

---

# **Dredd**

***Release 4.9.2***

**Jan 18, 2018**



---

## Contents

---

<b>1</b>	<b>Features</b>	<b>3</b>
<b>2</b>	<b>Contents</b>	<b>5</b>
<b>3</b>	<b>Useful Links</b>	<b>83</b>
<b>4</b>	<b>Example Applications</b>	<b>85</b>





# DREDD

No more outdated API Documentation



**Dredd is a language-agnostic command-line tool for validating API description document against backend implementation of the API.**

Dredd reads your API description and step by step validates whether your API implementation replies with responses as they are described in the documentation.



### 1.1 Supported API Description Formats

- API Blueprint
- Swagger

### 1.2 Supported Hooks Languages

Dredd supports writing *hooks* — a glue code for each test setup and teardown. Following languages are supported:

- *Go*
- *Node.js (JavaScript)*
- *Perl*
- *PHP*
- *Python*
- *Ruby*
- *Rust*
- Didn't find your favorite language? Add a new one

### 1.3 Supported Systems

- Linux, macOS, Windows, ...
- Travis CI, CircleCI, Jenkins, AppVeyor, ...





## 2.1 Installation

Dredd is a command-line application written in [CoffeeScript](#), a dialect of the JavaScript programming language. To run it on your machine or in your [Continuous Integration](#) server, you first need to have [Node.js](#) installed.

### 2.1.1 Install Node.js

#### macOS

1. Install Node.js.
  - If you're using [Homebrew](#), run `brew install node`.
  - Otherwise [download Node.js](#) from the official website and install Node.js using the downloaded installer.
2. Make sure both `node --version` and `npm --version` work in your Terminal.

#### Windows

1. [Download Node.js](#) from the official website and install Node.js using the downloaded installer.
2. Make sure both `node --version` and `npm --version` work in your Command Prompt.

#### Linux

1. Install Node.js as system package.
2. Make sure both `node --version` and `npm --version` work in your Terminal.

### Pro Tips

- [Continuous Integration](#) section in the [How-To Guides](#) can help you to install Dredd on CI server.
- To maintain multiple Node.js versions on your computer, check out [nvm](#).

### 2.1.2 Install Dredd

1. `npm install -g dredd`
2. `dredd --version`

If the second command works, you're done!

### Globally vs locally

The `-g` ensures Dredd will be installed “globally”. That means you'll be able to access it from any directory just by typing `dredd`.

If you work on projects installable by `npm`, i.e. projects containing `package.json`, you might want to have Dredd installed as a development dependency instead. Just install Dredd by `npm install dredd --save-dev`. See `package.json` of the [Dredd Example](#) repository for inspiration.

### Which Version?

- **For development**, always go with the latest version.
- **For testing in CI**, always pin your Dredd version to a specific number and upgrade to newer releases manually (but often!).

### Why Am I Seeing Network Errors?

In a restricted network (VPN, firewall, proxy) you can see errors similar to the following ones:

```
npmERR! Cannot read property 'path' of null
npmERR!code ECONNRESET
npmERR!network socket hang up
```

```
Error: Command failed: git config --get remote.origin.url
ssh: connect to host github.com port 22: Operation timed out
fatal: Could not read from remote repository.
```

To solve these issues, you need to set your proxy settings for both `npm` and `git`:

```
$ npm config set proxy "http://proxy.company.com:8080"
$ npm config set https-proxy "https://proxy.company.com:8080"

$ git config --global http.proxy "http://proxy.company.com:8080"
$ git config --global https.proxy "https://proxy.company.com:8080"
```

When using `git config`, make sure you have the port specified even when it's the standard `:80`. Also check out [how to set up Dredd to correctly work with proxies](#).

## Why I'm Seeing `node-gyp` or `python` Errors?

The installation process features compilation of some C++ components, which may not be successful. In that case, errors related to `node-gyp` or `python` are printed. However, if `dredd --version` works for you when the installation is done, feel free to ignore the errors.

In case of compilation errors, Dredd automatically uses a less performant solution written in pure JavaScript. Next time when installing Dredd, you can use `npm install -g dredd --no-optional` to skip the compilation step ([learn more about this](#)).

## Why Is the Installation So Slow?

The installation process features compilation of some C++ components, which may take some time ([learn more about this](#)). You can simplify and speed up the process using `npm install -g dredd --no-optional` if you are:

- using Dredd exclusively with [Swagger](#),
- using Dredd with small [API Blueprint](#) files,
- using Dredd on Windows or other environments with complicated C++11 compiler setup.

The `--no-optional` option avoids any compilation attempts when installing Dredd, but causes slower reading of the API Blueprint files, especially the large ones.

## Windows Support

There are still [several known limitations](#) when using Dredd on Windows, but the intention is to support it without any compromises. If you find any new issues, please [file them in the bug tracker](#).

## 2.2 Quickstart

In following tutorial you can quickly learn how to test a simple HTTP API application with Dredd. The tested application will be very simple backend written in [Express.js](#).

### 2.2.1 Install Dredd

```
$ npm install -g dredd
```

If you're not familiar with the Node.js ecosystem or you bump into any issues, follow the [installation guide](#).

**Note:** While Dredd seems to work on Windows for many users, it's not officially supported on the platform yet. See [details](#).

### 2.2.2 Document Your API

First, let's design the API we are about to build and test. That means you will need to create an API description file, which will document how your API should look like. Dredd supports two formats of API description documents:

- [API Blueprint](#)
- [Swagger](#)

If you choose API Blueprint, create a file called `api-description.apib` in the root of your project and save it with following content:

```
FORMAT: 1A

# GET /
+ Response 200 (application/json; charset=utf-8)

    {"message": "Hello World!"}
```

If you choose Swagger, create a file called `api-description.yml`:

```
swagger: "2.0"
info:
  version: "1.0"
  title: Example API
  license:
    name: MIT
host: www.example.com
basePath: /
schemes:
  - http
paths:
  /:
    get:
      produces:
        - application/json; charset=utf-8
      responses:
        200:
          description: ""
          schema:
            type: object
            properties:
              message:
                type: string
            required:
              - message
```

### 2.2.3 Implement Your API

As we mentioned in the beginning, we'll use [Express.js](#) to implement the API. Install the framework by npm:

```
$ npm init
$ npm install express --save
```

Now let's code the thing! Create a file called `app.js` with following contents:

```
var app = require('express')();

app.get('/', function(req, res) {
  res.json({message: 'Hello World!'});
})

app.listen(3000);
```

## 2.2.4 Test Your API

At this moment, the implementation is ready to be tested. Let's run the server as a background process and let's test it:

```
$ node app.js &
```

Finally, let Dredd validate whether your freshly implemented API complies with the description you have:

```
$ dredd api-description.apib http://127.0.0.1:3000 # API Blueprint
$ dredd api-description.yml http://127.0.0.1:3000 # Swagger
```

## 2.2.5 Configure Dredd

Dredd can be configured by *many CLI options*. It's recommended to save your Dredd configuration alongside your project, so it's easier to repeatedly execute always the same test run. Use interactive configuration wizard to create `dredd.yml` file in the root of your project:

```
$ dredd init
? Location of the API description document: api-description.apib
? Command to start API backend server e.g. (bundle exec rails server)
? URL of tested API endpoint: http://127.0.0.1:3000
? Programming language of hooks:
  nodejs
  python
  ruby
  ...
? Dredd is best served with Continuous Integration. Create CircleCI config for Dredd? 
  ↩ Yes
```

Now you can start test run just by typing `dredd`!

```
$ dredd
```

## 2.2.6 Use Hooks

Dredd's *hooks* enable you to write some glue code in your favorite language to support enhanced scenarios in your API tests. Read the documentation about hooks to learn more on how to write them. Choose your language and install corresponding hook handler library.

## 2.2.7 Advanced Examples

For more complex example applications, please refer to:

- [Express.js example application](#)
- [Ruby on Rails example application](#)
- [Laravel example application](#)

## 2.3 How It Works

In a nutshell, Dredd does following:

1. Takes your API description document,
2. creates expectations based on requests and responses documented in the document,
3. makes requests to tested API,
4. checks whether API responses match the documented responses,
5. reports the results.

### 2.3.1 Versioning

Dredd follows [Semantic Versioning](#). To ensure certain stability of your Dredd installation (e.g. in CI), pin the version accordingly. You can also use release tags:

- `npm install dredd` - Installs the latest published version including experimental pre-release versions.
- `npm install dredd@stable` - Skips experimental pre-release versions. Recommended for CI installations.

If the `User-Agent` header isn't overridden in the API description document, Dredd uses it for sending information about its version number along with every HTTP request it does.

### 2.3.2 Execution Life Cycle

Following execution life cycle documentation should help you to understand how Dredd works internally and which action goes after which.

1. Load and parse API description documents
  - Report parse errors and warnings
2. Pre-run API description check
  - Missing example values for URI template parameters
  - Required parameters present in URI
  - Report non-parseable JSON bodies
  - Report invalid URI parameters
  - Report invalid URI templates
3. Compile HTTP transactions from API description documents
  - Inherit headers
  - Inherit parameters
  - Expand URI templates with parameters
4. Load *hooks*
5. Test run
  - Report test run `start`
  - Run `beforeAll` hooks
  - For each compiled transaction:
    - Report `test start`
    - Run `beforeEach` hook

- Run `before` hook
  - Send HTTP request
  - Receive HTTP response
  - Run `beforeEachValidation` hook
  - Run `beforeValidation` hook
  - *Perform validation*
  - Run `after` hook
  - Run `afterEach` hook
  - Report `test end` with result for in-progress reporting
- Run `afterAll` hooks
6. Report `test run end` with result statistics

### 2.3.3 Automatic Expectations

Dredd automatically generates expectations on HTTP responses based on examples in the API description with use of `Gavel.js` library. Please refer to `Gavel` rules if you want know more.

#### Response Headers Expectations

- All headers specified in the API description must be present in the response.
- Names of headers are validated in the case-insensitive way.
- Only values of headers significant for content negotiation are validated.
- All other headers values can differ.

When using `Swagger`, headers are taken from `response.headers`. HTTP headers significant for content negotiation are inferred according to following rules:

- `produces` is propagated as response's `Content-Type` header.
- Response's `Content-Type` header overrides any `produces`.

#### Response Body Expectations

If the HTTP response body is JSON, Dredd validates only its structure. Bodies in any other format are validated as plain text.

To validate the structure Dredd uses `JSON Schema` inferred from the API description under test. The effective JSON Schema is taken from following places (the order goes from the highest priority to the lowest):

#### API Blueprint

1. + `Schema` section - provided custom JSON Schema (`Draft v4` and `v3`) will be used.
2. + `Attributes` section with data structure description in `MSON` - API Blueprint parser automatically generates JSON Schema from `MSON`.

3. + `Body` section with sample JSON payload - `Gavel.js`, which is responsible for validation in Dredd, automatically infers some basic expectations described below.

This order exactly follows the [API Blueprint specification](#).

### Swagger

1. `response.schema` - provided JSON Schema will be used.
2. `response.examples` with sample JSON payload - `Gavel.js`, which is responsible for validation in Dredd, automatically infers some basic expectations described below.

### Gavel's Expectations

- All JSON keys on any level given in the sample must be present in the response's JSON.
- Response's JSON values must be of the same JSON primitive type.
- All JSON values can differ.
- Arrays can have additional items, type or structure of the items is not validated.
- Plain text must match perfectly.

### Custom Expectations

You can make your own custom expectations in `hooks`. For instance, check out how to employ `Chai.js` assertions.

## 2.3.4 Making Your API Description Ready for Testing

It's very likely that your API description document will not be testable **as is**. This section should help you to learn how to solve the most common issues.

### URI Parameters

Both [API Blueprint](#) and [Swagger](#) allow usage of URI templates (API Blueprint fully implements [RFC6570](#), Swagger templates are much simpler). In order to have an API description which is testable, you need to describe all required parameters used in URI (path or query) and provide sample values to make Dredd able to expand URI templates with given sample values. Following rules apply when Dredd interpolates variables in a templated URI, ordered by precedence:

1. Sample value (available in Swagger as `x-example` vendor extension property).
2. Value of `default`.
3. First value from `enum`.

If Dredd isn't able to infer any value for a required parameter, it will terminate the test run and complain that the parameter is *ambiguous*.

**Note:** The implementation of API Blueprint's request-specific parameters is still in progress and there's only experimental support for it in Dredd as of now.



## Request Headers

In [Swagger](#) documents, HTTP headers are inferred from `"in": "header"` parameters. HTTP headers significant for content negotiation are inferred according to following rules:

- `consumes` is propagated as request's `Content-Type` header.
- `produces` is propagated as request's `Accept` header.
- If request body parameters are specified as `"in": "formData"`, request's `Content-Type` header is set to `application/x-www-form-urlencoded`.

**Note:** Processing `"in": "header"` parameters and inferring `application/x-www-form-urlencoded` from `"in": "formData"` parameters is not implemented yet ([apiaryio/fury-adapter-swagger#68](#), [apiaryio/fury-adapter-swagger#67](#)).

## Request Body

### API Blueprint

The effective request body is taken from following places (the order goes from the highest priority to the lowest):

1. `+ Body` section with sample JSON payload.
2. `+ Attributes` section with data structure description in [MSON](#) - API Blueprint parser automatically generates sample JSON payload from MSON.

This order [exactly follows the API Blueprint specification](#).

### Swagger

The effective request body is inferred from `"in": "body"` and `"in": "formData"` parameters.

If body parameter has `schema.example`, it is used as a raw JSON sample for the request body. If it's not present, Dredd's [Swagger Adapter](#) generates sample values from the JSON Schema provided in the `schema` property. Following rules apply when the adapter fills values of the properties, ordered by precedence:

1. Value of default.
2. First value from enum.
3. Dummy, generated value.

### Empty Response Body

If there is no body example or schema specified for the response in your API description document, Dredd won't imply any assertions. Any server response will be considered as valid.

If you want to enforce the incoming body is empty, you can use [hooks](#):

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

hooks.beforeEachValidation(function (transaction, done) {
  if (transaction.real.body) {
    transaction.fail = 'The response body must be empty';
  }
});
```

```
done ();  
});
```

In case of responses with 204 or 205 status codes Dredd still behaves the same way, but it warns about violating the [RFC7231](#) when the responses have non-empty bodies.

### 2.3.5 Choosing HTTP Transactions

#### API Blueprint

While [API Blueprint](#) allows specifying multiple requests and responses in any combination (see specification for the [action section](#)), Dredd currently supports just separated HTTP transaction pairs like this:

```
+ Request  
+ Response  
  
+ Request  
+ Response
```

In other words, Dredd always selects just the first response for each request.

**Note:** Improving the support for multiple requests and responses is under development. Refer to issues [#25](#) and [#78](#) for details. Support for URI parameters specific to a single request within one action is also limited. Solving [#227](#) should unblock many related problems. Also see [Multiple Requests and Responses](#) guide for workarounds.

#### Swagger

The [Swagger](#) format allows to specify multiple responses for a single operation. By default Dredd tests only responses with `2xx` status codes. Responses with other codes are marked as *skipped* and can be activated in [hooks](#) - see the [Multiple Requests and Responses](#) how-to guide.

In [produces](#) and [consumes](#), only JSON media types are supported. Only the first JSON media type in [produces](#) is effective, others are skipped. Other media types are respected only when provided with [explicit examples](#).

[Default response](#) is ignored by Dredd unless it is the only available response. In that case, the default response is assumed to have HTTP 200 status code.

### 2.3.6 Security

Depending on what you test and how, output of Dredd may contain sensitive data.

Mind that if you run Dredd in a CI server provided as a service (such as [CircleCI](#), [Travis CI](#), etc.), you are disclosing the CLI output of Dredd to third parties.

When using [Apiary Reporter](#) and [Apiary Tests](#), you are sending your testing data to [Apiary](#) (Dredd creators and maintainers). See their [Terms of Service](#) and [Privacy Policy](#). Which data exactly is being sent to Apiary?

- **Complete API description under test.** This means your API Blueprint or Swagger files. The API description is stored encrypted in Apiary.
- **Complete testing results.** Those can contain details of all requests made to the server under test and their responses. Apiary stores this data unencrypted, even if the original communication between Dredd and the API server under test happens to be over HTTPS. See [Apiary Reporter Test Data](#) for detailed description of what is sent. You can [sanitize it before it gets sent](#).

- **Little meta data about your environment.** Contents of environment variables `TRAVIS`, `CIRCLE`, `CI`, `DRONE`, `BUILD_ID`, `DREDD_AGENT`, `USER`, and `DREDD_HOSTNAME` can be sent to Apiary. Your `hostname`, version of your Dredd installation, and `type`, `release` and `architecture` of your OS can be sent as well. Apiary stores this data unencrypted.

See also [guidelines on how to develop Apiary Reporter](#).

### 2.3.7 Using HTTP(S) Proxy

You can tell Dredd to use HTTP(S) proxy for:

- downloading API description documents (the `positional argument` or the `--path` option accepts also URL)
- reporting to Apiary

Dredd respects `HTTP_PROXY`, `HTTPS_PROXY`, `NO_PROXY`, `http_proxy`, `https_proxy`, and `no_proxy` environment variables. For more information on how those work see [relevant section](#) of the underlying library's documentation.

Dredd intentionally **does not support HTTP(S) proxies for testing**. Proxy can deliberately modify requests and responses or to behave in a very different way then the server under test. Testing over a proxy is, in the first place, testing of the proxy itself. That makes the test results irrelevant (and hard to debug).

## 2.4 How-To Guides

In the following guides you can find tips and best practices how to cope with some common tasks. While searching this page for particular keywords can give you quick results, reading the whole section should help you to learn some of the Dredd's core concepts and usual ways how to approach problems when testing with Dredd.

### 2.4.1 Isolation of HTTP Transactions

Requests in the API description usually aren't sorted in order to comply with logical workflow of the tested application. To get the best results from testing with Dredd, you should ensure each resource action ([API Blueprint](#)) or operation ([Swagger](#)) is executed in isolated context. This can be easily achieved using [hooks](#), where you can provide your own setup and teardown code for each HTTP transaction.

You should understand that testing with Dredd is an analogy to **unit tests** of your application code. In unit tests, each unit should be testable without any dependency on other units or previous tests.

#### Example

Common case is to solve a situation where we want to test deleting of a resource. Obviously, to test deleting of a resource, we first need to create one. However, the order of HTTP transactions can be pretty much random in the API description.

To solve the situation, it's recommended to isolate the deletion test by [hooks](#). Providing `before` hook, we can ensure the database fixture will be present every time Dredd will try to send the request to delete a category item.

#### API Blueprint

```
FORMAT: 1A

# Categories API

## Categories [/categories]

### Create a Category [POST]
+ Response 201

## Category [/category/{id}]
+ Parameters
  + id: 42 (required)

### Delete a Category [DELETE]
+ Response 204

## Category Items [/category/{id}/items]
+ Parameters
  + id: 42 (required)

## Create an Item [POST]
+ Response 201
```

To have an idea where we can hook our arbitrary code, we should first ask Dredd to list all available transaction names:

```
$ dredd api-description.apib http://127.0.0.1:3000 --names
info: Categories > Create a category
info: Category > Delete a category
info: Category Items > Create an item
```

Now we can create a `hooks.js` file. The file will contain setup and teardown of the database fixture:

```
hooks = require('hooks');
db = require('./src/db');

beforeAll(function() {
  db.cleanup();
});

afterEach(function(transaction) {
  db.cleanup();
});

before('Category > Delete a Category', function() {
  db.createCategory({id: 42});
});

before('Category Items > Create an Item', function() {
  db.createCategory({id: 42});
});
```

## Swagger

```
swagger: "2.0"
info:
  version: "0.0.0"
```

```

title: Categories API
license:
  name: MIT
host: www.example.com
basePath: /
schemes:
  - http
consumes:
  - application/json
produces:
  - application/json
paths:
  /categories:
    post:
      responses:
        200:
          description: ""
  /category/{id}:
    delete:
      parameters:
        - name: id
          in: path
          required: true
          type: string
          enum:
            - "42"
      responses:
        200:
          description: ""
  /category/{id}/items:
    post:
      parameters:
        - name: id
          in: path
          required: true
          type: string
          enum:
            - "42"
      responses:
        200:
          description: ""

```

To have an idea where we can hook our arbitrary code, we should first ask Dredd to list all available transaction names:

```

$ dredd api-description.yml http://127.0.0.1:3000 --names
info: /categories > POST > 200 > application/json
info: /category/{id} > DELETE > 200 > application/json
info: /category/{id}/items > POST > 200 > application/json

```

Now we can create a `hooks.js` file. The file will contain setup and teardown of the database fixture:

```

hooks = require('hooks');
db = require('./src/db');

beforeAll(function() {
  db.cleanup();
});

```

```
afterEach(function(transaction) {
  db.cleanup();
});

before('/category/{id}', function() {
  db.createCategory({id: 42});
});

before('/category/{id}/items', function() {
  db.createCategory({id: 42});
});
```

### 2.4.2 Testing API Workflows

Often you want to test a sequence of steps, a scenario, rather than just one request-response pair in isolation. Since the API description formats are quite limited in their support of documenting scenarios, Dredd probably isn't the best tool to provide you with this kind of testing. There are some tricks though, which can help you to work around some of the limitations.

**Note:** API Blueprint prepares direct support for testing and scenarios. Interested? Check out [apiaryio/api-blueprint#21!](#)

To test various scenarios, you will want to write each of them into a separate API description document. To load them during a single test run, use the `--path` option ([docs](#)).

For workflows to work properly, you'll also need to keep **shared context** between individual HTTP transactions. You can use [hooks](#) in order to achieve that. See tips on how to [pass data between transactions](#).

### API Blueprint Example

Imagine we have a simple workflow described:

```
FORMAT: 1A

# My Scenario
## POST /login
+ Request (application/json)
    {"username": "john", "password": "d0e"}

+ Response 200 (application/json)
    {"token": "s3cr3t"}

## GET /cars
+ Response 200 (application/json)
    [
        {"id": "42", "color": "red"}
    ]

## PATCH /cars/{id}
```

```

+ Parameters
  + id: 42 (string, required)

+ Request (application/json)

  {"color": "yellow"}

+ Response 200 (application/json)

  {"id": 42, "color": "yellow"}

```

## Writing Hooks

To have an idea where we can hook our arbitrary code, we should first ask Dredd to list all available transaction names:

```

$ dredd api-description.apib http://127.0.0.1:3000 --names
info: /login > POST
info: /cars > GET
info: /cars/{id} > PATCH

```

Now we can create a `hooks.js` file. The code of the file will use global `stash` variable to share data between requests:

```

hooks = require('hooks');
db = require('./src/db');

stash = {}

// Stash the token we've got
after('/login > POST', function (transaction) {
  stash.token = JSON.parse(transaction.real.body).token;
});

// Add the token to all HTTP transactions
beforeEach(function (transaction) {
  if (stash.token) {
    transaction.request.headers['X-API-Key'] = stash.token;
  }
});

// Stash the car ID we've got
after('/cars > GET', function (transaction) {
  stash.carId = JSON.parse(transaction.real.body).id;
});

// Replace car ID in request with the one we've stashed
before('/cars/{id} > PATCH', function (transaction) {
  transaction.fullPath = transaction.fullPath.replace('42', stash.carId);
  transaction.request.uri = transaction.fullPath;
});

```

## Swagger Example

Imagine we have a simple workflow described:

```
swagger: "2.0"
info:
  version: "0.0.0"
  title: Categories API
  license:
    name: MIT
host: www.example.com
basePath: /
schemes:
  - http
consumes:
  - application/json
produces:
  - application/json
paths:
  /login:
    post:
      parameters:
        - name: body
          in: body
          required: true
          schema:
            type: object
            properties:
              username:
                type: string
              password:
                type: string
      responses:
        200:
          description: ""
          schema:
            type: object
            properties:
              token:
                type: string
  /cars:
    get:
      responses:
        200:
          description: ""
          schema:
            type: array
            items:
              type: object
              properties:
                id:
                  type: string
                color:
                  type: string
  /cars/{id}:
    patch:
      parameters:
        - name: id
          in: path
          required: true
          type: string
          enum:
```



```

    - "42"
  - name: body
    in: body
    required: true
    schema:
      type: object
      properties:
        color:
          type: string
  responses:
    200:
      description: ""
      schema:
        type: object
        properties:
          id:
            type: string
          color:
            type: string

```

## Writing Hooks

To have an idea where we can hook our arbitrary code, we should first ask Dredd to list all available transaction names:

```

$ dredd api-description.yml http://127.0.0.1:3000 --names
info: /login > POST > 200 > application/json
info: /cars > GET > 200 > application/json
info: /cars/{id} > PATCH > 200 > application/json

```

Now we can create a `hooks.js` file. The code of the file will use global `stash` variable to share data between requests:

```

hooks = require('hooks');
db = require('./src/db');

stash = {}

// Stash the token we've got
after('/login > POST > 200 > application/json', function (transaction) {
  stash.token = JSON.parse(transaction.real.body).token;
});

// Add the token to all HTTP transactions
beforeEach(function (transaction) {
  if (stash.token) {
    transaction.request.headers['X-API-Key'] = stash.token;
  }
});

// Stash the car ID we've got
after('/cars > GET > 200 > application/json', function (transaction) {
  stash.carId = JSON.parse(transaction.real.body).id;
});

// Replace car ID in request with the one we've stashed
before('/cars/{id} > PATCH > 200 > application/json', function (transaction) {

```

```
transaction.fullPath = transaction.fullPath.replace('42', stash.carId)
transaction.request.uri = transaction.fullPath
})
```

### 2.4.3 Making Dredd Validation Stricter

API Blueprint or Swagger files are usually created primarily with *documentation* in mind. But what's enough for documentation doesn't need to be enough for *testing*.

That applies to both **MSON** (a language powering API Blueprint's + `Attributes` sections) and **JSON Schema** (a language powering the Swagger format and API Blueprint's + `Schema` sections).

In following sections you can learn about how to deal with common scenarios.

#### Avoiding Additional Properties

If you describe a JSON body which has attributes `name` and `size`, the following payload will be considered as correct:

```
{"name": "Sparta", "size": 300, "luck": false}
```

It's because in both **MSON** and **JSON Schema** additional properties are not forbidden by default.

- In API Blueprint's + `Attributes` sections you can mark your object with `fixed-type`, which doesn't allow additional properties.
- In API Blueprint's + `Schema` sections and in Swagger you can use `additionalProperties: false` on the objects.

#### Requiring Properties

If you describe a JSON body which has attributes `name` and `size`, the following payload will be considered as correct:

```
{"name": "Sparta"}
```

It's because properties are optional by default in both **MSON** and **JSON Schema** and you need to explicitly specify them as required.

- In API Blueprint's + `Attributes` section, you can use `required`.
- In API Blueprint's + `Schema` sections and in Swagger you can use `required`, where you list the required properties. (Note this is true only for the **Draft v4** JSON Schema, in older versions the `required` functionality was done differently.)

#### Validating Structure of Array Items

If you describe an array of items, where each of the items should have a `name` property, the following payload will be considered as correct:

```
[{"name": "Sparta"}, {"title": "Athens"}, "Thebes"]
```

That's because in **MSON**, the default behavior is that you are specifying what *may* appear in the array.

- In API Blueprint's + `Attributes` sections you can mark your array with `fixed-type`, which doesn't allow array items of a different structure than specified.
- In API Blueprint's + `Schema` sections and in Swagger make sure to learn about how `validation of arrays` exactly works.

### Validating Specific Values

If you describe a JSON body which has attributes `name` and `size`, the following payload will be considered as correct:

```
{"name": "Sparta", "size": 42}
```

If the size should be always equal to 300, you need to specify the fact in your API description.

- In API Blueprint's + `Attributes` sections you can mark your property with `fixed`, which turns the sample value into a required value. You can also use `enum` to provide a set of possible values.
- In API Blueprint's + `Schema` sections and in Swagger you can use `enum` with one or more possible values.

## 2.4.4 Integrating Dredd with Your Test Suite

Generally, if you want to add Dredd to your existing test suite, you can just save Dredd configuration in the `dredd.yml` file and add call for `dredd` command to your task runner.

There are also some packages which make the integration a piece of cake:

- `grunt-dredd`
- `dredd-rack`
- `meteor-dredd`

To find more, search for `dredd` in your favorite language's package index.

## 2.4.5 Continuous Integration

It's a good practice to make Dredd part of your continuous integration workflow. Only that way you can ensure that application code you'll produce won't break the contract you provide in your API documentation.

Dredd's interactive configuration wizard, `dredd init`, can help you with setting up `dredd.yml` configuration file and with modifying or generating CI configuration files for [Travis CI](#) or [CircleCI](#).

If you prefer to add Dredd yourself or you look for inspiration on how to add Dredd to other continuous integration services, see examples below. When testing in CI, always pin your Dredd version to a specific number and upgrade to newer releases manually.

### `circle.yml` Configuration File for CircleCI

```
dependencies:
  pre:
    - npm install -g dredd@x.x.x
test:
  pre:
    - dredd apiary.apib http://127.0.0.1:3000
```

### .travis.yml Configuration File for Travis CI

```
before_install:
  - npm install -g dredd@x.x.x
before_script:
  - dredd apiary.apib http://127.0.0.1:3000
```

## 2.4.6 Authenticated APIs

Dredd supports all common authentication schemes:

- Basic access authentication
- Digest access authentication
- OAuth (any version)
- CSRF tokens
- ...

Use `user` setting in your configuration file or `--user` argument to provide HTTP basic authentication:

```
--user=user:password
```

Most of the authentication schemes use HTTP header for carrying the authentication data. If you don't want to add authentication HTTP header to every request in the API description, you can instruct Dredd to do it for you:

```
--headers="Authorization: Basic YmVuOnBhc3M="
```

## 2.4.7 Sending Multipart Requests

```
FORMAT: 1A
# Testing 'multipart/form-data' Request API
# POST /data
+ Request (multipart/form-data; boundary=CUSTOM-BOUNDARY)
  + Body
    --CUSTOM-BOUNDARY
    Content-Disposition: form-data; name="text"
    Content-Type: text/plain
    test equals to 42
    --CUSTOM-BOUNDARY
    Content-Disposition: form-data; name="json"
    Content-Type: application/json
    {"test": 42}
    --CUSTOM-BOUNDARY--
+ Response 200 (application/json; charset=utf-8)
```

**+ Body**

```
{"test": "OK"}
```

```
swagger: '2.0'
info:
  title: "Testing 'multipart/form-data' Request API"
  version: '1.0'
consumes:
  - multipart/form-data; boundary=CUSTOM-BOUNDARY
produces:
  - application/json; charset=utf-8
paths:
  '/data':
    post:
      parameters:
        - name: text
          in: formData
          type: string
          required: true
          x-example: "test equals to 42"
        - name: json
          in: formData
          type: string
          required: true
          x-example: '{"test": 42}'
      responses:
        200:
          description: 'Test OK'
          examples:
            application/json; charset=utf-8:
              test: 'OK'
```

## 2.4.8 Sending Form Data

```
FORMAT: 1A

# Testing 'application/x-www-form-urlencoded' Request API

# POST /data

+ Request (application/x-www-form-urlencoded)

  + Body

    test=42

+ Response 200 (application/json; charset=utf-8)

  + Body

    {"test": "OK"}
```

```
swagger: '2.0'
info:
  title: "Testing 'application/x-www-form-urlencoded' Request API"
  version: '1.0'
consumes:
  - application/x-www-form-urlencoded
produces:
  - application/json; charset=utf-8
paths:
  '/data':
    post:
      parameters:
        - name: test
          in: formData
          type: string
          required: true
          x-example: "42"
      responses:
        200:
          description: 'Test OK'
          examples:
            application/json; charset=utf-8:
              test: 'OK'
```

### 2.4.9 Multiple Requests and Responses

**Note:** For details on this topic see also [How Dredd Works With HTTP Transactions](#).

#### API Blueprint

To test multiple requests and responses within one action in Dredd, you need to cluster them into pairs:

```
FORMAT: 1A

# My API

## Resource [/resource/{id}]

+ Parameters
  + id: 42 (required)

### Update Resource [PATCH]

+ Request (application/json)

  {"color": "yellow"}

+ Response 200 (application/json)

  {"color": "yellow", "id": 1}

+ Request Edge Case (application/json)
```

```
    {"weight": 1}
```

+ Response 400 (`application/vnd.error+json`)

```
    {"message": "Validation failed"}
```

Dredd will detect two HTTP transaction examples and will compile following transaction names:

```
$ dredd api-description.apib http://127.0.0.1 --names
info: Beginning Dredd testing...
info: Resource > Update Resource > Example 1
info: Resource > Update Resource > Example 2
```

In case you need to perform particular request with different URI parameters and standard inheritance of URI parameters isn't working for you, try [modifying transaction before its execution in hooks](#).

## Swagger

When using [Swagger](#) format, by default Dredd tests only responses with 2xx status codes. Responses with other codes are marked as *skipped* and can be activated in *hooks*:

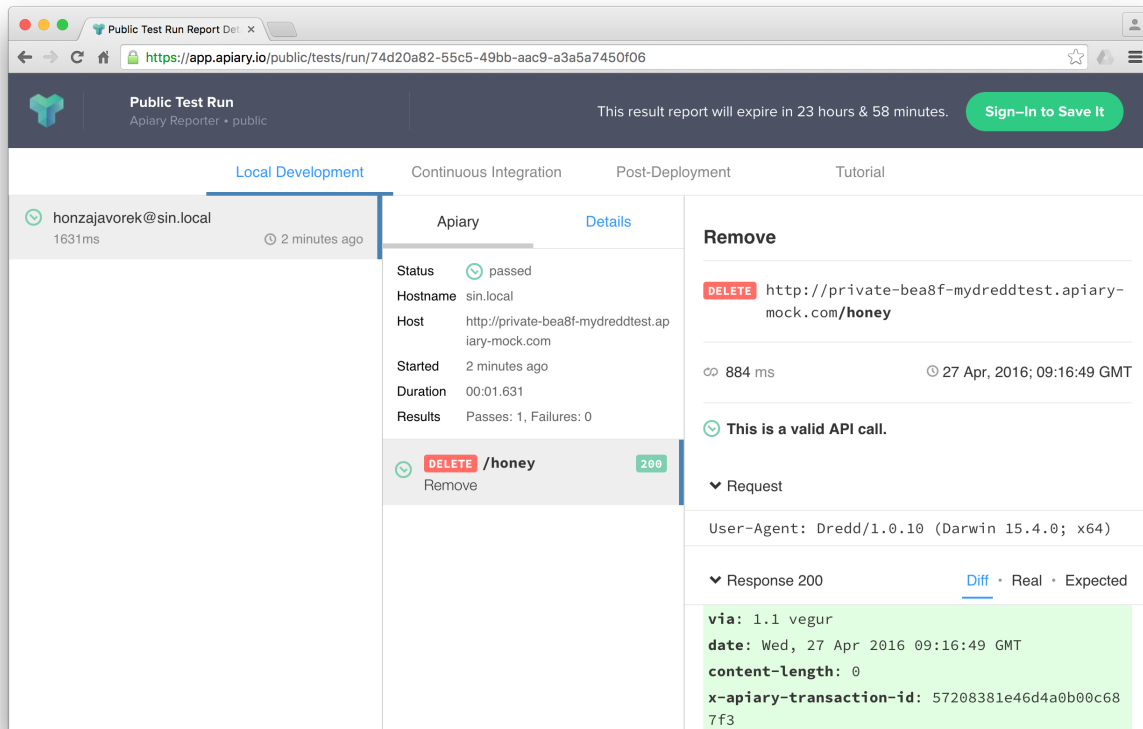
```
var hooks = require('hooks');

hooks.before('/resource > GET > 500 > application/json', function (transaction, done)
  ↪ {
    transaction.skip = false;
    done();
  ↪ });
```

### 2.4.10 Using Apiary Reporter and Apiary Tests

Command-line output of complex HTTP responses and expectations can be hard to read. To tackle the problem, you can use Dredd to send test reports to [Apiary](#). Apiary provides a comfortable interface for browsing complex test reports:

```
$ dredd apiary.apib http://127.0.0.1 --reporter=apiary
warn: Apiary API Key or API Project Subdomain were not provided. Configure Dredd to
↪ be able to save test reports alongside your Apiary API project: http://dredd.
↪ readthedocs.io/en/latest/how-to-guides/#using-apiary-reporter-and-apiary-tests
info: Beginning Dredd testing...
pass: DELETE /honey duration: 884ms
complete: 1 passing, 0 failing, 0 errors, 0 skipped, 1 total
complete: Tests took 1631ms
complete: See results in Apiary at: https://app.apiary.io/public/tests/run/74d20a82-
↪ 55c5-49bb-aac9-a3a5a7450f06
```

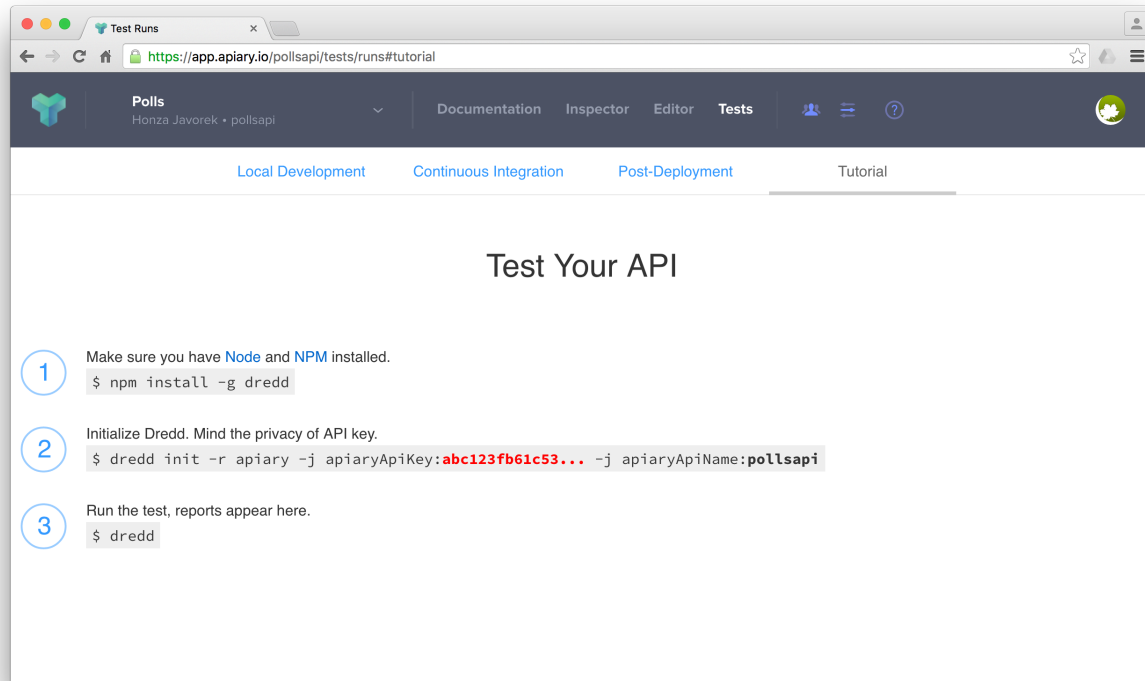


### Saving Test Reports under Your Account in Apiary

As you can see on the screenshot, the test reports are anonymous by default and will expire after some time. However, if you provide Apiary credentials, your test reports will appear on the *Tests* page of your API Project. This is great especially for introspection of test reports from Continuous Integration.

To get and setup credentials, just follow the tutorial in Apiary:





As you can see, the parameters go like this:

```
$ dredd -c apiaryApiKey:<Apiary API Key> -c apiaryApiName:<API Project Subdomain>
```

In addition to using parameters and `dredd.yml`, you can also use environment variables:

- `APIARY_API_KEY=<Apiary API Key>` - Alternative way to pass credentials to Apiary Reporter.
- `APIARY_API_NAME=<API Project Subdomain>` - Alternative way to pass credentials to Apiary Reporter.

When sending test reports to Apiary, Dredd inspects the environment where it was executed and sends some information about it alongside test results. Those are used mainly for detection whether the environment is Continuous Integration and also, they help you to identify individual test reports on the *Tests* page. You can use the following variables to tell Dredd what to send:

- `agent` (string) - `DREDD_AGENT` or current user in the OS
- `hostname` (string) - `DREDD_HOSTNAME` or `hostname` of the OS
- `CI` (boolean) - looks for `TRAVIS`, `CIRCLE`, `CI`, `DRONE`, `BUILD_ID`, ...

### 2.4.11 Example Values for Request Parameters

While example values are natural part of the API Blueprint format, the Swagger specification allows them only for body request parameters (`schema.example`).

However, Dredd needs to know what values to use when testing described API, so it supports `x-example` vendor extension property to overcome the Swagger limitation:

```
...
paths:
```

```
/cars:
  get:
    parameters:
      - name: limit
        in: query
        type: number
        x-example: 42
```

The `x-example` property is respected for all kinds of request parameters except of `body` parameters, where native `schema.example` should be used.

### 2.4.12 Removing Sensitive Data from Test Reports

Sometimes your API sends back sensitive information you don't want to get disclosed in [Apiary Tests](#) or in your CI log. In that case you can use [Hooks](#) to do sanitation. Before diving into examples below, do not forget to consider following:

- Be sure to read [section about security](#) first.
- Only the `transaction.test` object will make it to reporters. You don't have to care about sanitation of the rest of the `transaction` object.
- The `transaction.test.message` and all the `transaction.test.results.body.results.rawData.*.message` properties contain validation error messages. While they're very useful for learning about what's wrong on command line, they can contain direct mentions of header names, header values, body properties, body structure, body values, etc., thus it's recommended their contents are completely removed to prevent unintended leaks of sensitive information.
- Without the `transaction.test.results.body.results.rawData` property [Apiary reporter](#) won't be able to render green/red difference between payloads.
- You can use *Ultimate 'afterEach' Guard* to make sure you won't leak any sensitive data by mistake.
- If your hooks crash, Dredd will send an error to reporters, alongside with current contents of the `transaction.test` object. See the [Sanitation of Test Data of Transaction With Secured Erroring Hooks](#) example to learn how to prevent this.

#### Sanitation of the Entire Request Body

- [API Blueprint](#)
- [Hooks](#)

#### Sanitation of the Entire Response Body

- [API Blueprint](#)
- [Hooks](#)

#### Sanitation of a Request Body Attribute

- [API Blueprint](#)
- [Hooks](#)

### **Sanitation of a Response Body Attribute**

- API Blueprint
- Hooks

### **Sanitation of Plain Text Response Body by Pattern Matching**

- API Blueprint
- Hooks

### **Sanitation of Request Headers**

- API Blueprint
- Hooks

### **Sanitation of Response Headers**

- API Blueprint
- Hooks

### **Sanitation of URI Parameters by Pattern Matching**

- API Blueprint
- Hooks

### **Sanitation of Any Content by Pattern Matching**

- API Blueprint
- Hooks

### **Sanitation of Test Data of Passing Transaction**

- API Blueprint
- Hooks

### **Sanitation of Test Data When Transaction Is Marked as Failed in 'before' Hook**

- API Blueprint
- Hooks

### **Sanitation of Test Data When Transaction Is Marked as Failed in 'after' Hook**

- API Blueprint
- Hooks

## Sanitation of Test Data When Transaction Is Marked as Skipped

- API Blueprint
- Hooks

## Ultimate 'afterEach' Guard Using Pattern Matching

You can use this guard to make sure you won't leak any sensitive data by mistake.

- API Blueprint
- Hooks

## Sanitation of Test Data of Transaction With Secured Erroring Hooks

If your hooks crash, Dredd will send an error to reporters, alongside with current contents of the `transaction.test` object. If you want to prevent this, you need to add `try/catch` to your hooks, sanitize the test object, and gracefully fail the transaction.

- API Blueprint
- Hooks

## 2.5 Command-line Interface

### 2.5.1 Usage

```
$ dredd '<API Description Document>' '<API Location>' [OPTIONS]
```

Example:

```
$ dredd ./apiary.md http://127.0.0.1:3000
```

### 2.5.2 Arguments

#### API Description Document (string)

URL or path to the API description document (API Blueprint, Swagger). **Sample values:** `./api-blueprint.apib`, `./swagger.yml`, `./swagger.json`, `http://example.com/api-blueprint.apib`

#### API Location (string)

URL, the root address of your API. **Sample values:** `http://127.0.0.1:3000`, `http://api.example.com`

### 2.5.3 Configuration File

If you use Dredd repeatedly within a single project, the preferred way to run it is to first persist your configuration in a `dredd.yml` file. With the file in place you can then run Dredd every time simply just by:

```
$ dredd
```

Dredd offers interactive wizard to setup your `dredd.yml` file:

```
$ dredd init
```

See below how sample configuration file could look like. The structure is the same as of the [Dredd Class configuration object](#).

```
reporter: apiary
custom:
  - "apiaryApiKey:yourSecretApiaryAPIKey"
  - "apiaryApiName:apiName"
dry-run: null
hookfiles: "dreddhooks.js"
sandbox: false
server: rails server
server-wait: 3
init: false
custom: {}
names: false
only: []
output: []
header: []
sorted: false
user: null
inline-errors: false
details: false
method: []
level: info
timestamp: false
silent: false
path: []
blueprint: api-description.apib
endpoint: "http://127.0.0.1:3000"
```

**Note:** Do not get confused by Dredd using a keyword `blueprint` also for paths to Swagger documents. This is for historical reasons and will be changed in the future.

### 2.5.4 CLI Options Reference

Remember you can always list all available arguments by `dredd --help`.

#### **-color, -c**

Determines whether console output should include colors. **Default value:** `true`

#### **-config**

Path to `dredd.yml` config file. **Default value:** `./dredd.yml`

### **-custom, -j**

Pass custom key-value configuration data delimited by a colon. E.g. -j 'a:b' **Default value:** []

### **-details, -d**

Determines whether request/response details are included in passing tests.

### **-dry-run, -y**

Do not run any real HTTP transaction, only parse API description document and compile transactions.

### **-header, -h**

Extra header to include in every request. This option can be used multiple times to add multiple headers. **Default value:** []

### **-help**

Show usage information.

### **-hookfiles, -f**

Specifies a pattern to match files with before/after hooks for running tests. Files are executed in alphabetical order.

### **-hooks-worker-after-connect-wait**

How long to wait between connecting to hooks worker and start of testing. [ms] **Default value:** 100

### **-hooks-worker-connect-retry**

How long to wait between attempts to connect to hooks worker. [ms] **Default value:** 500

### **-hooks-worker-connect-timeout**

Total hook worker connection timeout (includes all retries). [ms] **Default value:** 1500

### **-hooks-worker-handler-host**

Host of the hook worker. **Default value:** "127.0.0.1"

### **-hooks-worker-handler-port**

Port of the hook worker. **Default value:** 61321

**-hooks-worker-term-retry**

How long to wait between attempts to terminate hooks worker. [ms] **Default value:** 500

**-hooks-worker-term-timeout**

How long to wait between trying to terminate hooks worker and killing it. [ms] **Default value:** 5000

**-hooks-worker-timeout**

How long to wait for hooks worker to start. [ms] **Default value:** 5000

**-init, -i**

Run interactive configuration. Creates dredd.yml configuration file.

**-inline-errors, -e**

Determines whether failures and errors are displayed as they occur (true) or aggregated and displayed at the end (false).

**-language, -a**

Language of hookfiles. Possible options are: nodejs, ruby, python, php, perl, go, rust **Default value:** "nodejs"

**-level, -l**

The level of logging to output. Options: silly, debug, verbose, info, warn, error. **Default value:** "info"

**-method, -m**

Restrict tests to a particular HTTP method (GET, PUT, POST, DELETE, PATCH). This option can be used multiple times to allow multiple methods. **Default value:** []

**-names, -n**

Only list names of requests (for use in a hookfile). No requests are made.

**-only, -x**

Run only specified transaction name. Can be used multiple times **Default value:** []

**-output, -o**

Specifies output file when using additional file-based reporter. This option can be used multiple times if multiple file-based reporters are used. **Default value:** []

### **-path, -p**

Additional API description paths or URLs. Can be used multiple times with glob pattern for paths. **Default value:** []

### **-reporter, -r**

Output additional report format. This option can be used multiple times to add multiple reporters. Options: xunit, nyan, dot, markdown, html, apiary. **Default value:** []

### **-sandbox, -b**

Load and run non trusted hooks code in sandboxed container

### **-server, -g**

Run API backend server command and kill it after Dredd execution. E.g. `rails server`

### **-server-wait**

Set delay time in seconds between running a server and test run. **Default value:** 3

### **-silent, -q**

Silences commandline output.

### **-sorted, -s**

Sorts requests in a sensible way so that objects are not modified before they are created. Order: CONNECT, OPTIONS, POST, GET, HEAD, PUT, PATCH, DELETE, TRACE.

### **-timestamp, -t**

Determines whether console output should include timestamps.

### **-user, -u**

Basic Auth credentials in the form `username:password`.

### **-version**

Show version number.



## 2.6 Using Dredd as a JavaScript Library

Dredd can be used directly from your JavaScript code. First, import and configure Dredd:

```
var Dredd = require('dredd');
var dredd = new Dredd(configuration);
```

Then you need to run the Dredd testing:

```
dredd.run(function (err, stats) {
  // err is present if anything went wrong
  // otherwise stats is an object with useful statistics
});
```

As you can see, `dredd.run` is a function receiving another function as a callback. Received arguments are `err` (error if any) and `stats` (testing statistics) with numbers accumulated throughout the Dredd run.

### 2.6.1 Configuration Object for Dredd Class

Let's have a look at an example configuration first. (Please also see [options source](#) to read detailed information about the options attributes).

```
{
  server: 'http://127.0.0.1:3000/api', // your URL to API endpoint the tests will run
  ↪against
  options: {
    'path': [], // Required Array if Strings; filepaths to API description
    ↪documents, can use glob wildcards

    'dry-run': false, // Boolean, do not run any real HTTP transaction
    'names': false, // Boolean, Print Transaction names and finish, similar to dry-
    ↪run

    'level': 'info', // String, log-level (info, silly, debug, verbose, ...)
    'silent': false, // Boolean, Silences all logging output

    'only': [], // Array of Strings, run only transaction that match these names

    'header': [], // Array of Strings, these strings are then added as headers
    ↪(key:value) to every transaction
    'user': null, // String, Basic Auth credentials in the form username:password

    'hookfiles': [], // Array of Strings, filepaths to files containing hooks (can
    ↪use glob wildcards)

    'reporter': ['dot', 'html'], // Array of possible reporters, see folder src/
    ↪reporters

    'output': [], // Array of Strings, filepaths to files used for output of file-
    ↪based reporters

    'inline-errors': false, // Boolean, If failures/errors are display immediately in
    ↪Dredd run

    'color': true,
```

```
'timestamp': false
},
'emitter': EventEmitterInstance, // optional - listen to test progress, your own
↳instance of EventEmitter

'hooksData': {
  'pathToHook' : '...'
}

'data': {
  'path/to/file': '...'
}
}
```

### 2.6.2 Properties

#### server (string)

Your choice of the API endpoint to test the API description against. It must be a valid URL (you can specify port, path and http or https protocol).

#### options (object)

Because `options.path` array is required, you must specify options. You'll end with errors otherwise.

#### options.path (object)

**Required** Array of filepaths to API description documents. Or it can also be an URL to download the API description from internet via http(s) protocol.

#### data (object)

**Optional** Object with keys as filename and value as blueprint-code.

Useful when you don't want to operate on top of filesystem and want to pass code of your API description as a string. You get the point.

#### hooksData (object)

**Optional** Object with keys as filename and strings with JavaScript hooks code.

Load hooks file code from string. Must be used together with sandboxed mode.

```
{
  'data': {
    './api-description.apib': 'FORMAT: 1A\n\n# My String API\n\nGET /url\n+ Response
↳200\n\n    Some content',
    './directory/another-api-description.apib': '# Another API\n\n## Group
↳Machines\n\n### Machine [/machine]\n\n#### Read machine [GET]\n\n...'
```

```
}  
}
```

## 2.7 Hook Scripts

Similar to any other testing framework, Dredd supports executing code around each test step. Hooks are code blocks executed in defined stage of *execution lifecycle*. In the hooks code you have an access to compiled HTTP *transaction object* which you can modify.

Hooks are usually used for:

- loading db fixtures
- cleanup after test step or steps
- handling authentication and sessions
- passing data between transactions (saving state from responses to *stash*)
- modifying request generated from API description
- changing generated expectations
- setting custom expectations
- debugging via logging stuff

### 2.7.1 Languages

You can interact with your server implementation in following languages:

- *Go*
- *JavaScript (Sandboxed)*
- *Node.js*
- *Perl*
- *PHP*
- *Python*
- *Ruby*

Dredd doesn't speak your language? **It's very easy to write support for your language.** Your contribution is more than welcome!

### 2.7.2 Using Hook Files

To use a hook file with Dredd, use the `--hookfiles` flag in the command line. You can use this flag multiple times or use a `glob` expression for loading multiple hook files. Dredd executes hook files in alphabetical order.

Example:

```
$ dredd single-get.apib http://machines.apiary.io --hookfiles=*_hooks.*
```

## 2.7.3 Getting Transaction Names

For addressing specific test steps is used the **transaction names** of the compiled HTTP transactions (*actions*) from the API description.

In order to retrieve transaction names please run Dredd with the `--names` argument last and it will print all available names of transactions.

For example, given an API Blueprint file `api-description.apib` as following:

```
FORMAT: 1A

# Machines API

# Group Machines

# Machines collection [/machines]

## Get Machines [GET]

- Response 200 (application/json; charset=utf-8)

  [{"type": "bulldozer", "name": "willy"}]
```

Run this command to retrieve all transaction names:

```
$ dredd single-get.apib http://machines.apiary.io --names
info: Machines > Machines collection > Get Machines
```

The `Machines > Machines collection > Get Machines` is the name of a transaction which you can use in your hooks. The same approach works also for Swagger documents.

## 2.7.4 Types of Hooks

Dredd supports following types of hooks:

- `beforeAll` called at the beginning of the whole test run
- `beforeEach` called before each HTTP transaction
- `before` called before some specific HTTP transaction
- `beforeEachValidation` called before each HTTP transaction is validated
- `beforeValidation` called before some specific HTTP transaction is validated
- `after` called after some specific HTTP transaction regardless its result
- `afterEach` called after each HTTP transaction
- `afterAll` called after whole test run

Refer to [Dredd execution lifecycle](#) when is each hook executed.

## Transaction Object Structure

The main purpose of hooks is to work with the transaction object they get as the first argument, in order to inspect or modify Dredd's behavior. See [transaction object reference](#) to learn more about its contents.

## 2.8 Writing Dredd Hooks In Go

GitHub repository

Go hooks are using *Dredd's hooks handler socket interface*. For using Go hooks in Dredd you have to have *Dredd already installed*. The Go library is called `goodman`.

### 2.8.1 Installation

```
$ go get github.com/snikch/goodman/cmd/goodman
```

### 2.8.2 Usage

Using Dredd with Go is slightly different to other languages, as a binary needs to be compiled for execution. The `--hookfiles` flags should point to compiled hook binaries. See below for an example `hooks.go` file to get an idea of what the source file behind the go binary would look like.

```
$ dredd apiary.apib http://127.0.0.1:3000 --server=./go-lang-web-server-to-test --
↪language=go --hookfiles=./hook-file-binary
```

### 2.8.3 API Reference

In order to get a general idea of how the Go Hooks work, the main executable from the package `$GOPATH/bin/goodman` is an HTTP Server that Dredd communicates with and an RPC client. Each hookfile then acts as a corresponding RPC server. So when Dredd notifies the Hooks server what transaction event is occurring the hooks server will execute all registered hooks on each of the hookfiles RPC servers.

You'll need to know a few things about the `Server` type in the hooks package.

1. The `hooks.Server` type is how you can define event callbacks such as `beforeEach`, `afterAll`, etc.
2. To get a `hooks.Server` struct you must do the following

```
package main

import (
    "github.com/snikch/goodman/hooks"
    trans "github.com/snikch/goodman/transaction"
)

func main() {
    h := hooks.NewHooks()
    server := hooks.NewServer(hooks.NewHooksRunner(h))

    // Define all your event callbacks here

    // server.Serve() will block and allow the goodman server to run your defined
    // event callbacks
    server.Serve()
    // You must close the listener at end of main()
    defer server.Listener.Close()
}
```

1. Callbacks receive a `Transaction` instance, or an array of them

2. A Server will run your Runner and handle receiving events on the dredd socket.

### Runner Callback Events

The Runner type has the following callback methods.

1. BeforeEach, BeforeEachValidation, AfterEach
  - accepts a function as a first argument passing a Transaction object as a first argument
2. Before, BeforeValidation, After
  - accepts transaction name as a first argument
  - accepts a function as a second argument passing a Transaction object as a first argument of it
3. BeforeAll, AfterAll
  - accepts a function as a first argument passing a Slice of Transaction objects as a first argument

Refer to Dredd execution lifecycle to find when each hook callback is executed.

### Using the Go API

Example usage of all methods.

```
package main

import (
    "fmt"

    "github.com/snikch/goodman/hooks"
    trans "github.com/snikch/goodman/transaction"
)

func main() {
    h := hooks.NewHooks()
    server := hooks.NewServer(hooks.NewHooksRunner(h))
    h.BeforeAll(func(t []*trans.Transaction) {
        fmt.Println("before all modification")
    })
    h.BeforeEach(func(t *trans.Transaction) {
        fmt.Println("before each modification")
    })
    h.Before("/message > GET", func(t *trans.Transaction) {
        fmt.Println("before modification")
    })
    h.BeforeEachValidation(func(t *trans.Transaction) {
        fmt.Println("before each validation modification")
    })
    h.BeforeValidation("/message > GET", func(t *trans.Transaction) {
        fmt.Println("before validation modification")
    })
    h.After("/message > GET", func(t *trans.Transaction) {
        fmt.Println("after modification")
    })
    h.AfterEach(func(t *trans.Transaction) {
        fmt.Println("after each modification")
    })
}
```

```

h.AfterAll(func(t []*trans.Transaction) {
    fmt.Println("after all modification")
})
server.Serve()
defer server.Listener.Close()
}

```

## 2.8.4 Examples

### How to Skip Tests

Any test step can be skipped by setting the `Skip` property of the `Transaction` instance to `true`.

```

package main

import (
    "fmt"

    "github.com/snikch/goodman/hooks"
    trans "github.com/snikch/goodman/transaction"
)

func main() {
    h := hooks.NewHooks()
    server := hooks.NewServer(hooks.NewHooksRunner(h))
    h.Before("Machines > Machines collection > Get Machines", func(t *trans.
↪Transaction) {
        t.Skip = true
    })
    server.Serve()
    defer server.Listener.Close()
}

```

### Failing Tests Programmatically

You can fail any step by setting the `Fail` field of the `Transaction` instance to `true` or any string with a descriptive message.

```

package main

import (
    "fmt"

    "github.com/snikch/goodman/hooks"
    trans "github.com/snikch/goodman/transaction"
)

func main() {
    h := hooks.NewHooks()
    server := hooks.NewServer(hooks.NewHooksRunner(h))
    h.Before("Machines > Machines collection > Get Machines", func(t *trans.
↪Transaction) {
        t.Fail = true
    })
}

```

```
h.Before("Machines > Machines collection > Post Machines", func(t *trans.
↳Transaction) {
    t.Fail = "POST is broken"
})
server.Serve()
defer server.Listener.Close()
}
```

### Modifying the Request Body Prior to Execution

```
package main

import (
    "fmt"

    "github.com/snikch/goodman/hooks"
    trans "github.com/snikch/goodman/transaction"
)

func main() {
    h := hooks.NewHooks()
    server := hooks.NewServer(hooks.NewHooksRunner(h))
    h.Before("Machines > Machines collection > Get Machines", func(t *trans.
↳Transaction) {
        body := map[string]interface{}{}
        json.Unmarshal([]byte(t.Request.Body), &body)

        body["someKey"] = "new value"

        newBody, _ := json.Marshal(body)
        t.Request.body = string(newBody)
    })
    server.Serve()
    defer server.Listener.Close()
}
```

## 2.9 JavaScript Hooks In Sandbox Mode

### 2.9.1 Usage

```
$ dredd apiary.apib http://127.0.0.1:3000 --sandbox --hookfiles=./hooks*.js
```

#### Dredd JS API Option

Sandbox mode can be enabled in Dredd JavaScript API

```
var Dredd = require('dredd');
var configuration = {
    server: "http://127.0.0.1",
    options: {
        path: "./test/fixtures/single-get.apib",
```



```

    sandbox: true,
    hookfiles: ['./test/fixtures/sandboxed-hook.js']
  }
};
var dredd = new Dredd(configuration);

dredd.run(function (error, stats) {
  // your callback code here
});

```

## 2.9.2 Sandboxed JavaScript Hooks API reference

The Sandbox mode can be used for running untrusted hook code. It can be activated with a CLI switch or with the JS API. In each hook file you can use following functions:

```

beforeAll(function)
beforeEach(function)
before(transactionName, function)
beforeEachValidation(function)
beforeValidation(transactionName, function)
after(transactionName, function)
afterEach(function)
afterAll(function)
log(string)

```

- A **Transaction Object** is passed as a first argument to the hook function for `before`, `after`, `beforeEach`, `afterEach`, `beforeValidation` and `beforeEachValidation`.
- An array of Transaction Objects is passed to `beforeAll` and `afterAll`.
- Sandboxed hooks don't have an asynchronous API. Loading and running of each hook happens in it's own isolated, sandboxed context.
- Hook maximum execution time is 500ms.
- Memory limit is 1M
- You can access global `stash` object variables in each separate hook file. `stash` is passed between contexts of each hook function execution. This `stash` object purpose is to allow *transportation* of user defined values of type `String`, `Number`, `Boolean`, `null` or `Object` and `Array` (no `Functions` or `callbacks`).
- Hook code is evaluated with `"use strict"` directive - [details at MDN](#)
- Sandboxed mode does not support hooks written in CoffeeScript language
- You can print to console (via Dredd's logger) with `log` function, taking into account the used logging level `--level` option (levels: `error` > `warn` > `hook` > `info`)

### Request Stash in Sandbox Mode

```
after('First action', function (transaction) {
  stash['id'] = JSON.parse(transaction.real.response);
});

before('Second action', function (transaction) {
  newBody = JSON.parse(transaction.request.body);
  newBody['id'] = stash['id'];
  transaction.request.body = JSON.stringify(newBody);
});
```

### Hook function context is not shared

When **sandboxed**, hook function **context is not shared** between even the same step hook functions.

Note: **This is wrong**. It throws an exception.

```
var myObject = {};

after('First action', function (transaction) {
  myObject['id'] = JSON.parse(transaction.real.response);
});

before('Second action', function (transaction) {
  newBody = JSON.parse(transaction.request.body);
  newBody['id'] = myObject['id'];
  transaction.request.body = JSON.stringify(newBody);
});
```

This will explode with: `ReferenceError: myObject is not defined`

## 2.10 Writing Dredd Hooks In Node.js

### 2.10.1 Usage

```
$ dredd apiary.apib http://127.0.0.1:30000 --hookfiles=./hooks*.js
```

### 2.10.2 API Reference

- For `before`, `after`, `beforeValidation`, `beforeEach`, `afterEach` and `beforeEachValidation` a `Transaction Object` is passed as the first argument to the hook function.
- An array of `Transaction Objects` is passed to `beforeAll` and `afterAll`.
- The second argument is an optional callback function for async execution.
- Any modifications on the `transaction` object are propagated to the actual HTTP transactions.
- You can use `hooks.log` function inside the hook function to print yours debug messages and other information.
- `configuration` object is populated on the `hooks` object

## Sync API

```

var hooks = require('hooks');

hooks.beforeAll(function (transactions) {
  hooks.log('before all');
});

hooks.beforeEach(function (transaction) {
  hooks.log('before each');
});

hooks.before("Machines > Machines collection > Get Machines", function (transaction) {
  hooks.log("before");
});

hooks.beforeEachValidation(function (transaction) {
  hooks.log('before each validation');
});

hooks.beforeValidation("Machines > Machines collection > Get Machines", function_
↪(transaction) {
  hooks.log("before validation");
});

hooks.after("Machines > Machines collection > Get Machines", function (transaction) {
  hooks.log("after");
});

hooks.afterEach(function (transaction) {
  hooks.log('after each');
});

hooks.afterAll(function (transactions) {
  hooks.log('after all');
})

```

## Async API

When the callback is used in the hook function, callbacks can handle asynchronous function calls.

```

var hooks = require('hooks');

hooks.beforeAll(function (transactions, done) {
  hooks.log('before all');
  done();
});

hooks.beforeEach(function (transaction, done) {
  hooks.log('before each');
  done();
});

hooks.before("Machines > Machines collection > Get Machines", function (transaction,
↪done) {
  hooks.log("before");
  done();
})

```

```
});

hooks.beforeEachValidation(function (transaction, done) {
  hooks.log('before each validation');
  done();
});

hooks.beforeValidation("Machines > Machines collection > Get Machines", function_
↪(transaction, done) {
  hooks.log("before validation");
  done();
});

hooks.after("Machines > Machines collection > Get Machines", function (transaction,
↪done) {
  hooks.log("after");
  done();
});

hooks.afterEach(function (transaction, done) {
  hooks.log('after each');
  done();
});

hooks.afterAll(function (transactions, done) {
  hooks.log('after all');
  done();
})
```

### 2.10.3 Examples

#### How to Skip Tests

Any test step can be skipped by setting skip property of the transaction object to true.

```
var before = require('hooks').before;

before("Machines > Machines collection > Get Machines", function (transaction) {
  transaction.skip = true;
});
```

#### Sharing Data Between Steps in Request Stash

You may pass data between test steps using the response stash.

```
var hooks = require('hooks');
var before = hooks.before;
var after = hooks.after;

var responseStash = {};

after("Machines > Machines collection > Create Machine", function (transaction) {

  // saving HTTP response to the stash
```

```

    responseStash[transaction.name] = transaction.real;
  });

before("Machines > Machine > Delete a machine", function (transaction) {
  //reusing data from previous response here
  var machineId = JSON.parse(responseStash['Machines > Machines collection > Create_
↵Machine'])['id'];

  //replacing id in URL with stashed id from previous response
  var url = transaction.fullPath;
  transaction.fullPath = url.replace('42', machineId);
});

```

### Failing Tests Programmatically

You can fail any step by setting `fail` property on transaction object to `true` or any string with descriptive message.

```

var before = require('hooks').before;

before("Machines > Machines collection > Get Machines", function (transaction) {
  transaction.fail = "Some failing message";
});

```

### Using Chai Assertions

Inside hook files, you can require `Chai` and use its `assert`, `should` or `expect` interface in hooks and write your custom expectations. Dredd catches Chai's expectation error in hooks and makes transaction to fail.

```

var hooks = require('hooks');
var before = hooks.before;
var assert = require('chai').assert;

after("Machines > Machines collection > Get Machines", function (transaction) {
  assert.isBelow(transaction.real.body.length, 100);
});

```

### Modifying Transaction Request Body Prior to Execution

```

var hooks = require('hooks');
var before = hooks.before;

before("Machines > Machines collection > Get Machines", function (transaction) {
  // parse request body from API description
  var requestBody = JSON.parse(transaction.request.body);

  // modify request body here
  requestBody['someKey'] = 'someNewValue';

  // stringify the new body to request
  transaction.request.body = JSON.stringify(requestBody);
});

```

## Adding or Changing URI Query Parameters to All Requests

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

hooks.beforeEach(function (transaction) {
  // add query parameter to each transaction here
  var paramToAdd = "api-key=23456"
  if(transaction.fullPath.indexOf('?') > -1){
    transaction.fullPath += "&" + paramToAdd;
  } else{
    transaction.fullPath += "?" + paramToAdd;
  }
});
```

## Handling sessions

```
var hooks = require('hooks');
var stash = {};

// hook to retrieve session on a login
hooks.after('Auth > /remoteauth/userpass > POST', function (transaction) {
  stash['token'] = JSON.parse(transaction.real.body)['sessionId'];
});

// hook to set the session cookie in all following requests
hooks.beforeEach(function (transaction) {
  if(stash['token'] != undefined){
    transaction.request['headers']['Cookie'] = "id=" + stash['token'];
  }
});
```

## Remove trailing newline character in expected plain text bodies

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

hooks.beforeEach(function(transaction) {
  if (transaction.expected.headers['Content-Type'] === 'text/plain') {
    transaction.expected.body = transaction.expected.body.replace(/^\s+|\s+$/g, "");
  }
});
```

## Using Babel

With this workaround you can use [Babel](#) for support of all the latest JS syntactic coolness in Dredd hooks:

```
npm install -g babel-cli babel-preset-es2015
echo '{ "presets": ["es2015"] }' > .babelrc
babel-node `which dredd` test/fixtures/single-get.apib http://127.0.0.1:3000 --
↪hookfiles=./es2015.js
```

## 2.11 Writing Dredd Hooks In Perl

GitHub repository

Perl hooks are using *Dredd's hooks handler socket interface*. For using Perl hooks in Dredd you have to have *Dredd already installed*

### 2.11.1 Installation

```
$ cpanm Dredd::Hooks
```

### 2.11.2 Usage

```
$ dredd apiary.apib http://127.0.0.1:3000 --language=dredd-hooks-perl --hookfiles=./
↳hooks*.pl
```

### 2.11.3 API Reference

Module `Dredd::Hooks::Methods` imports following decorators:

1. `beforeEach`, `beforeEachValidation`, `afterEach`
  - wraps a function and passes `Transaction` object as a first argument to it
2. `before`, `beforeValidation`, `after`
  - accepts `transaction name` as a first argument
  - wraps a function and sends a `Transaction` object as a first argument to it
3. `beforeAll`, `afterAll`
  - wraps a function and passes an Array of `Transaction` objects as a first argument to it

Refer to [Dredd execution life-cycle](#) to find when is each hook function executed.

#### Using Perl API

Example usage of all methods in

```
use Dredd::Hooks::Methods;

beforeAll( sub {
    print 'before all'
});

beforeEach( sub {
    print 'before each'
})

before( "Machines > Machines collection > Get Machines" => sub {
    print 'before'
});
```

```
beforeEachValidation(sub {
  print 'before each validation'
});

beforeValidation( "Machines > Machines collection > Get Machines" => sub {
  print 'before validations'
});

after( "Machines > Machines collection > Get Machines" => sub {
  print 'after'
});

afterEach( sub {
  print 'after_each'
});

afterAll( sub {
  print 'after_all'
});
```

## 2.11.4 Examples

### How to Skip Tests

Any test step can be skipped by setting skip property of the transaction object to true.

```
use Dredd::Hooks::Methods;
use Types::Serialiser;

before("Machines > Machines collection > Get Machines" => sub {
  my ($transaction) = @_;

  $transaction->{skip} = Types::Serialiser::true;
});
```

### Sharing Data Between Steps in Request Stash

If you want to test some API workflow, you may pass data between test steps using the response stash.

```
use JSON;
use Dredd::Hooks::Methods;

my $response_stash = {};

after("Machines > Machines collection > Create Machine" => sub {
  my ($transaction) = @_;

  # saving HTTP response to the stash
  $response_stash->{$transaction->{name}} = $transaction->{real}
});

before("Machines > Machine > Delete a machine" => sub {
  my ($transaction) = @_;
  #reusing data from previous response here
```



```

my $parsed_body = JSON->decode_json(
    $response_stash->{'Machines > Machines collection > Create Machine'}
);
my $machine_id = $parsed_body->{id};
#replacing id in URL with stashed id from previous response
$transaction->{fullPath} =~ s/42/$machine_id/;
});

```

## Failing Tests Programmatically

You can fail any step by setting fail property on transaction object to true or any string with descriptive message.

```

use Dredd::Hooks::Methods;

before("Machines > Machines collection > Get Machines" => sub {
    my ($transaction) = @_;
    $transaction->{fail} = "Some failing message";
});

```

## Modifying Transaction Request Body Prior to Execution

```

use JSON;
use Dredd::Hooks::Methods;

before("Machines > Machines collection > Get Machines" => sub {
    my ($transaction) = @_;

    # parse request body from API description
    my $request_body = JSON->decode_json($transaction->{request}{body});

    # modify request body here
    $request_body->{someKey} = 'some new value';

    # stringify the new body to request
    $transaction->{request}{body} = JSON->encode_json($request_body);
});

```

## Adding or Changing URI Query Parameters to All Requests

```

use Dredd::Hooks::Methods;

beforeEach( sub {
    my ($transaction) = @_;
    # add query parameter to each transaction here
    my $param_to_add = "api-key=23456";

    if ($transaction->{fullPath} =~ m/?/){
        $transaction->{fullPath} .= "&$param_to_add";
    } else {
        $transaction->{fullPath} .= "?$param_to_add";
    }
});

```

## Handling sessions

```

use JSON;
use Dredd::Hooks::Methods;

my $stash = {}

# hook to retrieve session on a login
after('Auth > /remoteauth/userpass > POST' => sub {
    my ($transaction) = @_;

    my $parsed_body = JSON->decode_json($transaction->{real}{body});
    my $stash->{token} = $parsed_body->{sessionId};
});

# hook to set the session cookie in all following requests
beforeEach( sub {
    my ($transaction) = @_;

    if (exists $stash->{token}){
        $transaction->{request}{headers}{Cookie} = "id=".$stash{token};
    }
});

```

## Remove trailing newline character in expected *plain text* bodies

```

use Dredd::Hooks::Methods;

beforeEach(
    my ($transaction) = @_;

    if( $transaction->{expected}{headers}{Content-Type} eq 'text/plain'){
        $transaction->{expected}{body} =.chomp($transaction->{expected}{body});
    }
});

```

## 2.12 Writing Dredd Hooks In PHP

GitHub repository

PHP hooks are using *Dredd's hooks handler socket interface*. For using PHP hooks in Dredd you have to have *Dredd* already installed

### 2.12.1 Installation

#### Requirements

- php version >= 5.4

Installing dredd-hooks-php can be easily installed through the package manager, composer.

```
$ composer require ddelnano/dredd-hooks-php --dev
```

## 2.12.2 Usage

```
$ dredd apiary.apib http://127.0.0.1:3000 --language=vendor/bin/dredd-hooks-php --
↳hookfiles=./hooks*.php
```

## 2.12.3 API Reference

The `Dredd\Hooks` class provides the static methods listed below to create hooks

1. `beforeEach`, `beforeEachValidation`, `afterEach`
  - accepts a closure as a first argument passing a `Transaction` object as a first argument
2. `before`, `beforeValidation`, `after`
  - accepts `transaction name` as a first argument
  - accepts a block as a second argument passing a `Transaction` object as a first argument of it
3. `beforeAll`, `afterAll`
  - accepts a block as a first argument passing an Array of `Transaction` objects as a first argument

Refer to [Dredd execution lifecycle](#) to find when is each hook function executed.

### Using PHP API

Example usage of all methods. **Very Important** The `$transaction` variable passed to the closure **MUST** be a reference. Otherwise the `$transaction` variable will be passed by value when the closure is executed and the changes will not be reflected.

```
<?php
use Dredd\Hooks;

Hooks::beforeAll(function(&$transaction) {
    echo "before all";
});

Hooks::beforeEach(function(&$transaction) {
    echo "before each";
});

Hooks::before("Machines > Machines collection > Get Machines", function(&
↳$transaction) {
    echo "before";
});

Hooks::beforeEachValidation(function(&$transaction) {
    echo "before each validation";
});

Hooks::beforeValidation("Machines > Machines collection > Get Machines", function(&
↳$transaction) {
```

```
    echo "before validation";
});

Hooks::after("Machines > Machines collection > Get Machines", function(&$transaction)
→{

    echo "after";
});

Hooks::afterEach(function(&$transaction) {

    echo "after each";
});

Hooks::afterAll(function(&$transaction) {

    echo "after all";
});
```

### 2.12.4 Examples

In the [dredd-hooks-php repository](#) there is an example laravel application with instructions in the [wiki](#)

#### How to Skip Tests

Any test step can be skipped by setting `skip` property of the `transaction` object to `true`.

```
<?php

use Dredd\Hooks;

Hooks::before("Machines > Machines collection > Get Machines", function(&
→$transaction) {

    $transaction->skip = true;
});
```

#### Failing Tests Programmatically

You can fail any step by setting `fail` property on `transaction` object to `true` or any string with descriptive message.

```
<?php

use Dredd\Hooks;

Hooks::before("Machines > Machines collection > Get Machines", function(&
→$transaction) {
```

```

    $transaction->fail = true;
});

```

## Modifying Transaction Request Body Prior to Execution

```

<?php
use Dredd\Hooks;

Hooks::before("Machines > Machines collection > Get Machines", function(&
↳$transaction) {

    $requestBody = $transaction->request->body;

    $requestBody['someKey'] = 'new value';

    $transaction->request->body = json_encode($requestBody);
});

```

## Adding or Changing URI Query Parameters to All Requests

```

<?php
use Dredd\Hooks;

Hooks::beforeEach(function(&$transaction) {

    // add query parameter to each transaction here

    $paramToAdd = 'api-key=23456';

    if (strpos($transaction->fullPath, "?") {

        $transaction->fullPath .= "&{$paramToAdd}";

    }

    else {

        $transaction->fullPath .= "?{$paramToAdd}";

    }

});

```

## Handling sessions

```

<?php
use Dredd\Hooks;

$stash = [];

Hooks::after("Auth > /remoteauto/userpass", function(&$transaction) use ($stash) {

```

```
$parsedBody = json_decode($transaction->real->body);

$stash['token'] = $parseBody->sessionId;
});

Hooks::beforeEach(function(&$transaction) use ($stash) {

    if ($transaction->token) {

        $transaction->request->headers->Cookie = "id=($stash['token'])s";
    }
});
```

## 2.13 Writing Dredd Hooks In Python

GitHub repository

Python hooks are using *Dredd's hooks handler socket interface*. For using Python hooks in Dredd you have to have *Dredd already installed*

### 2.13.1 Installation

```
$ pip install dredd_hooks
```

### 2.13.2 Usage

```
$ dredd apiary.apib http://127.0.0.1:3000 --language=python --hookfiles=./hooks*.py
```

### 2.13.3 API Reference

Module `dredd_hooks` imports following decorators:

1. `before_each`, `before_each_validation`, `after_each`
  - wraps a function and passes `Transaction` object as a first argument to it
2. `before`, `before_validation`, `after`
  - accepts `transaction name` as a first argument
  - wraps a function and sends a `Transaction` object as a first argument to it
3. `before_all`, `after_all`
  - wraps a function and passes an Array of `Transaction` objects as a first argument to it

Refer to [Dredd execution life-cycle](#) to find when is each hook function executed.

## Using Python API

Example usage of all methods in

```
import dredd_hooks as hooks

@hooks.before_all
def my_before_all_hook(transactions):
    print('before all')

@hooks.before_each
def my_before_each_hook(transaction):
    print('before each')

@hooks.before
def my_before_hook(transaction):
    print('before')

@hooks.before_each_validation
def my_before_each_validation_hook(transaction):
    print('before each validation')

@hooks.before_validation
def my_before_validation_hook(transaction):
    print('before validations')

@hooks.after
def my_after_hook(transaction):
    print('after')

@hooks.after_each
def my_after_each(transaction):
    print('after_each')

@hooks.after_all
def my_after_all_hook(transactions):
    print('after_all')
```

### 2.13.4 Examples

More complex examples are to be found in the Github repository [under the examples directory](#). If you want to share your own, don't hesitate and submit a PR.

### How to Skip Tests

Any test step can be skipped by setting skip property of the transaction object to true.

```
import dredd_hooks as hooks

@hooks.before("Machines > Machines collection > Get Machines")
def skip_test(transaction):
    transaction['skip'] = True
```

### Sharing Data Between Steps in Request Stash

If you want to test some API workflow, you may pass data between test steps using the response stash.

```
import json
import dredd_hooks as hooks

response_stash = {}

@hooks.after("Machines > Machines collection > Create Machine")
def save_response_to_stash(transaction):
    # saving HTTP response to the stash
    response_stash[transaction['name']] = transaction['real']

@hooks.before("Machines > Machine > Delete a machine")
def add_machine_id_to_request(transaction):
    # reusing data from previous response here
    parsed_body = json.loads(response_stash["Machines > Machines collection > Create_
↵Machine"])
    machine_id = parsed_body['id']
    # replacing id in URL with stashed id from previous response
    transaction['fullPath'] = transaction['fullPath'].replace('42', machine_id)
```

### Failing Tests Programmatically

You can fail any step by setting fail property on transaction object to true or any string with descriptive message.

```
import dredd_hooks as hooks

@hooks.before("Machines > Machines collection > Get Machines")
def fail_transaction(transaction):
    transaction['fail'] = "Some failing message"
```

### Modifying Transaction Request Body Prior to Execution

```
import json
import dredd_hooks as hooks

@hooks.before("Machines > Machines collection > Get Machines")
def add_value_to_body(transaction):
    # parse request body from API description
    request_body = json.loads(transaction['request']['body'])

    # modify request body here
    request_body['someKey'] = 'some new value'

    # stringify the new body to request
    transaction['request']['body'] = json.dumps(request_body)
```



## Adding or Changing URI Query Parameters to All Requests

```
import dredd_hooks as hooks

@hooks.before_each
def add_api_key(transaction):
    # add query parameter to each transaction here
    param_to_add = "api-key=23456"

    if '?' in transaction['fullPath']:
        transaction['fullPath'] = ''.join((transaction['fullPath'], "&", param_to_add))
    else:
        transaction['fullPath'] = ''.join((transaction['fullPath'], "?", param_to_add))
```

## Handling sessions

```
import json
import dredd_hooks as hooks

stash = {}

# hook to retrieve session on a login
@hooks.after('Auth > /remoteauth/userpass > POST')
def stash_session_id(transaction):
    parsed_body = json.loads(transaction['real']['body'])
    stash['token'] = parsed_body['sessionId']

# hook to set the session cookie in all following requests
@hooks.before_each
def add_session_cookie(transaction):
    if 'token' in stash:
        transaction['request']['headers']['Cookie'] = "id=" + stash['token']
```

## Remove trailing newline character in expected *plain text* bodies

```
import dredd_hooks as hooks

@hooks.before_each
def remove_trailing_newline(transaction):
    if transaction['expected']['headers']['Content-Type'] == 'text/plain':
        transaction['expected']['body'] = transaction['expected']['body'].rstrip()
```

## 2.14 Writing Dredd Hooks In Ruby

GitHub repository

Ruby hooks are using *Dredd's hooks handler socket interface*. For using Ruby hooks in Dredd you have to have *Dredd* already installed

### 2.14.1 Installation

```
$ gem install dredd_hooks
```

### 2.14.2 Usage

```
$ dredd apiary.apib http://127.0.0.1:3000 --language=ruby --hookfiles=./hooks*.rb
```

### 2.14.3 API Reference

Including module `Dredd::Hooks::Methods` expands current scope with methods

1. `@before_each`, `before_each_validation`, `after_each`
  - accepts a block as a first argument passing a `Transaction` object as a first argument
2. `before`, `before_validation`, `after`
  - accepts `transaction name` as a first argument
  - accepts a block as a second argument passing a `Transaction` object as a first argument of it
3. `before_all`, `after_all`
  - accepts a block as a first argument passing an Array of `Transaction` objects as a first argument

Refer to [Dredd execution lifecycle](#) to find when is each hook function executed.

#### Using Ruby API

Example usage of all methods in

```
include DreddHooks::Methods

before_all do |transactions|
  puts 'before all'
end

before_each do |transaction|
  puts 'before each'
end

before "Machines > Machines collection > Get Machines" do |transaction|
  puts 'before'
end

before_each_validation do |transaction|
  puts 'before each validation'
end

before_validation "Machines > Machines collection > Get Machines" do |transaction|
  puts 'before validations'
end

after "Machines > Machines collection > Get Machines" do |transaction|
  puts 'after'
end
```

```

end

after_each do |transaction|
  puts 'after_each'
end

after_all do |transactions|
  puts 'after_all'
end

```

## 2.14.4 Examples

### How to Skip Tests

Any test step can be skipped by setting skip property of the transaction object to true.

```

include DreddHooks::Methods

before "Machines > Machines collection > Get Machines" do |transaction|
  transaction['skip'] = true
end

```

### Sharing Data Between Steps in Request Stash

If you want to test some API workflow, you may pass data between test steps using the response stash.

```

require 'json'
include DreddHooks::Methods

response_stash = {}

after "Machines > Machines collection > Create Machine" do |transaction|
  # saving HTTP response to the stash
  response_stash[transaction['name']] = transaction['real']
do

before "Machines > Machine > Delete a machine" do |transaction|
  #reusing data from previous response here
  parsed_body = JSON.parse response_stash['Machines > Machines collection > Create_
↪Machine']
  machine_id = parsed_body['id']

  #replacing id in URL with stashed id from previous response
  transaction['fullPath'].gsub! '42', machine_id
end

```

### Failing Tests Programmatically

You can fail any step by setting fail property on transaction object to true or any string with descriptive message.

```
include DreddHooks::Methods

before "Machines > Machines collection > Get Machines" do |transaction|
  transaction['fail'] = "Some failing message"
end
```

### Modifying Transaction Request Body Prior to Execution

```
require 'json'
include DreddHooks::Methods

before "Machines > Machines collection > Get Machines" do |transaction|
  # parse request body from API description
  request_body = JSON.parse transaction['request']['body']

  # modify request body here
  request_body['someKey'] = 'some new value'

  # stringify the new body to request
  transaction['request']['body'] = request_body.to_json
end
```

### Adding or Changing URI Query Parameters to All Requests

```
include DreddHooks::Methods

hooks.before_each do |transaction|

  # add query parameter to each transaction here
  param_to_add = "api-key=23456"

  if transaction['fullPath'].include('?')
    transaction['fullPath'] += "&" + param_to_add
  else
    transaction['fullPath'] += "?" + param_to_add
  end
end
```

### Handling sessions

```
require 'json'
include DreddHooks::Methods

stash = {}

# hook to retrieve session on a login
hooks.after 'Auth > /remoteauth/userpass > POST' do |transaction|
  parsed_body = JSON.parse transaction['real']['body']
  stash['token'] = parsed_body['sessionId']
end

# hook to set the session cookie in all following requests
```

```
hooks.beforeEach do |transaction|
  unless stash['token'].nil?
    transaction['request']['headers']['Cookie'] = "id=" + stash['token']
  end
end
```

### Remove trailing newline character for in expected plain text bodies

```
include DreddHooks::Methods

before_each do |transaction|
  if transaction['expected']['headers']['Content-Type'] == 'text/plain'
    transaction['expected']['body'] = transaction['expected']['body'].gsub(/^\s+|\s+$/
    ↪g, "")
  end
end
```

## 2.15 Writing Dredd Hooks In Rust

GitHub repository

Rust hooks are using *Dredd's hooks handler socket interface*. For using Rust hooks in Dredd you have to have *Dredd already installed*. The Rust library is called `dredd-hooks` and the corresponding binary `dredd-hooks-rust`.

### 2.15.1 Installation

```
$ cargo install dredd-hooks
```

### 2.15.2 Usage

Using Dredd with Rust is slightly different to other languages, as a binary needs to be compiled for execution. The `--hookfiles` flags should point to compiled hook binaries. See below for an example `hooks.rs` file to get an idea of what the source file behind the Rust binary would look like.

```
$ dredd apiary.apib http://127.0.0.1:3000 --server=./rust-web-server-to-test --
↪language=rust --hookfiles=./hook-file-binary
```

### 2.15.3 API Reference

In order to get a general idea of how the Rust Hooks work, the main executable from the package `dredd-hooks` is an HTTP Server that Dredd communicates with and an RPC client. Each hookfile then acts as a corresponding RPC server. So when Dredd notifies the Hooks server what transaction event is occurring the hooks server will execute all registered hooks on each of the hookfiles RPC servers.

You'll need to know a few things about the `HooksServer` type in the `dredd-hooks` package.

1. The `HooksServer` type is how you can define event callbacks such as `beforeEach`, `afterAll`, etc..
2. To get a `HooksServer` struct you must do the following;

```
extern crate dredd_hooks;

use dredd_hooks::{HooksServer};

fn main() {
    let mut hooks = HooksServer::new();

    // Define all your event callbacks here

    // HooksServer::start_from_env will block and allow the RPC server
    // to receive messages from the main `dredd-hooks-rust` process.
    HooksServer::start_from_env(hooks);
}
```

1. Callbacks receive a `Transaction` instance, or an array of them.

### Runner Callback Events

The `HooksServer` type has the following callback methods.

1. `before_each`, `before_each_validation`, `after_each`
  - accepts a function as a first argument passing a `Transaction` object as a first argument
2. `before`, `before_validation`, `after`
  - accepts `transaction name` as a first argument
  - accepts a function as a second argument passing a `Transaction` object as a first argument of it
3. `before_all`, `after_all`
  - accepts a function as a first argument passing a `Vec` of `Transaction` objects as a first argument

Refer to [Dredd execution lifecycle](#) to find when each hook callback is executed.

### Using the Rust API

Example usage of all methods.

```
extern crate dredd_hooks;

use dredd_hooks::{HooksServer};

fn main() {
    let mut hooks = HooksServer::new();
    hooks.before("/message > GET", Box::new(move |tr| {
        println!("before hook handled");
        tr
    }));
    hooks.after("/message > GET", Box::new(move |tr| {
        println!("after hook handled");
        tr
    }));
    hooks.before_validation("/message > GET", Box::new(move |tr| {
        println!("before validation hook handled");
        tr
    }));
    hooks.before_all(Box::new(move |tr| {
```

```

        println!("before all hook handled");
        tr
    });
    hooks.after_all(Box::new(move |tr| {
        println!("after all hook handled");
        tr
    }));
    hooks.before_each(Box::new(move |tr| {
        println!("before each hook handled");
        tr
    }));
    hooks.before_each_validation(Box::new(move |tr| {
        println!("before each validation hook handled");
        tr
    }));
    hooks.after_each(Box::new(move |tr| {
        println!("after each hook handled");
        tr
    }));
    HooksServer::start_from_env(hooks);
}

```

## 2.15.4 Examples

### How to Skip Tests

Any test step can be skipped by setting the value of the `skip` field of the `Transaction` instance to `true`.

```

extern crate dredd_hooks;

use dredd_hooks::{HooksServer};

fn main() {
    let mut hooks = HooksServer::new();

    // Runs only before the "/message > GET" test.
    hooks.before("/message > GET", Box::new(|mut tr| {
        // Set the skip flag on this test.
        tr.insert("skip".to_owned(), true.into());
        // Hooks must always return the (modified) Transaction(s) that were passed in.
        tr
    }));
    HooksServer::start_from_env(hooks);
}

```

### Failing Tests Programmatically

You can fail any step by setting the value of the `fail` field of the `Transaction` instance to `true` or any string with a descriptive message.

```

extern crate dredd_hooks;

use dredd_hooks::{HooksServer};

```

```
fn main() {
    let mut hooks = HooksServer::new();
    hooks.before("/message > GET", Box::new(|mut tr| {
        // .into() can be used as an easy way to convert
        // your value into the desired Json type.
        tr.insert("fail".to_owned(), "Yay! Failed!".into());
        tr
    }));
    HooksServer::start_from_env(hooks);
}
```

### Modifying the Request Body Prior to Execution

```
extern crate dredd_hooks;

use dredd_hooks::{HooksServer};

fn main() {
    let mut hooks = HooksServer::new();
    hooks.before("/message > GET", Box::new(|mut tr| {
        // Try to access the "request" key as an object.
        // (This will panic should the "request" key not be present.)
        tr["request"].as_object_mut().unwrap()
            .insert("body".to_owned(), "Hello World!".into());

        tr
    }));
    HooksServer::start_from_env(hooks);
}
```

## 2.16 Writing Dredd hook handler for new language

### 2.16.1 Dredd hooks handler client

Dredd comes with concept of hooks language abstraction bridge via simple TCP socket.

When you run Dredd with `--language` argument, it runs the command in argument and tries to connect to `http://127.0.0.1:61321`. If connection to the hook handling server wasn't successful, it exits with exit code 3.

Dredd internally registers a function for each **type of hooks** and when this function is executed it assigns execution `uuid` to that event, serializes received function parameters (a **Transaction object** or an Array of it), sends it to the TCP socket to be handled (executed) in other language and waits until message with same `uuid` is received. After data reception it assigns received data back to the transaction, so other language can interact with transactions same way like *native Node.js hooks*.

### 2.16.2 Language agnostic test suite

Dredd hooks language abstraction bridge comes with the **language agnostic test suite**. It's written in Gherkin - language for writing **Cucumber** scenarios and **Aruba CLI testing framework** and it tests your new language handler integration with CLI Dredd and expected behavior from user's perspective.



### 2.16.3 What to implement

If you want to write a hook handler for your language you will have to implement:

- CLI Command running TCP socket server
  - Must return message `Starting to stdout`
- Hooks API in your language for registering code being executed during the Dredd lifecycle:
  - before all transactions
  - before each transaction
  - before transaction
  - before each transaction validation
  - before transaction validation
  - after transaction
  - after each transaction
  - after all transactions
- When CLI command is executed
  - It loads files passed in alphabetical order with paths resolved to absolute form
    - \* It exposes API similar to those in *Ruby*, *Python* and *Node.js* to each loaded file
    - \* It registers functions declared in files for later execution
  - starts a TCP socket server and starts listening on `http://127.0.0.1:61321`.
- When any data is received by the server
  - Adds every received character to a buffer
  - When delimiting newline (`\n`) character is received
    - \* It parses the *message* in the buffer as JSON
    - \* It looks for `event` key in received object and executes appropriate registered hooks functions
  - When the hook function is being executed
    - \* It passes value of `data` key from received object to the executed function
    - \* Hook function is able to modify data
  - When function was executed
    - \* It should serialize message to JSON
    - \* Send the serialized message back to the socket with same `uuid` as received
    - \* Send a newline character as message delimiter

### 2.16.4 Termination

When the testing is done, Dredd signals the hook handler process to terminate. This is done repeatedly with delays. When termination timeout is over, Dredd loses its patience and kills the process forcefully.

- **retry delays** can be configured by `--hooks-worker-term-retry`
- **timeout** can be configured by `--hooks-worker-term-timeout`

On Linux or macOS, Dredd uses the `SIGTERM` signal to tell the hook handler process it should terminate. On Windows, where signals do not exist, Dredd sends the `END OF TEXT` character (`\u0003`, which is ASCII representation of Ctrl+C) to standard input of the process.

### 2.16.5 TCP Socket Message format

- transaction (object)
  - `uuid`: `234567-asdfghjkl` (string) - Id used for event unique identification on both server and client sides
  - `event`: `event` (enum) - Event type
    - \* `beforeAll` (string) - Signals the hook handler to run the `beforeAll` hooks
    - \* `beforeEach` (string) - Signals the hook handler to run the `beforeEach` and `before` hooks
    - \* `beforeEachValidation` (string) - Signals the hook handler to run the `beforeEachValidation` and `beforeValidation` hooks
    - \* `afterEach` (string) - Signals the hook handler to run the `after` and `afterEach` hooks
    - \* `afterAll` (string) - Signals the hook handler to run the `afterAll` hooks
  - `data` (enum) - Data passed as a argument to the function
    - \* (object) - Single Transaction object
    - \* (array) - An array of Transaction objects, containing all transactions in the API description. Sent for `beforeAll` and `afterAll` events

### 2.16.6 Configuration Options

There are several configuration options, which can help you during development:

- `--hooks-worker-timeout` - docs
- `--hooks-worker-connect-timeout` - docs
- `--hooks-worker-connect-retry` - docs
- `--hooks-worker-after-connect-wait` - docs
- `--hooks-worker-term-timeout` - docs
- `--hooks-worker-term-retry` - docs
- `--hooks-worker-handler-host` - docs
- `--hooks-worker-handler-port` - docs

### 2.16.7 Need help? No problem!

If you have any questions, please:

- Have a look at the [Ruby](#), [Python](#), [Perl](#), and [PHP](#) hook handlers codebase for inspiration
- If you're writing a hook handler for a compiled language, check out the [Go](#) implementation
- File an [issue](#) in [Dredd repository](#)

## 2.17 Data Structures

Documentation of various data structures in both `Gavel.js` and `Dredd`. `MSON` notation is used to describe the data structures.

### 2.17.1 Transaction (object)

Transaction object is passed as a first argument to *hook functions* and is one of the main public interfaces in `Dredd`.

- `id`: GET (200) /greetings - identifier for this transaction
- `name`: `./api-description.apib > My API > Greetings > Hello, world! > Retrieve Message > Example 2` (string) - reference to the transaction definition in the original API description document (see also [Dredd Transactions](#))
- `origin` (object) - reference to the transaction definition in the original API description document (see also [Dredd Transactions](#))
  - `filename`: `./api-description.apib` (string)
  - `apiName`: `My Api` (string)
  - `resourceGroupName`: `Greetings` (string)
  - `resourceName`: `Hello, world!` (string)
  - `actionName`: `Retrieve Message` (string)
  - `exampleName`: `Example 2` (string)
- `host`: `127.0.0.1` (string) - server hostname without port number
- `port`: `3000` (number) - server port number
- `protocol`: `https`: (enum[string]) - server protocol
  - `https`: (string)
  - `http`: (string)
- `fullPath`: `/message` (string) - expanded [URI Template](#) with parameters (if any) used for the HTTP request `Dredd` performs to the tested server
- `request` (object) - the HTTP request `Dredd` performs to the tested server, taken from the API description
  - `body`: `Hello world!\n` (string)
  - `headers` (object) - keys are HTTP header names, values are HTTP header contents
  - `uri`: `/message` (string) - request URI as it was written in API description
  - `method`: `POST` (string)
- `expected` (object) - the HTTP response `Dredd` expects to get from the tested server
  - `statusCode`: `200` (string)
  - `headers` (object) - keys are HTTP header names, values are HTTP header contents
  - `body` (string)
  - `bodySchema` (object) - JSON Schema of the response body
- `real` (object) - the HTTP response `Dredd` gets from the tested server (present only in `after hooks`)
  - `statusCode`: `200` (string)

- headers (object) - keys are HTTP header names, values are HTTP header contents
- body (string)
- skip: `false` (boolean) - can be set to `true` and the transaction will be skipped
- fail: `false` (enum) - can be set to `true` or `string` and the transaction will fail
  - (string) - failure message with details why the transaction failed
  - (boolean)
- test (*Transaction Test*) - test data passed to Dredd's reporters
- results (*Transaction Results*) - testing results

### 2.17.2 Transaction Test (object)

- start (Date) - start of the test
- end (Date) - end of the test
- duration (number) - duration of the test in milliseconds
- startedAt (number) - unix timestamp, `transaction.startedAt`
- title (string) - `transaction.id`
- request (object) - `transaction.request`
- actual (object) - `transaction.real`
- expected (object) - `transaction.expected`
- status (enum) - whether the validation passed or not, defaults to empty string
  - `pass` (string)
  - `fail` (string)
  - `skip` (string)
- message (string) - concatenation of all messages from all *Gavel Errors* in `results` or Dredd's custom message (e.g. "failed in before hook")
- results (Dredd's `transaction.results`)
- valid (boolean)
- origin (object) - `transaction.origin`

### 2.17.3 Transaction Results (object)

This is a cousin of the *Gavel Validation Result*.

- general (object) - contains Dredd's custom messages (e.g. "test was skipped"), formatted the same way like those from Gavel
  - results (array[*Gavel Error*])
- statusCode (*Gavel Validator Output*)
- headers (*Gavel Validator Output*)
- body (*Gavel Validator Output*)

### 2.17.4 Gavel Validation Result (object)

Can be seen also [here](#).

- `statusCode` (*Gavel Validator Output*)
- `headers` (*Gavel Validator Output*)
- `body` (*Gavel Validator Output*)
- `version` (string) - version number of the Gavel Validation Result structure

### 2.17.5 Gavel Validator Output (object)

Can be seen also [here](#).

- `results` (array[*Gavel Error*])
- `realType` (string) - media type
- `expectedType` (string) - media type
- `validator` (string) - validator class name
- `rawData` (enum) - raw output of the validator, has different structure for every validator and is saved and used in Apiary to render graphical diff by [gavel2html](#)
  - (*JsonSchema Validation Result*)
  - (*TextDiff Validation Result*)

### 2.17.6 JsonSchema Validation Result (object)

The validation error is based on format provided by [Amanda](#) and is also “documented” [here](#). Although for validation of draft4 JSON Schema Gavel uses `tv4` library, the output then gets reshaped into the structure of Amanda’s errors.

This validation result is returned not only when validating against [JSON Schema](#), but also when validating against JSON example or when validating HTTP headers.

- `length`: 0 (number, default) - number of error properties
- `errorMessages` (object) - doesn’t seem to ever contain anything or be used for anything
- `0` (object) - validation error details, property is always a string containing a number (0, 1, 2, ...)
- `property` (array[string]) - path to the problematic property in format of `json-pointer’s parse()` output
- `propertyValue` (mixed) - real value of the problematic property (can be also undefined etc.)
- `attributeName: enum, required` (string) - name of the relevant JSON Schema attribute, which triggered the error
- `attributeValue` (mixed) - value of the relevant JSON Schema attribute, which triggered the error
- `message` (string) - error message (in case of `tv4` it contains [JSON Pointer](#) to the problematic property and for both [Amanda](#) and `tv4` it can directly mention property names and/or values)
- `validator: enum` (string) - the same as `attributeName`
- `validatorName: error, enum` (string) - the same as `attributeName`
- `validatorValue` (mixed) - the same as `attributeValue`

### 2.17.7 TextDiff Validation Result (string)

Block of text which looks extremely similar to the standard GNU diff/patch format. Result of the `patch_toText()` function of the `google-diff-match-patch` library.

### 2.17.8 Gavel Error (object)

Can also be seen as part of Gavel Validator Output [here](#).

- pointer (string) - JSON Pointer path
- severity (string) - severity of the error
- message (string) - error message

### 2.17.9 Apiary Reporter Test Data (object)

- testRunId (string) - ID of the *test run*, recieved from Apiary
- origin (object) - *test*.origin
- duration (number) - duration of the test in milliseconds
- result (string) - *test*.status
- startedAt (number) - *test*.startedAt
- resultData (object)
  - request (object) - *test*.request
  - realResponse (object) - *test*.actual
  - expectedResponse (object) - *test*.expected
  - result (*Transaction Results*) - *test*.results

### 2.17.10 Internal Apiary Data Structures

These are private data structures used in Apiary internally and they are documented incompletely. They're present in this document just to provide better insight on what and how Apiary internally saves. It is closely related to what you can see in documentation for [Apiary Tests API for anonymous test reports](#) and [Apiary Tests API for authenticated test reports](#).

#### Apiary Test Run (object)

Also known as `stats` in Dredd's code.

- result
  - tests: 0 (number, default) - total number of tests
  - failures: 0 (number, default)
  - errors: 0 (number, default)
  - passes: 0 (number, default)
  - skipped: 0 (number, default)

- start: 0 (number, default)
- end: 0 (number, default)
- duration: 0 (number, default)

### Apiary Test Step (object)

- resultData
  - request (object) - *test.request*
  - realResponse (object) - *test.actual*
  - expectedResponse (object) - *test.expected*
  - result (*Transaction Results*) - *test.results*

## 2.18 Contributing Guidelines

### 2.18.1 Quick Start

#### Ideas

- File an [issue](#).
- Explain why you want the feature. How does it help you? What for do you want the feature?

#### Bugs

- File an [issue](#).
- Ideally, write a failing test and send it as a Pull Request.

#### Coding

- Dredd is written in [CoffeeScript](#).
- Dredd uses *Semantic Release and Conventional Changelog*.

#### Recommended Workflow

1. Fork Dredd.
2. Create a feature branch.
3. Write tests.
4. Write code.
5. Lint what you created: `npm run lint`
6. Send a Pull Request.
7. Make sure [test coverage](#) didn't drop and all CI builds are passing.

### Semantic Release and Conventional Changelog

Releasing of new Dredd versions to npm is automatically managed by [Semantic Release](#). Semantic Release makes sure correct version numbers get bumped according to the **meaning** of your changes once your PR gets merged to `master`.

To make it work, it's necessary to follow [Conventional Changelog](#). That basically means all commit messages in the project should follow a particular format:

```
<type>: <subject>
```

Where `<type>` is:

- `feat` - New functionality added
- `fix` - Broken functionality fixed
- `perf` - Performance improved
- `docs` - Documentation added/removed/improved/...
- `chore` - Package setup, CI setup, ...
- `refactor` - Changes in code, but no changes in behavior
- `test` - Tests added/removed/improved/...

In the rare cases when your changes break backwards compatibility, the message must include string `BREAKING CHANGE:.` That will result in bumping the major version.

Seems hard?

- See [existing commits](#) as a reference
- [Commitizen CLI](#) can help you to create correct commit messages
- `npm run lint` validates format of your messages

## 2.18.2 Handbook for Contributors and Maintainers

### Maintainers

[Apiary](#) is the main author and maintainer of Dredd's [upstream repository](#). Currently responsible people are:

- [@netmilk](#) - product decisions, feature requests
- [@honzajavorek](#) - lead of development

### Programming Language

Dredd is written in [CoffeeScript](#) and is meant to be ran on server using Node.js. Before publishing to npm registry, it is compiled to plain ES5 JavaScript code (`throwaway lib` directory).

While tests are compiled on-the-fly thanks to CoffeeScript integration with the Mocha test framework, they actually need the code to be also pre-compiled every time because some integration tests use code linked from `lib`. This is certainly a flaw and it slows down day-to-day development, but unless we find out how to get rid of the `lib` dependency, it's necessary.

Also mind that CoffeeScript is production dependency (not dev dependency), because it's needed not only for compiling Dredd package before uploading to npm, but also for running user-provided hooks written in CoffeeScript.



## Compiled vs pure JavaScript

Dredd uses [Drafter](#) for parsing [API Blueprint](#) documents. Drafter is written in C++11 and needs to be compiled during installation. Because that can cause a lot of problems in some environments, there's also pure JavaScript version of the parser, [drafter.js](#). Drafter.js is fully equivalent, but it can have slower performance. Therefore there's [drafter-npm](#) package, which tries to compile the C++11 version of the parser and uses the JavaScript equivalent in case of failure.

Dredd depends on the [drafter-npm](#) package. That's the reason why you can see `node-gyp` errors and failures during the installation process, even though when it's done, Dredd seems to normally work and correctly parses API Blueprint documents.

## Forcing the JavaScript version

The `--no-optional` option forces the JavaScript version of Drafter and avoids any compilation attempts when installing Dredd:

```
$ npm install -g dredd --no-optional
```

## Troubleshooting the compilation

If you need the performance of the C++11 parser, but you are struggling to get it installed, it's usually because of the following problems:

- **Your machine is missing a C++11 compiler.** See how to fix this on [Windows](#) or [Travis CI](#).
- **npm was used with Python 3.** `node-gyp`, which performs the compilation, doesn't support Python 3. If your default Python is 3 (see `python --version`), [tell npm to use an older version](#).

## Supported Node.js Versions

Given the [table with LTS schedule](#), only versions marked as **Maintenance** or **Active** are supported, until their **Maintenance End**. The testing matrix of Dredd's CI builds must contain all currently supported versions and must not contain any unsupported versions. The same applies for the underlying libraries, such as [Dredd Transactions](#) or [Gavel.js](#).

In following files the latest supported Node.js version should be used:

- `appveyor.yml` - Windows CI builds
- `docs/install-node.sh` - ReadTheDocs docs builds

## Versioning

Dredd follows [Semantic Versioning](#). To ensure certain stability of Dredd installations (e.g. in CI builds), users can pin their version. They can also use release tags:

- `npm install dredd` - Installs the latest published version including experimental pre-release versions.
- `npm install dredd@stable` - Skips experimental pre-release versions.

When releasing, make sure you respect the tagging:

- To release pre-release, e.g. `42.1.0-pre.7`, use just `npm publish`.
- To release any other version, e.g. `42.1.0`, use `npm publish && npm dist-tag add dredd@42.1.0 stable`.

Releasing process for standard versions is currently automated by [Semantic Release](#). Releasing process for pre-releases is not automated and needs to be done manually, ideally from a special git branch.

### Testing

Use `npm test` to run all tests. Dredd uses [Mocha](#) as a test framework. It's default options are in the `test/mocha.opts` file.

### Windows

Dredd is tested on the [AppVeyor](#), a Windows-based CI. There are still [several known limitations](#) when using Dredd on Windows, but the intention is to support it without any compromises. Any help with fixing problems on Windows is greatly appreciated!

### Linting

Dredd uses [coffeelint](#) to lint the CoffeeScript codebase. There is a plan to converge with Apiary's [CoffeeScript Style Guide](#), but as most of the current code was written before the style guide was introduced, it's a long run. The effective settings are in the `coffeelint.json` file.

Linters is optional for local development to make easy prototyping and work with unpolished code, but it's enforced on CI level. It is recommended you integrate [coffeelint](#) with your favorite editor so you see violations immediately during coding.

### Changelog

Changelog is in form of [GitHub Releases](#). Currently it's automatically generated by [Semantic Release](#). See *above* to learn about how it works.

### Documentation

Dredd's documentation is written in [Markdown](#) using [Sphinx](#). [ReadTheDocs](#) is used to build and publish the documentation:

- <https://dredd.readthedocs.io> - preferred long URL
- <http://dredd.rtdf.org> - preferred short URL

Source of the documentation can be found in the `docs` directory. To render Dredd's documentation on your computer, you need Python 3 and Node.js installed.

### Installation and Development

1. Make sure `node` is an executable and `npm install` has been done for the Dredd directory. Extensions to the docs are written in Node.js and Sphinx needs to have a way to execute them.
2. [Get Python 3](#). On macOS, run `brew install python3`. [ReadTheDocs](#) build the docs with Python 3.5, so make sure you have that or higher.
3. Create a [virtual environment](#) and activate it:

```
python3 -m venv ./venv
. ./env/bin/activate
```

4. Install dependencies for the docs: `pip install -r docs/requirements.txt`

Once installed, you may use following commands:

- `npm run docs:build` - Builds the documentation
- `npm run docs:serve` - Runs live preview of the documentation on `http://127.0.0.1:8000`

## Installation on ReadTheDocs

The final documentation gets deployed on the [ReadTheDocs](#). The service, however, does not support Node.js. Therefore on ReadTheDocs, the `conf.py` configuration file for Sphinx runs `docs/install-node.sh`, which installs Node.js locally, using `npm`.

## ToC and Markdown

Traditionally, Sphinx only supported the `reStructuredText` format. Thanks to the `recommonmark` project it's possible to use also `Markdown`, *almost* as a format native to Sphinx. Dredd's docs are using the `AutoStructify` extension to be able to specify `toctree` and other stuff specific to `reStructuredText`. The ToC is generated from the `Contents` section in the `docs/index.md` file.

## Node.js Extensions

There are some extensions hooked into the build process of `Sphinx`, modifying how the documents are processed. They're written in Node.js, because:

- It's better to have them in the same language as Dredd.
- This way they're able to import source files (e.g. `src/options.coffee`).

By default, `Hercule` is attached as an extension, which means you can use the `:[Title](link.md)` syntax for including other Markdown files. All other extensions are custom and are automatically loaded from the `docs/_extensions` directory.

The extension is expected to be a `.js` or `.coffee` script file, which takes `docname` as an argument, reads the Markdown document from `stdin`, modifies it, and then prints it to `stdout`. When in need of templating, extensions are expected to use the bundled `ect` templating engine.

## Local References

Currently the `recommonmark` project has still some limitations in how references to local files work. That's why Dredd's docs have a custom implementation, which also checks whether the destination exists and fails the build in case of broken link. You can use following syntax:

- `[Title](link.md)` to link to other documents
- `[Title](link.md#section)` to link to sections of other documents

Any `id` HTML attributes generated for headings or manual `<a name="section"></a>` anchors are considered as valid targets. While this feels very natural for a seasoned writer of Markdown, mind that it is much more error prone than `reStructuredText`'s references.

### Redirects

Redirects are documented in the `docs/redirects.yml` file. They need to be manually set in the [ReadTheDocs administration](#). It's up to Dredd maintainers to keep the list in sync with reality.

You can use the `rtd-redirects` tool to programmatically upload the redirects from `docs/redirects.yml` to ReadTheDocs admin interface.

### Symlinked Contributing Docs

The `docs/contributing.md` file is a [symbolic link](#) to the `.github/CONTRIBUTING.md` file, where the actual content lives. This is to be able to serve the same content also as [GitHub contributing guidelines](#) when someone opens a Pull Request.

### Coverage

Dredd strives for as much test coverage as possible. [Coveralls](#) help us to monitor how successful we are in achieving the goal. If a Pull Request introduces drop in coverage, it won't be accepted unless the author or reviewer provides a good reason why an exception should be made.

The Travis CI build uses following commands to deliver coverage reports:

- `npm run test:coverage` - Tests Dredd and creates the `cov.info` file
- `npm run coveralls` - Uploads the `cov.info` file to Coveralls

The first mentioned command goes like this:

1. `coffee-coverage` is used to instrument the CoffeeScript code.
2. Instrumented code is copied into a separate directory. We run tests in the directory using Mocha with a special `lcov` reporter, which gives us information about which lines were executed in a standard `lcov` format.
3. Because some integration tests execute the `bin/dredd` script in a subprocess, we collect the coverage stats also in this file. The results are appended to a dedicated `lcov` file.
4. All `lcov` files are then merged into one using `lcov-result-merger` and sent to Coveralls.

### Notes

- Hand-made combined Mocha reporter is used to achieve running tests and collecting coverage at the same time.
- Both Dredd code and the combined reporter decide whether to collect coverage or not according to contents of the `COVERAGE_DIR` environment variable, which sets the directory for temporary `LCOV` files created during coverage collection. (If set, collecting takes place.)

### Hacking Apiary Reporter

If you want to build something on top of the Apiary Reporter, note that it uses a public API described in following documents:

- [Apiary Tests API for anonymous test reports](#)
- [Apiary Tests API for authenticated test reports](#)

Following data are sent over the wire to Apiary:

- [Apiary Reporter Test Data](#)

There is also one environment variable you could find useful:

- `APIARY_API_URL='https://api.apiary.io'` - Allows to override host of the Apiary Tests API.

### **Misc Tips**

- When using long CLI options in tests or documentation, please always use the notation with `=`. For example, use `--path=/dev/null`, not `--path /dev/null`. While both should work, the version with `=` feels more like standard GNU-style long options and it makes arrays of arguments for `spawn` more readable.
- Using `127.0.0.1` (in code, tests, documentation) is preferred over `localhost` (see [#586](#)).
- Prefer explicit `<br>` tags instead of [two spaces](#) at the end of the line when writing documentation in Markdown.



## CHAPTER 3

---

### Useful Links

---

- [GitHub Repository](#)
- [Bug Tracker](#)
- [Changelog](#)





## CHAPTER 4

---

### Example Applications

---

- Express.js
- Ruby on Rails