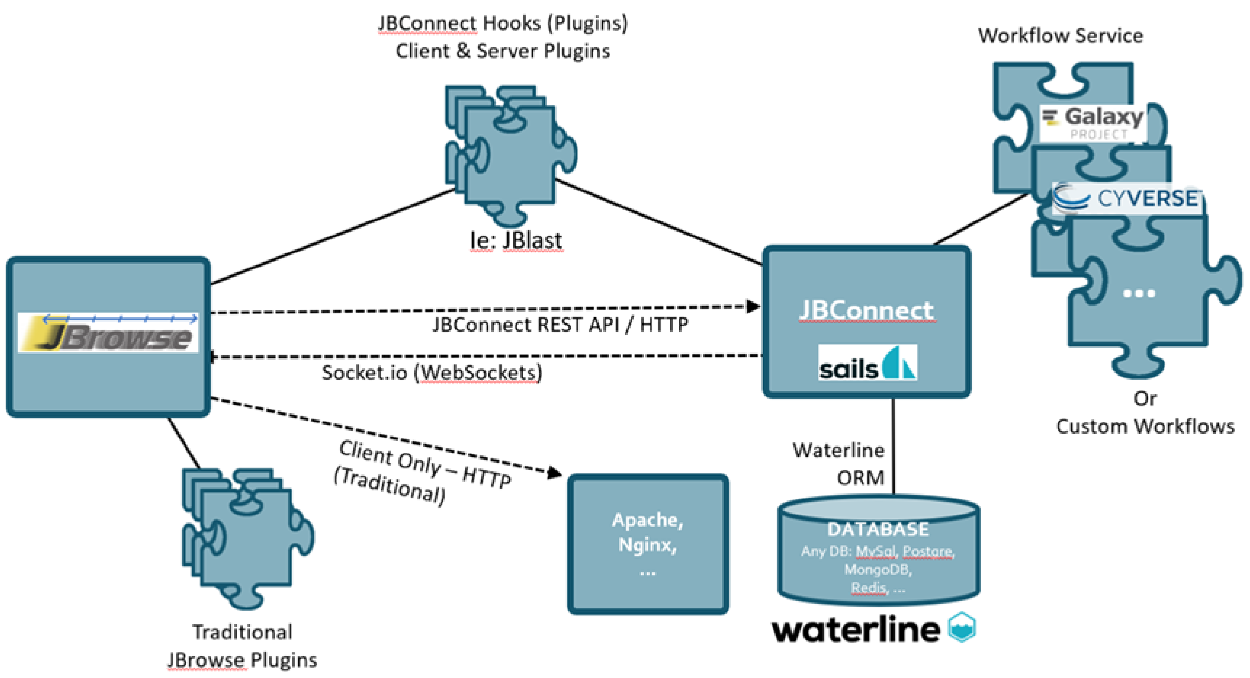
PROJECT_NAME Documentation

Release 0.1.0

AUTHOR

Mar 09, 2021

Contents



Paper: JBrowse Connect: A server API to connect JBrowse instances and users, Yao, Buels, Stein, Sen, Holmes, et al. 2020; PLOS Computational Biology

JBConnect provides the following features:

Sails JS - NodeJS/Expressed-based
Tightly integrated with JBrowse
Track & Dataset Management with JBrowse integration • RESTful track data access • Track API (CRUD) • Auth and Secure Tracks/Datasets/Assets • Track/Asset SubPub events with Socket.io
Flexible Authentication – (Passport.js) supporting strategies, like OAuth2, OpenID, etc.
User management services
Policy Engine f0r managing access to Tracks, Datasets, Services, Assets
Extensible server-side analysis with workflow abstraction and job queue
Waterline ORM (MongoDB, MySQL, Postgres, Redis, etc.) with integrated Blueprint object models
npm installable hook model supporting both client-side (JBrowse plugins) and server-side extensions in a single package.
Grunt – task management (minification, watches, etc.)

CHAPTER 1

Quick Start

The quick start instructions demonstrate installing JBConnect with JBrowse loaded as a an NPM module (since JB-Connect is generally intended to be a companion of JBrowse. JBrowse may also be installed in a separate directory. (See *JBrowse Installed In Separate Directory*.)

1.1 Pre-Install

JBConnect requires sailsjs and redis . *Redis* is only used by the queue framework (kue)

```
yum install redis
redis-server
npm install -g sails@1.0.2
```

1.2 Install

Install the JBConnect and JBrowse. jb_setup.js ensures the sample data is loaded.

```
git clone http://github.com/gmod/jbconnect
cd jbconnect
npm install
npm install @gmod/jbrowse@1.15.1
patch node_modules/@gmod/jbrowse/setup.sh fix_jbrowse_setup.patch
./utils/jb_setup.js
```

The patch operation is needed to make JBrowse 1.15.1 setup.sh run properly. If JBrowse is installed in another location, the patch should be run before setup.sh.

1.3 Run

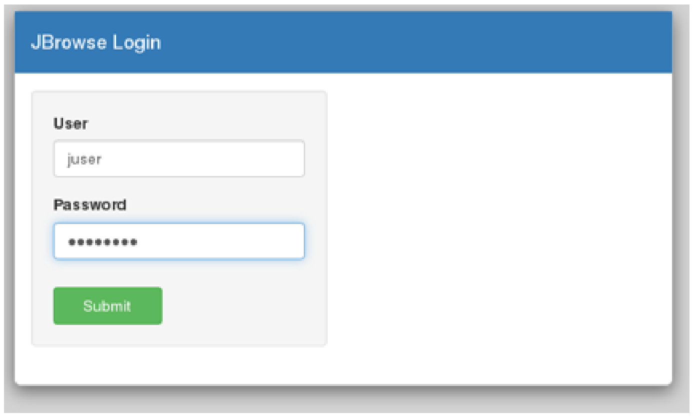
Launch the server.

`sails lift`

From a web browser, access the application.

`http://localhost:1337/login`

You will arrive at the following screen



The default username/password: juser/password

1.4 Vulnerabilty Warnings

npm currently reports a number of vulnerability warnings of various degrees of severity, most of which are coming from Sails. Many of these warnings are fixed in a later version of SailsJS. We hope in a future release of the software to correct more of these warnings.

CHAPTER 2

Contents

2.1 Features

JBConnect is a server/analysis framework for JBrowse and has the following features:

2.1.1 Directory Layout

```
JBConnect project
├── api                          Standard sails API layout, models, controllers,␣
↪etc.
├── assets                       contains client accessible assets
├── bin                          Utilities
├── config                       Configuration files.
│   └── globals.js               global config file for module
├── data                         Contains the local database file
│   └── localDiskDb.db           Local database file
├── docs                         Documentation
│   └── genapi-rst               jsdoc generated rst files
├── plugins                      Client-side Plugins
│   └── JBClient                 Client plugin
├── test                         Test
├── views                        Login / registration pages
├── Gruntfile.js                 Grunt config
├── jbutil                       JBConnect Utility (exe)
└── package.json
```

2.1.2 jbutil Command

`jbutil` is a setup/configuration utility for JBConnect. *-jbconnect-hook modules can extend `jbutil` command options. (see: jbs-hooks-extend)

This example shows that `JBlast` adds a number of commands to `jbutil` todo: update help

```
$ ./jbutil --help
Usage: jbutil [OPTION]
      --config            display aggregated config
      --blastdbpath=PATH  (jblast) existing database path
      --setupworkflows    (jblast) [install|<path>] "install" project wf, or specify .
↪ga file
      --setuptools        (jblast) setup jblast tools for galaxy
      --setupdata         (jblast) setup data and samples
      --setupindex        (jblast) setup index.html in the jbrowse directory
      --setuphistory      setup history
  -h, --help              display this help
```

See: **jbs-jbutilextending_**

2.1.3 Queue Framework

JBConnect uses Kue as the basis for the queue framework. However, Kue is encapsulated in the Job model/controller. Since Kue requires redis database, redis server must be running. An integrated job panel is available when the JBClient plugin is active. (see: *JBClient Plugin*)

For diagnostic purposes, a Kue utility can be used to view/manage the Kue database content: `http://localhost:1337/kue`

This route can be disabled with in config/http.js.

2.1.4 Configuration

JBConnect configurations are in `config/globals.js`

```
jbrowse: {
    jbrowseRest: "http://localhost:1337",       // path accessible by web browser
    jbrowsePath: jbPath,                        // or point to jbrowse directory (ie.
↪"/var/www/jbrowse/")
    routePrefix: "jbrowse",                     // jbrowse is accessed with http://
↪<addr>/jbrowse
    dataSet: [
        {
            dataPath: "sample_data/json/volvox" // registered datasets.
        }
    ]
}
```

2.1.5 Client-Side Plugins

Client-side plugins are in the *plugins* directory. Plugins will automatically be accessible by the client side. However, they need to be configured in the *plugins:* section of the particular dataset in JBrowse *trackList.json*.

Plugins are copied to the configured JBrowse instance upon `sails lift`.

2.1.6 Web Includes

libroutes maps dependancy routes for client-side access. These provide access to modules that are required for use by the client-side plugins or other client-side code. The framework looks for libroutes.js in , in their respective config directories

For example: for the module jquery, The module is installed with 'npm install jquery' The mapping the mapping 'jquery': '/jblib/jquery' makes the jquery directory accessible as /jblib/jquery from the client side.

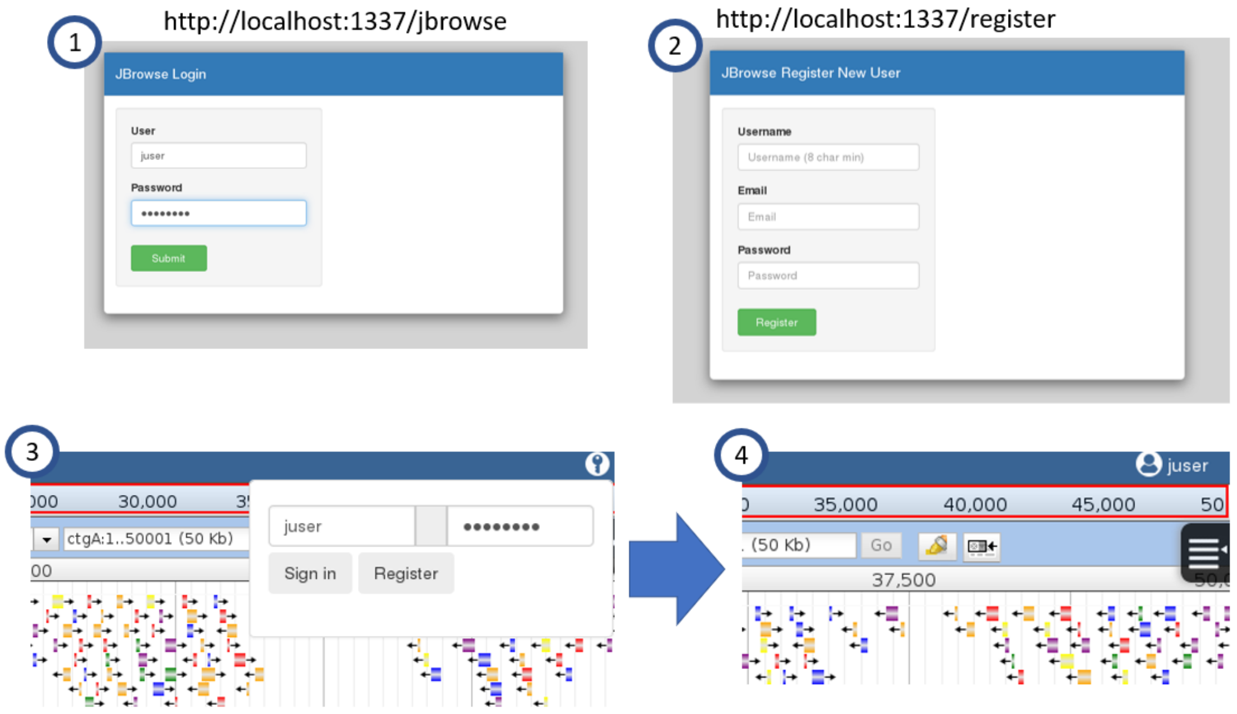
Library Routes are virtual routes, in that they only exist when the server is lifted. They are virtually mapped to their respective locations in the node_modules directory.

`config/libroutes.js`:

```
module.exports = {
    lib: {
            'jquery.mb.extruder':        '/jblib/mb.extruder',
            'jQuery-ui-Slider-Pips':     '/jblib/slider-pips',
            'jquery-ui-dist':            '/jblib/jquery-ui'
    }
};
```

2.1.7 Standalone Register / Login / Logout Routes

Stand-alone routes allow for basic register/login/logout functionality free from the JBrowse interface.



Logout: `http://<address>:1337/logout`

Get Login State: `http://<address>:1337/loginstate`

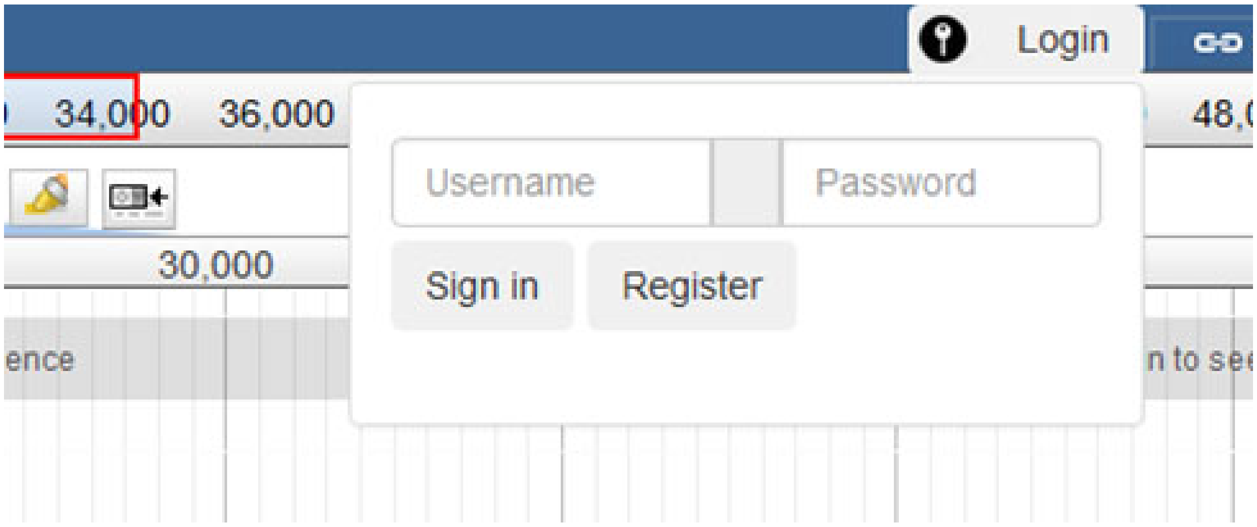
The routes are defined in *config/routes.js*.

At the moment, these are the only user related GUI interfaces there are. It is intended to add more complete management and password management interfaces down the road.

*Note: Stand-alone interfaces use 'bootstrap <http://getbootstrap.com/>'_*

Login/Logout Panel

Login Panel



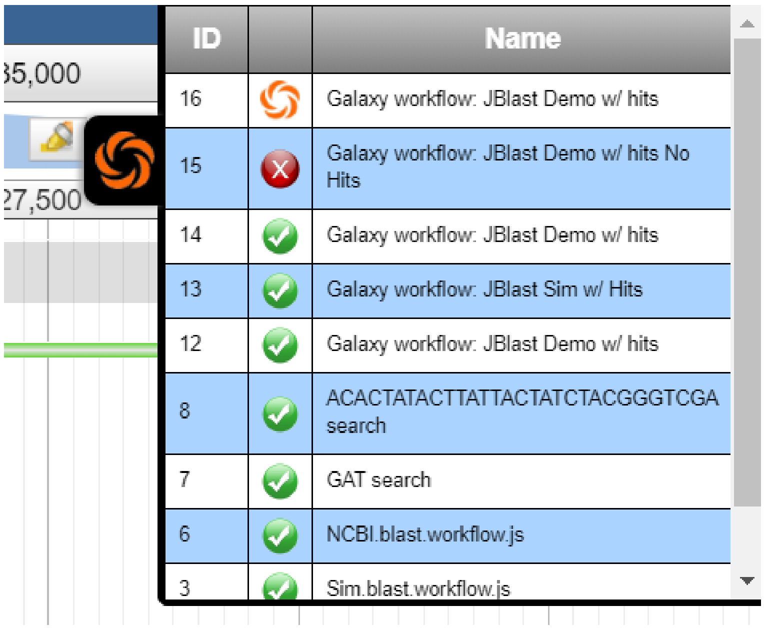
Loguot Panel

img/logout-integrated.jpg

Job Queue Panel

JBConnect uses *Kue* as the queue framework. Since Kue requires *redis* database, redis server must be running. An integrated job panel is available when the JBClient plugin is active. (see: *JBClient Plugin*)

Integrated Job Panel:



ID		Name
16		Galaxy workflow: JBlast Demo w/ hits
15	✗	Galaxy workflow: JBlast Demo w/ hits No Hits
14	✓	Galaxy workflow: JBlast Demo w/ hits
13	✓	Galaxy workflow: JBlast Sim w/ Hits
12	✓	Galaxy workflow: JBlast Demo w/ hits
8	✓	ACACTATACTTATTACTATCTACGGGTCGA search
7	✓	GAT search
6	✓	NCBI.blast.workflow.js
3	✓	Sim.blast.workflow.js

2.1.8 Test Framework

Test framework uses

- Mocha for unit test
- Nightwatch for end-to-end, supporting phantomjs, selenium and online service such as browserstack.
- Istanbul for coverage

To execute

```
npm test
```

by default nightwatch is setup for phantomjs. Selenium requires running an additional selenium server

`package.json`:

```
"scripts": {
  "test": "nyc node ./node_modules/mocha/bin/mocha test/bootstrap.test.js test/
↪integration/**/*.test.js test/e2e/**/*.test.js --nightwatch-test phantomjs",
},
```

The option `--nightwatch-test` can be:

- `phantomjs` - runs client tests with phantomjs
- `selenium` - runs client tests with selenium
- `browserstack` - runs client test with selenium through remote browserstack account.

2.1.9 Documentation Framework

For integrated documentation, JSdoc3 is used to generate API docs from code with jsdoc-sphinx, a jsdoc template that generates RestructuredText (RST) and Sphinx. This enables support for readthedocs.

See: RST/Sphinx Cheatsheet

Generate docs: `npm run gendocs`

This will generate `docs/api.rst`. This must be committed and pushed for it to appear in `jbconnect.readthedocs.io`.

2.2 Configuration Options

2.2.1 JBrowse Installed In Separate Directory

The JBrowse directory can also be configured manually. (See jbs-globals-js)

2.2.2 Configuration Files

A number of configuration files are in the `./config` directory. A few of the more important ones (ones that JBServer touches) are described mentioned in the table below. See Sails Configuration for a better description of the configuration framework.

jbs-globals-js	global configuration file
http.js	Custom middleware and /jbrowse route is setup here.
passport.js, policies.js	passport framework and auth policies config
routes.js	various route configurations
connections.js	choice of database - local, mongo, mysql, . . . (we use local by default.) The DB file is in the `./data/localDiskDb.db`.

globals.js

To view aggregate configuration: `./jbutil --config`

The aggregate config is the merged globals.js combined with the globals.js of server hook modules.

The aggregate config file is the merged config of JBConnect and its installed jbh- (hook) modules.

Edit config file: `nano config/globals.js`

```
jbrowse: {
    jbrowseRest: "http://localhost:1337",
    jbrowsePath: jbPath,                        // or "/var/www/jbrowse/"
    routePrefix: "jbrowse",                     // jbrowse is accessed with http://
↪<addr>/jbrowse

    dataSet: {
         Volvox: {path: "sample_data/json/volvox"}
    },

    // search service settings
    serverSearch: {
        resultPath: "ServerSearch",
        resultCategory: "Search Results",
        trackTemplate: "ServerSearchTrackTemplate.json",
        workflowScript: "ServerSearch.workflow.js",
        processScript:   'ServerSearchProcess.html'
    },
```

(continues on next page)

(continued from previous page)

```
    // search job service registration
    services: {
        'serverSearchService': {name: 'serverSearchService',  type: 'service'}
    },

    /*
     * Virtual Routes
     * These routes reference node_modules that are used by the client and
     * accessed by virtual route.
     */
    libRoutes: {
        // name         node_modules dir            virtual route
        'jquery':       {module: 'jquery',          vroute:'/jblib/jquery'},
        'bootstrap':    {module: 'bootstrap',       vroute:'/jblib/bootstrap'},
        'jqueryui':     {module: 'jquery-ui-dist',  vroute:'/jblib/jquery-ui'},
        'mbextruder':   {module: 'jquery.mb.extruder', vroute:'/jblib/mb.extruder'}
    },
    /*
     * Web Includes
     * These includes are injected into JBrowse ``index.html`` upon ``sails lift``.
     */
    webIncludes: {
        // key                    virtual route
        "css-bootstrap":         {lib: "/jblib/bootstrap/dist/css/bootstrap.min.css"},
        "css-mbextruder":        {lib: "/jblib/mb.extruder/css/mbExtruder.css"},
        "css-jqueryui":          {lib: "/jblib/jquery-ui/jquery-ui.min.css"},
        "css-jqueryuistructure": {lib: "/jblib/jquery-ui/jquery-ui.structure.min.css"}
↪,
        "css-jqueryuitheme":     {lib: "/jblib/jquery-ui/jquery-ui.theme.min.css"},
        "js-sailsio":            {lib: "/js/dependencies/sails.io.js"},
        "js-jquery":             {lib: "/jblib/jquery/dist/jquery.min.js" },
        "js-jqueryui":           {lib: "/jblib/jquery-ui/jquery-ui.min.js" },
        "js-bootstrap":          {lib: "/jblib/bootstrap/dist/js/bootstrap.min.js"},
        "js-mbextruderHover":    {lib: "/jblib/mb.extruder/inc/jquery.hoverIntent.min.
↪js"},
        "js-mbextruderFlip":     {lib: "/jblib/mb.extruder/inc/jquery.mb.flipText.js"}
↪,
        "js-mbextruder":         {lib: "/jblib/mb.extruder/inc/mbExtruder.js"}
    }
}
```

2.2.3 Limiting Query Size

The query size (# of base pairs) can be limited. This might be necessary to contain the processing or contain memory consumption of client and server., particularly with operations like BLAST where the BLAST database may be very large.

Add the following option to the trackList.json of the dataset configuration:

```
{
    ...
    "bpSizeLimit": 25000,
    ...
},
```

This will cause an alert message when the selected query size exceeds 25000 bp.

If omitted, the allowed size will be unlimited.

We definitely recommend using this setting for larger assemblies.

2.2.4 Installing JBConnect jbh-hooks

A 'JBConnect Hook' is basically an *installable sails hook* with specific methods for extending JBConnect. JBConnect hooks must have the prefix `jbh-` prepended to the name. For example: jbh-jblast. When the hook is installed (i.e. `npm install jbh-jblast`). JBConnect will automatically integrate a number of features of the hook directly into JBConnect upon `sails lift`.

The jbh- hook can extend JBConnect in the following ways:

- Extend models, controllers, policies and services
- Integrated client-side JBrowse plugins injection
- Integrated client-side npm module injection
- Integrated job services (see: **jbs-jobservice_**)
- Integrated configuration tool (jbutil)
- Aggregated configurations

Installing a hook:

`npm install jbh-<hook name>` (i.e. jbh-jblast)

For detailed info on jbh-hooks, see: *JBConnect Hooks*

2.2.5 JBClient Plugin

JBrowse GUI intetrated interfaces are available when the `JBClient` plugin is configured on in the JBrowse client.

To enable integrated features within the JBrowse app, modify the dataset's `trackList.json`, adding `JBClient` plugin to the configuration.

Note: the JBClient plugin is not physically in the JBrowse plugin directory. It is available as a route.

```
"plugins": [
  "JBClient",                    <-----
  "NeatHTMLFeatures",
  "NeatCanvasFeatures",
  "HideTrackLabels"
],
```

2.2.6 Job Service Configuration

Job services (*jservice*) are a special type of service that are used to extend RESTful API service and serve processing for job operations.

Configuration is defined in `config/globals.js` under the jbrowse section under service.

A definition: <indexname>: {name: <servicename>, type:<type>, alias:<alias> }

where:

- indexname - is the reference name service (generally the same as servicename)
- servicename - is the name of the service reference the service code in api/services.

- type - is the type of service. either “workflow” or “service”
- alias - (optional) if specified, the service can also be referenced by the alias name.

jservice type:

- workflow - service can serve job execution and RESTful interfaces
- service - service only serves RESTful interfaces

Job service config in `config/globals.js`:

```
// list of services that will get registered.
services: {
    'basicWorkflowService':     {name: 'basicWorkflowService',  type: 'workflow',␣
↪alias: "jblast"},
    'filterService':            {name: 'filterService',         type: 'service'},
    'entrezService':            {name: 'entrezService',         type: 'service'}
},
```

2.2.7 Extending jbutil

`jbutil` is a command line utility that is used to configure JBConnect in various ways. `jbutil` can be extended by a installable hook through `bin/jbutil-ext.js`.

`jbutil-ext.js` must imeplement these function:

```
module.exports = {
    // this return the options that the module support.  In this example,
    // we add -t or --test and --thing options to jbutil.

    getOptions: function() {
        return [
            ['t' , 'test=ARG', '(jbh-myhook) this is a test option'],
            ['' , 'thing',   , '(jbh-myhook) this is another test option']
        ];
    },

    // Extends the help display
    // In this example, we describe how to use --test with a parameter value "abc"

    getHelpText: function() {
        return "\nExample: ./jbutil --test abc\n";
    },

    // process options
    // where opt - the option list.
    //       path - path of the module that will process the option (i.e. "./node_
↪modules/jbh-jblast"
    //       config - the aggregate globals.js config.

    process: function(opt,path,config) {
        var tool = opt.options['setupindex'];
        if (typeof tool !== 'undefined') {
            jblib.exec_setupindex(this.config);
            jblib.exec_setupPlugins(this.config);
        }
```

(continues on next page)

(continued from previous page)

```
        var tool = opt.options['dbreset'];
        if (typeof tool !== 'undefined') {
    }
```

See npm module node-getopt for more info.

2.3 JBConnect Hooks

JBconnect-Hook leverages the Sails Installable Hook framework and adds facilities to extend it for JBConnect:

- Job Service integration - provides a runnable job that is launched by the job queue. This may also include an adapter for local or 3rd party server API access such as Galaxy. It may also implement REST APIs specific to the service. The actual adapter portion is optional.
- The Job Queue relies on the adapter to provide translated job state information and execution pre and post processing of the analysis operations.
- The REST API service of the Job Service is a different interface than the standard sails controller interfaces.
- JBrowse plugin / injection (plugins that are tightly integrated with the server-side hooks) along with client-side module dependencies, used by the JBrowse plugins. The injection occurs upon sails lift and copies the necessary plugins into the JBrowse server plugins directory.
- Model/controllers/services as provided by Sails Blueprints. These services are merged with native server models/controllers/services into globally accessible objects.
- Commands via (jbutil) - command options and implementation are merged with the function of jbutil, providing extended command capabilities specific to the JBConnect hook.
- Configurations are aggregated with the JBConnect server configurations.

2.3.1 Directory Layout

This is the standard directory layout of a JBConnect hook module

```
*-jbconnect-hook project
├── api                             Standard Sails modules layout
│   ├── controllers
│   ├── hooks
│   │   └── Myhook
│   │       └── index.js           The main hook
│   ├── models
│   ├── policies
│   └── services
├── bin
│   └── jbutil-ext.js               jbutil extension module
├── config
│   └── globals.js                  Config file for module
├── plugins                         Client-side Plugins
│   └── PluginA
└── package.json
```

2.3.2 package.json

JBConnect hooks extend sails hooks and are required to contain the following section in the `package.json`:

```
"sails": {
  "isHook": true,
  "hookName": "jblast-jbconnect-hook",
  "isJBConnectHook": true
}
```

Note the naming convention `*-jbconnect-hook` is required.

2.3.3 Configurations

This file contains the default config options that are specific to the hook module. These config options are merged with other JBConnect hooks and the JBConnect `config/globals.js`.

From JBConnect, use `./jbutil --config` to see the aggregated config.

Configurations in `globals.js` are intended to be project defaults. Configurations can also be in JBConnect's root directory, called `jbconnect.config.js`, which are user modified configurations.

2.3.4 Client-Side JBrowse Plugins

JBrowse plugin associated with the JBConnect hook can be deployed with the JBConnect hook. The framework provides for injecting JBrowse plugins into the working JBrowse directory along with any client-side dependency modules used by the plugin.

The following illustrates how to create a client-side plugin under the framework, with its various configation options.

Developing JBrowse Plugins Under the Framework

Refer to Writing JBrowse Plugins for more information.

Client-side plugins in plugins directory are copied to the target JBrowse plugins directories upon `sails lift`.

A plugin can be disabled if it has an entry in the `excludePlugins:` section of config/globals.js file or jbconnect.config.js.

```
jbrowse: {
    ...
    excludePlugins: {
        "ServerSearch": true    // doesn't work with JBrowse 1.13.0+
    },
    ...
}
```

Web Include (client dependencies)

Web Includes maps dependancies for client-side access. These are routes to modules that are required for use by the client-side plugins or other client-side code. The framework looks for globals.js in jbh- (hook modules), in their respective config directories

For example: for the dependency module jquery, Relevant assets are copied into assets/jblib by bin/postinstall.js The mapping the mapping 'js-jquery': '/jblib/jquery' makes the jquery directory accessible as /jblib/jquery.min.js from the client side.

`globals.js`

```
...

jbrowse: {
    /*
     * Web Includes
     * These includes are injected into JBrowse upon sails lift (see tasks/pipeline.
↪js).
     */
    webIncludes: {
        "css-bootstrap":         {lib: "/jblib/bootstrap.min.css"},
        "css-mbextruder":        {lib: "/jblib/mb.extruder/mbExtruder.css"},
        "css-jqueryui":          {lib: "/jblib/jquery-ui.min.css"},
        "css-jqueryuistructure": {lib: "/jblib/jquery-ui.structure.min.css"},
        "css-jqueryuitheme":     {lib: "/jblib/jquery-ui.theme.min.css"},
        "js-sailsio":            {lib: "/js/dependencies/sails.io.js"},
        "js-jquery":             {lib: "/jblib/jquery.min.js" },
        "js-jqueryui":           {lib: "/jblib/jquery-ui.min.js" },
        "js-bootstrap":          {lib: "/jblib/bootstrap.min.js"},
        "js-mbextruderHover":    {lib: "/jblib/mb.extruder/jquery.hoverIntent.min.js"}
↪,
        "js-mbextruderFlip":     {lib: "/jblib/mb.extruder/jquery.mb.flipText.js"},
        "js-mbextruder":         {lib: "/jblib/mb.extruder/mbExtruder.js"}
    },
}
...
```

2.3.5 Extending Commands

`jbutil` is a general command of JBConnect that are used for various operations. `jbutil-ext.js` can be used by the hook to extend options of jbutil.

- it can extend new command line options
- it can extend the help (i.e. `./jbutil --help`)

This is a simplified example of jbutil-ext.js.

```
module.exports = {

    // defining the options
    getOptions: function() {
        return [
            ['f' , 'fox'    , 'make a fox sound'],
            ['d' , 'dog'    , 'take out the dog'],
        ];
    },

    // this is displayed when the user uses the --help or -h option
    getHelpText: function() {
        return  "What does the fox say\n"+
                "./jbutil -fox\n"+
```

(continues on next page)

(continued from previous page)

```
                'Take out the dog\n"+
                "./jbutil -dog\n";

    },

    // processing the options
    process: function(opt,path,config) {
        if (opt.options['cat']) {
            ....
        }
        if (opt.options['dog']) {
            ....
        }

    },

    // do some pre initialization
    init: function(opt,path,config) {
        return 1; // successful init, or 0 if failed.
    }

};
```

More info about the command options processor can be found in node-getopt .

Additional non-jbutil commands

The hook can also deploy any additional commands in the JBConnect's `utils` directory.

2.3.6 Sails Module Layout

This is the standard sails directory layout for models, controllers, policies, and services of a sails hook. The framework uses marlinspike to integrate controllers, models, policies, and services into JBConnect.

```
hook project root
├── api                         Standard Sails modules layout
    ├── controllers             optional
    ├── hooks                   hook core index.js in here
    ├── models                  optional
    ├── policies                optional
    └── services                Job services and supporting modules in here.
```

A core index.js is in `api/hooks/<hook name>/index.js` and can be basically be copied from here .

This core fragment starts the initialization of the hook.

2.3.7 Job Service

A job service is a special service that can react to the job queue framework asking it to execute something.

The job service generally resides in `api/services` directory of the hook and is named `<something>Service.js`.

Function Map

Job services must contain a fmap section which defines the routes that the job service exposes. And there should be corresponding routes (or REST APIs) defined in the module. The fmap section must exist, but does not need to be populated.

```
module.exports = {
    fmap: {
        set_filter:         'post',
        get_blastdata:      'get',
        get_trackdata:      'get'
    },

    // each function should be implemented in the job service
    set_filter(req, res) {
        var requestData = req.allParams();
        ...
        return res.send(ret);
    },
    get_blastdata(req, res) {
        var requestData = req.allParams();
        ...
        return res.send(ret);
    },
    get_trackdata: function(req, res) {
        var requestData = req.allParams();
        ...
        return res.send(ret);
    },
```

For request parameters, see: Sails req

For response options, see: Sails res

Calling fmap functions

fmap functions are called with either GET or POST using the URL route (eg. "/service/exec/set_filter"). Parameters can be passed as data payload or as URL parameters.

Our handling functions generally use var requestData = req.allParams(), making the handlers rather indiscriminate to how the parameters are passed.

An example of a POST request:

```
var postData = {
      filterParams: filter,
      asset: "152_search_1517988101045", // usually the track.label name
      dataset: "sample_data/json/volvox"
}
$.post( "/service/exec/set_filter", postData , function(data) {
    console.log( data );
}, "json");
```

An example of a GET request:

```
$.get("/service/exec/get_blastdata/?asset="+browser.jblast.asset+'&dataset=
↪'+encodeURIComponent(browser.config.dataRoot), function(data){
    console.log( data );
    $('.blast-hit-data').html("Hits: ("+data.filteredHits+'/'+data.hits+")");
});
```

Function Name Overlap

If two job services have the same function name, the first the first job service registered will take precedent.

For example: Say serviceA and serviceB both have a fmap function called my_function, and serviceA is defined before serviceB, then calling `/service/exec/my_function` will execute serviceA.my_function.

However, serviceB.my_function can still be addressed with the service-specific calling format, `/service/exec/serviceB:my_function`.

Obligatory Functions for Job Runners

Job services that are job runners that react to job execution, must implement the following functions:

```
// job service parameter validation
// jservice calls this to determine if the parameters are sufficient to execute the␣
↪job.
validateParams: function(params) {
    if (typeof params.searchParams === 'undefined') return "searchParams not defined";
    if (typeof params.searchParams.expr === 'undefined') return "search string␣
↪undefined";
    return 0;   // success
},
// job name generator
// jservice framework calls this to determine the jobs user-readable name that␣
↪appears in the job queue.
generateName(params) {
    return params.searchParams.expr+' search';
},
// jservice calls this to execute the job.  ``kJob`` is the kue object.
beginProcessing(kJob) {
    if (successful) kJob.kDoneFn();
    if (failed) kJob.kDoneFn(Error("this job failed because..."));
}
```

Job Service Configuration

Job services are defined in `config/globals.js` or in `jbconnect.config.js`.

```
jbrowse: {
    // list of services that will get registered.
    services: {
        // service                  display name                type                ␣
↪  alias
        'basicWorkflowService':     {name: 'basicWorkflowService',  type: 'workflow',␣
↪alias: "jblast"},
        'filterService':            {name: 'filterService',         type: 'service'},
```

(continues on next page)

(continued from previous page)

```
        'entrezService':            {name: 'entrezService',         type: 'service'}
    },
```

where - *service* refers to the job service module name - *display name* is the human readable name of the service - *type* - `workflow` means it's a job runner and `service` means it only hosts route functions.

`service` can either be the service module name (ie. "basicWorkflowService") or an the alias, if an alias if defined, given the configuration example below.

Submitting a Job

A Job Service must be implemented as a job runner to be a queueable job. (See *jbs-jobrunner*)

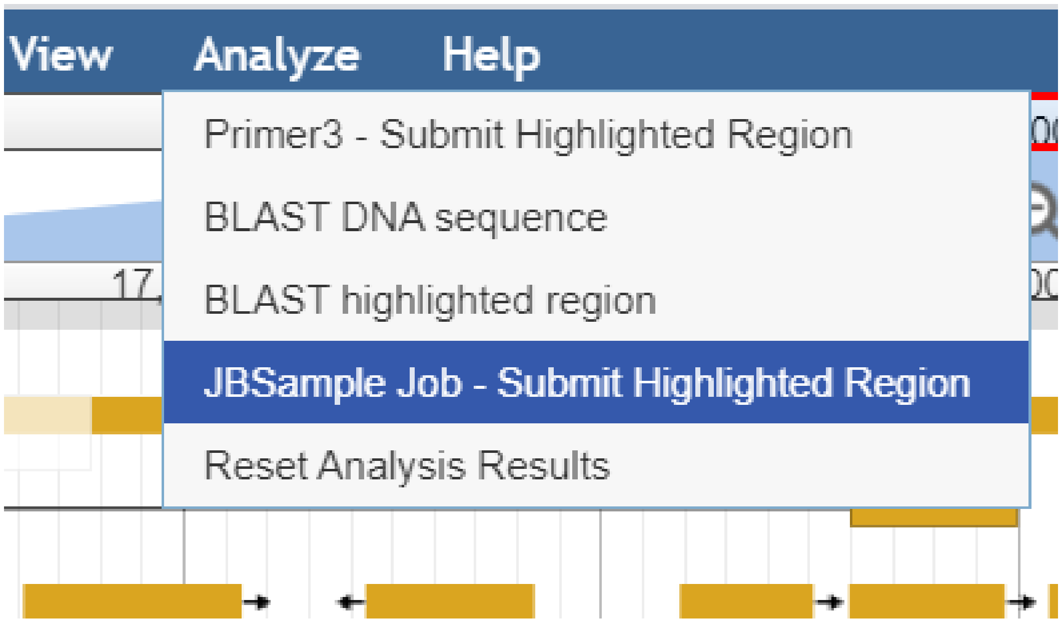
This is an example of job submission. The content of the POST data will depend of the type of job that is being submitted. However, `service:` must be included and reference an existing job service.

```
var postData = {
      service: "jblast",  // this can be the name of the job service or its alias
      dataset: "sample_data/json/volvox",
      // FASTA formated query sequence
      region: ">ctgA ctgA:44705..47713 (- strand) class=remark␣
↪length=3009\nacatccaatggcgaacataa...gcgagttt",
      workflow: "NCBI.blast.workflow.js"
  };
$.post( "/job/submit", postData , function( result ) {
    console.log( result );
}, "json");
```

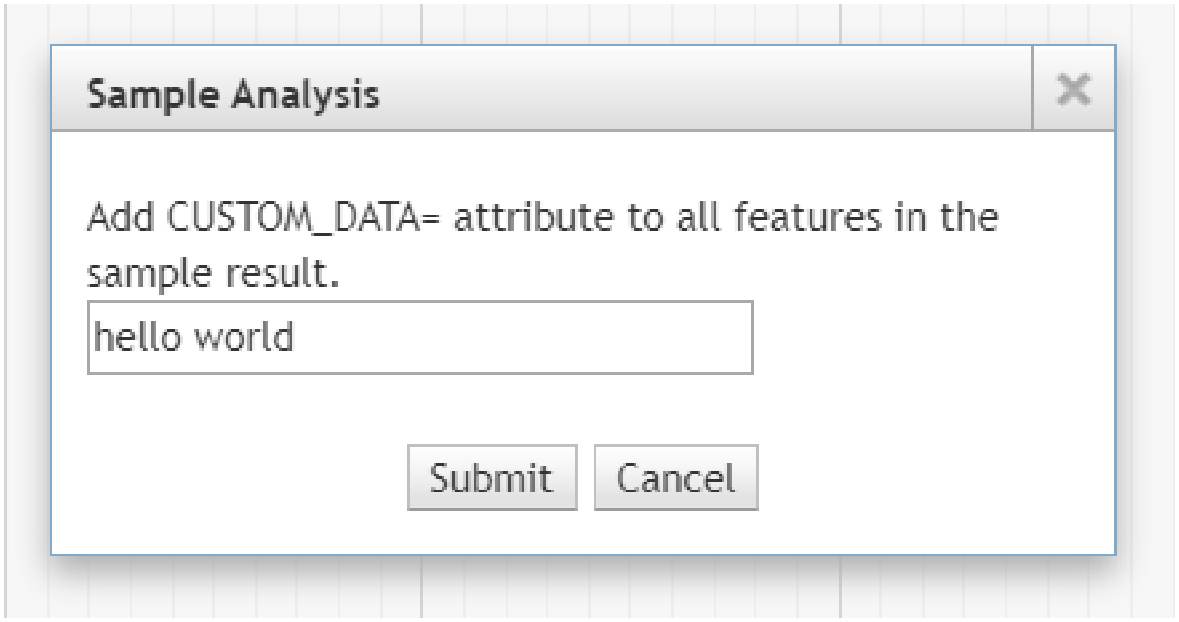
2.4 Tutorials

2.4.1 JBConnect Hook Tutorial

`api/services/localCommonService.js` is a workflow processing Job Service that can be used to execute general workflows scripts. In this example, we present demo analysis hook (demo-jbconnect-hook). We show how to create a client-side JBrowse plugin that integrates with JBrowse, adding a menu item under Analyze menu.



It's a fully functional demo module that has a server-side execution shell script and performs some arbitrary processing. The example also demonstrates the client-side plugin collects user data in its submit dialog box and passes it to be used by the execution script.


Sample Analysis
Add CUSTOM_DATA= attribute to all features in the sample result.
hello world
Submit
Cancel

The demo hook described in this section can be found here: <https://github.com/GMOD/demo-jbconnect-hook'_

JBCdemo JBrowse Plugin

This section describes a complete JBConnect installable hook.

The directory layout of a hook project is as such:

```
JBConnect project
├── api                                  Standard sails API layout, models, controllers,␣
↪etc.
```

(continues on next page)

(continued from previous page)

```
│    └── hook
│        └── index.js              hook index
├── bin                            Utilities
│    ├── jbutil-ext.js             jbutil command extensions
│    └── postinstall.js            package post installation
├── config                         Configuration files.
│    └── globals.js                global config file for module
├── plugins                        Client-side Plugins
│    └── JBCdemo                   Demo client plugin
├── package.json                   Node package description
└── workflows                      Workflows directory
     ├── demo-job.demo.wf.sh       Workflow script
     └── demo-job.TrackTemplate.json TrackTemplate
```

api/hook/index.js

The main purpose of this file is to facilitate merging the configurations, models and controllers with the main JBConnect, making them available globally.

Extending jbtuil command

jbutil-ext.js provides a means for the hook to extend the *jbutil* command.

This is further described in **'https://jbconnect.readthedocs.io/en/latest/configuration.html#extending-jbutil'_**

bin/postinstall.js

This performs the important roll of copying the workflows in the hook project into the workflows directory of JBConnect.

It can also be use to perform other post-install setup.

globals.js & workflowFilter

globals.js are merged with globals.js in JBConnect.

For the demo the workflowFitler applies to the get_workflow enumeration call

In main.js of the plugin the following structure is defined.

```
// analyze menu structure
browser.jbconnect.analyzeMenus.demo = {
    title: 'Demo Analysis',
    module: 'demo',
    init:initMenu,
    contents:dialogContent,
    process:processInput
};
```

Note that the module, defined as 'demo' here, is used by get_workflows call to filter the available workflows for a particular plugin. The definition in workflowFilter for 'demo' describes the filter. Only files that contain '.demo.wf' will be returned by get_workflows.

```
module.exports.globals = {
    jbrowse: {
        workflowFilter: {
            demo: {filter: '.demo.wf'},
        },
        ...
    }
};
```

In the file constructor of main.js, we find:

```
// analyze menu structure
browser.jbconnect.analyzeMenus.demo = {
    title: 'Demo Analysis',
    module: 'demo',
    init:initMenu,
    contents:dialogContent,
    process:processInput
};
```

The source can be found here: **‘https://github.com/GMOD/demo-jbconnect-hook/blob/master/plugins/JBCdemo/js/main.js‘_**

where,

- `title` is the title of the dialog box that is launched from the Analyze Menu.
- `module` is the module that is module name. Coincides with module name used in **‘Configuration of sample workflow script‘_**
- `init` is the function that initializes the selection items in the Analyze Menu for the module.
- `contents` is a function that builds the contents of the dialog box. This can be used to collect custom data prior to submitting.
- `process` is a function that collects the custom data from the fields created by `contents` to pass through the submit function.

In our example, the `initMenu()` does the setup of the Analyze menu item and when the item is selected by the user, it detects if a region has been highlighted. This is a pretty common thing that is check by nearly all our processing modules. If the region is not highlighted, We show an instructional dialog box instructing the user to highlight a region using JBrowse’s highlight feature.

In `dialogContent()`, we render an additional field in the submit dialog box for CUSTOM_DATA. The user can type any value in the input box. Upon submitting the job, the demonstration shows how data is passed from the user end to the execution script.

`processInput()` is called when the user clicks submit. Here we show the custom input field data are cellected and we show how to pass the field data to the system and ultimately submitted to JBConnect.

Again, processing begins through `localCommonService.js`, the thing that takes control of the job and launches the workflow script that was selected by the user. Note, the user will not see the workflow selection box unless there are more than one workflow. In our case, there is only one workflow script, so it will automatically be selected by the client plugin code.

demo-job.demo.wf.js Worflow Script

The workflow script resides in the workflow directory. In this example sample.samp.wf.js is a very simple script that copies sample.gff3 to the target directory; in demonstrating the passing of data from the client side to the server side script, it extracts the CUSTOM_DATA field that was captured by the JBSample plugin and appends the value to every feature of sample.gff3.

```
# cmd <id> <jobid> <jobdata> <tmpdir> <outdir>
echo "> my.sample.wf.sh " $0 $1 $2 $3 $4 $5

# copy sample.gff3 to target dir
cp ./bin/sample.gff3 "$5/$2.gff3"

# extract value of CUSTOM_DATA
MYVALUE=$(awk -v k=CUSTOM_DATA -F: '/{|}/{next}{gsub(/^ +|,$/,"");gsub(/"/,"");if(
↪$1==k)print $2}' $3)

# add CUSTOM_DATA=MYVALUE as attribute to all features
sed -e "s/$/;CUSTOM_DATA=$MYVALUE/" -i "$5/$2.gff3"
```

Note the 5 parameters that are passed to the command by `localCommonService`.

- $0 is the script path (ie: "/home/theuser/jbconnect/workflows/sample.samp.wf.sh")
- $1 <id> the job id (ie: "32")
- $2 <jobid> the job name (ie: "32-sample")
- $3 <jobdata> path of the job data file (ie: "/home/theuser/jbconnect/tmp/32-sample-jobdata.json")
- $4 <tmpdir> the directory where temporary or intermediate files might be placed.
- $5 <outdir> is the target directory where result files (like gff3 files) might be placed.

The full command looks something like this:

```
/home/theuser/jbconnect/workflows/sample.samp.wf.sh 32 32-sample
  /home/theuser/jbconnect/tmp/32-sample-jobdata.json
  /home/theuser/jbconnect/tmp /home/theuser/jb1151/sample_data/json/volvox/sample
```

`localCommonService` expects to see a file <outdir>/<jobid>.gff3. So, the script creates this result file in the target directory based on the given input parameters of the script. This is just the way `localCommonService` works. If the application requires other result files, a another Job Service would need to be created. (see *Creating a Stand-Alone Job Service for local workflow processing*)

The script can be found under the workflows dir, here

Configuration of localCommonService

The configuration is required to enable the system to recognize that the Job Service exists.

```
services: {
    'localCommonService':       {enable: true, name: 'localCommonService',  type:
↪'workflow', alias:'workflow'}
},
```

2.4.2 Creating a Stand-Alone Job Service for local workflow processing

This tutorial demonstrates how to create a job service that can be executed by the JBlast Plugin.

The source code for the tutorial can be found here

Job Runner functions

The function map defines the REST APIs that the job service supports. In the function map (`fmap`), `get_workflow` function is minimally require from the Process BLAST dialog. `get_hit_details` is not required since we don't actaully do a blast operation in the example.

```
module.exports = {

    fmap: {
        get_workflows:      'get'
    },
```

(required by Job Service)

Provides opportunity to initialize the Job Service module.

```
init(params,cb) {
    return cb();
},
```

(required by Job Runner Service)

Provides mechanism to validate parameters given by the job queuer. Since our example job is submitted by JBlast, we extect to see a region parameter.

```
validateParams(params) {
    if (typeof params.region === 'undefined') return "region not undefined";
    return 0;   // success
},
```

(required by Job Runner Service)

Job service generate readable name for the job that will appear in the job queue

```
generateName(params) {
    return "sample job";
},
```

(required by JBClient, not required for Job Services in general)

Return a list of available available options. This is used to populate the Plugin's Workflow. This should minimally return at least one item for JBlast client to work properly. Here, we are just passing a dummy list, which will be ignored by the rest of the example.

```
get_workflows (req, res) {

    wflist = [
        {
            id: "something",
            name: "sample do nothing job",
            script: "something",
            path: "./"
        }
    ];

    res.ok(wflist);
},
```

(required by Job Runner Service)

`beginProcessing()` is called by the job execution engine to begin processing. The kJob parameter is a reference to the Kue job.

```
beginProcessing(kJob) {
    let thisb = this;
    let nothingName = "sample nothing ";

    kJob.data.count = 10;   // 10 seconds of nothing
    let f1 = setInterval(function() {
        if (kJob.data.count===0) {
            clearInterval(f1);
            thisb._postProcess(kJob);
        }
        // update the job text
        kJob.data.name = nothingName+kJob.data.count--;
        kJob.update(function() {});
    },1000);
},

//  (not required)
//  After the job completes, we do some processing in postDoNothing() and then call
//  addToTrackList to insert a new track into JBrowse
_postProcess(kJob) {

    // insert track into trackList.json
    this.postDoNothing(kJob,function(newTrackJson) {
        postAction.addToTrackList(kJob,newTrackJson);
    });
},

//  (not required)
//  here, we do some arbitrary post prosessing.
//  in this example, we are setting up a jbrowse track from a canned template.
postDoNothing(kJob,cb) {

    let templateFile = approot+'/bin/nothingTrackTemplate.json';
    let newTrackJson = [JSON.parse(fs.readFileSync(templateFile))];

    let trackLabel = kJob.id+' sample job results';

    newTrackJson[0].label = "SAMPLEJOB_"+kJob.id+Math.random();
    newTrackJson[0].key = trackLabel;

    kJob.data.track = newTrackJson[0];
    kJob.update(function() {});

    cb(newTrackJson);
}
```

Note that queue data can be changed with the following:

```
kJob.data.name = nothingName+kJob.data.count--;
kJob.update(function() {});
```

Configuration

To enable: edit jbconnect.config.js add the `sampleJobService` line under `services` and disable the other services.

```
module.exports  = {
    jbrowse: {
        services: {
            'sampleJobService':         {enable: true,  name: 'sampleJobService',  ␣
↪type: 'workflow'},                      <====
            'localBlastService':        {enable: false, name: 'localBlastService', ␣
↪type: 'workflow', alias: "jblast"},
            'galaxyBlastService':       {enable: false, name: 'galaxyBlastService',␣
↪type: 'workflow', alias: "jblast"}
        },
    }
};
```

Monitoring processing

The job runner is responsible for monitoring the state of any potential lengthy analysis opertion. If the job runner service is intended to perform some lengthy analysis, there would have to be some mechanism to detect the completion of the operation.

Completion processing

To complete a job, call one of the following.

```
(success)  kJob.kDoneFn();
(fail)     kJob.kDoneFn(new Error("failed because something"));
```

This will change the status of the job to either completed or error.

In our example, the helper library postAction handles the completion:

```
postAction.addToTrackList(kJob,newTrackJson);
```

Upon calling `kJob.kDoneFn()`, the module is required to perform any necessary cleanup.

2.5 API

2.5.1 Namespace: `AuthController`

Local Navigation

- *Description*
- *Function:* `login`
- *Function:* `login`
- *Function:* `logout`

Description

Authentication Controller.

See also Passport model.

Function: `login`

Render the login page

The login form itself is just a simple HTML form:

```
<form role="form" action="/auth/local" method="post">
  <input type="text" name="identifier" placeholder="Username or Email">
  <input type="password" name="password" placeholder="Password">
  <button type="submit">Sign in</button>
</form>
```

You could optionally add CSRF-protection as outlined in the documentation: http://sailsjs.org/#!documentation/config.csrf

A simple example of automatically listing all available providers in a Handlebars template would look like this:

```
{{#each providers}}
  <a href="/auth/{{slug}}" role="button">{{name}}</a>
{{/each}}
```

The `next` parameter can specify the target URL upon successful login.

Example: `GET http://localhost:1337/login?next=http://localhost:1337/jbrowse?data=sample_data/json/volvox`

`login` (*req*, *res*)

Arguments

- **`req`** (*`Object`*) – request
- **`res`** (*`Object`*) – response

Function: `login`

`login` ()

Function: `logout`

Log out a user and return them to the homepage

Passport exposes a logout() function on req (also aliased as logOut()) that can be called from any route handler which needs to terminate a login session. Invoking logout() will remove the req.user property and clear the login session (if any).

For more information on logging out users in Passport.js, check out: http://passportjs.org/guide/logout/

Example: `GET http://localhost:1337/logout`

`logout`(*req*, *res*)

Arguments

- **`req`** (*`Object`*) – request
- **`res`** (*`Object`*) – response

Function: `register`

`register`()

Function: `loginstate`

get login state

`GET http://localhost:1337/loginstate`

Example Result:

```
{
    "loginstate": true,
    "user": {
        "username": "juser",
        "email": "juser@jbrowse.org"
    }
}
```

`loginstate`(*req*, *res*)

Arguments

- **`req`** (*`object`*) – request
- **`res`** (*`object`*) – response

Function: `provider`

Create a third-party authentication endpoint

`provider`(*req*, *res*)

Arguments

- **`req`** (*`Object`*) – request
- **`res`** (*`Object`*) – response

Function: `callback`

Create a authentication callback endpoint

This endpoint handles everything related to creating and verifying Pass- ports and users, both locally and from third-aprty providers.

Passport exposes a login() function on req (also aliased as logIn()) that can be used to establish a login session. When the login operation completes, user will be assigned to req.user.

For more information on logging in users in Passport.js, check out: http://passportjs.org/guide/login/

`callback` (*req*, *res*)

Arguments

- **`req`** (*`Object`*) – request
- **`res`** (*`Object`*) – response

Function: `disconnect`

`disconnect` ()

2.5.2 Module: `controllers/DatasetController`

Local Navigation

- *Description*
- *Function: `get`*

Description

REST Interfaces for Dataset model

Datasets are configure in `config/globals.js` file.

See Dataset Model

Subscribe to Dataset events:

```
io.socket.get('/dataset', function(resData, jwres) {console.log(resData);});
io.socket.on('dataset', function(event){
   consol.log(event);
}
```

Function: `get`

Enumerate or search datasets

`GET /dataset/get`

`get` (*req*, *res*)

Arguments

- **`req`** (*`object`*) – request data
- **`res`** (*`object`*) – response data

2.5.3 Module: `controllers/JobActiveController`

Local Navigation

Description

REST interfaces for JobActive model.

See: JobActive model.

Subscribe to JobActive events:

```
io.socket.get('/jobactive', function(resData, jwres) {console.log(resData);});
io.socket.on('jobactive', function(event){
   consol.log(event);
}
```

Function: `get`

Read job active record

`GET /jobactive/get`

`get` (*req*, *res*)

Arguments

- **`req`** (*`object`*) – request
- **`res`** (*`object`*) – response

2.5.4 Module: `controllers/JobController`

Local Navigation

Description

REST interfaces for Job model

See Job model.

Subscribe to Job events:

```
io.socket.get('/job', function(resData, jwres) {console.log(resData);});
io.socket.on('job', function(event){
   consol.log(event);
}
```

Function: `get`

Enumerate or search job list.

`GET /job/get`

Example usage (jQuery):

```
$.ajax({
    url: "/job/get",
    dataType: "text",
    success: function (data) {
       console.log(data)
    }
});
```

The returned `data` is a JSON array of *job* objects.

Example Job object:

```
{
    "id": 113,
    "type": "workflow",
    "progress": "100",
    "priority": 0,
    "data": {
      "service": "serverSearchService",
      "dataset": "sample_data/json/volvox",
      "searchParams": {
        "expr": "ttt",
        "regex": "false",
        "caseIgnore": "true",
        "translate": "false",
        "fwdStrand": "true",
        "revStrand": "true",
        "maxLen": "100"
      },
      "name": "ttt search",
      "asset": "113_search_1513478281528",
      "path": "/var/www/html/jbconnect/node_modules/jbrowse/sample_data/json/volvox/
↪ServerSearch",
      "outfile": "113_search_1513478281528.gff",
      "track": {
        "maxFeatureScreenDensity": 16,
        "style": {
```

(continues on next page)

(continued from previous page)

```
          "showLabels": false
        },
        "displayMode": "normal",
        "storeClass": "JBrowse/Store/SeqFeature/GFF3",
        "type": "JBrowse/View/Track/HTMLFeatures",
        "metadata": {
          "description": "Search result job: 113"
        },
        "category": "Search Results",
        "key": "113 ttt results",
        "label": "113_search_1513478281528",
        "urlTemplate": "ServerSearch/113_search_1513478281528.gff",
        "sequenceSearch": true
      }
    },
    "state": "complete",
    "promote_at": "1513478280038",
    "created_at": "1513478280038",
    "updated_at": "1513478292634",
    "createdAt": "2018-02-01T05:38:27.371Z",
    "updatedAt": "2018-02-01T05:38:27.371Z"
  }
```

get (*req*, *res*)

Arguments

- **req** (*object*) – request
- **res** (*object*) – response

Function: submit

Submit a job.

Example - submit sequence search:

```
var postData = {
    service: "serverSearchService",
    dataset: "sample_data/json/volvox,
    searchParams: searchParams
};
$.post("/job/submit, postData, function(retdata) {
   console.log(retdata)
},'json');
```

Returned data from job submit: `{ status: "success", jobId: 152 }`, where `jobId` is the id of the created job in the job queue.

submit (*req*, *res*)

Arguments

- **req** (*object*) – request
- **res** (*object*) – response

2.5.5 Module: `controllers/ServiceController`

Local Navigation

- *Description*
- *Function:* `get`

Description

REST interfaces for Service model.

See Service model

Subscribe to Service events:

```
io.socket.get('/service', function(resData, jwres) {console.log(resData);});
io.socket.on('service', function(event){
   consol.log(event);
}
```

Function: `get`

Enumerate job services (jservices)

`GET /service/get`

`get` (*req*, *res*)

Arguments

- **`req`** (*`object`*) – request
- **`res`** (*`object`*) – response

`GET or POST /service/exec/`

This example calls set_filter, a JBlast operation:

```
var postData = {
      filterParams: data,
      asset: "jblast_sample",
      dataset: "sample_data/json/volvox"
}
$.post( "/service/exec/set_filter", postData , function( data) {
    console.log( data );
}, "json");
```

The returned data depends on the service function that is called.

2.5.6 Module: `controllers/TrackController`

Local Navigation

Description

REST interfaces for TrackController

Subscribe to Track events:

```
io.socket.get('/track', function(resData, jwres) {console.log(resData);});
io.socket.on('track', function(event){
   consol.log(event);
}
```

Function: `get`

enumerate tracks or search track list.

Get all tracks `GET /track/get`

Get filtered tracks by dataset:

`GET /track/get?id=1` where id is the dataset id

`GET /track/get?path=sample_data/json/volvox` where path is the dataset path

get (*req*, *res*)

Arguments

- **req** (*object*) – request
- **res** (*object*) – response

Function: `get_tracklist`

get JBrowse tracklist

`GET /track/get_tracklist`

get_tracklist (*req*, *res*)

Arguments

- **req** (*object*) – request
- **res** (*object*) – response

Function: `add`

add a new track

`POST /track/add`

Must include "dataset" in the fields, which can be the path (string) or id (int)

Calling example:

```
let newTrack = {
    "dataset":"sample_data/json/volvox",
    "autocomplete": "all",
    "track": "EST",
    "style": {
        "className": "est"
    },
    "key": "HTMLFeatures - ESTs",
    "feature": [
        "EST_match:est"
    ],
    "storeClass": "JBrowse/Store/SeqFeature/NCList",
    "urlTemplate": "tracks/EST/{refseq}/trackData.json",
    "compress": 0,
    "label": "EST",
    "type": "FeatureTrack",
    "category": "Miscellaneous"
};
$.post( "/track/add", newTrack, function( data ) {
  console.log( "result", data );
}, "json");
```

`add`(*req*, *res*)

Arguments

- **`req`** (*`object`*) – request
- **`res`** (*`object`*) – response

Function: `modify`

modify an existing track

POST `/track/modify`

Calling example:

```
let modifyTrack = {
    "autocomplete": "all",
    "track": "EST",
    "style": {
        "className": "est"
    },
    "key": "HTMLFeatures - ESTs",
    "feature": [
        "EST_match:est"
    ],
    "storeClass": "JBrowse/Store/SeqFeature/NCList",
```

(continues on next page)

(continued from previous page)

```
    "urlTemplate": "tracks/EST/{refseq}/trackData.json",
    "compress": 0,
    "label": "EST",
    "type": "FeatureTrack",
    "category": "Miscellaneous"
};
$.post( "/track/modify", modifyTrack, function( data ) {
  console.log( "result", data );
}, "json");
```

`modify`(*req*, *res*)

Arguments

- **`req`** (`object`) – request
- **`res`** (`object`) – response

Function: `remove`

remove an existing track

`POST /track/remove`

Calling example:

```
$.post( "/track/remove", { id: 23 }, function( data ) {
  console.log( "result", data );
}, "json");
```

`remove`(*req*, *res*)

Arguments

- **`req`** (`object`) – request
- **`res`** (`object`) – response

2.5.7 Module: `controllers/UserController`

Local Navigation

- *Description*
- *Function:* `get`

Description

REST interfaces for UserController

Subscribe to User events:

```
io.socket.get('/user', function(resData, jwres) {console.log(resData);});
io.socket.on('user', function(event){
   consol.log(event);
}
```

Function: `get`

Enumerate or search users

`GET /user/get`

`get` (*req*, *res*)

Arguments

- **`req`** (*`object`*) – Enumerate or search users

`GET /user/get` :param object res: Enumerate or search users

`GET /user/get`

2.5.8 Module: `models/Dataset`

Local Navigation

Description

Dataset is a model that represents the JBrowse dataset. Generally, this includes path to the dataset and some of the data contained in trackList.json.

Datasets known to JBConnect are defined in config/globals.js (see: *Configuration*)

Dataset object:

```
{
  "name": "Volvox",
  "path": "sample_data/json/volvox",
  "createdAt": "2018-02-01T05:38:26.320Z",
  "updatedAt": "2018-02-01T05:38:26.320Z",
  "id": 1
}
```

Ref: Sails Models and ORM

Function: `Init`

Initializes datasets as defined in config/globals.js. (see: *Configuration*)

`Init` (*params*, *cb*)

Arguments

- **`params`** (*`object`*) – callback function
- **`cb`** (*`function`*) – callback function

Return undefined Initializes datasets as defined in config/globals.js.

(see: *Configuration*)

Function: `Get`

Get list of tracks based on critera in params

`Get` (*params*, *cb*)

Arguments

- **`params`** (*`object`*) – search critera (i.e. `{id: 1,user:'jimmy'}`)
- **`cb`** (*`function`*) – callback `function(err,array)`

Function: `Resolve`

Given either a dataset string (ie. “sample_data/json/volvox” or the database id of a dataset, it returns a dataset object in the form:

```
{
    path: "sample_data/json/volvox",
    id: 3
}
```

`Resolve` (*dval*)

Arguments

- **`dval`** (*`val`*) – dataset string (ie. “sample_data/json/volvox”) or id (int)

Code Example

```
{
    path: "sample_data/json/volvox",
    id: 3
}
```

Return object

- dataset object

dataset (string - i.e. “sample_data/json/volvox” if input was an id returns null if not found

Function: `Sync`

Sync datasets, defined in globals with database.

todo: need to improve, perhaps use async?

`Sync`()

Arguments

- **`Sync()`** – cb - callback function

2.5.9 Module: `models/Job`

Local Navigation

Description

Job model is an encapsulation of the Kue job framework.

Kue uses redis database. This model synchronizes the Job database with the redis data through the use of Kue's API.

Kue event messages are stuffed into a FIFO *_eventList* and dequeued with *_processNextEvent* to ensure order.

- **Example Job object:**

```
{
    "id": 113,
    "type": "workflow",
    "progress": "100",
    "priority": 0,
    "data": {
      "service": "serverSearchService",
      "dataset": "sample_data/json/volvox",
      "searchParams": {
        "expr": "ttt",
        "regex": "false",
        "caseIgnore": "true",
        "translate": "false",
        "fwdStrand": "true",
        "revStrand": "true",
        "maxLen": "100"
      },
      "name": "ttt search",
      "asset": "113_search_1513478281528",
      "path": "/var/www/html/jbconnect/node_modules/jbrowse/sample_data/json/volvox/
↪ServerSearch",
      "outfile": "113_search_1513478281528.gff",
      "track": {
        "maxFeatureScreenDensity": 16,
        "style": {
          "showLabels": false
        },
        "displayMode": "normal",
        "storeClass": "JBrowse/Store/SeqFeature/GFF3",
        "type": "JBrowse/View/Track/HTMLFeatures",
        "metadata": {
          "description": "Search result job: 113"
        },
        "category": "Search Results",
        "key": "113 ttt results",
        "label": "113_search_1513478281528",
        "urlTemplate": "ServerSearch/113_search_1513478281528.gff",
        "sequenceSearch": true
      }
    },
    "state": "complete",
    "promote_at": "1513478280038",
    "created_at": "1513478280038",
    "updated_at": "1513478292634",
    "createdAt": "2018-02-01T05:38:27.371Z",
    "updatedAt": "2018-02-01T05:38:27.371Z"
  }
```

Event Mappings:

Kue Events	Job Events
• queue-enqueue	create
• queue-start	update
• queue-failed	update
• queue-failed-attempt	update
• queue-progress	update
• queue-complete	update
• queue-remove	remove
• queue-promotion	unused

Ref: Sails Models and ORM

Function: `Init`

start the monitor

`Init` ()

Function: `Get`

Get list of tracks based on critera in params

`Get` (*params*, *cb*)

Arguments

- **`params`** (*`object`*) – search critera (i.e. `{id: 1,user:'jimmy'}`)
- **`cb`** (*`function`*) – callback function(err,array)

Function: `Remove`

`Remove` (*id*)

Arguments

- **`id`** (*`int`*) – id of the item to be removed
- **`Remove(id)`** – cb - callback function(err,

Function: `Submit`

`Submit`()

Function: `_jobRunner`

`_jobRunner`()

Function: `_kueEventMonitor`

`_kueEventMonitor`()

Function: `_pushEvent`

`_pushEvent`()

Function: `_processNextEvent`

`_processNextEvent`()

Function: `_createJob`

`_createJob`()

Function: `_updateJob`

`_updateJob`()

Function: `kJob`

`kJob`()

Function: `sJob`

`sJob`()

Function: `_destroyJob`

`_destroyJob`()

Function: `_listJobs`

`_listJobs`()

Function: `_syncJobs`

Synchronize all kue jobs (kJobs) and sails db jobs (sJobs) Called upon initialization of the Job model

if the kJob exists but sJob does not, then create the sJob from kJob. If the sJob exists but not kJob, then delete the sJob

`_syncJobs`()

Function: `kJobs`

`kJobs`()

Function: `sJobs`

`sJobs`()

Constant: `async`:

Constant: _:

Constant: `fetch`:

2.5.10 Module: `models/JobActive`

Local Navigation

Description

JobActive holds a count of the number of active jobs. It only contains one record that gets updated when the number of active jobs changes. A timer thread monitors the job queue for active jobs and updates the JobActive record with any changes to the number of active jobs. Subscribers to the record (clients) will get notification. JBClient plugin uses this to determine if a job is active and changes the activity icon of the job queue panel.

JobActive object example:

```
{
  "active": 0,
  "createdAt": "2017-11-23T00:53:41.864Z",
  "updatedAt": "2018-02-07T07:59:32.471Z",
  "id": 1
}
```

Function: `Init`

initialize starts the job active monitor

`Init` (*params*, *cb*)

Arguments

- **`params`** (*`object`*) – value is ignored
- **`cb`** (*`type`*) – callback `function cb(err)`

Function: `Get`

Get list of tracks based on critera in params

`Get` (*params*, *cb*)

Arguments

- **`params`** (*`object`*) – search critera (i.e. `{id: 1,user:'jimmy'}`)
- **`cb`** (*`function`*) – callback `function(err,array)`

Function: `_activeMonitor`

`_activeMonitor` ()

2.5.11 Module: `models/Passport`

Local Navigation

- *Description*
- *Function: `hashPassword`*

Description

The Passport model handles associating authenticators with users. An authen- ticator can be either local (password) or third-party (provider). A single user can have multiple passports, allowing them to connect and use several third-party strategies in optional conjunction with a password.

Since an application will only need to authenticate a user once per session, it makes sense to encapsulate the data specific to the authentication process in a model of its own. This allows us to keep the session itself as light- weight as possible as the application only needs to serialize and deserialize the user, but not the authentication data, to and from the session.

Function: `hashPassword`

Hash a passport password.

`hashPassword` (*password*, *next*)

Arguments

- **password** (*Object*) – password
- **next** (*function*) – next policy

2.5.12 Module: `models/Service`

Local Navigation

- *Description*

Description

The service module implements the job service frameowrk which are installable modules that can host web services and be a job execution processing for a particular type of job.

Installable services are generally named <servicename>Service.js and reside in the api/services directory. For example: a job service built into this project is serverSearchService.js

api/services/serviceProc.js is the bettr part of the implementation of service

Job services are defined in *config/globals.js* in the jbrowse/services section.

Example job service object:

```
{
  "name": "serverSearchService",
  "type": "service",
  "module": "search",
  "createdAt": "2018-02-01T05:38:26.289Z",
  "updatedAt": "2018-02-07T07:59:31.430Z",
  "id": 1
}
```

2.5.13 Module: `models/Track`

Local Navigation

- *Description*
- *Function:* `Init`
- *Function:* `StartWatch`
- *Function:* `PauseWatch`
- *Function:* `ResumeWatch`
- *Function:* `Get`
- *Function:* `GetTrackList`
- *Function:* `Add`
- *Function:* `Modify`

Description

Track is a model for a list of tracks that are in the `trackList.json`'s `[tracks]` section.

Ref: Sails Models and ORM

Track object example:

```
{
  "dataset": 1,
  "path": "sample_data/json/volvox",
  "lkey": "DNA",
  "trackData": {
    "seqType": "dna",
    "key": "Reference sequence",
    "storeClass": "JBrowse/Store/Sequence/StaticChunked",
    "chunkSize": 20000,
    "urlTemplate": "seq/{refseq_dirpath}/{refseq}-",
    "label": "DNA",
    "type": "SequenceTrack",
    "category": "Reference sequence"
  },
  "createdAt": "2018-02-01T05:38:26.339Z",
  "updatedAt": "2018-02-01T05:38:26.339Z",
  "id": 1
}
```

Function: `Init`

`Init` (*params*, *cb*)

Arguments

- **`params`** (*`type`*) – parameters
- **`cb`** (*`type`*) – callback function

Function: `StartWatch`

`StartWatch` ()

Function: `PauseWatch`

`PauseWatch` ()

Function: `ResumeWatch`

`ResumeWatch`()

Function: `Get`

Get list of tracks based on critera in params

`Get`(*params*, *cb*)

Arguments

- **`params`** (*`object`*) – search critera
- **`cb`** (*`function`*) – callback `function(err,array)`

Function: `GetTrackList`

Get JBrowse tracklist in JSON format of tracks based on critera in params

`GetTrackList`(*params*, *cb*)

Arguments

- **`params`** (*`object`*) – search critera
- **`cb`** (*`function`*) – callback `function(err,json)`

Function: `Add`

`Add`()

Function: `Modify`

`Modify`()

Function: `Remove`

`Remove`(*dataset*, *id*)

Arguments

- **`dataset`** (*`string`*) – (eg: "sample_data/json/volvox")
- **`id`** (*`int`*) – id of the item to be removed
- **`id)`** (*`Remove(dataset,`*) – cb - callback function(err,

Function: `Sync`

Sync tracklist.json tracks with Track model (promises version)

`Sync`(*dataset*)

Arguments

- **`dataset`** (*`string`*) – ie. ("sample_data/json/volvox")

Function: `SyncTest`

`SyncTest`()

Function: `cleanTracks`

remove all tracks for a given user. if params.session does not exist or user not logged in, returns false.

`cleanTracks`(*params*)

Arguments

- **`params`** (`object`) – remove all tracks for a given user.

if params.session does not exist or user not logged in, returns false. :return int: returns true if successful, false if nothing done

Constant: `Promise`:

Constant: `fs`:

Constant: `path`:

Constant: `deferred`:

Constant: `deepmerge`:

Constant: _:

2.5.14 Module: `models/User`

Local Navigation

- *Description*

Description

User is the data model for a user.

Example User object:

```
{
  "username": "juser",
  "email": "juser@jbrowse.org",
  "admin": true,
  "createdAt": "2017-11-29T21:00:56.726Z",
  "updatedAt": "2017-11-29T21:00:56.726Z",
  "id": 2
}
```

2.5.15 Module: `policies/bearerAuth`

Local Navigation

- *Description*

Description

bearerAuth Policy

Policy for authorizing API requests. The request is authenticated if the it contains the accessToken in header, body or as a query param. Unlike other strategies bearer doesn't require a session. Add this policy (in config/policies.js) to controller actions which are not accessed through a session. For example: API request from another client

2.5.16 Module: `policies/isAdmin`

Local Navigation

- *Description*
- *Function:* `nonAdminAction`

Description

isAdmin policy provides passage if the user contains the property admin: true.

req.session looks something like this: req.session Session {

> **cookie: { path: '/',** _expires: null, originalMaxAge: null, httpOnly: true
>
> }, passport: { user: 2 }, authenticated: true, (true if logged in, user: { username: 'juser', email: 'juser@jbrowse.org' }

}

Function: `nonAdminAction`

`nonAdminAction``()`

2.5.17 Module: `policies/passport`

Local Navigation

- *Description*

Description

Passport Middleware

Policy for Sails that initializes Passport.js and as well as its built-in session support.

In a typical web application, the credentials used to authenticate a user will only be transmitted during the login request. If authentication succeeds, a session will be established and maintained via a cookie set in the user's browser.

Each subsequent request will not contain credentials, but rather the unique cookie that identifies the session. In order to support login sessions, Passport will serialize and deserialize user instances to and from the session.

For more information on the Passport.js middleware, check out: http://passportjs.org/guide/configure/

2.5.18 Module: `policies/sessionAuth`

Local Navigation

Description

Simple policy to allow any authenticated user. Assumes that your login action in one of your controllers sets *req.session.authenticated = true;*

Ref: Sails Policies Concepts

2.5.19 Module: `services/jbutillib`

Local Navigation

Description

Support library for jbutil command

Function: `doExtScripts`

Traverse `jbutils-ext.js` of submodules (*-jbconnect-hook)

`doExtScripts`(*cb*)

Arguments

- **cb** (*function*) – callback

Function: `getMergedConfig`

Returned merged jbrowse config. Merged from `*-jbconnect-hook config/globals.js`, local `config/globals.js`

`getMergedConfig`()

Function: `mergeConfigJs`

`mergeConfigJs`()

Function: `getClientDependencies`

`getClientDependencies`(*filter*)

Arguments

- **filter** (*string*) – (ie. ".css" or ".js")

Return Array the aggregated client dependencies from webIncludes.

Function: `injectIncludesIntoHtml`

Inject css/js into JBrowse index.html

`injectIncludesIntoHtml`()

Function: `setupPlugins`

add plugins to `trackList.json`.

`setupPlugins`()

Function: `removeIncludesFromHtml`

remove css/js from JBrowse index.html

`removeIncludesFromHtml`()

Function: `unsetupPlugins`

remove plugins from `trackList.json`.

`unsetupPlugins`()

Function: `safeCopy`

copy src to targ, but if targ exists, it will backup the target by appending a number

`safeCopy`(*src*, *origTarg*)

Arguments

- **`src`** (*`string`*) – source
- **`origTarg`** (*`string`*) – target

Return string final target filename

Function: `safeWriteFile`

if content is the same as target, do nothing. if content is different than target, write new content to target file.

`safeWriteFile`(*content*, *origTarg*)

Arguments

- **`content`** (*`type`*) – content to write
- **`origTarg`** (*`type`*) – target file

Return string backuped up filename

Function: `install_database`

Install the sails database from `./bin`.

`install_database`(*overwrite*)

Arguments

- **`overwrite`** (*`int`*) – 0, do not overwrite db. 1, overwrite db.

Function: `zapRedis`

cleanout redis database

`zapRedis`()

Function: `injectPlugins`

Inject client-side plugins into the JBrowse plugins dir

Note: as of JBrowse 1.13.0, you must run *npm run build* after this function, webpack build. called in sails lift /tasks/register .. jb-inject-plugins to function properly

if env E2E_COVERAGE is defined, it will instrument the plugins before installing.

Example: // injects plugins and instruments JBClient plugin and builds webpack in JBrowse ./jbutil –pushplugins –coverage JBClient –buildwebpack

`injectPlugins`(*plugin*)

Arguments

- **`plugin`** (`string`) – if defined (ie. "JBClient"), it will instrument the given plugin

Return injectPlugins(plugin) (int) count - count of plugins injected.

Function: `injectTracklist`

`injectTracklist`()

Function: `buildWebpack`

`buildWebpack`()

Function: `removePlugins`

remove client side plugins from JBrowse index.html

`removePlugins`()

Function: `getPlugins`

get the list of plugins. This includes JBConnect plugins as well as plugins of JBConnect hook modules that are loaded.

`getPlugins`()

Return object array of plugin objects

Function: `addRoute`

Add a route

`addRoute`(*params*, *module*, *route*, *target*)

Arguments

- **`params`** (*object*) – eg. `{app: <app-object>,express: <express-object>}`
- **`module`** (*string*) – the module name (ie. `"jquery"`)
- **`route`** (*string*) – the route (ie. `"/jblib/jquery"`)
- **`target`** (*string*) – the target (ie `"/var/www/html/jbconnect/node_modules/jquery"`)

Function: `addPluginRoute`

Add a plugin route

`addPluginRoute`(*params*, *module*, *route*, *target*)

Arguments

- **`params`** (*object*) – eg. `{app: <app-object>,express: <express-object>}`
- **`module`** (*string*) – the module name (ie. `"jblast"`)
- **`route`** (*string*) – the route (ie. `"/jbrowse/plugins/JBlast"`)
- **`target`** (*string*) – the target (ie `"/var/www/html/jbconnect/node_modules/jblast-jbconnect-hook/plugins/JBlast"`)

Constant: `fs`:

Constant: `path`:

Constant: `approot`:

Constant: `glob`:

Constant: `sh`:

Constant: `merge`:

Constant: `config`:

Constant: `html2json`:

Constant: `json2html`:

Constant: `_`:

Constant: `async`:

2.5.20 Module: `services/passport`

Local Navigation

- *Description*

Description

Passport Service

A painless Passport.js service for your Sails app that is guaranteed to Rock Your Socks™. It takes all the hassle out of setting up Passport.js by encapsulating all the boring stuff in two functions:

> passport.endpoint() passport.callback()

The former sets up an endpoint (/auth/:provider) for redirecting a user to a third-party provider for authentication, while the latter sets up a callback endpoint (/auth/:provider/callback) for receiving the response from the third-party provider. All you have to do is define in the configuration which third-party providers you'd like to support. It's that easy!

Behind the scenes, the service stores all the data it needs within "Pass- ports". These contain all the information required to associate a local user with a profile from a third-party provider. This even holds true for the good ol' password authentication scheme – the Authentication Service takes care of encrypting passwords and storing them in Passports, allowing you to keep your User model free of bloat.

2.5.21 Module: `services/postAction`

Local Navigation

- *Description*
- *Function: `addToTrackList`*

Description

Used by hooks to add a track and announce to subscribers.

Function: `addToTrackList`

Add track to track list and notify.

`addToTrackList` (*kJob*, *newTrackJson*)

Arguments

- **`kJob`** (*`object`*) – kue job reference
- **`newTrackJson`** (*`JSON`*) – new track JSON

2.5.22 Module: `services/serviceProc`

Local Navigation

- *Description*
- *Function: `init`*
- *Function: `addService`*

Description

Support functions for Service model.

Function: `init`

initialize the job service framework

`init` (*params*, *cb2*)

Arguments

- **`params`** (`type`) – parameters
- **`cb2`** (`type`) – callback

Function: `addService`

add a service

`addService` (*service*, *cb*)

Arguments

- **`service`** (`object`) – service
- **`cb`** (`function`) – callback

Function: `execute`

`execute` ()

Constant: `async`:

Constant: `fs`:

Index