Trombone Documentation

Release 0.9

Johannes Hilden

March 25, 2015

Contents

CHAPTER 1

Project home page

- github.com/johanneshilden/trombone

CHAPTER 2

Contents

2.1 Introduction

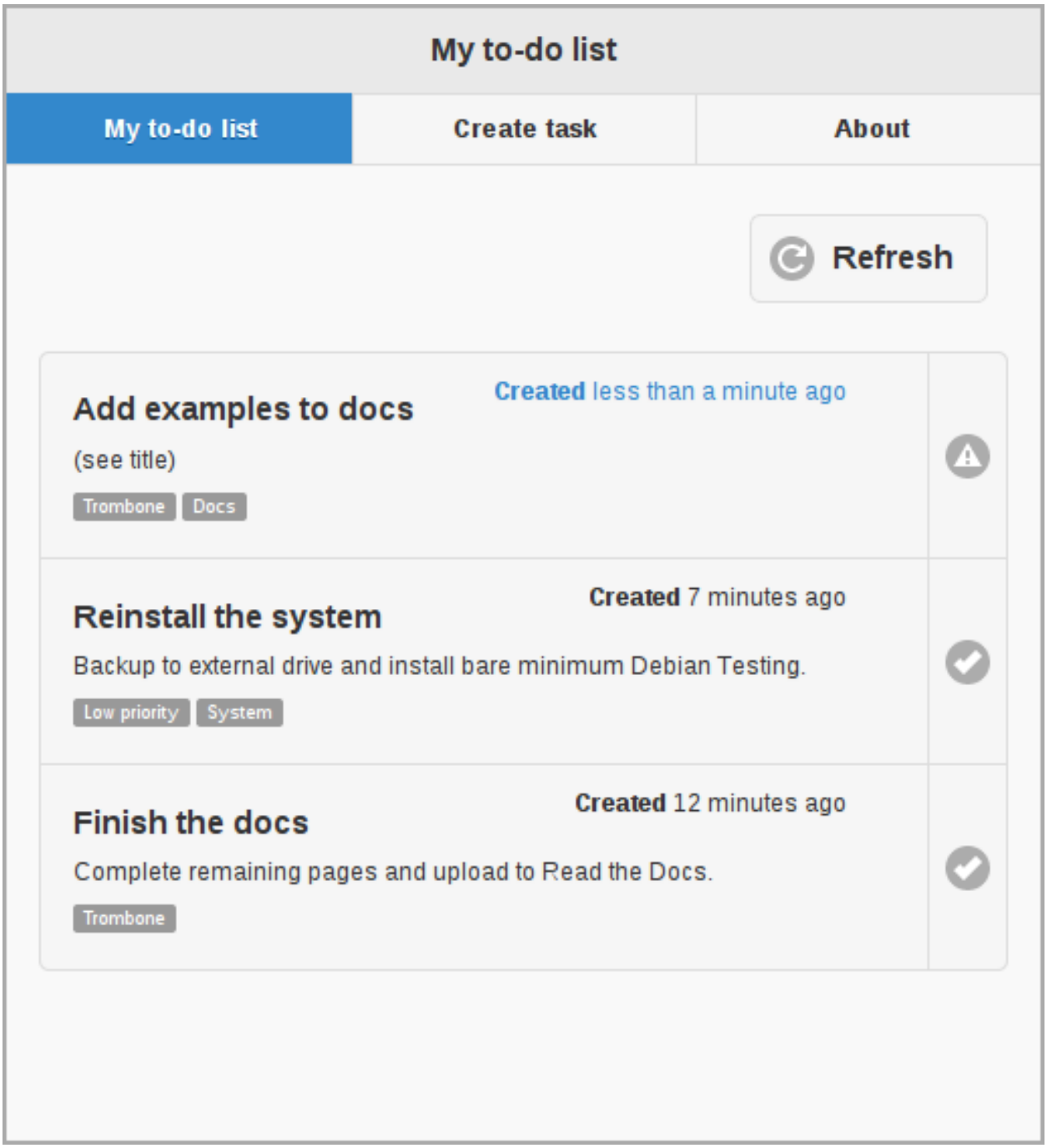
Trombone facilitates effortless adaptation of conventional SQL schemas to mobile-friendly APIs operating within the RESTful web service paradigm. It uses PostgreSQL as underlying RDBMS and translates JSON-formatted requests to database statements, according to rules layed out by a set of route templates, such as the one below.

```
GET resource/:id   ->   SELECT * FROM stuff WHERE id = {{:id}}
```

This format is described in more detail here.

2.1.1 Hello, World!

@todo



SQL

```
CREATE TABLE tasks (
    id          serial PRIMARY KEY,
    title       character varying(255)       NOT NULL,
    description text                         NOT NULL,
    created     timestamp with time zone     NOT NULL
);

CREATE TABLE tags (
```

```
    id          serial PRIMARY KEY,
    title       character varying(255)       NOT NULL
);

CREATE TABLE tasks_tags (
    id          serial PRIMARY KEY,
    task_id     integer                      NOT NULL,
    tag_id      integer                      NOT NULL,
    UNIQUE (task_id, tag_id)
);

INSERT INTO tags (title) VALUES ('Work'), ('Studies'), ('Pastime');
```

Configuration file

Method	Uri Request	JSON Format	Response
GET	/task	Retrieve all tasks, most recent first.	
POST	/task	Create a new task.	
DELETE	/task/:id	Delete a task.	
PUT	/task/:id	Update a task.	
GET	/tag	Load all tags.	
POST	/tag	Insert a tag.	
POST	/tag/task	Establish a task-tag association.	
POST	/!tag/task	Load task-tag associations for a collection of tasks.	

```
# Retrieve all tasks, most recent first
GET    /task  >>  SELECT id, title, description, created FROM tasks ORDER BY created DESC

# Create a new task
POST   /task  <>

    INSERT INTO tasks
      ( created
      , title
      , description )
    VALUES
      ( now()
      , {{title}}
      , {{description}} )

# Delete a task
DELETE /task/:id  --  DELETE FROM tasks WHERE id = {{:id}}

# Update a task
PUT    /task/:id  ><
```

```
    UPDATE tasks
       SET title       = {{title}}
         , description = {{description}}
    WHERE
      id = {{:id}}

# Load all tags
GET    /tag  >>  SELECT id, title FROM tags

# Insert a tag
POST   /tag  <>  INSERT INTO tags (title) VALUES ({{title}})

# Establish a task-tag association
POST   /tag/task  --

    INSERT INTO tasks_tags
      ( task_id
      , tag_id )
    VALUES
      ( {{taskId}}
      , {{tagId}} )

# Load task-tag associations for a collection of tasks
POST   /!tag/task  >>  SELECT id, task_id, tag_id FROM tasks_tags WHERE task_id IN ( {{ids}} )
```

2.2 Installation

2.2.1 Build Prerequisites

To build Trombone you need

- the PostgreSQL database server,
- pg_config (comes with postgresql-devel, or libpq-dev on Debian),
- the Cabal tool – a build system for Haskell programs, and
- a recent version of GHC (The Glasgow Haskell Compiler).

Cabal and GHC come bundled with the Haskell Platform, which is the recommended installation strategy unless you have more specific requirements. The Haskell Platform is available for all major operating systems.

Getting the Haskell Platform

Consult the search utility provided by your distribution's package manager to locate a suitable candidate, or follow the instructions on the download page relevant to your operating system.

Building Trombone

Once you have GHC and Cabal installed, run the command

```
$ cabal update
```

to download the most recent list of packages. Next, clone the repository,

```
$ git clone https://github.com/johanneshilden/trombone.git
```

and run the below sequence of commands. (The use of a sandbox here is optional, but recommended to avoid dependency problems.)

```
$ cd trombone
$ cabal sandbox init
$ cabal install --only-dependencies
$ cabal build

dist/build/trombone/trombone
```

2.2.2 Troubleshooting

Report bugs and other issues to github.com/johanneshilden/trombone/issues.

2.3 Basic Configuration

2.3.1 Running

To start the service on port 3010 (default) with the configuration file `my.conf`, connecting to `my_database`, run the following command:

```
$ trombone -d my_database -r my.conf
```

Some commonly used flags are:

`-C`	Enable CORS support.
`-r FILE`	Specify a (route) configuration file.
`--verbose`	Use verbose output.
`-x`	Disable HMAC authentication (for dev. environments).
`-t`	Bypass authentication for localhost.

For a complete list of flags and switches, see Command Line Flags, or give the command `trombone --help`.

Ping

To send a ping request to the server, we may then use a command line tool like `curl`:

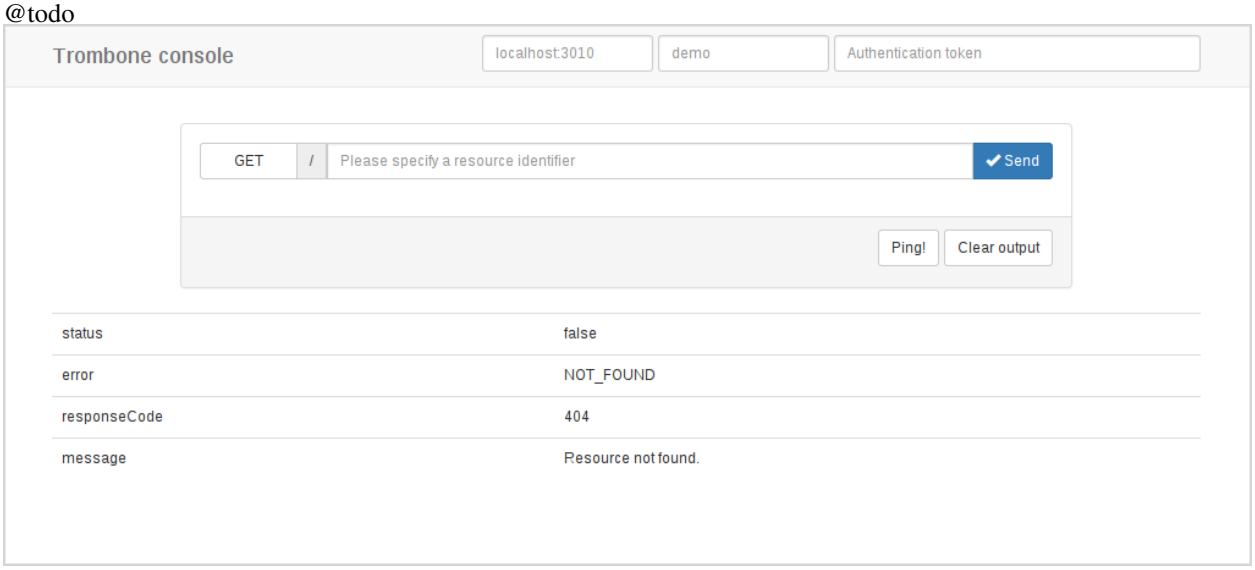
```
$ curl localhost:3010/ping
```

A typical response (if the service is running):

```
< HTTP/1.1 200
< Transfer-Encoding: chunked
< Content-Type: application/json; charset=utf-8
< Server: Trombone/0.8
{
   "status":true,
   "message":"Pong!"
}
```

Console app

@todo



2.3.2 Unix signal handlers

Trombone responds to `SIGHUP` by restarting the service, which causes configuration data to be reloaded. The `SIGTERM` handler completes all pending requests and thereafter shuts down the server.

Example

To send a `SIGHUP`:

```
kill -SIGHUP `ps -a | awk '/trombone/ {print $1}'`
```

2.3.3 Configuration data storage

The server will look for a database table called `trombone_config` in the event that a configuration file is not specified (i.e., the `-r` flag is omitted). This comes in useful if you cannot rely on persistent disk storage (e.g. on ephemeral file systems), or simply prefer to keep configuration data in the database.

```
CREATE TABLE IF NOT EXISTS trombone_config (
    id   serial                PRIMARY KEY,
    key  character varying(40) UNIQUE        NOT NULL,
    val  text                                NOT NULL
);
```

Note: This table is automatically created when the server starts, unless it already exists.

2.4 Route Format

A Trombone configuration file consists of a collection of route patterns. The format of a single route item is given by the following (high-level) grammar.

```
<route> ::= <method> <uri> <symbol> <action>
```

For a more detailed description of the syntactic rules involved in this route schema, please see BNF grammar. What we consider here is a more general overview.

As an example of a simple configuration file:

```
# Return all customers
GET  /customer      >>  SELECT * FROM customers

# Return a single customer, or a 404 error
GET  /customer/:id  ->  SELECT * FROM customers WHERE id = {{:id}}

# Create a new customer
POST /customer      <>

    INSERT INTO customers
        ( name
        , phone
        , industry )
    VALUES
        ( {{name}}
        , {{phone}}
        , {{industry}} )
```

The server scans the list of routes during dispatch, carefully looking for a pattern that matches the uri components and HTTP method used in the request.

The arrow symbol specifies the type of route and the response object's expected format. See below for explanations of these symbols. E.g., the particular arrow used here (`->`) denotes an SQL query with a singleton result.

2.4.1 Placeholders

Placeholders are denoted by a double pair of surrounding curly-braces (akin to e.g., Handlebars.js). Trombone templates acknowledge three types of placeholder variables:

- JSON value `{{placeholders}}`;
- Uri segment `{{:variables}}`; and
- DRY-block placeholders `{{..}}`.

Request body JSON values

When a JSON-formatted request body is present, the dispatch handler will first try to parse the object and substitute placeholders in the template with values whose keys correspond to the names of the variables under consideration.

Route configuration:

```
POST /customer  <>  INSERT INTO customer (name, address, phone)
                    VALUES ( {{name}}, {{address}}, {{phone}} )
```

Request object:

```
{
    "name": "OCP",
    "address": "Delta City",
    "phone": "555-MEGACORP"
}
```

Actual SQL query:

```
INSERT INTO customer (name, address, phone)
VALUES ('OCP', 'Delta City', '555-MEGACORP')
```

Note: Use the `--verbose` command-line option to inspect the final query string after a template is instantiated.

Uri variables

Uri variables are simple placeholders that may conceal text or integer values, supplied as part of the request uri.

```
GET customer/:id   ->   SELECT * FROM customer WHERE id = {{:id}}
```

Notice that the variable appears both in the query template (right-hand side of the arrow), and in the route's uri pattern, where it is bound to a specific path segment. The variable name must consist of only alphanumeric characters, hyphens and underscores. Furthermore, it is always prefixed with a single colon to make the distinction clear from ordinary request body placeholders.

DRY-block placeholders

DRY-block notation is explained under DRY-block Notation.

2.4.2 Comments

Comments start with a single octothorpe (#) character and may appear at the end of a route definition;

```
GET photo       >>  SELECT * FROM photo   # Retreive all photos!
```

or stretch over an entire line;

```
# Return some specific photo.
GET photo/:id   ->  SELECT * FROM photo WHERE id = {{:id}}
```

2.4.3 Multi-line expressions

SQL routes are allowed to span across multiple lines, as long as each subsequent, non-empty line is indented with, at least, one blank space; as in the example below.

```
GET resource  >>

      select name,
             address,
             phone,
             shoe_size
      from customer
      order by id
```

This, however, is *not* valid:

```
GET resource  >>

select name,
       address,
       phone,
       shoe_size
from customer
order by id
```

Except from this "single-space" requirement, indentation does not matter. Hence, the following is a valid route description.

```
GET resource  >>  select name
                       , address
                       , phone
                       , shoe_size
                  from customer
                  order by
                    id
```

2.4.4 Types of Routes

Database routes

Symbol	Explanation
`--`	An SQL statement that does not return any result.
`>>`	A query of a type that returns a collection.
`~>`	A query that returns a single item.
`->`	Identical to `~>` except that an 'Ok' status message is added to the JSON response.
`<>`	An `INSERT` statement that should return a 'last insert id'.
`><`	A statement that returns a row count result (e.g. `UPDATE`).

Other routes

The following, additional route formats all share the common trait that they do not interact directly with the database.

Symbol	Explanation
`||`	A request pipeline. (Followed by a pipeline identifier.)
`|>`	An inline request pipeline. (Followed by a pipeline definition.)
`<js>`	A node.js route. (Followed by a file path to the script.)
`{..}`	A static route. (Followed by a JSON object.)

These are explained here.

2.4.5 Parameter hints

With joins, and more complex queries, the server can have a difficult time figuring out the attribute names to return, from looking at the template alone. In such cases, and in situations where more control is needed, it is therefore possible (and necessary) to specify the list of property names. This list should appear immediately before the query template, enclosed in parentheses.

```
GET /customer  >>

    (id, name, phone)
```

```
SELECT a.a, a.b, a.c
FROM customer
  AS a
JOIN something
  AS b...
```

A similar syntax is available for `INSERT` statements, which can be used if the server is unable to infer the table name and sequence necessary to obtain the last inserted id.

```
POST /customer  <>  (tbl_name, sequence) INSERT INTO...
```

2.4.6 Special Considerations

SELECT * FROM

`SELECT * FROM`-type of queries are accepted as a convenient shorthand. The server will attempt to expand the column names during preprocessing of the configuration file. However, this is not guaranteed to work. In some cases you will have to explicitly write out the column names, e.g., `SELECT id, name, favorite_cheese FROM...`.

Wildcard operators

Since string values are automatically wrapped in single quoutes before they are inserted into a template, the following will not work as expected,

```
SELECT * FROM customer WHERE customer.name LIKE '%{{q}}%'
```

E.g., `{"q": "ACME"}` would translate to `customer.name LIKE '%'ACME'%'`.

This is clearly not what we intended. Instead, define your template as

```
SELECT * FROM customer WHERE customer.name LIKE {{q}}
```

and insert the `%`-characters inside the string property of the object sent to the server:

```
{
   "q": "%ACME%"
}
```

2.4.7 DRY-block Notation

A common pattern is to have multiple database queries that are similar in one way or another.

```
GET customer/all        >>
   select id, name, phone, address from customer order by id

GET customer/:id        ->
   select id, name, phone, address from customer where id = {{:id}}

GET customer/area/:id   >>
   select id, name, phone, address from customer where area_id = {{:id}} order by id
```

To avoid repetition, an alternative DRY notation can be employed in cases such as this. The following is an equivalent route definition using a DRY-block.

```
DRY
     select id, name, phone, address from customer {{..}}      # base template
{
     GET customer/all       >>  order by id                          ;
     GET customer/:id       ->  where id = {{:id}}                   ;
     GET customer/area/:id  >>  where area_id = {{:id}} order by id
}
```

A DRY-block consists of a *base template* and a number of *stubs*, each with the segment of the statement unique to its corresponding route.

```
<method> <uri> <symbol> <stub>
```

Here are some important observations.

- The `{{..}}`-placeholder must appear in the base query to indicate where the stub should be inserted. The preprocessor looks at each item within the block, expands it by inserting the base query with the stub replaced for `{{..}}`.
- A semi-colon delimiter is required to separate the stubs within the block. (It may be omitted for the last item.)
- Each block item must be indented with at least one blank space. The opening and closing brackets should appear on their own lines (without indentation):

```
{
    GET /..
    GET /..
}
```

2.5 Non-SQL Routes

This is an overview of the various route types that are not interacting directly with the database.

Symbol	Explanation
`||`	A request pipeline. (Followed by a pipeline identifier.)
`|>`	An inline request pipeline. (Followed by a pipeline definition.)
`<js>`	A node.js route. (Followed by a file path to the script.)
`{..}`	A static route. (Followed by a JSON object.)

2.5.1 Pipelines

Pipelines offer a simple, declarative syntax for composition of routes using ordinary JSON objects.

Pipelines can be declared in two different ways; either in a separate file or as inline definitions.

Pipeline configuration file

Inline pipeline syntax

Basic Format

Structure of a pipeline

```
GET  /my-pipeline  |>
{
    "processors": [
    ],
    "connections": [
    ]
}
```

Processors

@todo

Connections

@todo

Filters

@todo

Equal-to

@todo

Not-equal-to

@todo

Greater-than

@todo

Greater-than-or-equal

@todo

Less-than

@todo

Less-than-or-equal

@todo

Transformers

@todo

Exclude

@todo

Include

@todo

Bind

@todo

Rename

@todo

Copy

@todo

Aggregate

@todo

2.5.2 node.js

http://nodejs.org/

Example 1.

```
GET /stuff  <js>  node/demo1.js
```

```
// node/demo1.js

var response = {
    statusCode : 200,
    body       : 'Just saying "hello".'
};
```

```
console.log(JSON.stringify(response));
```

Example 2.

```
POST /oracle <js>  node/demo2.js

// node/demo2.js

var fs = require('fs');

function parseStdin() {
    var data = fs.readFileSync('/dev/stdin').toString();
    if (data) {
        return JSON.parse(data);
    } else {
        return null;
    }
};

// Parse request object
var obj = parseStdin();

// Do some heavy computation
obj.string = obj.string.replace(/\%1/, '42');

// Send response
var response = {
    statusCode : 200,
    body       : obj
};

console.log(JSON.stringify(response));

$ curl http://localhost:3010/oracle -d '{"string": "The answer is %1."}'
The answer is 42.
```

2.5.3 Static Objects

The `{..}` syntax enables for static JSON response objects to be embedded directly in the route description.

```
GET /stuff  {..}  {"status":"Ok.","response":[1,2,3,4]}
```

A possible use-case for this is to deliver machine readable documentation as part of a service (self-describing APIs), where participants automatically can determine their abilities against a communication endpoint using the `OPTIONS` HTTP method. See, e.g., http://zacstewart.com/2012/04/14/http-options-method.html for a discussion of this approach.

> *At the very least, services should be responding with a 200 and the Allow header. That's just correct web server behavior. But there's really no excuse for JSON APIs not to be returning a documentation object.*

```
OPTIONS /photo  {..}  {"GET":{"description":"Retreive a list of all photos."},
                       "POST":{"description":"Create a new photo."}}
```

The rationale for the `OPTIONS` method is outlined in RFC 2616, Section 9.2.

> *The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI. This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval.*

Special <Allow> keyword

Static JSON response routes support a special `<Allow>` keyword, the primary intent of which is to support the interaction pattern described above.

```
OPTIONS /photo  {..}  {"<Allow>":"GET,POST,OPTIONS",
            "GET":{"description":"Retreive a list of all photos."},
            "POST":{"description":"Create a new photo."}}
```

A typical response would then be:

```
< HTTP/1.1 200
< Allow: 'GET,POST,OPTIONS'
< Content-Type: application/json; charset=utf-8
{"GET":{"description":"Retreive a list of all customers."},
 "POST":{"description":"Create a new customer."}}
```

2.6 Usage Patterns & Conventions

2.6.1 Naming

Trombone makes two fairly idiomatic assumptions; namely that,

- database tables and columns follow the `lowercase_separated_by_underscores` naming convention, and that
- JSON objects use `camelCase` formatting.

Conversion between these two formats is usually implicit.

2.6.2 Array Actions

```
curl http://localhost:3010 --verbose -d '[{}, {}]'
```

```
curl http://localhost:3010 --verbose -d '[{"summary":"","name":""}, {"summary":"","name":""}, {"summa
```

```
var obj = [
    {
        name: 'Object #1',
        summary: '...'
    },
    {
        name: 'Object #2',
        summary: '...'
    },
    {
        name: 'Object #3',
        summary: '...'
    }
```

```
];

Trombone.request({
    host     : 'http://localhost:3010',
    client   : 'demo',
    key      : 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
    type     : 'POST',
    resource : 'util',
    data     : obj,
    nonce    : Date.now()/10 | 0,
    success  : function() { alert('Ok.'); }
});
```

2.6.3 Response Codes

Code	Title	Explanation
200	Ok	A normal response.
202	Accepted	This response code indicates that the result is a collection (array). That is, each individual response item must be considered separately and no claim is made as to the state of success w.r.t. these. See Array Actions.
400	Bad Request	The request contains malformed JSON or is otherwise invalid.
401	Unauthorized	HMAC authentication failed.
404	Not Found	No route matches the request, or the record doesn't exist for the route. E.g., a `SELECT * FROM tbl WHERE id = {{id}}` query returning an empty result.
500	Internal Server Error	An error occured during processing of the request. Refer to the attached error code for details.
503	Service Unavailable	The server is shutting down or restarting.

2.6.4 Error Codes

@todo

```
{
    "status"       : false,
    "error"        : "NOT_FOUND",
    "responseCode" : 404,
    "message"      : "Resource not found."
}
```

Code	Comment
BAD_REQUEST	
NOT_FOUND	
UNAUTHORIZED	
CONFLICT	
SQL_FOREIGN_KEY_CONSTRAINT_VIOLATION	
SQL_UNIQUE_CONSTRAINT_VIOLATION	
SQL_ERROR	
SERVER_CONFIGURATION_ERROR	
SERVICE_UNAVAILABLE	
INTERNAL_SERVER_ERROR	

2.6.5 Arrays

@todo

```
POST   /tag/task  >>

    SELECT * FROM tasks_tags WHERE task_id IN ( {{ids}} )
```

(Note the brackets.)

```
{
  "ids": [1,2,3,4,5,6,7,39]
}
```

2.6.6 Notes about HTTP Methods

@todo

GET

POST

PUT

DELETE

Idempotency in a nutshell

OPTIONS

PATCH

2.7 Authentication

2.7.1 Security model

To establish the authenticity of a request, the server must perform a message integrity check, operating on a cryptographic primitive known as a HMAC (hash-based message authentication code). A MAC is attached to each request, in the form of an `API-Access` header. During dispatch, a subsequent code is computed from the request object using

- a token (secure key) associated with the client application,
- an incremental nonce (see below), and
- the request method together with the path info.

The result of this operation is compared with the original MAC attached to the request, in order to verify its authenticity.

A valid key is a random, 40-character long, hexadecimal string.

```
53d5864520d65aa0364a52ddbb116ca78e0df8dc
```

Table schema

The `trombone_keys` table maintains client-key associations.

```
CREATE TABLE trombone_keys (
    id serial,
    client character varying(40) NOT NULL,
    key character varying(40) NOT NULL,
    nonce bigint NOT NULL
);

ALTER TABLE ONLY trombone_keys
    ADD CONSTRAINT trombone_keys PRIMARY KEY (id);

ALTER TABLE ONLY trombone_keys
    ADD CONSTRAINT unique_trombone_keys_client UNIQUE (client);
```

Note: This table is automatically created when the server starts with authentication enabled (i.e., in default mode), unless it already exists.

Authenticating client applications

In order for a client application to be granted access to the service, it must;

1. be present in the `trombone_keys` table with a unique identifier and its secure token; as well as
2. supply the following HTTP header with each request:

```
API-Access: <client_id>:<nonce>:<hash>
```

where `<client_id>` is replaced with the name of the application (as it appears in the `trombone_keys` table), and `<hash>` with the MAC code obtained by hashing a concatenated string – the constituents of which are given below, using the HMAC-SHA1 algorithm and aforementioned key.

The `<nonce>` is an integer value introduced to prevent an adversary from reusing a hash under a, so called, replay attack. The client implementation must therefore ensure that the nonce is strictly increasing for each request. This can be achieved using a timestamp, such as the one used in the reference implementation.

Hash string format

The format of the string given as input to the hashing algorithm must be as follows:

```
<client_id>:<method>:<uri>:<nonce>:<json_body>
```

SHA1 implementations are available for many programming languages. The following have been tested with Trombone:

JavaScript	https://code.google.com/p/crypto-js/
Haskell	http://hackage.haskell.org/package/Crypto/docs/Data-HMAC.html

For complete, working examples, see Reference Implementations.

Client key administration

Trombone includes the `keyman` utility, which can be used for command line administration of client keys.

See Tools & Utilities.

Disable HMAC authentication

Message authentication can be disabled with the `-x` command line switch. Doing so in a production setting is not recommended.

> **Warning:** Deactivating message authentication gives everyone access to your server interface. To mitigate the risk of unauthorized access to production data, only use the `-x` flag in a safe environment.

Allowing access from localhost

To bypass HMAC authentication specifically for requests originating from the local host, instead use the `-t`, or `--trust-localhost` option.

2.7.2 Reference Implementations

```
CREATE DATABASE basic_auth_demo;

\c basic_auth_demo

CREATE TABLE IF NOT EXISTS utilities (
    id          serial PRIMARY KEY,
    name        character varying(255)        NOT NULL,
    summary     character varying(255)        NOT NULL
);

INSERT INTO utilities (name, summary) VALUES
    ('ls',   'list directory contents'),
    ('htop', 'interactive process viewer'),
    ('df',   'report file system disk usage'),
    ('pwd',  'print name of current/working directory'),
    ('awk',  'pattern scanning and text processing language');

CREATE TABLE IF NOT EXISTS trombone_config (
    id          serial PRIMARY KEY,
    key         character varying(40) UNIQUE NOT NULL,
    val         text                         NOT NULL
);

INSERT INTO trombone_config (key, val) VALUES
    ('routes', E'GET /utils >> SELECT * FROM utilities\nPOST /util <> INSERT INTO utilities (name, su
```

Create a file `basic-keyman.conf`:

```
host     = 'localhost'
port     =  5432
dbname   = 'basic_auth_demo'
user     = 'postgres'
password = 'postgres'
```

(Modify the file as required.)

```
$ ./keyman register demo -c basic-keyman.conf

Client registered:
demo: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Start the server

```
$ trombone -d basic_auth_demo -C
```

JavaScript

Insert the generated `demo` key on line 15.

```
1  // auth-example.js
2
3  $(document).ready(function() {
4
5      var render = function(obj) {
6          $('#response').html('<pre>' + JSON.stringify(obj, null, 4) + '</pre>');
7      };
```

```

    var onError = function(e) {
        render(JSON.parse(e.responseText));
    };

    var defaults = {
        host     : 'http://localhost:3010',
        key      : 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
        client   : 'demo',
        type     : 'GET',
        error    : onError
    };

    $('#insert-action').click(function() {

        var name    = $('#insert-title').val(),
            summary = $('#insert-description').val();

        if (!summary || !name) {
            $('#response').html('Please fill out both fields.');
            return;
        }

        var obj = {
            summary : summary,
            name    : name
        };

        Trombone.request($.extend({}, defaults, {
            data     : obj,
            nonce    : Date.now()/10 | 0,
            type     : 'POST',
            resource : 'util',
            success  : function() {
                $('#response').html('Ok.');
            }
        }));

    });

    $('#request-action').click(function() {

        Trombone.request($.extend({}, defaults, {
            nonce    : Date.now()/10 | 0,
            resource : 'utils',
            success  : render
        }));

    });
});

<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Trombone data access service example: Request authentication</title>
    </head>
    <body>
```

```
        <div>
            <a id="request-action" href="javascript:">Request some data</a>
        </div>
        <div>
            <div><input id="insert-title" type="text"></div>
            <div><textarea id="insert-description"></textarea></div>
            <div><a id="insert-action" href="javascript:">Insert some data</a></div>
        </div>
        <div id="response"></div>

        <script src="http://code.jquery.com/jquery-2.1.1.min.js"></script>
        <script src="http://crypto-js.googlecode.com/svn/tags/3.1.2/build/rollups/aes.js"></script>
        <script src="http://crypto-js.googlecode.com/svn/tags/3.1.2/build/rollups/hmac-sha1.js"></sc
        <script src="js/trombone.request.min.js"></script>
        <script src="js/auth-example.js"></script>
    </body>
</html>
```

Haskell

@todo

Purescript

@todo

C++/Qt

@todo

2.8 Command Line Flags

Flag	Long option	Description
-V	--version	Display version number and exit
-?	--help	Display this help and exit
-x	--disable-hmac	Disable message integrity authentication (HMAC)
-C	--cors	Enable support for cross-origin resource sharing
-A[USER:PASS]	--amqp[=USER:PASS]	Enable RabbitMQ messaging middleware [username:password]
	--amqp-host=HOST	RabbitMQ host [host]
-i[FILE]	--pipelines[=FILE]	Read request pipelines from external file [config. file]
-s PORT	--port=PORT	server port
-l[FILE]	--access-log[=FILE]	Enable logging to file [log file]
	--colors	Use colors in log output
	--size=SIZE	log file size
-h HOST	--db-host=HOST	database host
-d DB	--db-name=DB	database name
-u USER	--db-user=USER	database user
-p PASS	--db-password=PASS	database password
-P PORT	--db-port=PORT	database port
-r FILE	--routes-file=FILE	route pattern configuration file
-t	--trust-localhost	Bypass HMAC authentication for requests from localhost
	--pool-size=SIZE	Number of connections to keep in PostgreSQL connection pool
	--verbose	Print various debug information to stdout

2.8.1 Defaults

Many of these settings have sensible default values.

Option	Value
AMQP user	"guest"
AMQP password	"guest"
Server port	3010
Log file	"log/access.log"
Log size	4,096 bytes
DB-host	"localhost"
DB-name	"trombone"
DB-user	"postgres"
DB-password	"postgres"
DB-port	5432
Pipelines file	"pipelines.conf"
Pool size	10

2.9 Middleware

Middlewares are built-in, auxiliary software components adaptable to suit specific needs. These components are normally disabled (with the exception of file serving) and must be enabled at run-time. See respective section for details on how to activate and configure a component.

2.9.1 Available Components

- RabbitMQ
- CORS (cross-origin resource sharing)
- Logging
- Static File Serving

2.9.2 RabbitMQ

RabbitMQ is a a messaging system based on the Advanced Message Queuing Protocol – an emerging standard for multi-purpose, asynchronous message delivery. The AMQP middleware integrates Trombone with RabbitMQ and makes it possible for third-party applications to receive notifications when server resources are modified.

Flags
Enable with `--amqp[=USER:PASS]` or `-A` and, optionally, supply a host name using `--amqp-host[=HOST]` (if you leave out this option, `localhost` is assumed).

AMQP Endpoint

When a request of type `POST`, `PUT`, `DELETE`, or `PATCH` is accepted and produces a regular `200 OK` response, a subsequent message is published to an exchange managed by the server.

Trombone AMQP Exchange

Name	`exchange/trombone/api`
Type	`fanout`

Messages follow the format `<method> <uri>:<response-body>`; e.g.,

```
POST customer/new:{"status":true,"id":49,"message":"Ok."}
```

Using AMQP in JavaScript applications

To configure and run RabbitMQ with STOMP Over WebSocket enabled, follow these instructions to install the Web-Stomp plugin.

STOMP is a simple text-orientated messaging protocol. It defines an interoperable wire format so that any of the available STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among languages and platforms.

For more information on STOMP Over WebSocket, see http://jmesnil.net/stomp-websocket/doc/.

JavaScript Example

For this example, you need stomp.js, and sock.js.

- http://jmesnil.net/stomp-websocket/doc/#download
- http://cdn.sockjs.org/sockjs-0.3.min.js

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Trombone/RabbitMQ over STOMP</title>
    </head>
    <body>

        <div id="notification"></div>

        <script type="text/javascript" src="js/sockjs.min.js"></script>
        <script type="text/javascript" src="js/stomp.min.js"></script>
        <script type="text/javascript">

            // See: http://www.rabbitmq.com/web-stomp.html
            var ws = new SockJS('http://127.0.0.1:55674/stomp'),
                client = Stomp.over(ws);

            // Heartbeats won't work with SockJS.
            client.heartbeat.outgoing = 0;
            client.heartbeat.incoming = 0;

            var onConnect = function() {
                client.subscribe('/exchange/trombone/api', function(msg) {
                    var div = document.getElementById('notification');
                    div.innerHTML += msg.body + '<br>';
```

```
            });
        };

        var onError = function() {
            console.log('Error connecting to RabbitMQ server.');
        };

        client.connect('guest', 'guest', onConnect, onError, '/');

    </script>
  </body>
</html>
```

2.9.3 CORS

The CORS component provisions Trombone with the ability to accept cross-domain requests. It implements the handshake and response headers mandated by CORS-compliant client applications, such as modern web browsers.

CORS introduces a standard mechanism that can be used by all browsers for implementing cross-domain requests. The spec defines a set of headers that allow the browser and server to communicate about which requests are (and are not) allowed. CORS continues the spirit of the open web by bringing API access to all.

Note: CORS involves coordination between both server and client. For more information regarding client requirements, as well as cross-origin resource sharing in general, please see: enable-cors.org.

Flags
Enable using `--cors` or `-C`.

2.9.4 Logging

The logging format is similar to Apache's log file output.

Flags
Enable using `--access-log[=FILE]` or `-l`, and specify `--colors` to enable colors in the log file.

Typical output

@todo

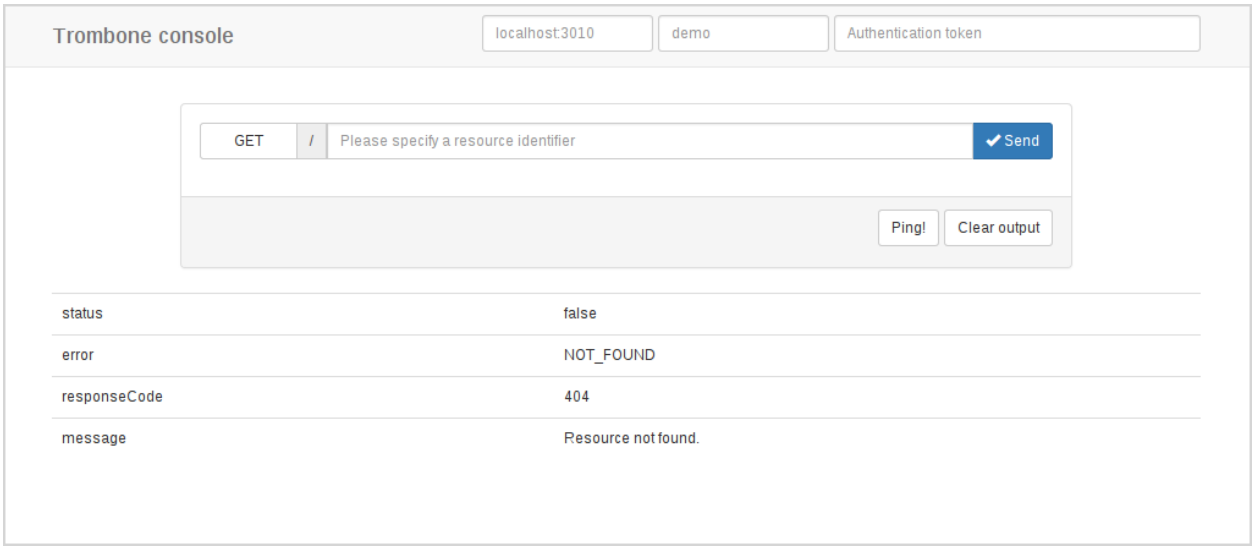
2.9.5 Static File Serving

Trombone can also act as a simple file server. Files located under the `public/` directory or any of its subdirectories are HTTP accessible.

```
public/image.png   <~>   http://localhost:3010/image.png
```

2.10 Tools & Utilities

2.10.1 Console



@todo

2.10.2 Keyman

The `keyman` utility implements a simple CRUD interface, suitable for command line administration of client keys.

```
Usage:
  keyman list [--config=<file>]
  keyman (register|renew) <client> [<key>] [--config=<file>]
  keyman revoke <client> [--config=<file>]
  keyman --help

Options:
  -c --config=<file>  Path to database connection file.
  -? --help           Display this help.
```

The configuration file contains a list of parameters (identical to those described here) used to establish a database connection. Note that the default location for this file is `~/.config/trombone/keyman.conf`.

Sample `keyman.conf` file:

```
host     = 'localhost'
port     =  5432
dbname   = 'trombone'
user     = 'postgres'
password = 'postgres'
```

Keyman usage

To list existing client keys:

```
$ ./keyman list

generic            : 14ad0ef86bf392b38bad6009113c2a5a8a1d993a
batman             : 53d5864520d65aa0364a52ddbb116ca78e0df8dc
spock              : 78a302b6d3e0e37d2e37cf932955781900c46eca
```

Register a new client:

```
$ ./keyman register my_application

Client registered:
my_application: 53d5864520d65aa0364a52ddbb116ca78e0df8dc
```

A token is automatically generated for the new client. Alternatively, an existing key (a 40 character long hexadecimal string) may be specified as an extra, trailing argument, e.g., `keyman register my_application 53d5864520d65aa0364a52ddbb116ca78e0df8dc`. Subsequent to registering the application, we can confirm that it appears in the client list with its new key.

```
$ ./keyman list | grep my_application

my_application      : 53d5864520d65aa0364a52ddbb116ca78e0df8dc
```

To remove a client, use:

```
$ ./keyman revoke unwanted_client
```

2.10.3 JavaScript libraries

```
trombone.request.js
trombone.request.min.js
```

@todo

2.11 BNF Grammar

@todo

```
<route>     ::= <method> <uri> <action>

<method>    ::= "GET" | "POST" | "PUT" | "PATCH" | "DELETE" | "OPTIONS"

<uri>       ::= [ <delim> ] { <item> <delim> }

<delim>     ::= "/"

<item>      ::= <variable> | <atom>

<variable>  ::=

<atom>      ::=

<action>    ::= <sql-route>
              | <pipeline-route>
              | <inline-route>
              | <static-route>
```

```
              | <node-js-route>

<sql-route> ::= <sql-no-result>
              | <sql-item>
              | <sql-item-ok>
              | <sql-collection>
              | <sql-last-insert>
              | <sql-count>

<sql-no-result>   ::= "--"
<sql-item>        ::= "~>"
<sql-item-ok>     ::= "->"
<sql-collection>  ::= ">>"
<sql-last-insert> ::= "<>"
<sql-count>       ::= "><"

<pipeline-route>  ::= "||"

<inline-route>    ::= "|>"

<static-route>    ::= "{..}"

<node-js-route>   ::= "<js>"
```

Note: A full treatment of the SQL syntax is beyond the scope of this document.

2.12 Examples

@todo

2.13 Deployment

2.13.1 Heroku

Trombone has been successfully deployed to Heroku, although the process described here should be considered somehow experimental.

We will use Joe Nelson's (begriffs) excellent buildpack to deploy to Heroku, using git. Various other Haskell buildpacks are available, but none of these have been tested with Trombone.

We assume you have your app running and the Heroku Toolbelt installed on your local machine. The first step, unless you have an existing git repository, is to initialize (`git init`) one and add the project files.

Then create the Heroku app using the `heroku-buildpack-ghc` buildpack.

```
heroku create --stack=cedar --buildpack https://github.com/begriffs/heroku-buildpack-ghc.git
```

Log in to Heroku's web panel and add a PostgreSQL database service to your application.

Now add a `Procfile` to your project. The file should be in the root directory (along with `Main.hs` etc.).

Procfile template

```
web: dist/build/trombone/trombone -C -s $PORT -h <hostname> -d <database> --db-user=<db-user> --db-pa
```

Insert `<hostname>` and other details. The hostname is typically of the format `xxxxxxxxx.amazonaws.com`. These particulars are available from the Heroku web admin after creating the database.

```
git add Procfile
git commit -m 'Added Procfile for Heroku.'
```

We can now go ahead and push to heroku.

```
git push heroku master
```

Note that if you are using a local branch other than master, you should use the command `git push heroku localbranch:master`.

15 minutes of Fail

The build will most likely fail due to Heroku's 15-minute time limit. Follow the instructions here.

@todo

@todo

@todo

2.14 About

Trombone is written in Haskell and available for use under the BSD license.

- Issue Tracker: github.com/johanneshilden/trombone/issues
- Source Code: github.com/johanneshilden/trombone