’. To replace an existing dataset, just upload the new zip file with the first link, and copy it live with the second. There’s no need to delete the old data.

Sometimes the server will hold files open, which can interfere with dataset deletion or copying. If you have any problems, restart the containers and retry the operation.

1. Building the genomic database

You don’t need to build the genomic data unless you want to be able to browse it, but here is the process as it stands – expect this to change.

Genomic data is, for the time being, stored in the Github repo under /static/study\_data. The MySQL database needs to be built from it, but the data doesn’t need to be prepared in the exports directory as for Rep-Seq data.

- Browse to http://localhost/update\_genomic  
- When this completes, browse to <http://localhost/build_gff>  
- After this, a script needs to be run which depends on SAMTools. As I develop under Windows, and SAMTools doesn’t work with Windows, I use the Windows Bash Shell for this. At any rate, on some device that has SAMTools installed, cd to static/gff and run the script make\_bam

At this point the genomic data should be available and browsable through the API.

1. Running Reports

Reports run as separate processes, managed by Celery. Status is returned via Redis. You don’t need to run Celery or Redis unless you want to run reports.

Celery is a Python package and is installed as one of the requirements in requirements.txt. Redis needs to be installed. There are packages for most Linux distributions, but there isn’t an installation for Windows: to use it with Windows, install in WSL as documented here: <https://redislabs.com/blog/redis-on-windows-10/>

To start redis under WSL, once it’s installed, open a WSL shell and use the command

sudo service redis-server restart

On another version of Linux, you might have to use a slightly different command, depending on your service management. With WSL, note that services do not persist once the shell is closed: so you will have to open a shell and restart the service like this whenever you want redis to be running.

Redis runs on its default port 6379. You can test it is running by executing this Python code: the last line should return True:

import redis

pool=redis.ConnectionPool(host='localhost', port=6379, db=0)

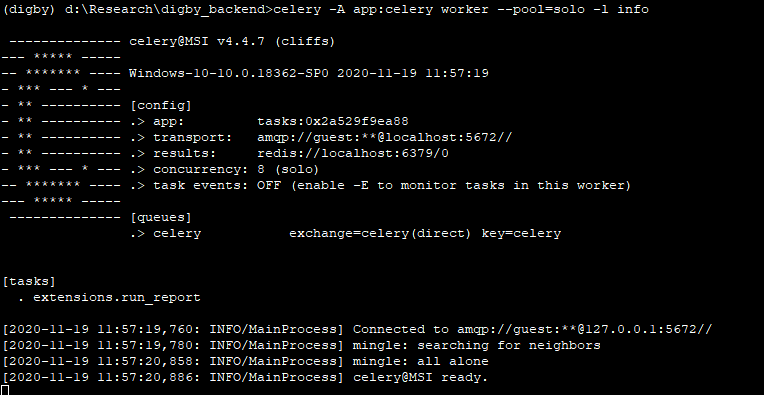
r=redis.Redis(connection\_pool=pool)

r.set('foo', 'bar')

To start Celery, open a fresh shell, cd to digby\_backend\_test, activate your Python environment and type:

celery -A app:celery worker --pool=solo -l info

You should see something similar to this:



You can now request reports from the front end, and observe them processing through Celery. Where the report is spawned to R, you will see the full command line, including associated data file pathnames. The files are preserved after the report is run, so you cam execute R separately with the same command line, in whatever debug environment you choose.

1. Brief notes on report structure

Code for reports is in api/reports. The set of available reports is defined in reports.json. In that file, the key (e.g. rep\_single\_genotype) refers to the file that corresponds to the report: in this case rep\_single\_genotype.py. That file is expected to define a function run(). When a report is requested, a Celery task will be started and it will call the corresponding run(). In the json, a params block defines parameters which are requested from the user before the report is requested: those are provided to run(), alongside the list of genomic and rep-seq samples on which the report should run, and the required format, selected from the set defined in reports.json.. Consult existing reports for examples.

Typically, run() will format the data required by the report, and then call the helper function run\_rscript(), If the report is written entirely in Pythion, the code can just be implemented in run(). The results are sent back to the user via the helper function send\_report().

1. Debugging python report code

You can’t step in to the report code run() function while celery is running, but you can disable it temporarily. In reports.py, find the get() function in ReportsRunApi and identify the call that is passed to celery:

run\_report.delay(report\_name, args.format, args.species, genomic\_samples, rep\_samples, params)

Remove the .delay in the function call and the run() code will be called directly rather than via celery:

run\_report.delay(report\_name, args.format, args.species, genomic\_samples, rep\_samples, params)