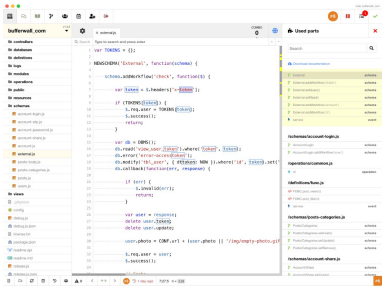
Core framework training

What is Total.js Platform

Ecosystem of related libraries

- Core Framework
- UI components
- DBMS ORM

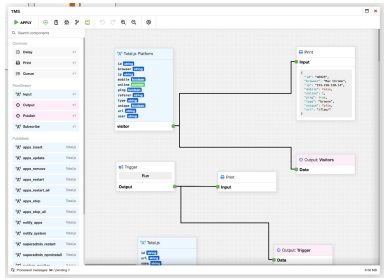


Tools

- Code Editor.
- UI designer
- SuperAdmin
- AppBuilder

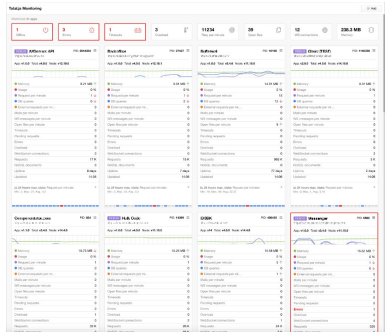
Applications

- OpenPlatform
- Flow + FlowStream
- CMS
- Files
- TMS Integrator



Services

- AppMonitor
- Total API
- Total Cloud
- Support
- Code review



NEW

- OpenSocial
- OpenSync
- OpenDB
- Extensible

What makes Total.js Platform different

Written in pure JS

No 3rd party dependencies

Tiny footprint

High performance

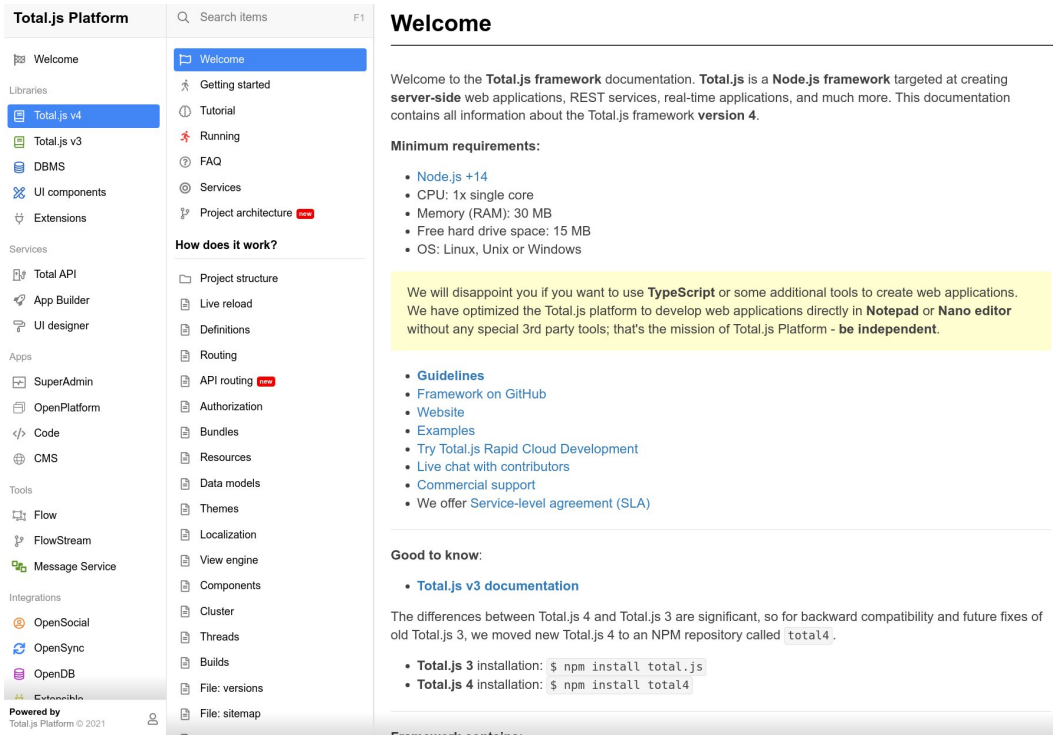
Fully open-sourced, licensed under MIT

No boilerplate code

Provides out-of-the-box all you need to build awesome applications

- Routing
- View-Engine
- NoSQL and Table flat-file databases
- SMTP sender
- WebSocket support
- Workers
- CLI utility
- Extensive documentation
- Application templates

...and more...



First steps with Total.js

Where to find help

- https://www.totaljs.com
- https://docs.totaljs.com/
- https://github.com/totaljs
- https://github.com/totaljs/examples
- Telegram/@totaljs

Prerequisites

- Node.js +14

Installation

```
$ npm install -g total4
```

CLI

```
$ total4 help

$ total4 templates

$ total4 create <template> <path>
```

Minimal application - create a file (e.g. index.js) and add the following line:

```
require('total4');
//do something
const options = {};
require('total4/debug')(options);
//require('total4/release')(options)
```

Running total.js applications

```
$ node index.js

$ node index.js –release

$ node index.js –servicemode
```

Running a simple HTTP server

```
$ total4 PORT
```

Project structure

User created:	
/builds/	files generated by the Application Builder
/bundles/	bundled application components, application parts, whole applications
/components/	reusable components, client side html, css, js, server side js packed together
/controllers/	controller functions
/databases/	TextDB database definitions, helpers
/definitions/	define own/rewrite existing functionality, initialize DBs, etc.
/jsonschemas/	jsonschemas definitions for schemas
/modules/	reusable application modules
/private/	private data like certificates
/public/	publicly accessible files
/resources/	localization files
/schemas/	schemas definitions
/views/	html files for use with the View Engine
/updates/	database updates script runs only one
/workers/	JS code evaluated and executed in separate threads

Realtime created	*not to be modified manually:*
/.src/	extracted and merged bundles
/tmp/	internal framework files
/logs/	logs generated by the framework

GLOBALS

CONF

The way to access the configuration. Stored in the `config` file in the root of the project.

```
CONF.key            //access the config in the code
@{CONF.key}         <!— access in the HTML templates —>
LOADCONFIG({        //load the configuration from an object
      author: 'Peter Sirka',
      name: 'The application',
      database: 'postgres://use:pass@localhost:5432/theapp'
})
```

DEF

Shared definitions defined in the definitions directory.

```
DEF.blacklist['127.0.0.1'] = true; //blacklist definition
DEF.validators.email               //regex validators
DEF.helpers.sum = (a,b)=>a+b       //@{sum(2,3)} in the
                                   //HTML templates
```

PATH

resolves different paths in the app

FUNC

Shared global function definitions

```
FUNC.sum = (a,b)=>a+b          //global function
                               //called as FUNC.sum(2,3)
```

MAIN or *REPO*

Global objects used to store and share the internal state of the application.

```
MAIN.data = {name: 'Peter Sirka'}
REPO.data = {name: 'Peter Sirka'}
```

PREF

Persist and manipulate the user preferences data.

```
PREF.margin                    //get preferences
PREF.set('margin', 10)         //set preferences
```

The data will be eventually saved in the `database/preferences.json` file.

This behavior can be modified on `DEF.onPrefLoad` and `DEF.onPrefSave` callbacks.

TEMP

Temporary data, cleaned up once per ~7 minutes.

CACHE

Cached data storage. Supports automatic expiration.

```
CACHE(key, [value], [expiration], [persistence])
CACHE('key', 12, '1 minute')
CACHE('key')
```

GLOBALS

U (FrameworkUtils)

Huge number of built-in functions

```
U.atob(), U.btoa()                    //base64 encode/decode
U.httpstatus(403)                     //'403: Forbidden'
U.getContentType('gif')               //'image/gif'
U.link('/app1', '&arg=23')            //'/app1/&arg=23'
U.join('dir1', 'dir2')                //'/dir1/dir2'
U.getName('/dir1/dir2/file.ext')      //'file.ext'
U.getExtension('/dir1/dir2/file.ext')//'ext'
U.parseFloat('3.14')                  //3.14
U.parseInt('34')                      //34
U.parseBoolean('12')                  //true
U.parseXML(                         //XML to JS object
`<message>                          //{
  <to>devs</to>                     // message.to:'devs'
  <from>Peter</from>                // message.from:'Peter'
  <heading>Hi!</heading>            // message.heading:'Hi!'
  <body>There</body>                // message.body:'There'
</message>`)                        //}
```

Other

```
UID(), UID16(), GUID([length]) //create ids
AUDIT([name],$,message,[type]) //write log to logs/name.log
CLONE(source, [skip])          //clone object
COUNTER('name')                //create persistent counter
ENCRYPT(value, key, [unique])  //encrypt value
DECRYPT(value, key)            //decrypt value
MAIL(address, subject, name)   //send email (accepts more
                               //arguments)
ON(name, callback)             //register callback on an
                               //event
OFF(name, [callback])          //de-register callback(s)
ONCE(name, callback)           //register callback on event
EMIT(name, [arg])              //emit event
EMIT2(name, [arg])             //emit event to all threads
NOSQL(name)                    //get NOSQL TextDB instance
TABLE(name)                    //get TABLE TextDB instance
```

Prototypes

```
Request/Response
String
Array
Number
Date
```

Built-in databases

Total.js provides built-in TextDB database in two variants. Data is by default stored in the `databases` directory.

NOSQL db

- NoSQL like database stored as serialized JSON
- Usage: `NOSQL('database_name')`

Table db

- SQL like database type
- Usage: `TABLE('database_name')`
- DB schema must be defined in the `config` file

```
table_NAME: id:uid,name:string,email:string(120)
```

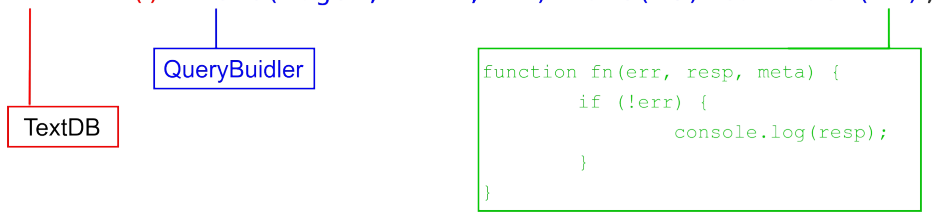
- Allowed data types

```
string, string(MAX_LENGTH)
number
Boolean
Date
object
uid
```

Operations

```
const db = NOSQL('users');

db.find().where('age', '<=', 45).take(10).callback(fn);
```



TextDB:

find, find2, list, insert, remove, count, clean, drop, alter

QueryBuilder:

where, take, skip, sort, paginate, in, notin, or, join, insert, ...

Routing

Routes need to be registered using the `ROUTE` global function which has to be 'installed' by declaring `exports.install` function. There can be more than one exports.install declarations.

Suggested directory for all routes and controllers is `controllers`, for auth and middleware is `definitions`.

```
exports.install = function() {

    ROUTE(...);

}
```

`ROUTE` accepts different arguments depending on the intended usage

View routing

```
+    authorized
-    unauthorized
none all

AUTH(function($){
    //$ - AuthOptions
    return (Math.random() < 0.5)?$.invalid():$.success();
})
```

```
ROUTE('AUTH METHOD RELATIVE_PATH', VIEW);
```

```
GET     /
GET     /about
```

Static file routing

```
const fn = function(req, res)
{
    res.file('/img/1.jpg');
}
```

```
ROUTE('FILE RELATIVE_PATH', HANDLER);
```

```
FILE    /images/*.jpg
FILE    /files/*.*
```

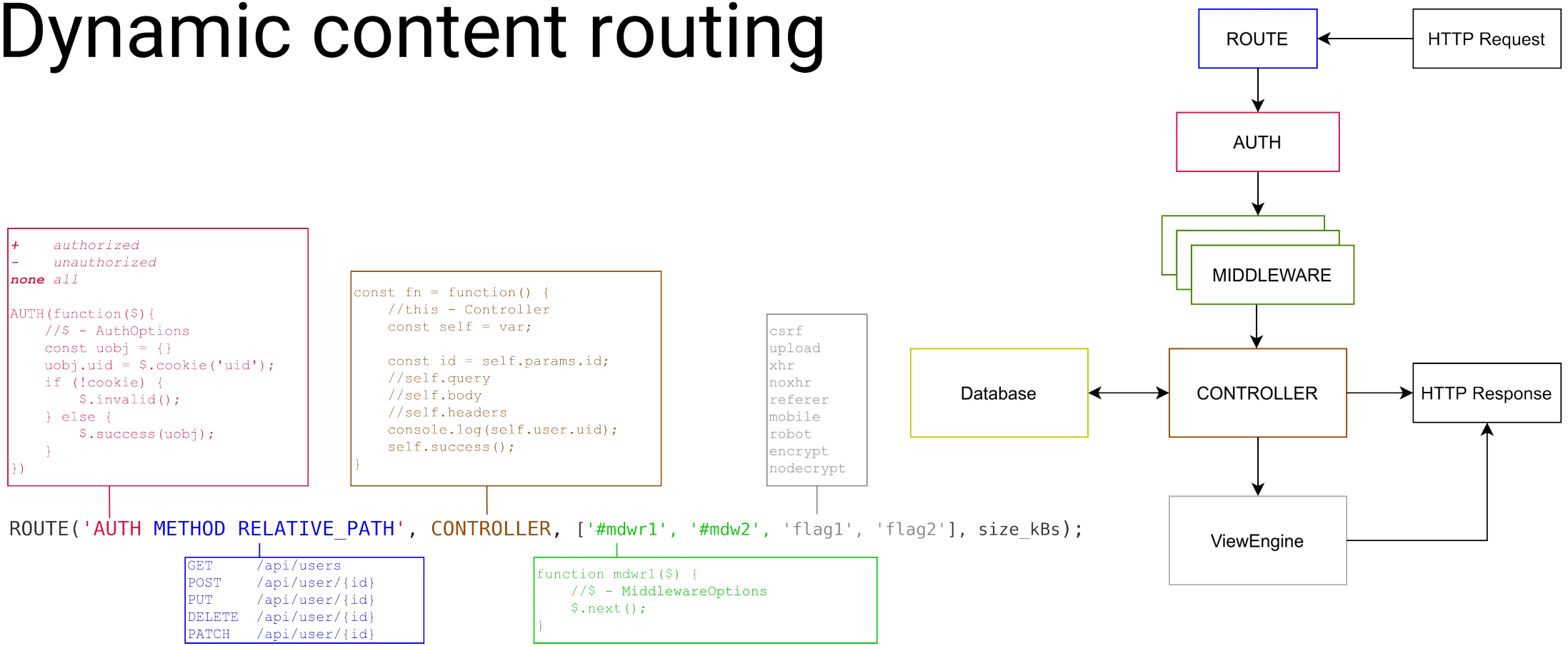
Error routing

```
const fn = function() {
    //this - Controller
    var self = this;
    self.status = 404;
    self.plain('404: Not found');
}
```

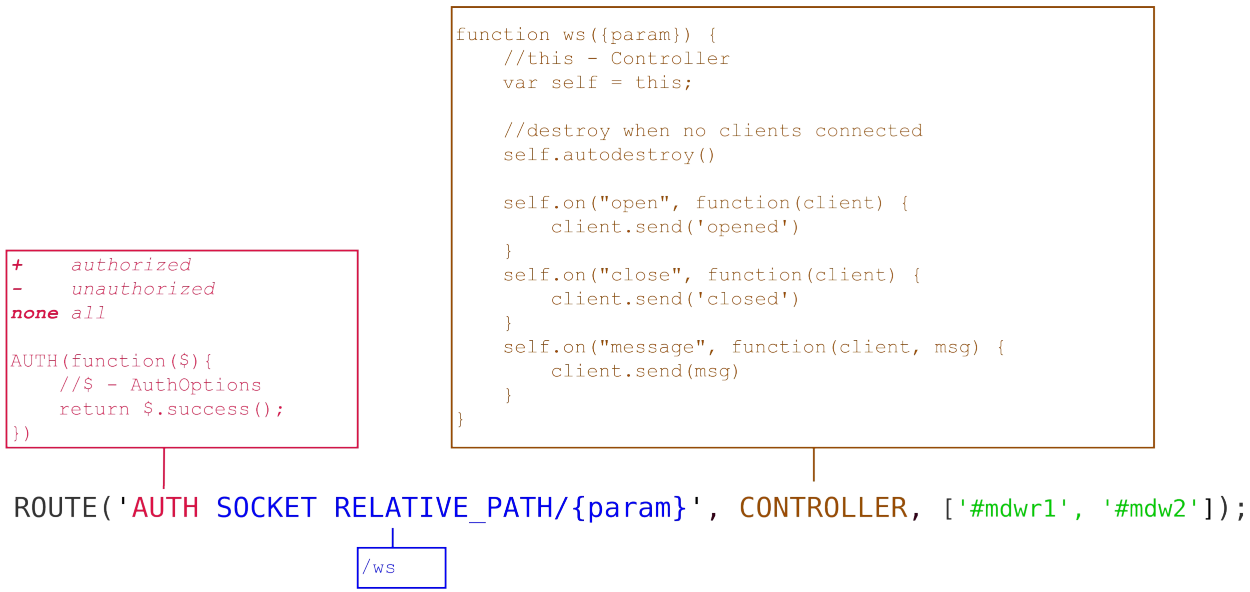
```
ROUTE('#404', CONTROLLER);
```

```
#400
#401
#403
#404
#409
#439
#500
#503
```

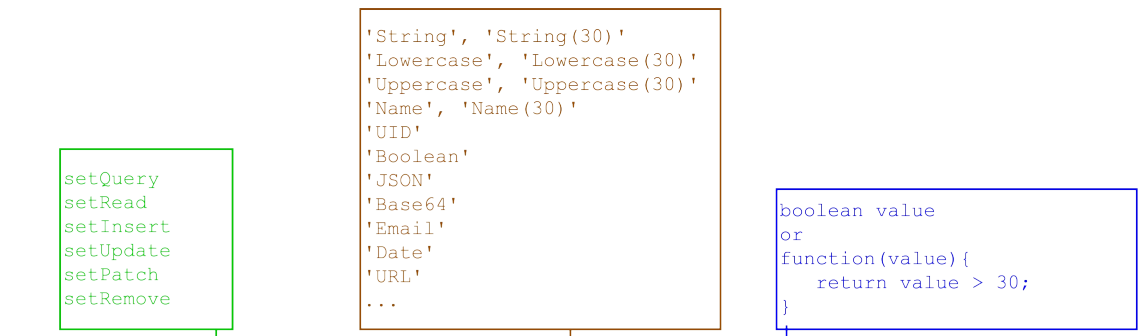
Dynamic content routing



Websocket routing



Schemas routing




```
NEWSCHEMA('Schema_name', function(schema) {
    schema.define(field_name, field_type, required, error_msg_string);

    schema.setQuery(function($, model) {
        //$ - SchemaOptions
        var x = model.field_name;
    })

    schema.addWorkflow('wf_name', function($, model) {
    })

});

ROUTE('AUTH METHOD RELATIVE_PATH  *Schema_name --> query');
ROUTE('AUTH METHOD RELATIVE_PATH  *Schema_name --> query (response) wf_name');
ROUTE('AUTH METHOD RELATIVE_PATH/{PARAMS}  *Schema_name --> query (response) wf_name');
```

API routing

```
ROUTE('API RELATIVE_PATH AUTHAPI_OPERATION *Schema_name --> WORKFLOWS');
ROUTE('API RELATIVE_PATH AUTHAPI_OPERATION/{PARAMS} *Schema_name --> WORKFLOWS');

POST request to RELATIVE_PATH
{
    "schema": "API_OPERATION/{PARAMS}",
    "data": {
        "field_name": value,
        ...
    }
}


ROUTE('API @SOCKET_PATH AUTHAPI_OPERATION/{PARAMS} *Schema_name --> WORKFLOWS');
ROUTE('SOCKET @RELATIVE_PATH @SOCKET_PATH');

Websocket message
{
    "TYPE": "SOCKET_PATH",
    "callbackid": "CUSTOM_CALLBACK_ID" //will be returned back
    "data": {
        "schema": "API_OPERATION/{PARAMS}"
        "data": {
            "field_name": value,
            ...
        }
    }
}
```

Clustering

Cluster

Based on the node.js cluster API. Requests are processed in the round-robin fashion. It's extremely easy to setup in the configuration object passed as the argument to the total4 library:

```
const options = {};
require('total4')(options);
```

– auto-scale configuration

```
options.cluster = 'auto';
options.cluster_limit = 10;
```

– Fixed number of threads configuration

```
options.cluster = 10;
```

Threads in the cluster can communicate with each other using

`EMIT2('event_name', ...args)` and `ON('event_name', fn)`

or via external resources (Redis, Memcached, etc.)

Threads

Threads allow for processing different requests by different threads. The configuration is specified done in the configuration object; as well as by separating the code in the `threads` directory. Can be combined with the cluster configuration

```
const options = {};
options.timeout = 5000;
options.threads = true;
// or options.threads = path;
require('total4')(options);
```

Code in the `threads` directory:

```
threads
      thread1
            controllers
            views
            ...
      thread2
            Controllers
```

Requests:

```
GET host_name/path/thread1

POST host_name/path/thread2
```

Threads won't work on the Windows platform because Unix sockets technology is used under the hood