QMachine Documentation

Release 1.2.4

Sean Wilkinson

May 25, 2018

Contents

CHAPTER 1

Overview

QMachine (QM) is a web service for distributed computing. Its design relaxes the usual requirements of a distributed computer so far that it can be powered completely by web browsers – without installing anything. As a model for computation, QM has been detailed in a recent paper, QMachine: commodity supercomputing in web browsers. This manual details QM as a software platform, with particular focuses on how it works, how to use it, and how to contribute to the open-source project.

1.1 How it works

Coordination of a distributed computing effort requires a common data interchange format and a common communications protocol. As a web service, QM provides an Application Programming Interface (API) that uses JavaScript Object Notation (JSON) as its common data interchange format and Hypertext Transfer Protocol (HTTP) as its communications protocol. In short, QM's API allows JSON-encoded messages sent to URLs over HTTP to become a message-passing interface for distributed computing. The use of web-friendly technologies is deliberate – QM is designed to be as universal as the World Wide Web itself.

Most QM-related software falls into two categories. API clients are programs that consume QM's API, and API servers are programs that provide the API. Web servers represent a third category, however, that is important only for hosting content such as the browser client library.

1.1.1 API clients

An API client is a program that consumes QM's API by sending JSON-encoded messages over HTTP to specific URLs, as defined by QM's *HTTP API* (page 3). The vast majority of programs will fall into this category, and most of these programs will use a client library for convenience. Currently, the only client library supported by the QM project is the *browser client* (page 5), which is written completely in JavaScript. An outdated Node.js client is now being resurrected, and a Ruby client is planned.

1.1.2 API servers

An API server, by way of contrast, is a program that provides QM's API by listening for and responding to specific HTTP requests, as defined by QM's *HTTP API* (page 3). There are two implementations to choose from: the *original reference version* (page 7) written in Node.js and the *"teaching version"* (page 9) written in Ruby. The Node.js version is recommended for production.

1.1.3 Web servers

For convenience in hosting, basic web servers are packaged alongside the API servers. A web server listens for and responds to HTTP requests for files and other resources that are published online. A web server is not strictly necessary as part of QM, but it is useful for making the browser client available to web browsers. The web servers bundled with the API servers are present only for convenience, and it is trivial to use off-the-shelf web servers like Apache or Nginx instead. (In fact, QM's own homepage uses Nginx.)

1.2 How to use it

See *Running a local sandbox* (page 11) and *Running on Platform-as-a-Service* (page 15).

1.3 How to contribute

See *Source code* (page 17).

CHAPTER 2

HTTP API

QM's Application Programming Interface (API) uses JavaScript Object Notation (JSON) as its common data interchange format and Hypertext Transfer Protocol (HTTP) as its communications protocol. It also supports Cross-Origin Resource Sharing.

2.1 Stable

This table specifies the "routes" understood by QM's API server. Request and response data use JSON format, but data may be omitted where values are left blank. JSON objects are denoted `{}`, and JSON arrays are denoted `[]`.

	HTTP Request			HTTP Response	
Purpose	Method	URL	Data	Code	Data
Get avar	GET	/box/cardboard?key=982770f29		200	`{}`
Get list	GET	/box/cardboard?status=waiting		200	`[]`
Set avar	POST	/box/cardboard?key=982770f29	`{}`	201	

The data model is based on Quanah's asynchronous variables ("avars"). An avar is a JavaScript object that acts as a generic container for other types by storing data in its "val" property. QM extends this model by adding "box", "key", and optional "status" properties. The "box" property is useful for grouping avars in various ways, while the "key" property is a unique identifier for each avar. The optional "status" property is present on avars that represent job descriptions.

2.1.1 Get avar

To get the value of an avar, a client must request it by known box and known key. For the example shown in the table, an avar with "cardboard" as its box, "982770f29" as its key, and 2 as its value would look like

```
{"box":"cardboard","key":"982770f29","val":2}
```

2.1.2 Get list

Because job descriptions are avars that must be accessed by a known box and a known key, a volunteer which knows only the box cannot run a job until it also knows the job's key. Clients may request a list of jobs' keys using a known box and a known status; this list is represented as a JSON array. For the example shown in the table, an array containing three avars' keys might look like

```
["job_key_1","job_key_2","job_key_3"]
```

2.1.3 Set avar

To set the value of an avar, a client must send the new value in a request by known box and known key. No response data will be returned, but a successful response will be indicated by an HTTP status code of 201.

2.2 Experimental

The latest version of the API (1.2.4) includes extensions that enable versioned API calls. These routes were added in anticipation of future needs to support legacy APIs without running legacy servers. Although deprecating the word "box" in favor of "v1" seems desirable, it is possible that it anticipates needs that will never arise in practice. Discussion and input here would be much appreciated.

	HTTP Request			HTTP Response	
Purpose	Method	URL	Data	Code	Data
Get avar	GET	/v1/cardboard?key=982770f29		200	{ }
Get list	GET	/v1/cardboard?status=waiting		200	[ ]
Set avar	POST	/v1/cardboard?key=982770f29	{ }	201	

CHAPTER 3

Browser client

3.1 Getting started

The first step in getting started with any piece of software is installation, but "installation" is a misnomer when developing web applications. Browsers use JavaScript (JS) as their "native" programming language, but JS programs are never truly installed because they cannot alter or extend the browsers themselves. Instead, JS programs are downloaded according to the contents of webpages, and they run in "disposable" sandboxed environments that exist only while the webpages are open. Thus, the "installation" of QM's browser client into a webpage is as simple as adding a single line of HTML:

```
<script src="https://www.qmachine.org/qm.js"></script>
```

When the webpage loads, its JS environment will contain a `QM` object that will allow other programs to submit jobs to and volunteer to execute jobs from the official QM servers – for free!

Note: Modern web browsers can often be programmed "externally" to the webpages themselves. For example, browsers may load custom user scripts, or they may be scripted by external programs to run unit tests. These capabilities are not leveraged in any way by the QM browser client. It may or may not run correctly within Web Worker contexts, but the client described in this manual expects to run within the "ordinary" webpage context of a modern web browser, unassisted by applets, extensions, plugins, etc.

For the hardcore software engineers out there, QM's browser client is available for "installation" with Bower:

```
$ bower install qm
```

3.2 Basic use

3.2.1 Submitting jobs

See http://www.biomedcentral.com/1471-2105/15/176#sec4 for now.

3.2.2 Using volunteers' machines

Two convenience functions, `QM.start` and `QM.stop`, are provided in order to control a simple non-blocking (asynchronous) event loop externally. The loop "fires" approximately once per second, and if appropriate, it runs the `QM.volunteer` function. This internal event loop is neither externally configurable nor necessary for using QM – it simply wraps the `QM.volunteer` function for convenience, rather than forcing application code to implement its own non-blocking loop.

3.3 Advanced use

QM's browser client leverages asynchronous variables ("avars") extensively to manage concurrency issues in an object-oriented way, and this programming model is provided by Quanah and its JavaScript library. Tutorials for advanced use are forthcoming, but they will essentially discuss working with avars. For now, the best reference on Quanah is its manual.

CHAPTER 4

Node.js

4.1 Getting started

A Node.js module is available for installation with Node Package Manager (NPM). To install it in the current directory,

```
$ npm install qm
```

4.2 API client

A client for Node.js is planned, but it not yet available as part of the module. When it becomes available, it will be possible to submit jobs from a Node.js program to be executed by volunteer compute nodes.

4.3 API server

The original, reference version of QM is available as part of the module, and a basic web server is also provided to enable the use of web browsers as compute nodes if so desired.

A QM server can be launched by a Node.js program as shown in the following example, which includes default configuration values and commented database connection strings:

```
var qm = require('qm');

qm.launch_service({
    avar_ttl:           86400,      //- expire avars after _ seconds
    enable_api_server:  false,
    enable_cors:        false,
    enable_web_server:  false,
    gc_interval:        60,         //- evict unused avars every _ seconds
    hostname:           '0.0.0.0',  //- aka INADDR_ANY
```

(continues on next page)

(continued from previous page)

```
    log: function (request) {
     // This function is the default logging function.
        return {
            host: request.headers.host,
            method: request.method,
            timestamp: new Date(),
            url: request.url
        };
    },
    match_hostname:     false,
    max_body_size:      65536,      //- 64 * 1024 = 64 KB
    max_http_sockets:   500,
    persistent_storage: {
     // couch:           'http://127.0.0.1:5984/db',
     // mongo:           'mongodb://localhost:27017/test'
     // postgres:        'postgres://localhost:5432/' + process.env.USER
     // redis:           'redis://127.0.0.1:6379'
     // sqlite:          'qm.db'
    },
    port:               8177,
    static_content:     'katamari.json',
    trafficlog_storage: {
     // couch:           'http://127.0.0.1:5984/traffic'
     // mongo:           'mongodb://localhost:27017/test'
     // postgres:        'postgres://localhost:5432/' + process.env.USER
    },
    worker_procs:       1
});
```

The Node.js version of the API server can use any of five different databases to provide persistent storage for its message-passing interface: CouchDB, MongoDB, PostgreSQL, Redis, and SQLite. It can also log traffic data into CouchDB, MongoDB, or PostgreSQL if desired.

4.3.1 Try it live

Live, interactive demonstrations of the API and web servers are available at runnable.com. These will give you a different insight into the guts of QM by allowing you to make your own copies to play with and debug, completely for free.

CHAPTER 5

Ruby

5.1 Getting started

A Ruby gem is available for installation with RubyGems. To install it globally,

```
$ gem install qm
```

5.2 API client

A client for Ruby is under development. Basic low-level functions are now available in the stable release, but no effort has been made to leverage any of Ruby's language-level concurrency features. It is possible to use the current functions in a Ruby program to submit jobs to be executed by volunteer compute nodes, but it is not yet convenient.

5.3 API server

The "teaching version" of QMachine has now been merged into the Ruby gem, and a basic web server is also provided to enable the use of web browsers as compute nodes if so desired.

A QM server can be launched by a Ruby program as shown in the following example, which includes default configuration values and commented database connection strings:

```
require 'qm'

QM.launch_service({
    avar_ttl:           86400,      # expire avars after _ seconds
    enable_api_server:  false,
    enable_cors:        false,
    enable_web_server:  false,
    gc_interval:        60,         # evict unused avars every _ seconds
```

(continues on next page)

(continued from previous page)

```
    hostname:            '0.0.0.0',
    max_body_size:       65536,       # 64 * 1024 = 64 KB
    persistent_storage: {
      # mongo:           'mongodb://localhost:27017/test'
      # postgres:        'postgres://localhost:5432/' + ENV['USER']
      # redis:           'redis://127.0.0.1:6379'
      # sqlite:          'qm.db'
    },
    port:                8177,
    public_folder:       'public',
    trafficlog_storage: {
      # mongo:           'mongodb://localhost:27017/test'
      # postgres:        'postgres://localhost:5432/' + ENV['USER']
    },
    worker_procs:        1
})
```

The Ruby version of the API server has less flexibility than the original Node.js version does. There are now four choices to persist storage for the message-passing interface, but MongoDB is strongly recommended. Experimental support for PostgreSQL, Redis, and SQLite is available, but that support may be removed in the future. Currently, MongoDB is the only supported database for logging traffic data.

CHAPTER 6

Running a local sandbox

QMachine (QM) can be installed and run locally, which can be extremely useful for development as well as for deployment behind firewalls.

6.1 Prerequisites

QM is developed on Mac and Linux systems, and the Makefile located in the root of the main project's Git repository contains live instructions for building and testing *everything*.

To get up and running, you will need GNU Make, a standard POSIX development environment, and Git.

To build and run an API server, you will need MongoDB and either Node.js or Ruby. If you choose the Node.js version, you can use CouchDB, PostgreSQL, Redis, or SQLite instead, but the Makefile assumes Mongo by default.

To build the homepage and/or custom images, you will need ImageMagick.

PhantomJS is used for regression testing and for creating snapshots that can be converted into thumbnails.

6.1.1 Mac OS X

To get started on Mac OS X with your own local sandbox, you will want to install the Homebrew package manager using directions from its website. It will allow you to install all of the software mentioned previously by launching Terminal and typing

```
$ brew install couchdb imagemagick mongodb node phantomjs postgresql redis
```

I highly recommend installing Git through Homebrew, as well, but it isn't required. I prefer Apache's official CouchDB.app and Heroku's Postgres.app over the distributions installed by Homebrew because they include convenient launchers that live in the menu bar.

6.1.2 Ubuntu Linux

Ubuntu has a great built-in package manager that we can take advantage of, but package names vary by version. Incantations are forthcoming.

6.2 Node.js

First, make sure that you have NPM installed:

```
$ which npm || echo 'NPM is missing'
```

If NPM is missing, refer to its documentation for the installation procedure.

Next, check out QM's source code from GitHub:

```
$ git clone --recursive https://github.com/qmachine/qmachine.git
```

Now, select your local copy of the repository as the current directory:

```
$ cd qmachine/
```

Finally, start MongoDB and then launch QM on localhost:

```
$ make node-app
```

QM defaults to MongoDB for storage, but a different database can be specified explicitly:

```
$ make node-app db=couch
$ make node-app db=mongo
$ make node-app db=postgres
$ make node-app db=redis
$ make node-app db=sqlite
```

6.3 Ruby

A Ruby implementation of QM is also available in the project repository, and its directions are similar to those for Node.js. If you don't already use RVM, you should – it's fantastic and highly recommended.

First, make sure that you have Bundler installed:

```
$ which bundler || echo 'Bundler is missing'
```

If Bundler is missing, install it:

```
$ gem install bundler
```

Next, check out QM's source code from GitHub:

```
$ git clone --recursive https://github.com/qmachine/qmachine.git
```

Now, select your local copy of the repository as the current directory:

```
$ cd qmachine/
```

Finally, start MongoDB and then launch QM on localhost:

```
$ make ruby-app
```

QM defaults to MongoDB for storage, but the Ruby version also has experimental support for Postgres, Redis, and SQLite. Thus, possible command-line incantations include:

```
$ make ruby-app db=mongo
$ make ruby-app db=postgres
$ make ruby-app db=redis
$ make ruby-app db=sqlite
```

CHAPTER 7

Running on Platform-as-a-Service

QMachine is easy to deploy using Platform-as-a-service (PaaS) because its design was driven by the goal to be as far "above the metal" as possible.

7.1 One-click deployment to Heroku

In fact, QM is so far above the metal that it can be deployed to Heroku with a single click. If you're reading this in a digital format like HTML or PDF, you can do it without even leaving this page.

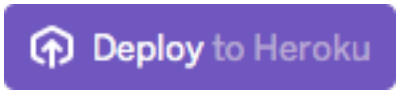

Fig. 1: To deploy your own turnkey system, click the button. Seriously, try it.

It's okay if you're leery of clicking things, if your computer's security settings blocked the coolness, or if you're using a hard copy of the manual. The idea here appears again and again in QM: a workflow can be launched simply by loading a URL. In this case, your click sends a message to Heroku to create a new app from a template in a version-controlled repository, https://github.com/qmachine/qm-ruby-turnkey. This template contains the "blueprint" for a turnkey QM system, complete with an API server, a web server, and a barebones webpage that loads the browser client. It uses the *Ruby version* (page 9) of QM for simplicity and the Heroku Button for convenience.

7.2 Other platforms

Full directions for platforms such as Bluemix, Heroku, and OpenShift are forthcoming.

CHAPTER 8

Source code

There are a number of repositories associated with QMachine, and they are all being transferred to the QMachine organization on GitHub.

8.1 Browser client

The source code for the *browser client* (page 5) is available at https://github.com/qmachine/qm-browser-client.

8.2 Main project

The main project repository is the best place to start with development. It is structured as a superproject, which means that cloning it requires cloning its submodules, too:

```
$ git clone --recursive https://github.com/qmachine/qmachine.git
```

There are also "mirrors" available at Bitbucket, CodePlex, GitLab, and SourceForge. The repositories at Gitorious and Google Code will no longer be updated because those hosting services are being shut down.

8.3 Node.js

The source code for the *Node.js module* (page 7) is available at https://github.com/qmachine/qm-nodejs.

8.4 Python

The source code for a *very* incomplete Python package is available at https://github.com/qmachine/qmachine-python.

8.5 R

The source code for an experimental R package is available at https://github.com/qmachine/qm-r.

8.6 Ruby

The source code for the *Ruby gem* (page 9) is available at https://github.com/qmachine/qm-ruby.