Dredd

Release latest

Apr 13, 2019

Contents

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No more outdated API Documentation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.

CHAPTER 1

Features

1.1 Supported API Description Formats

- API Blueprint
- OpenAPI 2 (formerly known as Swagger)
- OpenAPI 3 (experimental, contributions welcome!)

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, ...

CHAPTER 2

Contents

2.1 Installation

There are several options how to run Dredd on your machine or in your *Continuous Integration*.

2.1.1 Docker

If you are familiar with Docker, you can get started with Dredd quickly by using the ready-made apiaryio/dredd image. Specifics of running Dredd inside Docker are:

- you won't be able to use the `--server` option (see *Docker Compose*)
- setting up non-JavaScript *hooks* is less straightforward (see *Hooks inside Docker*)

macOS, Linux

Following line runs the `dredd` command using the apiaryio/dredd Docker image:

```
$ docker run -it -v $PWD:/api -w /api apiaryio/dredd dredd
```

As an example of how to pass arguments, following line runs the `dredd init` command:

```
$ docker run -it -v $PWD:/api -w /api apiaryio/dredd dredd init
```

When testing a service running on host (e.g. `localhost:8080`), you need to use `--network host` parameter in Docker command. If you are using Docker for Mac, you should use host.docker.internal instead of 127.0.0.1/localhost.

Windows

Following line runs the `dredd` command using the apiaryio/dredd Docker image:

```
C:\Users\Susan> docker run -it -v ${pwd}:/api -w /api apiaryio/dredd dredd
```

As an example of how to pass arguments, following line runs the `dredd init` command:

```
C:\Users\Susan> docker run -it -v ${pwd}:/api -w /api apiaryio/dredd dredd init
```

Docker Compose

Inside Docker it's impossible for Dredd to manage child processes, so the `--server` and `--language` options won't work properly.

Instead, you should have separate containers for each process and run them together with Dredd using Docker Compose. You can use –abort-on-container-exit and –exit-code-from with Docker Compose to manage the tear down of all the other containers when the Dredd tests finish.

2.1.2 npm

Dredd is a command-line application written in JavaScript (to be more precise, in Node.js) and as such can be installed using npm.

Installing Node.js and npm

macOS

- If you're using Homebrew, run `brew install node`
- Otherwise download Node.js from the official website and install it using the downloaded installer
- Make sure both `node --version` and `npm --version` work in your Terminal
- Node.js needs to be at least version 6

Linux

- Install Node.js as a system package
- In case your Linux distribution calls the Node.js binary `nodejs`, please follow this advice to have it as `node` instead
- Make sure both `node --version` and `npm --version` work in your Terminal
- Node.js needs to be at least version 6

Windows

- Download Node.js from the official website and install it using the downloaded installer
- Make sure both `node --version` and `npm --version` work in your Command Prompt
- Node.js needs to be at least version 6

Note: If your internet connection is restricted (VPN, firewall, proxy), you need to configure npm:

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

Otherwise you'll get similar errors during Dredd installation:

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

Later be sure to read *how to set up Dredd to correctly work with proxies*.

Installing Dredd

Now that you have everything prepared, you can finally run npm to install Dredd:

```
npm install dredd --global
```

Note: If you get `EACCES` permissions errors, try one of the officially recommended solutions. In the worst case, you can run the command again with `sudo`.

You can verify Dredd is correctly installed by printing its version number:

```
dredd --version
```

Now you can *start using Dredd*!

Adding Dredd as a dev dependency

If your API project is also an npm package, you may want to add Dredd as a dev dependency instead of installing it globally.

- Make sure your project is an npm package with a `package.json` file
- In the root of the project run `npm install dredd --save-dev`
- Once the installation is complete, you can run Dredd from the root of the project as `npx dredd`

This is how Dredd is installed in the dredd-example repository, so you may want to see it for inspiration.

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*.

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
- OpenAPI 2 (formerly known as Swagger)

API Blueprint

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!"}
```

OpenAPI 2

If you choose OpenAPI 2, 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!'});
})
```

(continues on next page)

(continued from previous page)

```
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:

API Blueprint

```
$ dredd api-description.apib http://127.0.0.1:3000
```

OpenAPI 2

```
$ dredd api-description.yml http://127.0.0.1:3000
```

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 the Gavel library. Please refer to Gavel's 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 OpenAPI 2, headers are taken from `response.headers` (spec). HTTP headers significant for content negotiation are inferred according to following rules:

- `produces` (spec) 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 4 and Draft 3) 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, which is responsible for validation in Dredd, automatically infers some basic expectations described below.

This order exactly follows the API Blueprint specification.

OpenAPI 2

1. `response.schema` (spec) - provided JSON Schema will be used.
2. `response.examples` (spec) with sample JSON payload - Gavel, 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 OpenAPI 2 allow usage of URI templates (API Blueprint fully implements **RFC 6570**, OpenAPI 2 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, in OpenAPI 2 available as the `x-example` vendor extension property (*docs*).
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 OpenAPI 2 documents, HTTP headers are inferred from `"in": "header"` parameters (spec). HTTP headers significant for content negotiation are inferred according to following rules:

- `consumes` (spec) is propagated as request's `Content-Type` header.
- `produces` (spec) 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`.

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.

OpenAPI 2

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

If body parameter has `schema.example` (spec), it is used as a raw JSON sample for the request body. If it's not present, Dredd's OpenAPI 2 adapter generates sample values from the JSON Schema provided in the `schema` (spec) 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*:

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

hooks.beforeEachValidation((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 **RFC 7231** 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.

OpenAPI 2

The OpenAPI 2 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` (spec) and `consumes` (spec), 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 OpenAPI 2 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 *`api-description-document`* 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 (OpenAPI 2) 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('./lib/db');

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

afterEach(function(transaction) {
  db.cleanUp();
```

(continues on next page)

(continued from previous page)

```
});

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

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

OpenAPI 2

```
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:
```

(continues on next page)

(continued from previous page)

```
        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('./lib/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 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.

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
```

(continues on next page)

(continued from previous page)

```
# 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('./lib/db');

stash = {}

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

(continues on next page)

(continued from previous page)

```
// 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
})
```

OpenAPI 2 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
```

(continues on next page)

(continued from previous page)

```
          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('./lib/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 OpenAPI 2 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 OpenAPI 2 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` (spec), which doesn't allow additional properties.
- In API Blueprint's Schema sections and in OpenAPI 2 you can use `additionalProperties: false` (spec) 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` (spec).
- In API Blueprint's Schema sections and in OpenAPI 2 you can use `required` (spec), 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` (spec), which doesn't allow array items of a different structure then specified.
- In API Blueprint's Schema sections and in OpenAPI 2 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` (spec), which turns the sample value into a required value. You can also use `enum` (spec) to provide a set of possible values.
- In API Blueprint's Schema sections and in OpenAPI 2 you can use `enum` (spec) 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.

`.circleci/config.yml` Configuration File for CircleCI

```
version: 2
jobs:
  build:
    docker:
      - image: circleci/node:latest
    steps:
      - checkout
      - run: npm install dredd@x.x.x --global
      - run: dredd apiary.apib http://127.0.0.1:3000
```

`.travis.yml` Configuration File for Travis CI

```
before_install:
  - npm install dredd@x.x.x --global
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 the *`--user`* option 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 by the *--header* option:

```
--header="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
```

(continues on next page)

(continued from previous page)

```
          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 Working with Images and other Binary Bodies

The API description formats generally do not provide a way to describe binary content. The easiest solution is to describe only the media type, to *leave out the body*, and to handle the rest using *Hooks*.

Binary Request Body

API Blueprint

```
FORMAT: 1A

# Images API

## Resource [/image.png]

### Send an Image [PUT]

+ Request (image/png)

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

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

OpenAPI 2

```
swagger: "2.0"
info:
  version: "1.0"
  title: Images API
schemes:
  - http
consumes:
  - image/png
produces:
  - application/json
paths:
  /image.png:
    put:
      parameters:
        - name: binary
          in: body
          required: true
          schema:
            type: string
            format: binary
      responses:
        200:
          description: 'Test OK'
          examples:
            application/json; charset=utf-8:
              test: 'OK'
```

Hooks

In hooks, you can populate the request body with real binary data. The data must be in a form of a Base64-encoded string.

```
const hooks = require('hooks');
const fs = require('fs');
const path = require('path');

hooks.beforeEach((transaction, done) => {
  const buffer = fs.readFileSync(path.join(__dirname, '../image.png'));
  transaction.request.body = buffer.toString('base64');
  transaction.request.bodyEncoding = 'base64';
  done();
});
```

Binary Response Body

API Blueprint

```
FORMAT: 1A

# Images API

## Resource [/image.png]

### Retrieve Representation [GET]

+ Response 200 (image/png)
```

OpenAPI 2

```
swagger: "2.0"
info:
  version: "1.0"
  title: Images API
schemes:
  - http
produces:
  - image/png
paths:
  /image.png:
    get:
      responses:
        200:
          description: Representation
          schema:
            type: string
            format: binary
          examples:
            "image/png": ""
```

Note: Do not use the explicit `binary` or `bytes` formats with response bodies, as Dredd is not able to properly work with those (fury-adapter-swagger#193).

Hooks

In hooks, you can either assert the body:

```
const hooks = require('hooks');
const fs = require('fs');
const path = require('path');

hooks.beforeEachValidation((transaction, done) => {
  const bytes = fs.readFileSync(path.join(__dirname, '../image.png'));
  transaction.expected.body = bytes.toString('base64');
  done();
});
```

Or you can ignore it:

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

hooks.beforeEachValidation((transaction, done) => {
  transaction.real.body = '';
  done();
});
```

2.4.10 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"}
```

(continues on next page)

(continued from previous page)

```
+ 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: 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.

OpenAPI 2

When using OpenAPI 2 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();
});
```

2.4.11 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: https://dredd.org/
↪en/latest/how-to-guides/#using-apiary-reporter-and-apiary-tests
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
```

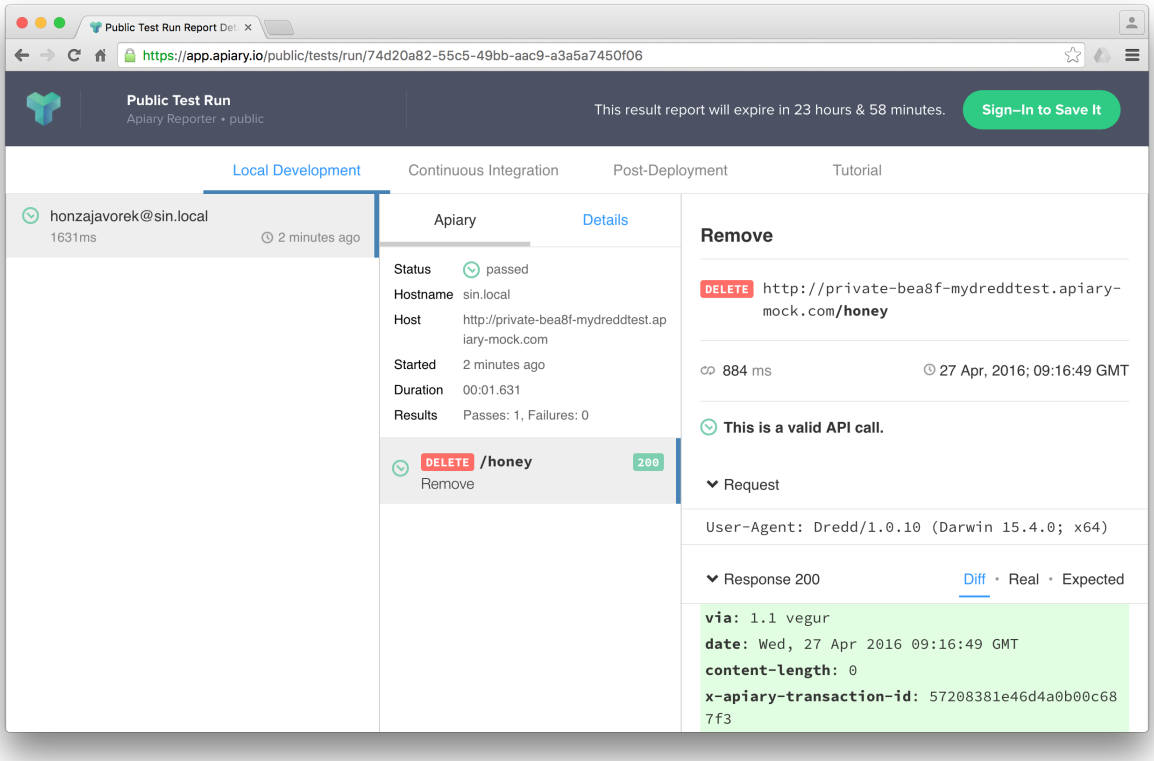

Fig. 1: Apiary Tests

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:

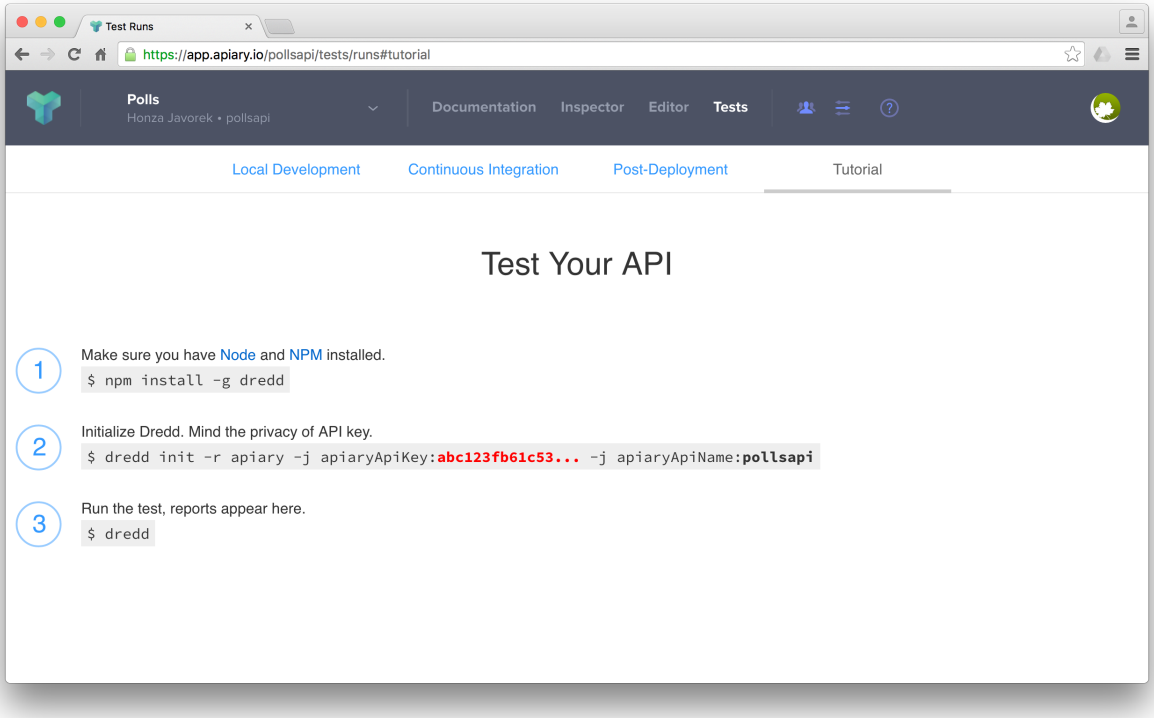

Fig. 2: Apiary Tests Tutorial

As you can see, the parameters go like this:

```
$ dredd -j apiaryApiKey:<Apiary API Key> -j 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`, ...

2.4.12 Example Values for Request Parameters

While example values are natural part of the API Blueprint format, the OpenAPI 2 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 OpenAPI 2 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.

2.4.13 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` (*docs*) object will make it to reporters. You don't have to care about sanitation of the rest of the `transaction` (*docs*) 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` (*docs*) 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` (*docs*) 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
URL or path to the API description document (API Blueprint, OpenAPI 2). **Sample values:** `./api-blueprint.apib`, `./openapi2.yml`, `./openapi2.json`, `http://example.com/api-blueprint.apib`

api-location
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"
server: rails server
server-wait: 3
init: false
names: false
only: []
output: []
header: []
sorted: false
user: null
inline-errors: false
details: false
method: []
loglevel: warning
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 OpenAPI 2 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
Use –color/–no-color to enable/disable colored output **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. **Default value:** `false`

--dry-run, -y
Do not run any real HTTP transaction, only parse API description document and compile transactions. **Default value:** `null`

--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
Path to hook files. Can be used multiple times, supports glob patterns. Hook files are executed in alphabetical order. **Default value:** `null`

--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. **Default value:** `false`

--inline-errors, -e
Determines whether failures and errors are displayed as they occur (true) or aggregated and displayed at the end (false). **Default value:** `false`

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

--loglevel, -l
Application logging level. Supported levels: 'debug', 'warning', 'error', 'silent'. The value 'debug' also displays timestamps. **Default value:** `"warning"`

--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. **Default value:** `false`

--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:** `[]`

--require
When using nodejs hooks, require the given module before executing hooks **Default value:** `null`

--server, -g
Run API backend server command and kill it after Dredd execution. E.g. *rails server* **Default value:** `null`

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

--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. **Default value:** `false`

--user, -u
Basic Auth credentials in the form username:password. **Default value:** `null`

--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 the *CLI options* 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
    loglevel: 'warning', // String, logging level (debug, warning, error, silent)
    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 lib/
↪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
    require: null,      // String, When using nodejs hooks, require the given module␣
↪before executing hooks
    color: true,
  },
  emitter: new EventEmitter(), // listen to test progress, your own instance of␣
↪EventEmitter
  apiDescriptions: ['FORMAT: 1A\n# Sample API\n']
}
```

`configuration`

`configuration.`**`server`**
: The HTTP(S) address of the API server to test against the API description(s). A valid URL is expected, e.g. `http://127.0.0.1:8000`

  Type string

  Required yes

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

  Type object

  Required yes

`configuration.options.`**`path`**
: Array of paths or URLs to API description documents.

Type array

Required yes

`configuration.`**`emitter`**
Listen to test progress by providing your own instance of EventEmitter.

Type EventEmitter

`configuration.`**`apiDescriptions`**
API descriptions as strings. Useful when you don't want to operate on top of the filesystem.

Type array

2.7 Hooks

Dredd supports *hooks*, which are blocks of arbitrary code that run before or after each test step. The concept is similar to XUnit's `setUp` and `tearDown` functions, Cucumber hooks, or Git hooks. Hooks are usually used for:

- Loading database fixtures,
- cleaning up after test step(s),
- handling auth and sessions,
- passing data between transactions (saving state from responses),
- modifying a request generated from the API description,
- changing generated expectations,
- setting custom expectations,
- debugging by logging stuff.

2.7.1 Getting started

Let's have a description of a blog API, which allows to list all articles, and to publish a new one.

API Blueprint

```
FORMAT: 1A

# Blog API
## Articles [/articles]
### List articles [GET]

+ Response 200 (application/json)

        [
          {
            "id": 1,
            "title": "Creamy cucumber salad",
            "text": "Slice cucumbers..."
          }
        ]

### Publish an article [POST]
```

(continues on next page)

(continued from previous page)

```
+ Request (application/json)

        {
          "title": "Crispy schnitzel",
          "text": "Prepare eggs..."
        }

+ Response 201 (application/json)

        {
          "id": 2,
          "title": "Crispy schnitzel",
          "text": "Prepare eggs..."
        }
```

OpenAPI 2

```
swagger: "2.0"
info:
  title: "Blog API"
  version: "1.0"
consumes:
  - "application/json"
produces:
  - "application/json"
paths:
  "/articles":
    x-summary: "Articles"
    get:
      summary: "List articles"
      description: "Retrieve a list of all articles"
      responses:
        200:
          description: "Articles list"
          examples:
            "application/json":
              - id: 1
                title: "Creamy cucumber salad"
                text: "Slice cucumbers..."
    post:
      summary: "Publish an article"
      description: "Create and publish a new article"
      parameters:
        - name: "body"
          in: "body"
          schema:
            example:
              title: "Crispy schnitzel"
              text: "Prepare eggs..."
      responses:
        201:
          description: "New article"
          examples:
            "application/json":
              id: 2
              title: "Crispy schnitzel"
              text: "Prepare eggs..."
```

Now let's say the real instance of the API has the POST request protected so it is not possible for everyone to publish new articles. We do not want to hardcode secret tokens in our API description, but we want to get Dredd to pass the auth. This is where the hooks can help.

Writing hooks

Hooks are functions, which are registered to be ran for a specific test step (HTTP transaction) and at a specific point in Dredd's *execution life cycle*. Hook functions take one or more *transaction objects*, which they can modify. Let's use hooks to add an Authorization header to Dredd's request.

Dredd supports *writing hooks in multiple programming languages*, but we'll go with JavaScript hooks in this tutorial as they're available out of the box.

API Blueprint

Let's create a file called `hooks.js` with the following content:

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

hooks.before('Articles > Publish an article', (transaction) => {
  transaction.request.headers.Authorization = 'Basic: YWxhZGRpbjpvcGVuc2VzYW1l';
});
```

As you can see, we're registering the hook function to be executed **before** the HTTP transaction `Articles > Publish an article`. This path-like identifier is a *transaction name*.

OpenAPI 2

Let's create a file called `hooks.js` with the following content:

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

hooks.before('Articles > Publish an article > 201 > application/json', (transaction)␣
↪=> {
  transaction.request.headers.Authorization = 'Basic: YWxhZGRpbjpvcGVuc2VzYW1l';
});
```

As you can see, we're registering the hook function to be executed **before** the HTTP transaction `Articles > Publish an article > 201 > application/json`. This path-like identifier is a *transaction name*.

Running Dredd with hooks

With the API instance running locally at `http://127.0.0.1`, you can now run Dredd with hooks using the *`--hookfiles`* option:

API Blueprint

```
dredd ./blog.apib http://127.0.0.1 --hookfiles=./hooks.js
```

OpenAPI 2

```
dredd ./blog.yaml http://127.0.0.1 --hookfiles=./hooks.js
```

Now the tests should pass even if publishing new article requires auth.

2.7.2 Supported languages

Dredd itself is written in JavaScript, so it supports *JavaScript hooks* out of the box. Running hooks in other languages requires installing a dedicated *hook handler*. Supported languages are:

Writing Dredd Hooks In Node.js

Usage

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

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` (*docs*) 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");
```

(continues on next page)

(continued from previous page)

```
});

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) {
```

(continues on next page)

(continued from previous page)

```
  hooks.log('after all');
  done();
});
```

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;
```

(continues on next page)

(continued from previous page)

```
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);
});
```

Modifying Multipart Transaction Request Body Prior to Execution

Dependencies:

- multi-part
- stream-to-string

```
const hooks = require('hooks');
const fs = require('fs');
const Multipart = require('multi-part');
const streamToString = require('stream-to-string');

var before = hooks.before;

before("Machines > Machines collection > Create Machines", async function␣
↪(transaction, done) {
    const form = new Multipart();
```

(continues on next page)

(continued from previous page)

```
    form.append('title', 'Foo');
    form.append('photo', fs.createReadStream('./bar.jpg'));
    transaction.request.body = await streamToString(form.getStream());
    transaction.request.headers['Content-Type'] = form.getHeaders()['content-type'];
    done();
});
```

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

You can use Babel for support of all the latest JS syntactic coolness in Dredd by using babel-register:

```
npm install -g babel-register @babel/preset-env
echo '{ "presets": [["env", { "target": { "node":6 } }]] }' > .babelrc
dredd test/fixtures/single-get.apib http://127.0.0.1:3000 --hookfiles=./es2015.js --
↪require=@babel/register
```

Using CoffeScript

You can use CoffeeScript in hooks by registering it as a compiler.

```
dredd test/fixtures/single-get.apib http://127.0.0.1:3000 --hookfiles=./hooks.coffee -
↪-require=coffeescript/register
```

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`.

Installation

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

Usage

Using Dredd with Go is slightly different to other languages, as a binary needs to be compiled for execution. The *--hookfiles* options 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
```

Note: If you're running *Dredd inside Docker*, read about *specifics of getting it working together with non-JavaScript hooks*.

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 occuring 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()
}
```

2. Callbacks receive a `Transaction` instance, or an array of them
3. 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"
```

(continues on next page)

(continued from previous page)

```
    "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()
}
```

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) {
```

(continues on next page)

(continued from previous page)

```
        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{}{}
```

(continues on next page)

(continued from previous page)

```
        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()
}
```

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*

Installation

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

Usage

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

Note: If you're running *Dredd inside Docker*, read about *specifics of getting it working together with non-JavaScript hooks*.

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'
});
```

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';
```

(continues on next page)

(continued from previous page)

```
    # 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) = @_;
```

(continues on next page)

(continued from previous page)

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

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*

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
```

Usage

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

Note: If you're running *Dredd inside Docker*, read about *specifics of getting it working together with non-JavaScript hooks*.

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";
});
```

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";

    }
});
```

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*

Installation

```
$ pip install dredd_hooks
```

Usage

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

Note: If you're running *Dredd inside Docker*, read about *specifics of getting it working together with non-JavaScript hooks*.

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')
```

(continues on next page)

(continued from previous page)

```
@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')
```

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 sumbit 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']
```

(continues on next page)

(continued from previous page)

```
  #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
```

(continues on next page)

(continued from previous page)

```
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()
```

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*

Installation

```
$ gem install dredd_hooks
```

Usage

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

Note: If you're running *Dredd inside Docker*, read about *specifics of getting it working together with non-JavaScript hooks*.

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

after_each do |transaction|
  puts 'after_each'
end

after_all do |transactions|
  puts 'after_all'
end
```

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|
```

(continues on next page)

(continued from previous page)

```
  # 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

```

(continues on next page)

(continued from previous page)

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

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 correspondig binary `dredd-hooks-rust`.

Installation

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

Usage

Using Dredd with Rust is slightly different to other languages, as a binary needs to be compiled for execution. The *`--hookfiles`* options 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
```

Note: If you're running *Dredd inside Docker*, read about *specifics of getting it working together with non-JavaScript hooks*.

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 occuring 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};

```

(continues on next page)

(continued from previous page)

```
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);
}
```

3. 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
```

(continues on next page)

(continued from previous page)

```
    }));
    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);
}
```

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};
```

(continues on next page)

(continued from previous page)

```
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);
}
```

Writing Dredd hook handler for new language

Dredd hooks handler client

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

When you run Dredd with *--language* option, it runs the given command 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*.

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.

What to implement

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

- CLI Command runnning TCP socket server
  - *Must return message ‘‘Starting‘* to stdout <https://github.com/apiaryio/dredd-hooks-template/blob/master/features/tcp_server.feature#L5>‘__
- 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

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.

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

Configuration Options

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

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

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

Note: If you don't see your favorite language, *it's fairly easy to contribute support for it*! Join the *Contributors Hall of Fame* where we praise those who added support for additional languages.

(Especially if your language of choice is **Java**, there's an eternal fame and glory waiting for you - see #875)

2.7.3 Transaction names

Transaction names are path-like strings, which allow hook functions to address specific HTTP transactions. They intuitively follow the structure of your API description document.

You can get a list of all transaction names available in your API description document by calling Dredd with the `--names` option:

API Blueprint

```
$ dredd ./blog.apib http://127.0.0.1 --names
info: Articles > List articles
skip: GET (200) /articles
info: Articles > Publish an article
skip: POST (201) /articles
complete: 0 passing, 0 failing, 0 errors, 2 skipped, 2 total
complete: Tests took 9ms
```

As you can see, the document `./blog.apib` contains two transactions, which you can address in hooks as:

- `Articles > List articles`
- `Articles > Publish an article`

OpenAPI 2

```
$ dredd ./blog.yaml http://127.0.0.1 --names
info: Articles > List articles > 200 > application/json
skip: GET (200) /articles
info: Articles > Publish an article > 201 > application/json
skip: POST (201) /articles
complete: 0 passing, 0 failing, 0 errors, 2 skipped, 2 total
complete: Tests took 9ms
```

As you can see, the document `./blog.yaml` contains two transactions, which you can address in hooks as:

- `Articles > List articles > 200 > application/json`
- `Articles > Publish an article > 201 > application/json`

Note: The transaction names and the `--names` workflow mostly do their job, but with many documented flaws. A successor to transaction names is being designed in #227

2.7.4 Types of hooks

Hooks get executed at specific points in Dredd's *execution life cycle*. Available types of hooks are:

- `beforeAll` called at the beginning of the whole test run
- `beforeEach` called before each HTTP transaction

- `before` called before a specific HTTP transaction
- `beforeEachValidation` called before each HTTP transaction is validated
- `beforeValidation` called before a specific HTTP transaction is validated
- `after` called after a specific HTTP transaction regardless its result
- `afterEach` called after each HTTP transaction
- `afterAll` called after whole test run

2.7.5 Hooks inside Docker

As mentioned in *Supported languages*, running hooks written in languages other than JavaScript requires a dedicated hook handler. Hook handler is a separate process, which communicates with Dredd over a TCP socket.

If you're *running Dredd inside Docker*, you may want to use a separate container for the hook handler and then run all your containers together as described in the *Docker Compose* section.

However, hooks were not originally designed with this scenario in mind. Dredd gets a name of (or path to) the hook handler in `--language` and then starts it as a child process. To work around this, fool Dredd with a dummy script and set `--hooks-worker-handler-host` together with `--hooks-worker-handler-port` to point Dredd's TCP communication to the other container.

Note: The issue described above is tracked in #755.

2.8 Data Structures

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

2.8.1 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)
  - bodyEncoding (enum) - can be manually set in *hooks*
    - `utf-8` (string) - indicates `body` contains a textual content encoded in UTF-8
    - `base64` (string) - indicates `body` contains a binary content encoded in Base64
  - 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)
  - bodyEncoding (enum)
    - `utf-8` (string) - indicates `body` contains a textual content encoded in UTF-8
    - `base64` (string) - indicates `body` contains a binary content encoded in Base64
- 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 (object)*) - test data passed to Dredd's reporters
- results (*Transaction Results (object)*) - testing results

2.8.2 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 Error (object)* in `results` or Dredd's custom message (e.g. "failed in before hook")
- results (Dredd's *transaction*.results)
- valid (boolean)
- origin (object) - *transaction*.origin

2.8.3 Transaction Results (object)

This is a cousin of the *Gavel Validation Result (object)*.

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

2.8.4 Gavel Validation Result (object)

Can be seen also here.

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

2.8.5 Gavel Validator Output (object)

Can be seen also here.

- results (array[*Gavel Error (object)*])
- 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 (object)*)
  - (*TextDiff Validation Result (string)*)

2.8.6 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`

2.8.7 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 (docs).

2.8.8 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

2.8.9 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 (object)*) - *test*.results

2.8.10 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 (object)*) - *test*.results

2.9 Internals

Dredd itself is a command-line Node.js application written in modern JavaScript. Contents:

2.9.1 Maintainers

Apiary is the main author and maintainer of Dredd's upstream repository. Currently responsible people are:

- @paraskakis - product decisions, feature requests
- @honzajavorek - lead of development

Dredd supports many programming languages thanks to the work of several contributors. They deserve eternal praise for dedicating time to create, improve, and maintain the respective *hook handlers*:

- @ddelnano (*PHP*, *Go*)
- @gonzalo-bulnes (*Ruby*)
- @hobofan (*Rust*)
- @snikch (*Go*)
- @ungrim97 (*Perl*)

2.9.2 Contributing

We are grateful for any contributions made by the community. Even seemingly small contributions such as fixing a typo in the documentation or reporting a bug are very appreciated!

To learn the basics of contributing to Dredd, please read the contributing documentation, placed in Dredd's GitHub repository.

Installing Dredd for development

To hack Dredd locally, clone the repository and run `npm install` to install JavaScript dependencies. Then run `npm test` to verify everything works as expected. If you want to run Dredd during development, you can do so using `./bin/dredd`.

Note: See also the full *installation guide*.

Commit message format

Semantic Release automatically manages releasing of new Dredd versions to the npm registry. It makes sure correct version numbers get increased according to the **meaning** of your changes once they are added to the `master` branch. This requires all commit messages to be in a specific format, called Conventional Changelog:

```
<type>: <message>
```

Where `<type>` is a prefix, which tells Semantic Release what kind of changes you made in the commit:

- `feat` - New functionality added (results in _minor_ version bump)
- `fix` - Broken functionality fixed (results in _patch_ version bump)
- `refactor` - Changes in code, but no changes in behavior
- `perf` - Performance improved
- `style` - Changes in code formatting
- `test` - Changes in tests
- `docs` - Changes in documentation
- `chore` - Changes in package or repository configuration

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

```
feat: add option "--require" to support custom transpilers

Remove bult-in compilation of CoffeeScript.

Close #1234

BREAKING CHANGE: Hookfiles using CoffeeScript are not supported
out of the box anymore. Instead manually install the coffeescript
module and add --require=coffeescript/register to your command.
```

- See existing commits as a reference
- Commitizen CLI can help you to create correct commit messages
- Run `npm run lint` to validate format of your messages
- Use `refactor` together with `BREAKING CHANGE:` for changes in code which only remove features (there doesn't seem to be a better category for that use case) – see real-world example

GitHub labels

Todo: This section is not written yet. See #808.

Programming language

Dredd is written in modern JavaScript, ran by Node.js, and distributed by npm.

Previously Dredd was written in CoffeeScript, and it was only recently converted to modern JavaScript. That's why sometimes the code does not feel very nice. Any efforts to refactor the code to something more human-friendly are greatly appreciated.

C++ dependencies

Dredd uses Drafter for parsing API Blueprint documents. Drafter is written in C++ 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++ version of the parser and in case of failure it falls back to the JavaScript equivalent. Dredd depends on the drafter-npm package.

That still proved problematic for Dredd though. The current solution is to provide an npm-shrinkwrap.json file with the Dredd Transactions library, which completely excludes protagonist, i.e. the compiled C++ binding. Unlike `package-lock.json`, the file can be distributed inside an npm package. The exclusion is performed by a `postshrinkwrap` npm script. This didn't work well with Dredd's `package-lock.json`, so currently Dredd's dependency tree is not locked for local or CI installations.

Supported Node.js versions

Given the table with LTS schedule, only versions marked as **Current**, **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. In `appveyor.yml` the latest supported Node.js version should be used. When dropping support for Node.js versions, remember to update the *installation guide*.

When dropping support for a certain Node.js version, it should be removed from the testing matrix, and it **must** be delivered as a breaking change, which increments Dredd's major version number.

Dependencies

New versions of dependencies are monitored by David and Greenkeeper. Vulnerabilities are monitored by Snyk.

Dependencies should not be specified in a loose way - only exact versions are allowed. This is ensured by `.npmrc` and the lock file. Any changes to dependencies (version upgrades included) are a subject to internal policies and must be first checked and approved by the maintainers before merged to `master`. This is because we are trying to be good Open Source citizens and to do our best to comply with licenses of all our dependencies.

As a contributor, before adding a new dependency or upgrading an existing one, please try to make sure the project and all its transitive dependencies feature standard permissive licenses, including correct copyright holders and license texts.

Versioning

Dredd follows Semantic Versioning. The releasing process is fully automated by Semantic Release.

There are two release tags: `latest` and `stable`. Currently they both point to the latest version. The `stable` tag exists only for backward compatibility with how Dredd used to be distributed in the past. It might get removed in the future.

Testing

Use `npm test` to run all tests. Dredd uses Mocha as a test framework. Its default options are in the `test/mocha.opts` file.

Linting

Dredd uses eslint to test the quality of the JavaScript codebase. We are adhering to the Airbnb's styleguide. Several rules are disabled to allow us to temporarily have dirty code after we migrated from CoffeeScript to JavaScript. The long-term intention is to remove all these exceptions.

The linter is optional for local development to make easy prototyping and working with unpolished code, but it's enforced on the CI level. It is recommended you integrate eslint 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.

We want to have a one-page changelog in the documentation as well - see #740.

Coverage

Tests coverage is a metric which helps developer to see which code **is not** tested. This is useful when introducing new code in Pull Requests or when maintaining under-tested old code (coverage shows that changes to such code are without any safety net).

We strive 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.

Note: Currently the integration is broken and while we're sending data to Coveralls, they do not report back under Pull Requests. Multiple sessions to debug the problem were not successful and we are considering to replace the service.

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

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

The first mentioned command does following:

1. Uses istanbul to instrument the JavaScript code
2. Runs the tests on the instrumented code using Mocha with a special lcov reporter, which gives us information about which lines were executed in the 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 the lcov-result-merger utility and sent to Coveralls

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 the variable is 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*

The `APIARY_API_URL` environment variable allows the developer to override the host of the Apiary Tests API.

2.9.3 Contributing to documentation

The documentation is written as code in the reStructuredText format and its source files are located in the docs directory. It is published automatically by the ReadTheDocs when the `master` branch is updated.

Even though alternatives exist (dredd.readthedocs.io, dredd.rtfd.io, or dredd.io), the documentation should always be linked canonically as https://dredd.org.

Building documentation locally

The documentation is built by Sphinx. To render it on your computer, you need Python 3.

1. Get Python 3. ReadTheDocs build the documentation with Python 3.6, so make sure you have this version.
2. Create a virtual environment and activate it:

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

3. Install dependencies for the docs:

```
(venv)$ pip install -r docs/requirements.txt
```

> **Note:** We are not using pipenv as it is not yet properly supported by ReadTheDocs.

Now you can use following commands:

- `npm run docs:lint` - Checks quality of the documentation (broken internal and external links, reStructuredText markup mistakes, etc.)
- `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 published by ReadTheDocs. We force their latest build image in the `readthedocs.yml` to get Python 3.

Writing documentation

- Read the reStructuredText primer
- No explicit newlines, please - write each paragraph as a single long line and turn on word wrap in your editor
- Explicit is better than implicit:
  - Bad: `npm i -g`
  - Good: `npm install --global`
- When using Dredd's long CLI options in tests or documentation, please always use the notation with = wherever possible:
  - Bad: `--path /dev/null`
  - Good: `--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.
- Do not title case headings, life's too short to spend it figuring out title casing correctly
- Using `127.0.0.1` (in code, tests, documentation) is preferred over `localhost` (see #586)
- Be consistent

Sphinx extensions

There are several extensions to Sphinx, which add custom directives and roles to the reStructuredText syntax:

CLI options Allows to automatically generate documentation of Dredd's CLI options from the JSON file which specifies them. Usage: `.. cli-options:: ./path/to/file.json`

GitHub issues Simplifies linking GitHub issues. Usage: `:ghissue:`drafter#123``

API Blueprint spec Simplifies linking the API Blueprint spec. Usage: `:apib:`schema-section``

MSON spec Simplifies linking the MSON spec. Usage: `:mson:`353-type-attribute``

OpenAPI 2 spec Simplifies linking the OpenAPI 2 spec. Usage: `:openapi2:`parameterobject``

OpenAPI 3 spec Simplifies linking the OpenAPI 3 spec. Usage: `:openapi3:`parameterobject``

RFCs Simplifies linking the RFCs. Not a custom extension in fact, this is provided by Sphinx out of the box. Usage: `:rfc:`1855``

The extensions are written in Python 3 and are heavily based on the knowledge shared in the FOSDEM 2018 talk by Stephen Finucane. Extensions use Python's unittest for tests. You can use `npm run docs:test-extensions` to run them.

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 the ReadTheDocs admin interface.

2.9.4 Windows support

Dredd is tested on the AppVeyor, a Windows-based CI. There are still several known issues when using Dredd on Windows, but the long-term intention is to support it without any compromises.

2.9.5 API description parsing

Todo: This section is not written yet. See #820.

2.9.6 Architecture

Todo: This section is not written yet. See #820.

CHAPTER 3

Useful Links

- GitHub Repository
- Bug Tracker
- Changelog

CHAPTER 4

Example Applications

- Express.js
- Ruby on Rails

Index

Symbols

A

C

R