Apollo Documentation

Release 2.0.6

Apollo

Jun 23, 2017

Contents

Apollo - An instantaneous, collaborative, genome annotation editor.

The application's technology stack includes a Grails-based Java web application with flexible database backends and a Javascript client that runs in a web browser as a JBrowse plugin.

You can find the latest release here: https://github.com/GMOD/Apollo/releases/latest and our setup guide: http://genomearchitect.readthedocs.io/en/latest/Setup.html

For general information on Apollo, go to: https://genomearchitect.github.io/

For more information on JBrowse, please visit: http://jbrowse.org

Note: This documentation covers release versions 2.x of Apollo. For the 1.0.4 installation please refer to the installation guide found at http://genomearchitect.readthedocs.io/en/1.0.4/

Contents:

CHAPTER 1

Pre-requisites

Client pre-requisites

Apollo is a web-based application, so the only client side requirement is a web browser. Apollo has been tested on Chrome, Firefox, and Safari and matches the web browser requirements for JBrowse (see jbrowse.org for details).

Server-side pre-requisites

Note: see the Apollo 2.x quick-start for the quickest way to take care of pre-requisites.

- System pre-requisites (see quick-start guide for simple setup)
  - Any Unix like system (e.g., Unix, Linux, Mac OS X).
  - Servlet container (must support servlet spec 3.0+) such as tomcat 8 for production (not needed for development).
  - Java 8+ OpenJDK or Oracle should work.
  - npm 2.X or better / node.js
  - Grails (optional, but good for development). The easiest way to install is using sdkman, see Apollo 2.x quick-start for this step).
  - Ant 1.8+ (most package managers will have this).
  - A database (RDMS) system. Sample configurations for PostgreSQL and MySQL are available. H2 configuration does not require any manual installation.
  - Basic tools like Git, Curl, a text editor, etc.
- Data generation pipeline pre-requisites (for full list see http://gmod.org/wiki/JBrowse_Configuration_Guide)
  - System packages:
    - libpng12-0 (optional, for JBrowse imagetrack)

        * libpng12-dev (optional, for JBrowse imagetrack)
        * zlib1g (Debian/Ubuntu)
        * zlib1g-dev (Debian/Ubuntu)
        * zlib (RedHat/CentOS)
        * zlib-devel (RedHat/CentOS)
        * libexpat1-dev (Debian/Ubuntu)
        * expat-dev (RedHat/CentOS)
- Perl pre-requisites:
  - Apollo will automatically try to install all perl-pre-requisites.
  - If you are building Apollo in "release" mode, perl 5.10 or up will be required
- Sequence search (optional).
  - Blat (download Linux or OSX binaries).

Package manager commands

To install system pre-requisites, you can try the following commands

Debian/Ubuntu 16

```
sudo apt-get install openjdk-8-jdk curl libexpat1-dev postgresql postgresql-server-dev-all
git
```

CentOS/RedHat

```
sudo yum install postgresql postgresql-server postgresql-devel expat-devel
tomcat git curl
```

MacOSX/Homebrew

```
brew install postgresql tomcat git
```

CHAPTER 2

Quick-start Developer's guide

Here we will introduce how to setup Apollo on your server. In general, there are two modes of deploying Apollo.

There is "development mode" where the application is launched in a temporary server (automatically) and there is "production mode", which will typically require an external separate database and tomcat server where you can deploy the generated `war` file.

This guide will cover the "development mode" scenario which should be easy to start. **To setup in a production environment, please see the setup guide.**

Java / JDK

You have to install Java and the Java Development Kit (JDK) 8 or higher to run Apollo. Both the Oracle and OpenJDK versions have been tested.

Node.js / NPM / Bower

You will need to install node.js, which includes NPM (the node package manager) to build Apollo.

Once node.js / npm is installed, install bower using (you may need sudo):

```
npm install -g bower
```

Grails / Groovy / Gradle (optional)

Installing Grails (application framework), Groovy (development language), or Gradle (build environment) is not required (they will install themselves), but it is suggested for doing development.

This is most easily done by using SDKMAN (formerly GVM) which can automatically setup grails for you.

1. `curl -s http://get.sdkman.io | bash`
2. `sdk install grails 2.5.5`
3. `sdk install gradle 2.11`
4. `sdk install groovy`

Get the code

To setup Apollo, you can download our latest release from our official releases as compressed zip or tar.gz file (link at the bottom).

Alternatively you can check it out from git as directly as follows:

1. git clone https://github.com/GMOD/Apollo.git Apollo
2. cd Apollo
3. git checkout 2.X.Y # where X.Y is the tagged version you want from here: https://github.com/GMOD/Apollo/tags

Verify install requirements

We can now perform a quick-start of the application in "development mode" with this command:

```
./apollo run-local
```

The jbrowse and perl pre-requisites will be installed during this step, and if there is a success, then a temporary server will be automatically launched at `http://localhost:8080/apollo`.

Note: You can also supply a port number e.g. `apollo run-local 8085` if there are conflicts on port 8080.

Also note: if there are any errors at this step, check the setup.log file for errors. You can refer to the troubleshooting guide and often it just means the pre-requisites or perl modules failed.

Also also note: the "development mode" uses an in-memory H2 database for storing data by default. The setup guide will show you how to configure custom database settings.

Setting up the application

Setup a production server

To setup in a production environment, please see the setup guide. To setup (as opposed to a development server as above), you must properly configure a servlet container like Tomcat or Jetty with sufficient memory.

Adding data to Apollo

After we have a server setup, we will want to add a new organism to the panel. If you are a new user, you will want to setup this data with the jbrowse pre-processing scripts. You can see the data loading guide for more details, but essentially, you will want to load a reference genome and an annotations file at a minimum:

```
bin/prepare-refseqs.pl --fasta yourgenome.fasta --out /opt/apollo/data

bin/flatfile-to-json.pl --gff yourannotations.gff --type mRNA \
        --trackLabel AnnotationsGff --out /opt/apollo/data
```

Login to the web interface

After you access your application at http://localhost:8080/apollo/ then you will be prompted for login information

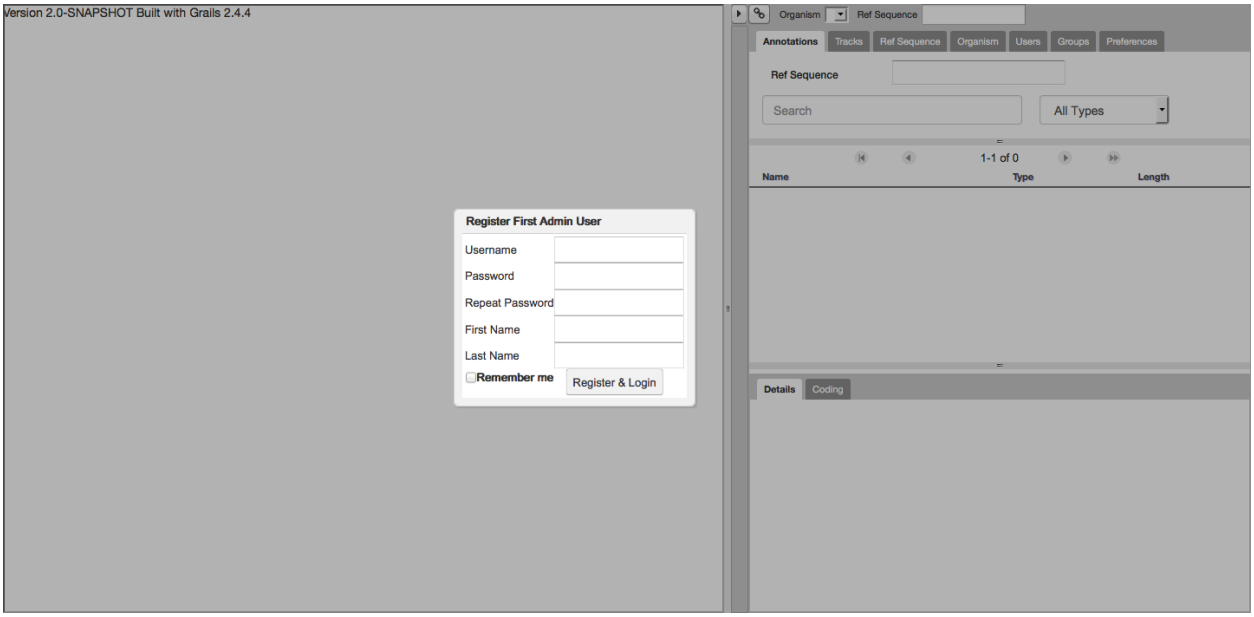


Figure 1. "Register First Admin User" screen allows you to create a new admin user.

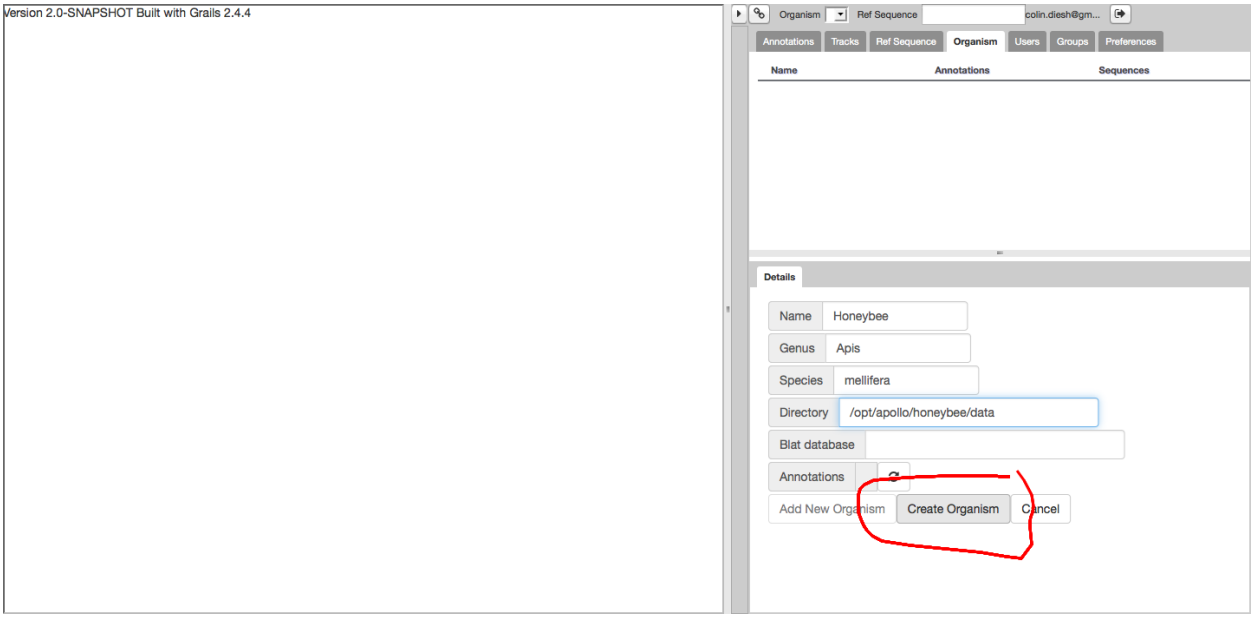


Figure 2. Navigate to the "Organism tab" and select "Create new organism". Then enter the new information for your organism. Importantly, the data directory refers to a directory that has been prepared with the JBrowse data loading scripts from the command line. See the data loading section for details.

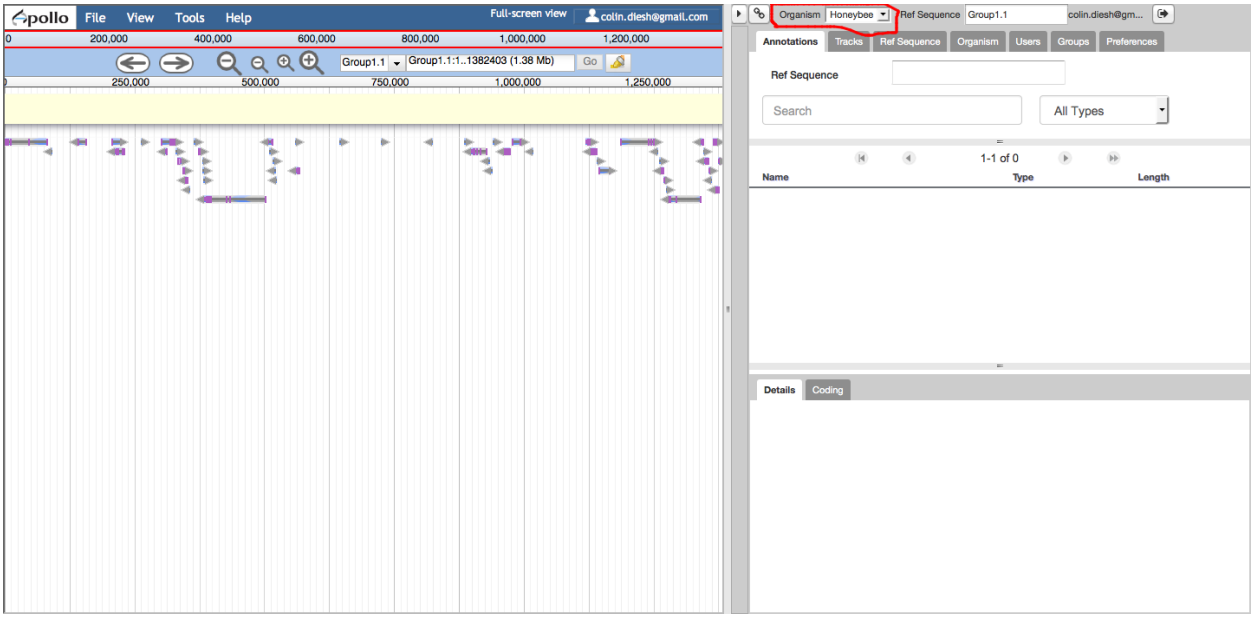


Figure 3. Open up the new organism from the drop down tab on the annotator panel.

Conclusion

If you completed this setup, you can then begin adding new users and performing annotations. Please continue to the setup guide for deploying the webapp to production or visit the troubleshooting guide if you encounter problems during setup.

CHAPTER 3

Setup guide

The quick-start guide showed how to quickly launch a temporary instance of Apollo, but deploying the application to production normally involves some extra steps.

The general idea behind your deployment is to create a `apollo-config.groovy` file from some existing sample files which have sample settings for various database engines.

Production pre-requisites

You will minimally need to have Java 8 or greater, Grails, git, ant, a servlet container e.g. tomcat7+, jetty, or resin. An external database such as PostgreSQL or MySQL is generally used for production, but instructions for the H2 database is also provided.

Important note: The default memory for Tomcat and Jetty is insufficient to run Apollo (and most other web apps).You should increase the memory according to these instructions.

Other possible build settings for JBrowse (based on an Ubuntu 16 install):

```
sudo apt-get update && sudo apt-get install zlib1g-dev libpng-dev libgd2-noxpm-dev␣
↪build-essential git python-software-properties
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
sudo apt-get install nodejs
```

NOTE: npm (installed with nodejs) must be version 2 or better. If not installed from the above instructions, most stable versions of node.js will supply this.

```
sudo npm install -g bower
```

NOTE: you must link nodejs to to node if your system installs it as a `nodejs` binary instead of a node one. E.g.,

```
sudo ln -s /usr/bin/nodejs /usr/bin/node
```

Build settings for Apollo specifically. Recent versions of tomcat7 will work, though tomcat8 is preferred. If it does not install automatically there are a number of ways to build tomcat on linux:

```
sudo apt-get install ant openjdk-8-jdk
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/  # or set in .bashrc / .project
```

Download Apollo from the latest release under source-code and unzip. Test installation by running `./apollo run-local` and see that the web-server starts up on http://localhost:8080/apollo/. To setup for production continue onto configuration below after install .

If you get an `Unsupported major.minor error` or similar, please confirm that the version of java that tomcat is running `ps -ef | grep java` is the same as the one you used to build. Setting JAVA_HOME to the Java 8 JDK should fix most problems.

Database configuration

Apollo supports several database backends, and you can choose sample configurations from using H2, Postgres, or MySQL by default.

Each has a file called `sample-h2-apollo-config.groovy` or `sample-postgres-apollo-config.groovy` that is designed to be renamed to apollo-config.groovy before running `apollo deploy`. Additionally there is a `sample-docker-apollo-config.groovy` which allows control of the configuration via environment variables.

Furthermore, the `apollo-config.groovy` has different groovy environments for test, development, and production modes. The environment will be selected automatically selected depending on how it is run, e.g:

- `apollo deploy` or `apollo release` use the production environment (i.e. when you copy the war file to your production server `apollo run-local` or `apollo debug` use the development environment (i.e. when you are running it locally)
- `apollo test` uses the test environment (i.e. only when running unit tests)

Configure for H2:

- H2 is an embedded database engine, so no external setups are needed. Simply copy sample-h2-apollo-config.groovy to apollo-config.groovy.

Configure for PostgreSQL:

- Create a new database with postgres and add a user for production mode. Here are a few ways to do this in PostgreSQL.
- Copy the sample-postgres-apollo-config.groovy to apollo-config.groovy.

Configure for MySQL:

- Create a new MySQL database for production mode (i.e. run ‘‘create database ‘apollo-production‘‘‘ in the mysql console) and copy the sample-postgres-apollo-config.groovy to apollo-config.groovy.

Configure for Docker:

- Set up and export all of the environment variables you wish to configure. At bare minimum you will likely wish to set `WEBAPOLLO_DB_USERNAME`, `WEBAPOLLO_DB_PASSWORD`, `WEBAPOLLO_DB_DRIVER`, `WEBAPOLLO_DB_DIALECT`, and `WEBAPOLLO_DB_URI`

- Create a new database in your chosen database backend and copy the sample-docker-apollo-config.groovy to apollo-config.groovy.
- Instructions and a script for launching docker with apollo and PostgreSQL.

Database schema

After you startup the application, the database schema (tables, etc.) is automatically setup. You don't have to initialize any database schemas yourself.

Deploy the application

The `apollo run-local` command only launches a temporary server and should really not be used in production, so to deploy to production, we build a new WAR file with the `apollo deploy` command. After you have setup your `apollo-config.groovy` file, and it has the appropriate username, password, and JDBC URL in it, then we can run the command:

```
./apollo deploy
```

This command will package the application and it will download any missing pre-requisites (jbrowse) into a WAR file in the "target/" subfolder. After it completes, you can then copy the WAR file (e.g. `apollo-2.0.4.war`) from the target folder to the `web-app` folder of your web container installation. If you name the file `apollo.war` in your webapps folder, then you can access your app at "http://localhost:8080/apollo"

We test primarily on Apache Tomcat (7.0.62+ and 8). **Make sure to set your Tomcat memory to an appropriate size or Apollo will run slow / crash.**

Alternatively, as we alluded to previously, you can also launch a temporary instance of the server which is useful for testing

```
./apollo run-local 8085
```

This temporary server will be accessible at "http://localhost:8085/apollo"

Note on database settings

If you use the `apollo run-local` command, then the "development" section of the apollo-config.groovy is used (or an temporary in-memory H2 database is used if no apollo-config.groovy exists).

If you use the WAR file generated by the `apollo deploy` command on your own webserver, then the "production" section of the apollo-config.groovy is used.

Detailed build instructions

While the shortcut `apollo deploy` takes care of basic application deployment, understanding the full build process of Apollo can help you to optimize and improve your deployed instances.

To learn more about the architecture of webapollo, view the architecture guide but the main idea here is to learn how to use `apollo release` to construct a build that includes javascript minimization

Pre-requisites for Javascript minimization

In addition to the system pre-requisites, the javascript compilation will use nodejs, which can be installed from a package manager on many platforms. Recommended setup for different platforms:

```
sudo apt-get install nodejs
sudo yum install epel-release npm
brew install node
```

Install extra perl modules

Building apollo in release mode also requires some extra Perl modules, namely Text::Markdown and DateTime. One way to install them:

```
bin/cpanm -l extlib DateTime Text::Markdown
```

Performing the javascript minimization

To build a Apollo release with Javascript minimization, you can use the command

```
./apollo release
```

This will compile JBrowse and Apollo javascript code into minimized files so that the number of HTTP requests that the client needs to make are reduced.

In all other respects, `apollo release` is exactly the same as `apollo deploy` though.

Performing active development

To perform active development of the codebase, use

```
./apollo debug
```

This will launch a temporary instance of Apollo by running `grails run-app` and `ant devmode` at the same time, which means that any changes to the Java files will be picked up, allowing fast iteration.

If you modify the javascript files (i.e. the client directory), you can run `scripts/copy_client.sh` and these will be picked up on-the-fly too.

CHAPTER 4

Apollo Configuration

Apollo includes some basic configuration parameters that are specified in configuration files. The most important parameters are the database parameters in order to get Apollo up and running. Other options besides the database parameters can be configured via the config files, but note that many parameters can also be configured via the web interface.

Note: Configuration options may change over time, as more configuration items are integrated into the web interface.

Main configuration

The main configuration settings for Apollo are stored in `grails-app/conf/Config.groovy`, but you can override settings in your `apollo-config.groovy` file (i.e. the same file that contains your database parameters). Here are the defaults that are defined in the Config.groovy file:

```
// default apollo settings
apollo {
  default_minimum_intron_size = 1
  history_size = 0
  overlapper_class = "org.bbop.apollo.sequence.OrfOverlapper"
  track_name_comparator = "/config/track_name_comparator.js"
  use_cds_for_new_transcripts = true
  user_pure_memory_store = true
  translation_table = "/config/translation_tables/ncbi_1_translation_table.txt"
  is_partial_translation_allowed = false // unused so far
  get_translation_code = 1
  sequence_search_tools = [
    blat_nuc: [
      search_exe: "/usr/local/bin/blat",
      search_class: "org.bbop.apollo.sequence.search.blat.
↪BlatCommandLineNucleotideToNucleotide",
      name: "Blat nucleotide",
      params: ""
    ],
    blat_prot: [
```

```
      search_exe: "/usr/local/bin/blat",
      search_class: "org.bbop.apollo.sequence.search.blat.
↪BlatCommandLineProteinToNucleotide",
      name: "Blat protein",
      params: ""
      tmp_dir: "/opt/apollo/tmp" //optional param, uses system tmp dir by default
    ]
  ]



  splice_donor_sites = [ "GT" ]
  splice_acceptor_sites = [ "AG"]
  gff3.source= "." bootstrap = false

  info_editor = {
    feature_types = "default"
    attributes = true
    dbxrefs = true
    pubmed_ids = true
    go_ids = true
    comments = true
  }
}
```

These settings are essentially the same familiar parameters from a config.xml file from previous Apollo versions. The defaults are generally sufficient, but as noted above, you can override any particular parameter in your `apollo-config.groovy` file, e.g. you can add override configuration any given parameter as follows:

```
grails {
  apollo.get_translation_code = 1
  apollo {
    use_cds_for_new_transcripts = true
    default_minimum_intron_size = 1
    get_translation_code = 1  // identical to the dot notation
  }
}
```

JBrowse Plugins

You can add / remove jbrowse plugins by copying a jbrowse section into your `apollo-config.groovy`.

There are two sections, `plugins` and `main`, which specifies the jbrowse version.

The main section can either contain a `git` block or a `url` block, both of which require `url`. If a git block a `tag` or `branch` can be specified.

In the `plugins` section, options are `included` (part of the JBrowse release), `url` (requiring a url parameter), or `git`, which can include a `tag` or `branch` as above.

Options for `alwaysRecheck` and `alwaysRepull` always check the branch and tag and always pull respectiviely.

Warning: The `NeatHTMLFeatures` and `NeatCanvasFeatures` plugins work very well in JBrowse instances. We are still in the process of testing and improving their performance in combination with the Apollo plugin. Until we finalize this process, we strongly advise caution if enabling them for use in your Apollo instances.

```
jbrowse {
    git {
        url= "https://github.com/GMOD/jbrowse"
//        tag = "1.12.1-release"
        branch = "master"
        alwaysPull = true
        alwaysRecheck = true
    }
//    url {
//        // always use dev for apollo
//        url = "http://jbrowse.org/wordpress/wp-content/plugins/download-monitor/
↪download.php?id=102"
//        type ="zip"
//        fileName = "JBrowse-1.12.0-dev"
//    }
    plugins {
        WebApollo{
            included = true
        }
        NeatHTMLFeatures{
            included = true
        }
        NeatCanvasFeatures{
            included = true
        }
        RegexSequenceSearch{
            included = true
        }
        HideTrackLabels{
            included = true
        }
//        MyVariantInfo {
//            git = 'https://github.com/GMOD/myvariantviewer'
//            branch = 'master'
//            alwaysRecheck = "true"
//            alwaysPull = "true"
//        }
//        SashimiPlot {
//            git = 'https://github.com/cmdcolin/sashimiplot'
//            branch = 'master'
//            alwaysPull = "true"
//        }
    }
}
```

Translation tables

The default translation table is 1

To use one of the others set the number in the `apollo-config.groovy` file as:

```
apollo {
...
  get_translation_code = "11"
}
```

These correspond to the NCBI translation tables.

To add a custom translation table, you can add it to to the `web-app/translation_tables` directory as:

```
web-app/translation_tables/ncbi_customname_translation_table.txt
```

and specify the `customname` as:

In apollo-config.groovy:

```
apollo {
...
  get_translation_code = "customname"
}
```

Logging configuration

To over-ride the default logging, you can look at the logging configurations from Config.groovy and override or modify them in `apollo-config.groovy`.

```
log4j.main = {
    error 'org.codehaus.groovy.grails.web.servlet',  // controllers
          'org.codehaus.groovy.grails.web.pages',    // GSP
          'org.codehaus.groovy.grails.web.sitemesh', // layouts
           ...
    warn 'grails.app'
}
```

Additional links for log4j:

- Advanced log4j configuration: http://blog.andresteingress.com/2012/03/22/grails-adding-more-than-one-log4j-configurations/
- Grails log4j guide: http://grails.github.io/grails-doc/2.4.x/guide/single.html#logging

Canned Elements

Canned comments, canned keys (tags), and canned values are configured using the Admin tab from the Annotator Panel on the web interface; these can no longer be created or edited using the configuration files. For more details on how to create and edit Canned Elements see Canned Elements.

View your instances page for more details. For example

- http://localhost:8080/apollo/cannedComment/
- http://localhost:8080/apollo/cannedKey/
- http://localhost:8080/apollo/cannedValue/

Search tools

Apollo can be configured to work with various sequence search tools. UCSC's BLAT tool is configured by default and you can customize it as follows by making modifications in the `apollo-config.groovy` file. Here we replace blat with blast (there is an existing wrapper for Blast). The database for each file will be passed in via params (globally)

or using the `Blat database` field in the organism tab. For blast the database will be the root name of the blast database files without the suffix.

```
apollo{
    sequence_search_tools {
        blat_nuc {
            search_exe = "/usr/local/bin/blastn"
            search_class = "org.bbop.apollo.sequence.search.blast.BlastCommandLine"
            name = "Blast nucleotide"
            params = ""
        }
        blat_prot {
            search_exe = "/usr/local/bin/tblastn"
            search_class = "org.bbop.apollo.sequence.search.blast.BlastCommandLine"
            name = "Blast protein to translated nucleotide"
            params = ""
            //tmp_dir: "/opt/apollo/tmp" optional param
        }
        your_custom_search_tool {
          search_exe = "/usr/local/customtool"
          search_class = "org.your.custom.Class"
          name: "Custom search"
        }
    }
}
```

When you setup your organism in the web interface, you can then enter the location of the sequence search database for BLAT.

Note: If the BLAT binaries reside elsewhere on your system, edit the search_exe location in the config to point to your BLAT executable.

Data adapters

Data adapters for Apollo provide the methods for exporting annotation data from the application. By default, GFF3 and FASTA adapters are supplied. They are configured to query your IOService URL e.g. http://localhost:8080/apollo/IOService with the customizable query

```
data_adapters = [[
  permission: 1,
  key: "GFF3",
  data_adapters: [[
    permission: 1,
    key: "Only GFF3",
    options: "output=file&format=gzip&type=GFF3&exportGff3Fasta=false"
  ],
  [
    permission: 1,
    key: "GFF3 with FASTA",
    options: "output=file&format=gzip&type=GFF3&exportGff3Fasta=true"
  ]]
],
[
  permission: 1,
  key : "FASTA",
```

```
  data_adapters :[[
    permission : 1,
    key : "peptide",
    options : "output=file&format=gzip&type=FASTA&seqType=peptide"
  ],
  [
    permission : 1,
    key : "cDNA",
    options : "output=file&format=gzip&type=FASTA&seqType=cdna"
  ],
  [
    permission : 1,
    key : "CDS",
    options : "output=file&format=gzip&type=FASTA&seqType=cds"
  ]]
]]
```

Default data adapter options

The options available for the data adapters are configured as follows

- type: `GFF3` or `FASTA`
- output: can be `file` or `text`. `file` exports to a file and provides a UUID link for downloads, text just outputs to stream.
- format: can by `gzip` or `plain`. `gzip` offers gzip compression of the exports, which is the default.
- exportSequence: `true` or `false`, which is used to include FASTA sequence at the bottom of a GFF3 export

Supported annotation types

Many configurations will require you to define which annotation types the configuration will apply to. Apollo supports the following "higher level" types (from the Sequence Ontology):

- sequence:gene
- sequence:pseudogene
- sequence:transcript
- sequence:mRNA
- sequence:tRNA
- sequence:snRNA
- sequence:snoRNA
- sequence:ncRNA
- sequence:rRNA
- sequence:miRNA
- sequence:repeat_region
- sequence:transposable_element

Apache / Nginx configuration

Oftentimes, admins will put use Apache or Nginx as a reverse proxy so that the requests to a main server can be forwarded to the tomcat server. This setup is not necessary, but it is a very standard configuration as is making modification to iptables.

Note that we use the SockJS library, which will downgrade to long-polling if websockets are not available, but since websockets are preferable, it helps to take some extra steps to ensure that the websocket calls are proxied or forwarded in some way too. If you are using tomcat 7, please make sure to use the most recent stable version, which supports web sockets by default. Using older versions (e.g. 7.0.26) websockets may not be included by default and you will need to include an additional .jar file.

Apache Proxy

The most simple setup on apache is as follows.. Here is the most basic configuration for a reverse proxy:

```
ProxyPass  /apollo http://localhost:8080/apollo
ProxyPassReverse  /apollo http://localhost:8080/apollo
```

Note: that a reverse proxy *does not* use `ProxyRequests On` (which turns on forward proxying, which is dangerous)

Also note: This setup will use downgrade to use AJAX long-polling without the websocket proxy being configured.

To setup the proxy for websockets, you can use mod_proxy_wstunnel, first load the module

```
LoadModule proxy_wstunnel_module libexec/apache2/mod_proxy_wstunnel.so
```

Then add extra ProxyPass calls for the websocket "endpoint" called `/apollo/stomp`

```
ProxyPass /apollo/stomp  ws://localhost:8080/apollo/stomp
ProxyPassReverse /apollo/stomp ws://localhost:8080/apollo/stomp
```

Debugging proxy issues

Note: if your webapp is accessible but it doesn't seem like you can login, you may need to customize the ProxyPassReverseCookiePath

For example, if you proxied to a different path, you might have something like this

```
ProxyPass  /testing http://localhost:8080
ProxyPassReverse  /testing http://localhost:8080
ProxyPassReverseCookiePath / /testing
```

Then your application might be accessible from http://localhost/testing/apollo

Nginx Proxy (from version 1.4 on)

Your setup may vary, but setting the upgrade headers can be used for the websocket configuration http://nginx.org/en/docs/http/websocket.html

```
    map $http_upgrade $connection_upgrade {
        default upgrade;
        ''      close;
    }
```

```
    server {
        # Main
        listen   80; server_name  myserver;

        # http://nginx.org/en/docs/http/websocket.html
        location /ApolloSever {
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $connection_upgrade;
            proxy_pass      http://127.0.0.1:8080;
        }
    }
```

Adding extra tabs

Extra tabs can be added to the side panel by over-riding the apollo configuration extraTabs:

```
    extraTabs = [
            ['title': 'extra1', 'url': 'http://localhost:8080/apollo/annotator/report/
↪'],
            ['title': 'extra2', 'content': '<b>Apollo</b> documentation <a href=
↪"http://genomearchitect.org" target="_blank">linked here</a>']
    ]
```

Upgrading existing instances

There are several scripts for migrating from older instances. See the migration guide for details. Particular notes:

Note: Apollo does not require using the `add-webapollo-plugin.pl` because the plugin is loaded implicitly by including the client/apollo/json/annot.json file at run time.

Upgrading existing JBrowse data stores

It is not necessary to change your existing JBrowse data directories to use Apollo 2.x, you can just point to existing data directories from your previous instances.

More information about JBrowse can also be found in their FAQ.

Adding custom CSS for track styling for JBrowse

There are a variety of different ways to include new CSS into the browser, but the easiest might be the following

Add the following statement to your trackList.json:

```
    "css" : "data/yourfile.css"
```

Then just place your CSS file in your organism's data directory.

Adding custom CSS globally for JBrowse

If you want to add CSS that is used globally for JBrowse, you can edit the CSS in the client/apollo/css folder, but since you need to re-deploy the app every time for updates, it is easier to just edit the data directories for your organisms (you do not need to re-deploy the app when you are editing organism specific data, since this is outside of the webapp directory and is not deployed with the WAR file)

Adding custom CSS globally for the GWT app

If you want to style the GWT sidebar, generally the bootstrap theme is used but extra CSS is also included from web-app/annotator/theme.css which overrides the bootstrap theme

Adding / using proxies

If you are https, or choose to use separate services rather than the default provided, you can setup a pass-through proxy or modify a particular URL.

This service is only available to logged-in users.

The internal proxy URL is:

`<apollo url>/proxy/request/<encoded_proxy_url>/`

For example if your URL the URL we want to proxy:

`http://golr.geneontology.org/solr/select`

encoded:

`http%3A%2F%2Fgolr.geneontology.org%2Fsolr%2Fselect`

If you user is logged-in and you pass in:

`http://localhost/apollo/proxy/request/http%3A%2F%2Fgolr.geneontology.org%2Fsolr%2Fselect?testkey=asdf&anotherkey=zxcv`

This will get proxied to:

`http://golr.geneontology.org/solr/select?testkey=asdf&anotherkey=zxcv`

If you choose to use another proxy service, you can go to the "Proxy" page (as administrator). Internally used proxies are provided by default. The order the final URL is chosen in is 'active' and then 'fallbackOrder'.

Register admin in configuration

If you want to register your admin user in the configuration, you can add a section to your `apollo-config.groovy` like:

```
apollo{
// other stuff
    admin{
        username = "super@duperadmin.com"
        password = System.getenv("APOLLO_ADMIN_PASSWORD")?:"demo"
        firstName = "Super"
        lastName = "Admin"
    }
}
```

It should only add the user a single time. User details can be retrieved from passed in text or from the environment depending on user preference.

Admin users will be added on system startup. Duplicate additions will be ignored.

Other authentication strategies

By default Apollo uses a username / password to authenticate users. However, additional strategies may be used.

To configure them, add them to the `apollo-config.groovy` and set active to true for the ones you want to use to authenticate.

```
apollo{
    // other stuff
    authentications = [
        ["name":"Username Password Authenticator",
         "className":"usernamePasswordAuthenticatorService",
         "active":true,
        ]
        ,
        ["name":"Remote User Authenticator",
         "className":"remoteUserAuthenticatorService",
         "active":false,
        ]
    ]
}
```

URL modifications

You should be able to pass in most JBrowse URL modifications to the `loadLink` URL.

You should use `tracklist=1` to force showing the native tracklist (or use the checkbox in the Track Tab in the Annotator Panel).

Use `openAnnotatorPanel=0` to close the Annotator Panel explicitly on startup.

Phone Home

In order to determine our usage and the current versions of Apollo being used (which helps us to provide Apollo for free), the server and the client will phone home and to google analytics.

To turn off the server phone home set the configuration this way.

```
apollo.phone.phoneHome = false
```

To add your own google analytics code set the code up this way:

```
google_analytics = ["UA-62921593-1","Your Google Analytics ID"]
```

If you don't want any reporting set:

```
google_analytics = []
```

CHAPTER 5

Chado Export Configuration

Following are the steps for setting up a Chado data source that is compatible with Apollo Chado Export.

Create a Chado database

First create a database in PostgreSQL for Chado.

Note: Initial testing has only been done on PostgreSQL.

Default name is `apollo-chado` and `apollo-production-chado` for development and production environment, respectively.

Create a Chado user

Now, create a database user that has all access privileges to the newly created Chado database.

Load Chado schema and ontologies

Apollo assumes that the Chado database has Chado schema v1.2 or greater and has the following ontologies loaded:

1. Relations Ontology
2. Sequence Ontology
3. Gene Ontology

The quickest and easiest way to do this is to use prebuilt Chado schemas. Apollo provides a prebuilt Chado schema with the necessary ontologies. (thanks to Eric Rasche at Center for Phage Technology, TAMU)

Users can load this prebuilt Chado schema as follows:

```
scripts/load_chado_schema.sh -u <USER> -d <CHADO_DATABASE> -h <HOST> -p <PORT> -s
↪<CHADO_SCHEMA_SQL>
```

If there is already an existing database with the same name and if you would like to dump and create a clean database:

```
scripts/load_chado_schema.sh -u <USER> -d <CHADO_DATABASE> -h <HOST> -p <PORT> -s
↪<CHADO_SCHEMA_SQL> -r
```

The '-r' flag tells the script to perform a pg_dump if <CHADO_DATABASE> exists.

e.g.,

```
scripts/load_chado_schema.sh -u postgres -d apollo-chado -h localhost -p 5432 -r -s␣
↪chado-schema-with-ontologies.sql.gz
```

The file chado-schema-with-ontologies.sql.gz can be found in Apollo/scripts/ directory.

The load_chado_schema.sh script creates log files which can be inspected to see if loading the schema was successful.

Note that you will also need to do this for your testing and production instances, as well.

Configure data sources

In apollo-config.groovy, uncomment the configuration for datasource_chado and specify the proper database name, database user name and database user password.

Export via UI

Users can export existing annotations to the Chado database via the Annotator Panel -> Ref Sequence -> Export.

Export via web services

Users can also leverage the Apollo web services API to export annotations to Chado. As a demonstration, a sample script, export_annotations_to_chado.groovy is provided.

Usage for the script:

```
export_annotations_to_chado.groovy -organism ORGANISM_COMMON_NAME -username APOLLO_
↪USERNAME -password APOLLO_PASSWORD -url http://localhost:8080/apollo
```

CHAPTER 6

Data generation pipeline

The data generation pipeline is based on the typical jbrowse commands such as prepare-refseqs.pl and flatfile-to-json.pl, and these scripts are automatically copied to a local bin/ directory when you run the setup scripts (e.g. `apollo run-local` or `apollo deploy` or `install_jbrowse.sh`).

If you don't see a bin/ subdirectory containing these scripts after running the setup, check setup.log and check the troubleshooting guide for additional tips or feel free to post the error and setup.log on GitHub or the mailing list.

prepare-refseqs.pl

The first step to setup the genome browser is to load the reference genome data. We'll use the `prepare-refseqs.pl` script to output to the data directory.

```
bin/prepare-refseqs.pl --fasta pyu_data/scf1117875582023.fa --out /opt/apollo/data
```

Note: the output directory is used later when we load the organism into the browser with the "Create organism" form

flatfile-to-json.pl

The flatfile-to-json.pl script can be used to load GFF3 files and you can customize the feature types. Here, we'll start off by loading data from the MAKER GFF for the Pythium ultimum data. The simplest loading command specifies a –trackLabel, the –type of feature to load, the –gff file and the –out directory.

```
bin/flatfile-to-json.pl --gff pyu_data/scf1117875582023.gff --type mRNA \
        --trackLabel MAKER --out /opt/apollo/data
```

Note: you can also use the command `bin/maker2jbrowse` for loading the MAKER data.

Also see the section Customizing features section for more information on customizing the CSS styles of the Apollo features.

Note: Apollo uses features that are loaded at the "transcript" level. If your GFF3 has "gene" features with "transcript"/"mRNA" child features, make sure that you use the argument –type mRNA or –type transcript.

generate-names.pl

Once data tracks have been created, you can generate a searchable index of names using the generate-names.pl script:

```
bin/generate-names.pl --verbose --out /opt/apollo/data
```

This is optional but useful step to index of names and features and refseq names. If you have some tracks that have millions of features, consider only indexing select tracks with the –tracks argument or disabling autocomplete with `--completionLimit 0`.

add-bam-track.pl

Apollo natively supports BAM files and the file can be read (in chunks) directly from the server with no preprocessing.

To add a BAM track, copy the .bam and .bam.bai files to your data directory, and then use the add-bam-track.pl to add the file to the tracklist.

```
mkdir /opt/apollo/data/bam
cp pyu_data/simulated-sorted.bam /opt/apollo/data/bam
cp pyu_data/simulated-sorted.bam.bai /opt/apollo/data/bam
bin/add-bam-track.pl --bam_url bam/simulated-sorted.bam \
   --label simulated_bam --key "simulated BAM" -i /opt/apollo/data/trackList.json
```

Note: the `bam_url` parameter is a URL that is relative to the data directory. It is not a filepath! Also, the .bai will automatically be located if it is simply the .bam with .bai appended to it.

add-bw-track.pl

Apollo also has native support for BigWig files (.bw), so no extra processing of these files is required either.

To use this, copy the BigWig data into the jbrowse data directory and then use the add-bw-track.pl to add the file to the tracklist.

```
mkdir /opt/apollo/data/bigwig
cp pyu_data/*.bw /opt/apollo/data/bigwig
bin/add-bw-track.pl --bw_url bigwig/simulated-sorted.coverage.bw \
    --label simulated_bw --key "simulated BigWig"
```

Note: the `bw_url` parameter is a URL that is relative to the data directory. It is not a filepath!

Customizing different annotation types (advanced)

To change how the different annotation types look in the "User-created annotation" track, you'll need to update the mapping of the annotation type to the appropriate CSS class. This data resides in `client/apollo/json/annot.json`, which is a file containing Apollo tracks that is loaded by default. You'll need to modify the JSON entry whose label is `Annotations`. Of particular interest is the `alternateClasses` element. Let's look at that default element:

```
"alternateClasses": {
    "pseudogene" : {
       "className" : "light-purple-80pct",
       "renderClassName" : "gray-center-30pct"
    },
    "tRNA" : {
       "className" : "brightgreen-80pct",
       "renderClassName" : "gray-center-30pct"
    },
    "snRNA" : {
       "className" : "brightgreen-80pct",
       "renderClassName" : "gray-center-30pct"
    },
    "snoRNA" : {
       "className" : "brightgreen-80pct",
       "renderClassName" : "gray-center-30pct"
    },
    "ncRNA" : {
       "className" : "brightgreen-80pct",
       "renderClassName" : "gray-center-30pct"
    },
    "miRNA" : {
       "className" : "brightgreen-80pct",
       "renderClassName" : "gray-center-30pct"
    },
    "rRNA" : {
       "className" : "brightgreen-80pct",
       "renderClassName" : "gray-center-30pct"
    },
    "repeat_region" : {
       "className" : "magenta-80pct"
    },
    "transposable_element" : {
       "className" : "blue-ibeam",
       "renderClassName" : "blue-ibeam-render"
    }
}
```

For each annotation type, you can override the default class mapping for both `className` and `renderClassName` to use another CSS class. Check out the Customizing features section for more information on customizing the CSS classes.

Customizing features

The visual appearance of biological features in Apollo (and JBrowse) is handled by CSS stylesheets with HTMLFeatures tracks. Every feature and subfeature is given a default CSS "class" that matches a default CSS style in a CSS stylesheet. These styles are are defined in `client/apollo/css/track_styles.css` and `client/apollo/css/webapollo_track_styles.css`. Additional styles are also defined in these files, and can be used by explicitly specifying them in the –className, –subfeatureClasses, –renderClassname, or –arrowheadClass parameters to flatfile-to-json.pl (see data loading section).

Apollo differs from JBrowse in some of it's styling, largely in order to help with feature selection, edge-matching, and dragging. Apollo by default uses invisible container elements (with style class names like "container-16px") for features that have children, so that the children are fully contained within the parent feature. This is paired with another styled element that gets rendered *within* the feature but underneath the subfeatures, and is specified by the

`--renderClassname` argument to `flatfile-to-json.pl`. Exons are also by default treated as special invisible containers, which hold styled elements for UTRs and CDS.

It is relatively easy to add other stylesheets that have custom style classes that can be used as parameters to `flatfile-to-json.pl`. For example, you can create `/opt/apollo/data/custom_track_styles.css` which contains two new styles:

```
.gold-90pct,
.plus-gold-90pct,
.minus-gold-90pct  {
    background-color: gold;
    height: 90%;
    top: 5%;
    border: 1px solid gray;
}

.dimgold-60pct,
.plus-dimgold-60pct,
.minus-dimgold-60pct  {
    background-color: #B39700;
    height: 60%;
    top: 20%;
}
```

In this example, two subfeature styles are defined, and the *top* property is being set to (100%-height)/2 to assure that the subfeatures are centered vertically within their parent feature. When defining new styles for features, it is important to specify rules that apply to plus-*stylename* and minus-*stylename* in addition to *stylename*, as Apollo adds the "plus-" or "minus-" to the class of the feature if the the feature has a strand orientation.

You need to tell Apollo where to find these styles by modifying the JBrowse config or the plugin config, e.g. by adding this to the trackList.json

```
"css" : "data/custom_track_styles.css"
```

Then you may use these new styles using –subfeatureClasses, which uses the specified CSS classes for your features in the genome browser, for example:

```
bin/flatfile-to-json.pl --gff MyFile.gff \
   --type mRNA --trackLabel MyTrack      \
   --subfeatureClasses '{"CDS":"gold-90pct","UTR": "dimgold-60pct"}'
```

Bulk loading annotations to the user annotation track

GFF3

You can use the `tools/data/add_features_from_gff3_to_annotations.pl` script to bulk load GFF3 files with transcripts to the user annotation track. Let's say we want to load our `maker.gff` transcripts.

```
tools/data/add_features_from_gff3_to_annotations.pl \
    -U localhost:8080/Apollo -u web_apollo_admin -p web_apollo_admin \
    -i scf1117875582023.gff -t mRNA -o "name of organism"
```

The default options should be able to handle most GFF3 files that contain genes, transcripts, and exons.

You can still use this script even if the GFF3 file that you are loading does not contain transcripts and exon types. Let's say we want to load `match` and `match_part` features as transcripts and exons respectively. We'll use the

`blastn.gff` file as an example.

```
    tools/data/add_features_from_gff3_to_annotations.pl \
       -U localhost:8080/Apollo -u web_apollo_admin -p web_apollo_admin \
       -i cf1117875582023.gff -t match -e match_part -o "name of organism"
```

You can view the add_features_from_gff3_to_annotations.pl help (`-h`) option for all available options.

Note: Apollo makes a clear distinction between a transcript and an mRNA. Genes that have mRNA as its child feature are treated as protein coding annotations and Genes that have transcript as its child feature are treated as non-coding annotations, specifically a pseudogene.

If you would like to look at a compatible representative GFF3, export annotations from Apollo via GFF3 export.

Disable draggable

Apollo has a number of specific track config parameters

```
overrideDraggable (boolean)
determines whether to transform the alignments tracks to draggable alignments

overridePlugins (boolean)
determines whether to transform alignments and sequence tracks
```

These can be specified on a specific track or in a global config.

CHAPTER 7

How to contribute code to Apollo

Audience

These guidelines are for developers of Apollo software, whether internal or in the broader community.

Basic principles of the Apollo-flavored GitHub Workflow

Principle 1: Work from a personal fork

- Prior to adopting the workflow, a developer will perform a *one-time setup* to create a personal Fork of apollo and will subsequently perform their development and testing on a task-specific branch within their forked repo. This forked repo will be associated with that developer's GitHub account, and is distinct from the shared repo managed by GMOD.

Principle 2: Commit to personal branches of that fork

- Changes will never be committed directly to the master branch on the shared repo. Rather, they will be composed as branches within the developer's forked repo, where the developer can iterate and refine their code prior to submitting it for review.

Principle 3: Propose changes via pull request of personal branches

- Each set of changes will be developed as a task-specific *branch* in the developer's forked repo, and then create a pull request will be created to develop and propose changes to the shared repo. This mechanism provides a way for developers to discuss, revise and ultimately merge changes from the forked repo into the shared Apollo repo.

Principle 4: Delete or ignore stale branches, but don't recycle merged ones

- Once a pull request has been merged, the task-specific branch is no longer needed and may be deleted or ignored. It is bad practice to reuse an existing branch once it has been merged. Instead, a subsequent branch and pull-request cycle should begin when a developer switches to a different coding task.
- You may create a pull request in order to get feedback, but if you wish to continue working on the branch, so state with "DO NOT MERGE YET".

Table of contents

One Time Setup - Forking a Shared Repo

The official shared `Apollo` repository is intended to be modified solely via pull requests that are reviewed and merged by a set of responsible 'gatekeeper' developers within the Apollo development team. These pull requests are initially created as task-specific named branches within a developer's personal forked repo.

Typically, a developer will fork a shared repo once, which creates a personal copy of the repo that is associated with the developer's GitHub account. Subsequent pull requests are developed as branches within this personal forked repo. The repo need never be forked again, although each pull request will be based upon a new named branch within this forked repo.

Step 1 - Backup your existing repo (optional)

The Apollo team has recently adopted the workflow described in this document. Many developers will have an existing clone of the shared repo that they have been using for development. This cloned local directory must be *moved aside* so that a proper clone of the forked repo can be used instead.

If you do not have an existing local copy of the shared repo, then skip to Step 2 *below.*

Step 2 - Fork `apollo` via the Web

The easiest way to fork the `apollo` repository is via the GitHub web interface:

- Ensure you are logged into GitHub as your GitHub user.
- Navigate to the apollo shared repo at https://github.com/GMOD/apollo.
- Notice the 'Fork' button in the upper right corner. It has a number to the right of the button. images/githubForkButton.p
- Click the Fork button. The resulting behavior will depend upon whether your GitHub user is a member of a GitHub organization. If not a member of an organization, then the fork operation will be performed and the forked repo will be created in the user's account.
- If your user is a member of an organization (e.g., GMOD or acme-incorporated), then GitHub will present a dialog for the user to choose where to place the forked repo. The user should click on the icon corresponding to their username. images/githubForkTarget.png
- *If you accidentally click the number, you will be on the Network Graphs page and should go back.*

Step 3 - Clone the Fork Locally

At this point, you will have a fork of the shared repo (e.g., apollo) stored within GitHub, but it is not yet available on your local development machine. This is done as follows:

```
# Assumes that directory ~/MI/ will contain your Apollo repos.
# Assumes that your username is MarieCurie.
# Adapt these instructions to suit your environment
> cd ~/MI
> git clone git@github.com:MarieCurie/apollo.git
> cd apollo
```

Notice that we are using the SSH transport to clone this repo, rather than the HTTPS transport. The telltale indicator of this is the `git@github.com:MarieCurie...` rather than the alternative `https://github.com/MarieCurie...`.

Note: If you encounter difficulties with the above `git clone`, you may need to associate your local public SSH key with your GitHub account. See Which remote URL should I use? for information.

Step 4 - Configure the local forked repo

The `git clone` above copied the forked repo locally, and configured the symbolic name 'origin' to point back to the *remote* GitHub fork. We will need to create an additional *remote* name to point back to the shared version of the repo (the one that we forked in Step 2). The following should work:

```
# Assumes that you are already in the local apollo directory
> git remote add upstream https://github.com/GMOD/apollo.git
```

Verify that remotes are configured correctly by using the command `git remote -v`. The output should resemble:

```
upstream    https://github.com/GMOD/apollo.git (fetch)
upstream    https://github.com/GMOD/apollo.git (push)
origin  git@github.com:MarieCurie/apollo.git (fetch)
origin  git@github.com:MarieCurie/apollo.git (push)
```

Step 5 - Configure `.bashrc` to show current branch (optional)

One of the important things when using Git is to know what branch your working directory is tracking. This can be easily done with the `git status` command, but checking your branch periodically can get tedious. It is easy to configure your `bash` environment so that your current git branch is always displayed in your bash prompt.

If you want to try this out, add the following to your `~/.bashrc` file:

```
function parse_git_branch()
{
  git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ \1/'
}
LIGHT_GRAYBG="\[\033[0;47m\]"
LIGHT_PURPLE="\[\033[0;35m\]"
NO_COLOR="\[\033[0m\]"
export PS1="$LIGHT_PURPLE\w$LIGHT_GRAYBG\$(parse_git_branch)$NO_COLOR \$ "
```

You will need to open up a new Terminal window (or re-login to your existing terminal) to see the effect of the above `.bashrc` changes.

If you cd to a git working directory, the branch will be displayed in the prompt. For example:

```
~ $
~ $ # This isn't a git directory, so no branch is shown
~ $
```

```
~ $ cd /tmp
/tmp $
/tmp $ # This isn't a git directory, so no branch is shown
/tmp $
/tmp $ cd ~/MI/apollo/
~/MI/apollo fix-feedback-button $
~/MI/apollo fix-feedback-button $ # The current branch is shown
~/MI/apollo fix-feedback-button $
~/MI/apollo fix-feedback-button $ git status
On branch fix-feedback-button
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    ... remaining output of git status elided ...
```

Typical Development Cycle

Once you have completed the One-time Setup above, then it will be possible to create new branches and pull requests using the instructions below. The typical development cycle will have the following phases:

- Refresh and clean up local environment
- Create a new task-specific branch
- Perform ordinary development work, periodically committing to the branch
- Prepare and submit a Pull Request (PR) that refers to the branch
- Participate in PR Review, possibly making changes and pushing new commits to the branch
- Celebrate when your PR is finally Merged into the shared repo.
- Move onto the next task and repeat this cycle

Refresh and clean up local environment

Git will not automatically sync your Forked repo with the original shared repo, and will not automatically update your local copy of the Forked repo. These tasks are part of the developer's normal *cycle*, and should be the first thing done prior to beginning a new development effort and creating a new branch. In addition, this

Step 1 - Fetch remotes

In the (likely) event that the *upstream* repo (the apollo shared repo) has changed since the developer last began a task, it is important to update the local copy of the upstream repo so that its changes can be incorporated into subsequent development.

```
> git fetch upstream        # Updates the local copy of shared repo BUT does not␣
↪affect the working directory, it simply makes the upstream code available locally␣
↪for subsequent Git operations. See step 2.
```

Step 2 - Ensure that 'master' is up to date

Assuming that new development begins with branch 'master' (a good practice), then we want to make sure our local 'master' has all the recent changes from 'upstream'. This can be done as follows:

```
> git checkout master
> git reset --hard upstream/master
```

The above command is potentially dangerous if you are not paying attention, as it will remove any local commits to master (which you should not have) as well as any changes to local files that are also in the upstream/master version (which you should not have). In other words, the above command ensures a proper clean slate where your local master branch is identical to the upstream master branch.

Some people advocate the use of `git merge upstream/master` or `git rebase upstream/master` instead of the `git reset --hard`. One risk of these options is that unintended local changes accumulate in the branch and end up in an eventual pull request. Basically, it leaves open the possibility that a developer is not really branching from upstream/master, but is branching from some developer-specific branch point.

Create a new branch

Once you have updated the local copy of the master branch of your forked repo, you can create a named branch from this copy and begin to work on your code and pull-request. This is done with:

```
> git checkout -b fix-feedback-button   # This is an example name
```

This will create a local branch called 'fix-feedback-button' and will configure your working directory to track that branch instead of 'master'.

You may now freely make modifications and improvements and these changes will be accumulated into the new branch when you commit.

If you followed the instructions in *Step 5 - Configure `.bashrc` to show current branch (optional)*, your shell prompt should look something like this:

```
~/MI/apollo fix-feedback-button $
```

Changes, Commits and Pushes

Once you are in your working directory on a named branch, you make changes as normal. When you make a commit, you will be committing to the named branch by default, and not to master.

You may wish to periodically `git push` your code to GitHub. Note the use of an explicit branch name that matches the branch you are on (this may not be necessary; a git expert may know better):

```
> git push origin fix-feedback-button   # This is an example name
```

Note that we are pushing to 'origin', which is our forked repo. We are definitely NOT pushing to the shared 'upstream' remote, for which we may not have permission to push.

Reconcile branch with upstream changes

If you have followed the instructions above at *Refresh and clean up local environment*, then your working directory and task-specific branch will be based on a starting point from the latest-and-greatest version of the shared repo's

master branch. Depending upon how long it takes you to develop your changes, and upon how much other developer activity there is, it is possible that changes to the upstream master will conflict with changes in your branch.

So it is a good practice to periodically pull down these upstream changes and reconcile your task branch with the upstream master branch. At the least, this should be performed prior to submitting a PR.

Fetching the upstream branch

The first step is to fetch the update upstream master branch down to your local development machine. Note that this command will NOT affect your working directory, but will simply make the upstream master branch available in your local Git environment.

```
> git fetch upstream
```

Rebasing to avoid Conflicts and Merge Commits

Now that you've fetched the upstream changes to your local Git environment, you will use the `git rebase` command to adjust your branch

```
> # Make that your changes are committed to your branch
> # before doing any rebase operations
> git status
    # ... Review the git status output to ensure your changes are committed
    # ... Also a good chance to double-check that you are on your
    # ... task branch and not accidentally on master
> git rebase upstream/master
```

The rebase command will have the effect of adjusting your commit history so that your task branch changes appear to be based upon the most recently fetched master branch, rather than the older version of master you may have used when you began your task branch.

By periodically rebasing in this way, you can ensure that your changes are in sync with the rest of Apollo development and you can avoid hassles with merge conflicts during the PR process.

Dealing with merge conflicts during rebase

Sometimes conflicts happen where another developer has made changes and committed them to the upstream master (ideally via a successful PR) and some of those changes overlap with the code you are working on in your branch. The `git rebase` command will detect these conflicts and will give you an opportunity to fix them before continuing the rebase operation. The Git instructions during rebase should be sufficient to understand what to do, but a very verbose explanation can be found at Rebasing Step-by-Step

Advanced: Interactive rebase

As you gain more confidence in Git and this workflow, you may want to create PRs that are easier to review and best reflect the intent of your code changes. One technique that is helpful is to use the *interactive rebase* capability of Git to help you clean up your branch prior to submitting it as a PR. This is completely optional for novice Git users, but it does produce a nicer shared commit history.

See squashing commits with rebase for a good explanation.

Submitting a PR (pull request)

Once you have developed code and are confident it is ready for review and final integration into the upstream version, you will want to do a final `git push origin ...` (see Changes, Commits and Pushes above). Then you will use the GitHub website to perform the operation of creating a Pull Request based upon the newly pushed branch.

See submitting a pull request.

Reviewing a pull request

The set of open PRs for the apollo can be viewed by first visiting the shared apollo GitHub page at https://github.com/GMOD/apollo.

Click on the 'Pull Requests' link on the right-side of the page: images/githubPullRequest.png

Note that the Pull Request you created from your forked repo shows up in the shared repo's Pull Request list. One way to avoid confusion is to think of the shared repo's PR list as a queue of changes to be applied, pending their review and approval.

Respond to TravisCI tests

The GitHub Pull Request mechanism is designed to allow review and refinement of code prior to its final merge to the shared repo. After creating your Pull Request, the TravisCI tests for apollo will be executed automatically, ensuring that the code that 'worked fine' on your development machine also works in the production-like environment provided by TravisCI. The current status of the tests can be found near the bottom of the individual PR page, to the right of the Merge Request symbol: images/githubTravisPrStatus.png

TBD - Something should be written about developers running tests PRIOR to TravisCI and the the PR. This may already be in the README.html, but should be cited.

Respond to peer review

The GitHub Pull Request mechanism is designed to allow review and refinement of code prior to its final merge to the shared repo. After creating your Pull Request, the TravisCI tests for apollo will be executed automatically, ensuring that the code that 'worked fine' on your development machine also works in the production-like environment provided by TravisCI. The current status of the tests can be found

Repushing to a PR branch

It's likely that after created a Pull Request, you will receive useful peer review or your TravisCI tests will have failed. In either case, you will make the required changes on your development machine, retest your changes, and you can

then push your new changes back to your task branch and the PR will be automatically updated. This allows a PR to evolve in response to feedback from peers. Once everyone is satisfied, the PR may be merged. (see below).

Merge a pull request

One of the goals behind the workflow described here is to enable a large group of developers to meaningfully contribute to the Apollo codebase. The Pull Request mechanism encourages review and refinement of the proposed code changes. As a matter of informal policy, Apollo expects that a PR will not be merged by its author and that a PR will not be merged without at least one reviewer approving it (via a comment such as +1 in the PR's Comment section).

Celebrate and get back to work

You have successfully gotten your code improvements into the shared repository. Congratulations! The branch you created for this PR is no longer useful, and may be deleted from your forked repo or may be kept. But in no case should the branch be further developed or reused once it has been successfully merge. Subsequent development should be on a new branch. Prepare for your next work by returning to *Refresh and clean up local environment*.

GitHub Tricks and Tips

- Add `?w=1` to a GitHub file compare URL to ignore whitespace differences.

References and Documentation

- The instructions presented here are derived from several sources. However, a very readable and complete article is Using the Fork-and-Branch Git Workflow. Note that the article doesn't make clear that certain steps like Forking are one-time setup steps, after which Branch-PullRequest-Merge steps are used; the instructions below will attempt to clarify this.
- New to GitHub? The GitHub Guides are a great place to start.
- Advanced GitHub users might want to check out the GitHub Cheat Sheet

CHAPTER 8

Troubleshooting guide

Tomcat memory

Typically, the default memory allowance for the Java Virtual Machine (JVM) is too low. The memory requirements for Web Apollo will depend on many variables, but in general, we recommend at least 1g for the heap size and 256m for the PermGen size as a starting point.

Suggested Tomcat memory settings

```
export CATALINA_OPTS="-Xms512m -Xmx1g \
        -XX:+CMSClassUnloadingEnabled \
        -XX:+CMSPermGenSweepingEnabled \
        -XX:+UseConcMarkSweepGC"
```

In cases where the assembled genome is highly fragmented, additional tuning of memory requirements and garbage collection will be necessary to maintain the system stable. Below is an example from a research group that maintains over 40 Apollo instances with assemblies that range from 1,000 to 150,000 scaffolds (reference sequences):

```
export CATALINA_OPTS="-Xmx12288m -Xms8192m \
        -XX:ReservedCodeCacheSize=64m \
        -XX:+UseG1GC \
        -XX:+CMSClassUnloadingEnabled \
        -Xloggc:$CATALINA_HOME/logs/gc.log \
        -XX:+PrintHeapAtGC \
        -XX:+PrintGCDetails \
        -XX:+PrintGCTimeStamps"
```

To change your settings, you can *usually* edit the setenv.sh script in `$TOMCAT_BIN_DIR/setenv.sh` where `$TOMCAT_BIN_DIR` is the directory where the Tomcat binaries reside. It is possible that this file doesn't exist by default, but it will be picked up when Tomcat restarts. Make sure that tomcat can read the file.

In most cases, creating the setenv.sh should be sufficient but you may have to edit a catalina.sh or another file directly depending on your system and tomcat setup. For example, on Ubuntu, the file /etc/default/tomcat7 often contains these

settings.

Confirm your settings

Your CATALINA_OPTS settings from setenv.sh can be confirmed with a tool like jvisualvm or via the command line with the `ps` tool. e.g. `ps -ef | grep java` should yield something like the following allowing you to confirm that your memory settings have been picked up.

```
root      9848     1  0 Oct22 ?        00:36:44 /usr/lib/jvm/java-7-openjdk-amd64/bin/
↪java -Djava.util.logging.config.file=/usr/local/tomcat/current/conf/logging.
↪properties -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Xms1g␣
↪-Xmx2g -XX:+CMSClassUnloadingEnabled -XX:+CMSPermGenSweepingEnabled -
↪XX:+UseConcMarkSweepGC -Dj
```

Re-install after changing settings

If you start seeing memory leaks (`java.lang.OutOfMemoryError: Java heap space`) after doing an update, you might try re-installing, as the live re-deploy itself can cause memory leaks or an inconsistent software state.

If you have named your web application named `Apollo.war` then you can remove all of these files from your webapps directory and re-deploy.

- Run `apollo deploy` (or `apollo release` for javascript-minimization)
- Undeploy any existing Apollo instances
- Stop tomcat
- Copy the war file to the webapps folder
- Start tomcat

Tomcat permissions

Preferably, when running Apollo or any webserver, you should not run Tomcat as root. Therefore, when deploying your war file to tomcat or another web application server, you may need to tune your file permissions to make sure Tomcat is able to access your files.

On many production systems, tomcat will typically belong to a user and group called something like 'tomcat'. Make sure that the 'tomcat' user can read your "webapps" directory (where you placed your war file) and write into the annotations and any other relevant directory (e.g. tomcat/logs). As such, it is sometimes helpful to add the user you logged-in as to the same group as your tomcat user and set group write permissions for both.

Consider using a package manager to install Tomcat so that proper security settings are installed, or to use the jsvc http://tomcat.apache.org/tomcat-7.0-doc/security-howto.html#Non-Tomcat_settings

Errors with JBrowse

JBrowse tools don't show up in `bin` directory (or install at all) after install or typing `install_jbrowse.sh`

If the `bin` directory with JBrowse tools doesn't show up after calling `install_jbrowse.sh` JBrowse is having trouble installing itself for a few possible reasons. If these do not work, please observe the JBrowse troubleshooting and JBrowse install pages, as well and the `setup.log` file created during the installation process.

cpanm or other components are not installed

Make sure the appropriate JBrowse libraries are installed on your system.

If you see `chmod: cannot access `web-app/jbrowse/bin/cpanm': No such file or directory` make sure to install cpanm.

Git tool is too old

Git expects to clone a single branch which is supported in git 1.7.10 and greater. The output when that fails looks something like this:

```
Buildfile: build.xml

copy.apollo.plugin.webapp:

setup-jbrowse:

git.clone:
[exec] Result: 129
```

The solution is to upgrade git to 1.7.10 or greater or remove the line with the `--single-branch` option in `build.xml`.

Accessing git behind a firewall.

If you are behind a firewall, checking out code using the `git://` protocol may not be allowed, but that is the default. The output will look something like this:

```
setup-jbrowse:

git.clone:
     [exec] Submodule 'src/FileSaver' (git://github.com/dkasenberg/FileSaver.js.git)␣
↪registered for path 'src/FileSaver'
     [exec] Submodule 'src/dbind' (git://github.com/rbuels/dbind.git) registered for␣
↪path 'src/dbind'
    . . . .
     [exec] Submodule 'src/xstyle' (git://github.com/kriszyp/xstyle.git) registered␣
↪for path 'src/xstyle'
     [exec] Result: 1
```

with possibly more output below.

Type:

```
git config --global url."https://".insteadOf git://
```

in the command-line and then re-install using `./apollo clean-all ./apollo run-local` (or deploy or release).

e.g. "Can't locate Hash/Merge.pm in @INC" or "Can't locate JBlibs.pm in @INC"

If you are trying to run the jbrowse binaries but get these sorts of errors, try running `install_jbrowse.sh` which will initialize as many pre-requisites as possible including JBLibs and other JBrowse dependencies.

Rebuilding JBrowse

You can manually clear jbrowse files from web-app/jbrowse and re-run `apollo deploy` to rebuild JBrowse.

RequestError: Unable to load ... Apollo2/jbrowse/data/trackList.json status: 500

Apollo2 does fairly strict JSON validation so make sure your trackList.json file is valid JSON

If you still get this error after validating please forward the issue to our github issue tracker.

Complaints about 8080 being in use

Please check that you don't already have a tomcat running `netstat -tan | grep 8080`. Sometimes tomcat does not exit properly. `ps -ef | grep java` and then `kill -9` the offending processing.

Note that you can also configure tomcat to run on different ports, or you can launch a temporary instance of apollo with `apollo run-local 8085` for example to avoid the port conflict.

Unable to open the h2 / default database for writing

If you receive an error similar to this:

```
SEVERE: Unable to create initial connections of pool.
org.h2.jdbc.JdbcSQLException: Error opening database:
    "Could not save properties /var/lib/tomcat7/prodDb.lock.db" [8000-176]
```

Then this is due to the production server trying to write an h2 instance in an area it doesn't have permissions to. If you use H2 (which is great for testing or single-user user, but not for full-blown production) make sure that:

You can modify the specified data directory for the H2 database in the apollo-config.groovy. For example, using the /tmp/ directory, or some other directory:

```
url = "jdbc:h2:/tmp/prodDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT=FALSE"
```

This will write a H2 db file to `/tmp/prodDB.db`. If you don't specify an absolute path it will try to write in the same directory that tomcat is running in e.g., `/var/lib/tomcat7/` which can have permission issues.

More detail on database configuration when specifying the `apollo-config.groovy` file is available in the setup guide.

Grails cache errors

In some instances you can't write to the default cache location on disk. Part of an example config log:

```
2015-07-03 14:37:39,675 [main] ERROR context.GrailsContextLoaderListener  - Error␣
↪initializing the application: null
java.lang.NullPointerException
        at grails.plugin.cache.ehcache.GrailsEhCacheManagerFactoryBean
↪$ReloadableCacheManager.rebuild(GrailsEhCacheManagerFactoryBean.java:171)
        at grails.plugin.cache.ehcache.EhcacheConfigLoader.reload(EhcacheConfigLoader.
↪groovy:63)
        at grails.plugin.cache.ConfigLoader.reload(ConfigLoader.groovy:42)
```

There are several solutions to this, but all involve updating the `apollo-config.groovy` file to override the caching defined in the Config.groovy.

Disabling the cache:

```
    grails.cache.config = {
        cache {
            enabled = false
            name 'globalcache'
        }
    }
```

This can also be done by removing the plugin. In `grails-app/conf/BuildConfig` remove / comment out the line and re-building:

```
 compile ':cache-ehcache:1.0.5'
```

Disallow writing overflow to disk

Can be used for small instances

```
    grails.cache.config = {
        // avoid ehcache naming conflict to run multiple WA instances
        provider {
            name "ehcache-apollo-"+(new Date().format("yyyyMMddHHmmss"))
        }
        cache {
            enabled = true
            name 'globalcache'
            eternal false
            overflowToDisk false   // THIS IS THE IMPORTANT LINE
            maxElementsInMemory 100000
        }
    }
```

Specify the overflow directory

Best for high load servers, which will need the cache. Make sure your tomcat / web-server user can write to that directory:

```
// copy from Config.groovy except where noted
grails.cache.config = {
...
    cache {
    ...
        maxElementsOnDisk 10000000
        // this is the important part, below!
        diskStore{
            path '/opt/apollo/cache-directory'
        }
    }
    ...
}
```

Information on the grails ehcache plugin (see "Overriding values") and ehcache itself.

Mysql invalid TimeStamp error

For certain version of MySQL we might get errors of this nature:

> SQLException occurred when processing request: [GET] /apollo/annotator/getAppState Value '0000-00-00 00:00:00' can not be represented as java.sql.Timestamp. Stacktrace follows: java.sql.SQLException: Value '0000-00-00 00:00:00' can not be represented as java.sql.Timestamp

The fix is to set the `zeroDateTimeBehavior=convertToNull` to the url connect screen. Originally identified here. Here is an example URL:

```
jdbc:mysql://localhost/apollo_production?zeroDateTimeBehavior=convertToNull&
↪autoReconnect=true&characterEncoding=UTF-8&characterSetResults=UTF-8
```

CHAPTER 9

Migration guide

This guide explains how to prepare your Apollo 2.x instance, and to migrate data from previous Web Apollo versions into 2.0.

In all cases you will need to follow the guide for setting up your 2.x instance.

Migration from Evaluation to Production:

If you are running your evaluation/development version using `./apollo run-local` when you setup your production instance, any prior annotations will use a separate database.

If you are using the same production instance you can use scripts to delete all annotations and preferences:

```
scripts/delete_all_features.sh
```

or just the annotations:

```
scripts/delete_only_features.sh
```

If you want to start from scratch (including reloading organisms and users), you can just drop the database (when the server is not running) and the proper tables will be recreated on startup.

Migration from 2.0.X to 2.0.Y on production:

Installation from a downloaded release

- Download the desired Apollo release from the bottom of each release. Official releases will be tagged as "Release" and have a green label.
- Expand the archive.
- Copy your existing apollo-config.groovy file into the directory.
- Always backup your database!

- Create a new war file as below: `./apollo deploy`.
- Turn off tomcat and remove the old apollo directory and `.war` file in the webapps folder.
- Copy in new .war file with the same name.
- Restart tomcat and you are ready to go.

Note if you you choose to have two different versions of Apollo running, though need to point to different database instances or you will experience problems.

Installation from a checked out github

If you want bleeding and only moderately tested code (not recommended unless you feel you know what you're doing), you can clone Apollo directly from our source page https://github.com/GMOD/Apollo/

Any upgrading can be taken care of during a pull. Please note that as we sometimes change the version of JBrowse, so you should do:

```
./apollo clean-all
```

before building a target for production.

You can the follow the directions for deploying a downloaded release, above.

Migration from 1.0 to 2.0:

We provide examples in the form of [migration scripts](https://github.com/gmod/apollo/tree/master/docs/web_services/examples) in the docs/web_services/examples directory. These tools are also described in the command line tools section.

We have written many of the command line tools examples using the groovy language, but mostly any language will work (Perl, shell/curl, Python, etc.).

Migrate Annotations

We provide a [migration script](https://github.com/gmod/apollo/tree/master/docs/web_services/examples/groovy/ migrate_annotations1to2.groovy) that connects to a single Web Apollo 1 instance and populates the annotations for an organism for a set of sequences / (confusingly called tracks as well). It would be best to develop your script on a development instance of Apollo2 for restricted sequences.

To get the scripts working properly, you'll need to provide the list of sequences (or tracks) to migrate for each organism. You can get the list of tracks by either using the database (`select * from tracks ;`) or looking in the Web Apollo annotations directory

```
ls -1 /opt/apollo/annotations/ | grep Annotations | grep -v history | paste -s -d"," -
```

Migrate Users

You have to add users de novo using something like the add_users.groovy script. In this case you create a csv file with the email, name, password, and role ('user' or 'admin'). This is passed into the add_users.groovy script and users are added.

From Web Apollo 1, you should be able to pull user names out of the database `select * from users ;`, but there is not much overlap between users in Web Apollo1.x and Apollo2.x.

If you have only a few users, however, just adding them manually on the users will likely be easier.

Add Organisms

If possible adding organisms on the organisms tab is the easiest option if you only have a handful of organisms.

The [add_organism.groovy script](https://github.com/gmod/apollo/tree/master/docs/web_services/examples/groovy/add_organism.groovy) can help automate this process if you have a large number of migrations to handle.

CHAPTER 10

Permissions guide

Global

- **admin**: access to everything
- **user**: only guarantees a login with permissions configured on organism basis

Organism

Can only view things related to that organism.

- **read**: view / search only, no annotation

```
Annotations: lock detail / coding
RefSeq: hide export
Organism: hide
User: hide
Group: hide
Preferences: hide
JBrowse: disable UcA track
```

- **export**: same as read, but can use the export screen

```
RefSeq: show export
```

- **write**: same as above, but can add / edit annotations

```
Annotations: allow editing
JBrowse: enable UcA track
```

- **admin**: access to everything for that organism

```
    Organism: show
    User: show
    Group: show
    Preferences: (still hide)
```

Table of permissions:

```
| Permission | Annotator          | Users/groups  | Annotations         | Organism
↪              |
|------------|--------------------|---------------|---------------------|-------------
↪--------------|
| READ       | visible / locked   | hide          | visible / no export | visible
↪              |
| EXPORT     | visible / locked   | hide          | visible / export    | visible
↪              |
| WRITE      | visible + editable | hide          | visible / export    | visible
↪              |
| ADMIN      | visible + editable | visible       | visible /export     | visible +
↪admin functions |
| NONE       | not available      | not available | not available       | not visible
↪              |
```

The Preference panel is available only for GLOBAL admin.

CHAPTER 11

Automated testing architecture

The Apollo unit testing framework uses the grails testing guidelines extensively, which can be reviewed here: http://grails.github.io/grails-doc/2.4.3/guide/testing.html

Our basic methodology is to run the full test suite with the apollo command:

```
apollo test
```

More specific tests can also be run for example by running specific commands for `grails test-app`

```
grails test-app :unit-test
```

This runs ALL of the tests in "test/unit". If you want to test a specific function then write it something like this:

```
grails test-app org.bbop.apollo.FeatureService :unit
```

Notes about the test suites:

1. @Mock includes any domain objects you'll use. Unit tests don't use the database.
2. The setup() function is run for each test
3. The test is composed of blocks of code with `when:` and `then:`. You have to have both or it is not a test.

Example test:

```
@TestFor(FeatureService)
@Mock([Sequence,FeatureLocation,Feature])
  class FeatureServiceSpec extends Specification {
  void setup(){}
  void "convert JSON to Feature Location"(){

  when: "We have a valid json object"
  JSONObject jsonObject = new JSONObject()
  Sequence sequence = new Sequence(name: "Chr3",
```

```
    seqChunkSize: 20, start:1, end:100, length:99).save(failOnError: true)
  jsonObject.put(FeatureStringEnum.FMIN.value,73)
  jsonObject.put(FeatureStringEnum.FMAX.value,113)
  jsonObject.put(FeatureStringEnum.STRAND.value, Strand.POSITIVE.value)


  then: "We should return a valid FeatureLocation"
  FeatureLocation featureLocation =
    service.convertJSONToFeatureLocation(jsonObject,sequence)
  assert featureLocation.sequence.name == "Chr3"
  assert featureLocation.fmin == 73
  assert featureLocation.fmax == 113
  assert featureLocation.strand ==Strand.POSITIVE.value
} }
```

There are 3 "special" types of things to test, which are all important and reflect the grails special functions: Domains, Controllers, Services. They will all be in the "test" directory and all be suffixed with "Spec" for a Spock test.

Chado

If you test with the chado export you will need to make sure you load ontologies into your chado database or integration steps will fail. If you don't specify chado in your apollo-config.groovy then no further action would be necessary.

```
./scripts/load_chado_schema.sh -u nathandunn -d apollo-chado-test -s chado-schema-
↪with-ontologies.sql.gz -r
```

CHAPTER 12

Architecture notes

Overview and quick-start

See the build doc for the official quick-start guide.

Minimally, the apollo application can be launched by running `apollo run-local`. This starts up a temporary tomcat server automatically. It will also simply use a in-memory H2 database if a different database configuration isn't setup yet.

For development purposes, you can also enable automatic code reloading which helps for fast iteration.

- `grails -reloading run-app` will allow changes to the server side code to be auto-reloaded.
- `ant devmode` will provide auto-reloading of GWT code changes
- `scripts/copy_client.sh` will copy the plugin code to the web-apps folder to update the plugin javascript

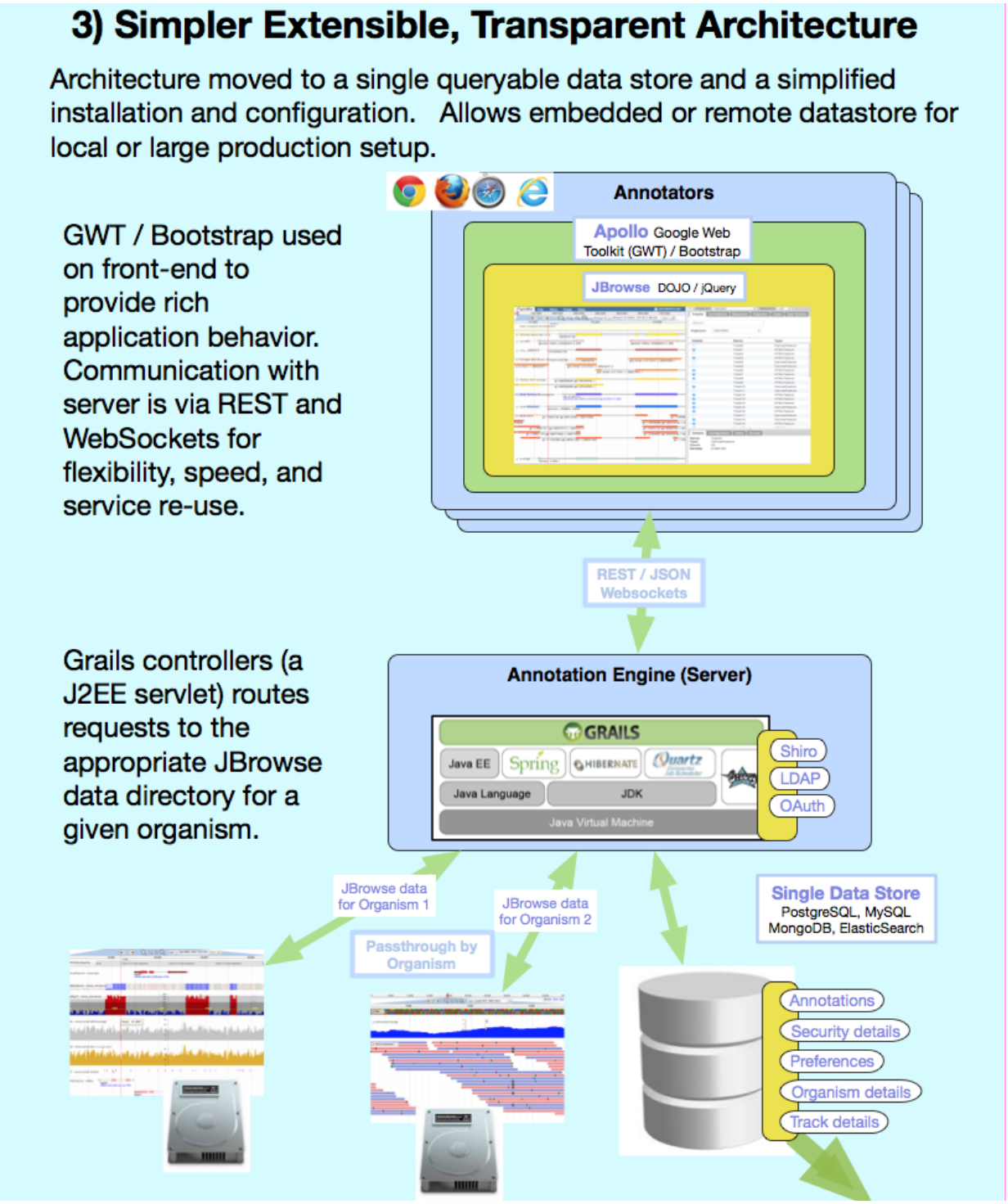
The `apollo` script automatically does several of these functions.

Note: Changes to domain/database objects will require an application restart, but, a very cool feature of our application is that the whole database doesn't need reloading after a database change.

If you look at the `apollo` binary, you'll see that the code for `grails run-app` and others are automatically launched during `apollo run-local`.

Also, as always during web development, yoe will want to clear the cache to see changes ("shift-reload" on most browsers).

Overview



PDF schema

The main components of the Apollo 2.x application are:

- Grails 2 Server with the current version set in the application.properties

- Datastore: configured via Hibernate / Grails whcih can use most anything supported by JDBC / hibernate (primarily, Postgres, MySQL, H2)
- JBrowse / Apollo Plugin: JS / HTML5 JBrowse doc and main site
- GWT client: provides the sidebar. Can be written in another front-end language, as well. GWT doc

Basic layout

- Grails code is in normal grails directories under "grails-app"
- GWT-only code is under "src/gwt"
  - Code shared between the client and the server is under "src/gwt/org/bbop/apollo/gwt/shared"
- Client code is under "client" (still)
- Tests are under "test"
- Old (presumably inactive code) is under "src/main/webapp"
- New source (loaded into VM) is under "src/java" or "src/groovy" except for grails specific code.
- Web code (not much) is either under "web-app" (and where jbrowse is copied) or under "grails-app/assets" (these are compiled down).
- GWT-specifc CSS can also be found in: "src/gwt/org/bbop/apollo/gwt/client/resources/" but it inherits the CSS on its current page, as well.

Main components

The main components of the Apollo 2.x application (the four most important are 1 through 4):

1. The domain classes; these are the main objects
2. Controllers, which route those domains and provide URL routes; provides rest services
3. Views: annotator and index and the only ones that matter for Apollo
4. Services: very important because all of the controllers should typically have routes, then particular business logic should go into the service.
5. Configuration files: The grails-app/conf folder contains central conf files, but the apollo-config.groovy file in your root directory can override these central configs (i.e. it is not necessary to edit DataSource.groovy)
6. Grails-app/assets: all your javascript live here. efficient way to deliver this stuff
7. Resources: web-app directory: css, images, and the jbrowse directory + WA plugin are initialized here.
8. Client directory: The WA plugin is copied or compiled along with jbrowse to the web-app directory

Schema/domain classes

Domain classes: the most important domain class everywhere is the Feature; it is the key to everything that we do. The way a domain class is built:

The domain classes represent a database table. The way it works with "Feature", which is inherited by many other classes, is that all features are stored in the same table, the difference is that in SQL, there is a class table and when

it pulls these tables from the database — it queries it and then converts it into the right class. There are a number of constrains you can set.

Very important: the hasMany maps the one-to-many relationship within the database. It can have many locations. the parentFeatureRelationships is where you map this one-to-many relationship. You also have to have a single item relationship.

You can add extra methods to the domain objects, but this is generally not necessary.

Note: In the DataStore configuration, setting called "auditable = true" means that a new table, a feature auditing tool, is keeping track of history for the specified objects

Feature class

All features inherit an ontologyId and specify a cvTerm, although CvTerms are being phased out.

Subclasses of "Feature" will specify the ontologyId, but "Feature" itself is too generic, for example, so it does not have an ontologyId.

Sequence class

Sequences are the method for WA to grabs sequences used to have a cache built-in mechanism doesn't want to have that anymore to avoid running into memory problems.

Feature locations

Features such as genes all have a feature location belongs to a particular sequence. If you have a feature with subclasses, it can exist within many locations, and each location belongs to its own sequence.

Feature relationship

Feature relationships can define parent/child relationships as well as SO terms i.e. SO "part_of" relationships

Feature enums

The FeatureString enum: allows for mapping names for concepts, and it is useful to use these enums without worrying about string mappings inside the application.

Running the application

If you go through and run this grails application when you send the URL request, then methods that are sent through the AnnotationEditorController (formerly called AnnotationEditorService) dynamically calls a method using handleOperation.

The AnnotatorController serves the page that the annotator is on. This doesn't map to a particular domain object.

In most cases when we have these methods, it unwraps the data that is sent through into JSON object as a set of variables. Then it is processed into java objects and routed back to JSON to send back.

When annotator creates a transcript, it is then released to requestHandlingService and it sends it to an annotation event, which sends it to a WebSocket, and it's then broadcasted to everyone.

Websockets and listeners

All clients subscribe to AnnotationNotifications for new transcripts and events.

If an add_transcript operation occurs, this is broadcasted via the websocket. The server side broadcasts this event, and then it does a JSON roundtrip to render the results and sends the return object that belongs to an AnnotationEvent.

Procedure transcript is created –> goes to the server –> adds a transcript locally –> announces it to everyone.

We used to use long polling request model for "push notifications" but now we use Spring with the SockJS, which uses websockets but it can fall back to long-polling.

There is another component of the broadcasting called brokerMessagingTemplate is the converter to broadcast the event

Controllers

Grails controllers are a fairly easy concept for "routing" URLs and info to methods in the code.

Services

Grails services are classes that perform business logic. (In IntelliJ, these are indicated by green buttons on the definitions to show that these are Injected Spring Bean classes)

The word @Transactional means that every operation that is not private is handled via a transaction. In the old model there were a lot of files that were recreated each time, even though they did the same. Now we define a class and can use it again and again. And there can be transactions within transaction. I could also call other services within services.

addTranscript generateTranscript

The different services do exactly what their name implies. It may not always be clear in what particular service each class should be in, but it can be changed later. It is easy also to make changes to the names as well.

Grails views

- Most of Views are under grails-app
  - everything conforms to the MVC backend model for the Grails application.
- Most of java, css, html is under web-app directory
  - Application logic for groovy, gwt, java, etc live here. we could put our old servlets there, but not recommended.

Main configuration

The central configuration files are defined in grails-app/conf/ folder, however the user normally only edits their personal config in apollo-config.groovy. That is because the user config file will override those in the central configuration. See Configure.html for details.

Database configuration

The "root" database configuration is specified by grails-app/conf/DataSource.groovy but it is generally over-ridden by the user's apollo-config.groovy

It is recommended that the user takes sample-postgres-apollo-config.groovy or sample-mysql-apollo-config.groovy and copies it to apollo-config.groovy for their application.

The default database driver is the h2 database, which is an "embedded" database that doesn't require installing postgres or mysql, but it is not generally seen as performant as postgres or mysql though.

Note: there are three environments that can be setup: a development environment, a test environment, and a production environment, and these are basically assigned automatically depending on how you deploy the app.

- Development environment - "apollo run-local" or "apollo debug"
- Test environment - "apollo test"
- Production environment - "apollo deploy" or "apollo release"

Note: If there are no users and no annotations, a bootstrap procedure can also automatically create some annotations and users to start up the app so there is something in there to begin with.

UrlMapping configuration:

The UrlMappings are stored in grails-app/conf/UrlMappings.groovy

The UrlMappings sets up a mapping from routes to controllers

Standard and customized mappings go in here. The way we route jbrowse to organism data directories is also controlled here. The organismJBrowseDirectory is set for a particular session, per user. If none specified, it brings up a default one.

Build configuration

The build configuration is stored in grails-app/conf/BuildConfig.groovy

If there are libraries that are missing are are to be added, you can add them here.

Additionally, the build system uses the "apollo" script and the "build.xml" to control the compilation and resource steps.

Central config

The central configuration is stored in grails-app/conf/Config.groovy

The central Grails config contains logging, app config, and also can reference external configs. The external config can override settings without even touching the application code using this method

In our application, we use the apollo-config.groovy then everything in there supersedes this file.

The log4j area can enable logging levels. You can turn on the "debug grails.app" to output all the webapollo debug info, or also set the "grails.debug" environment variable for java too.

There is also some Apollo configuration here, and it is mostly covered by the configuration section.

GWT web-app

When GWT compiles, it loads files into the web-app directory. When it loads up annotator, it goes to annotator index (the way things get loaded) it does an include annotator.nocache.js file, and with that, it includes all GWT stuff for the /annotator/index route. The src/gwt/org/bbop/apollo/gwt/ contains much code and the src/gwt/org/bbop/apollo/gwt/Annotator.gwt.xml is a central config file for the GWT web-app.

User interface definitions

A Bootstrap/GWT interface handles the tabs on the right for the new UI. The annotator object is at the root of everything.

Example definition: MainPanel.ui.xml

Tests

Unit tests

Unit tests and some basic javascript tests are running on Travis-CI (see .travis.yml for example script).

You can also run "apollo test" to run the tests locally. It will use the "test" database configuration automatically.

Also see the testing notes for more details.

CHAPTER 13

Command line tools

The command line tools offer a number of interesting features that can be used to help setup and retrieve data from the application.

Overview

The command line tools are located in docs/web_services/examples, and they are mostly small scripts that automate the usage of the the web services API.

get_gff3.groovy

Example:

```
get_gff3.groovy -organism Amel_4.5 -username admin@webapollo.com \
    -password admin_password -url http://localhost:8080/apollo > my output.gff3
```

This command can accept an -output argument to output to file, or the stdout can be redirected.

The -username and -password can be specified via the command line or if omitted, the user will be prompted.

get_fasta.groovy

Example:

```
get_fasta.groovy -organism Amel_4.5 -username admin@webapollo.com \
    -password admin_password -seqtype cds/cdna/peptide -url http://localhost:8080/
↪apollo > output.fa
```

This command can accept an -output argument to output to file, or the stdout can be redirected.

The -username and -password can be specified via the command line (similar to `get_gff3.groovy`) or if omitted, the user will be prompted.

add_users.groovy

Example:

```
add_users.groovy -username admin@webapollo.com -password admin_password \
    -newuser newuser@test.com -newpassword newuserpass \
    -destinationurl http://localhost:8080/apollo
```

The -username and -password refer to the admin user, and they can also be specified via stdin instead of the command line if they are omitted.

A list of users specified in a csv file can also be used as input.

add_organism.groovy

Example:

```
add_organism.groovy -name yeast -url http://localhost:8080/apollo/ \
    -directory /opt/apollo/yeast -username admin@webapollo.com -password admin_
↪password
```

The -directory refers to the jbrowse data directory containing the output from prepare-refseqs.pl, flatfile-to-json.pl, etc. The -blatdb is optional, -genus, and -species are optional.

The -username and -password refer to the admin user, and they can also be specified via stdin instead of the command line if they are omitted.

delete_annotations_from_organism.groovy

Example:

```
docs/web_services/examples/groovy/delete_annotations_from_organism.groovy  -
↪destinationurl http://localhost:8080/apollo\
     -organismname honeybee2
```

This script will delete any annotations associated with a given organism.

CHAPTER 14

Web Service API

The Apollo Web Service API is a JSON-based REST API to interact with the annotations and other services of Apollo. Both the request and response JSON objects can contain feature information that are based on the Chado schema. We use the web services API scripting examples and we also use them in the Apollo JBrowse plugin.

The most up to date Web Service API documentation is deployed from the source code rest-api-doc annotations.

See http://icebox.lbl.gov/Apollo2/jbrowse/web_services/api for details

Warning

If you are sending password you care about over the wire (even if not using web services) it is *highly recommended* that you use https (which adds encryption ssl) instead of http.

Examples

We provide an examples directory.

```
curl -b cookies.txt -c cookies.txt -e "http://localhost:8080" \
    -H "Content-Type:application/json" \
    -d "{'username': 'demo', 'password': 'demo'}" \
    "http://localhost:8080/apollo/Login?operation=login"
```

Login expects two parameters: username and password, and optionally rememberMe for a persistent cookie.

A successful login returns a empty JSON object

Python Client

A python client has been provided over many of the Apollo web services, which is easy to setup:

```
pip install apollo
arrow init # provide Apollo credentials
arrow -h
## have fun
arrow groups get_groups
```

Documentation on commands and some examples working with jq:

What is the Web Service API?

For a given Apollo server url (e.g., `https://localhost:8080/apollo` or any other Apollo site on the web), the Web Service API allows us to make requests to the various "controllers" of the application and perform operations.

The controllers that are available for Apollo include the AnnotationEditorController, the OrganismController, the IOServiceController for downloads of data, and the UserController for user management.

Most API requests will take:

- The proper url (e.g., to get features from the AnnotationEditorController, we can send requests to (e.g `http://localhost/apollo/annotationEditor/getFeatures`)
- username - an authorized user (also uses session if none specified)
- password - password (also uses session if none specified)
- organism - (if applicable) the "common name" of the organism for the operation – will also pull from the "user preferences" if none is specified.
- track/sequence - (if applicable) reference sequence name (shown in sequence panel / genomic browse)
- uniquename - (if applicable) the uniquename is a UUID used to guarantee a unique ID

Errors If an error has occurred, a proper HTTP error code (most likely 400 or 500) and an error message. is

returned, in JSON format:

```
{ "error": "error message" }
```

Cookies

The Apollo Login creates a JSESSIONID cookie and rememberMe cookie (if applicable) and these can be used in downstream API requests (for example, by setting -b cookies.txt in curl will preserve the cookie in the request).

You can also pass username/password to individual API requests and these will authenticate each individual request.

Representing features in JSON

Most requests and responses will contain an array of `feature` JSON objects named `features`. The `feature` object is based on the Chado `feature`, `featureloc`, `cv`, and `cvterm` tables.

```
{
    "residues": "$residues",
    "type": {
        "cv": {
            "name": "$cv_name"
        },
        "name": "$cv_term"
    },
    "location": {
        "fmax": $rightmost_intrabase_coordinate_of_feature,
        "fmin": $leftmost_intrabase_coordinate_of_feature,
        "strand": $strand
    },
    "uniquename": "$feature_unique_name"
    "children": [$array_of_child_features]
    "properties": [$array_of_properties]
}
```

where:

- `residues` - A sequence of alphabetic characters representing biological residues (nucleic acids, amino acids) [string]
- `type.cv.name` - The name of the ontology [string] `type.name` - The name for the cvterm [string]
- `location.fmax` - The rightmost/maximal intrabase boundary in the linear range [integer]
- `location.fmin` - The leftmost/minimal intrabase boundary in the linear range [integer]
- `strand` - The orientation/directionality of the location. Should be 0, -1 or +1 [integer]
- `uniquename` - The unique name for a feature [string]
- `children` - Array of child feature objects [array]
- `properties` - Array of properties (including frameshifts for transcripts) [array]

Note that different operations will require different fields to be set (which will be elaborated upon in each operation section).

Web Services API

The most up to date Web Service API documentation is deployed from the source code rest-api-doc annotations

See http://icebox.lbl.gov/Apollo2/jbrowse/web_services/api for details

CHAPTER 15

Example Build Script on Unix with MySQL

This is an example build script. It may **NOT** be appropriate for your environment but does demonstrate what a typical build process **might** look like on a Unix system using MySQL.

Please consult our Setup and Configuration documentation for additional information.

```
# Install prereqs
apt-get install tomcat8 git ant openjdk-8-jdk nodejs
# Upped tomcat memory per Apollo devs instructions:
echo "export CATALINA_OPTS="-Xms512m -Xmx1g \
              -XX:+CMSClassUnloadingEnabled \
              -XX:+CMSPermGenSweepingEnabled \
              -XX:+UseConcMarkSweepGC" >> /etc/default/tomcat8

# Download and extract their tarball
npm install -g bower
wget https://github.com/GMOD/Apollo/archive/2.0.4.tar.gz
mv 2.0.4.tar.gz Apollo-2.0.4.tar.gz
tar xf Apollo-2.0.4.tar.gz
# Setup apollo mysql user and database
CREATE USER 'apollo'@'localhost' IDENTIFIED BY 'THE_PASSWORD';
CREATE DATABASE `apollo-production`;
GRANT ALL PRIVILEGES ON `apollo-production`.* To 'apollo'@'localhost' IDENTIFIED BY
↪'THE_PASSWORD';
# Configure apollo for mysql.
cd ~/src/Apollo-2.0.4
# Let's store the config file outside of the source tree.
mkdir ~/apollo.config
# Copy the template
cp sample-mysql-apollo-config.groovy ~/apollo.config/apollo-config.groovy
ln -s ~/apollo.config/apollo-config.groovy
# For now, turn off tomcat8 so that we can see if the locally-run version works␣
↪service tomcat8 stop
# Run the local version, which verifies install reqs, and does a bunch of stuff (see␣
↪below)
cd Apollo-2.0.4
```

```
./apollo run-local

# Some of what the Apollo installer does:
# Clones a bunch of git submodules into apollo-2.0.4/src
# Does a bunch of java compiling.
# Downloads and installs grails for you here: $HOME/.grails .
# Installs perl modules here: $HOME/.cpanm
# Installs java stuff here: $HOME/.java and $HOME/.m2

# If a pre-installed instance:
rm -rf /var/lib/tomcat/webapps/apollo
rm -f /var/lib/tomcat/webapps/apollo.war
# Startup tomcat again
service tomcat8 start

# ... with javascript minimization:
./apollo release
# ... without javascript minimization
#  ./apollo deploy
# Above creates this file: target/apollo-2.0.4.war
sudo cp target/apollo-2.0.4.war /var/lib/tomcat/webapps/apollo.war

# Prepare JBrowse data
# Add the FASTA assembly
~/src/Apollo-2.0.4/bin/prepare-refseqs.pl \
--fasta /research/dre/assembly/assembly1.fasta.gz \
--out ~/organisms/dre

# Add annotations
~/src/Apollo-2.0.4/bin/flatfile-to-json.pl \
--gff /research/dre/annotation/FINAL_annotations/ssc_v4.gff \
--type mRNA --trackLabel Annotations --out ~/organisms/dre

# In interface point to directory ~/organisms/dre
```