Easer

User's Manual

Tamás Benke

2022-11-21

Contents

1 Getting Started

1.1 Overview

1.1.1 Motivation

RESTful APIs are widespread nowadays. We use edge servers for backing services in many areas. These edge servers usually provide generic functionalities, such as acting as web middleware that makes content negotiation, error handling, compression, authorization, etc. The business logic behind the REST API can be either built into the server itself, or can be detached and placed into distributed modules like it happens in case of microservices.

To implement a web server that provides both the generic middleware features and the business logic is simple at a basic level, but can get more complex if we want to make it well. It might have to contain several authorization strategies, tracing, etc. Reimplementing this again-and-again is time consuming, error prone, and even unnecessary, since these functionalities are the same in most of the cases, and usually the business logic is different only.

In principle, the REST API can be completely described via the Swagger/OpenApi configuration files. The generic web middleware features also can be controlled with some additional configuration via the environment, CLI parameters, or a config file. So in case if we move all the business logic into dedicated backing service applications which communicates with the edge server via a messaging middleware, then the edge server can be written to be fully generic, and no modification is needed, only defining the working parameters is necessary.

> The main goal with the implementation of `easer` is to have a general purpose, cloud ready server that connects client applications with backend services using configuration only, but no infrastructure coding is required.

The clients want to access to the backend services through standard synchronous REST APIs, and/or asynchronous websocket channels. `easer` makes this possible, and it needs only some configuration parameter and a standard description of the REST API.

1.1.2 Features

`easer` is a generic web server built on top of express, that has pre-built middlewares and components to deliver the following features:

- Acts as static web content server.
- Provides REST API that is described by Swagger/OpenApi descriptors.

- Serves the examples defined in the swagger files in mocking mode.
- Acts as edge server. Accomplishes messaging gateway functionality that maps the REST API calls to synchronous NATS calls towards service implementations, that can be implemented in different programming languages.
- Connects the frontend applications to backing services and pipelines via asynchronous topic-like messaging channels using websocket and NATS.
- Implements internal features required for graceful shutdown, logging, monitoring, etc.

1.1.3 Working Modes and Use-Cases

These are the typical usage scenarios:

1. **Static Web Server**;
2. **Mock Server**;
3. **REST API / NATS Gateway**:
   Exposes Micro Services through the REST API via NATS topics;
4. **WebSocket / NATS Gateway**:
   WebSocket Server and Gateway to NATS topics using Pattern Driven Micro Service calls and asynchronous data pipelines.

The Figure below is an example for the architecture of a system that uses all of the features of `easer`:

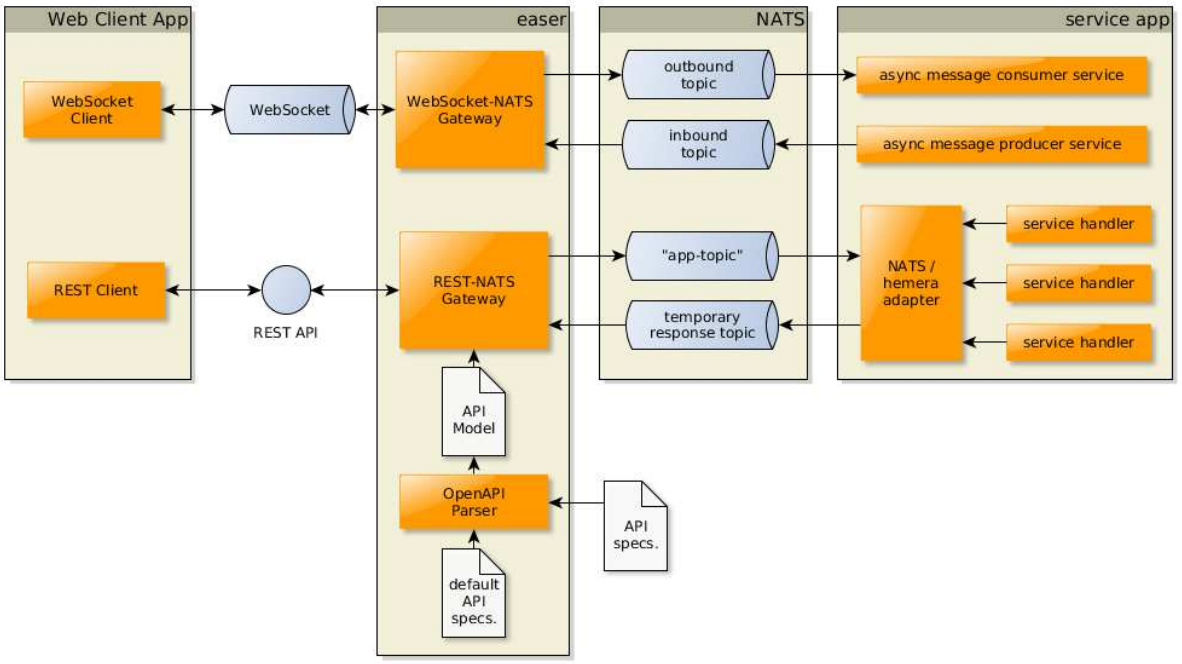

Figure 1: Full Architecture

The following sections describe the working of each part of this system.

1.2 Installation

1.2.1 Prerequisites

In order to run the server, you need to have the Node.js and npm installed on your machine.

1.2.2 Installation

The `easer` is made to act as a standalone application server, so it's preferred installation is:

```
npm install -g easer
```

For development purposes clone the easer server into a folder:

```
clone git@github.com:../website/static.git
```

Install the required dependencies:

```
cd easer
npm install
```

1.3 Basic Operations

1.3.1 Get version

```
$ easer --version

4.0.0
```

1.3.2 Get help

```
$ easer --help

Options:
      --version          Show version number                           [boolean]
  -c, --config           The name of the configuration file
                                                    [default: "config.yml"]
  -b, --basePath         The base-path URL prefix to each REST endpoints
                                                    [string] [default: "/"]
  -d, --dumpConfig       Print the effective configuration object to the console
                                                 [boolean] [default: false]
  -l, --logLevel         The log level               [string] [default: "info"]
  -t, --logFormat        The log (`plainText` or `json`)
                                               [string] [default: "plainText"]
  -p, --port             The port the server will listen[string] [default: 3007]
  -r, --restApiPath      The path to the REST API descriptors
                           [string] [default: "/home/tombenke/topics../websi
  -s, --useCompression   Use middleware to compress response bodies for all
                         request                       [boolean] [default: false]
  -u, --useMessaging     Use messaging middleware to forward REST API calls
                                                 [boolean] [default: false]
      --topicPrefix      The topic prefix for messaging based forwarding of REST
                         API calls                  [string] [default: "easer"]
      --parseRaw         Enable the raw body parser for the web server.
                                                   [boolean] [default: true]
```

```
      --parseJson         Enable the JSON body parser for the web server.
                                                    [boolean] [default: false]
      --parseXml          Enable the XML body parser for the web server.
                                                    [boolean] [default: false]
      --parseUrlencoded   Enable the URL Encoded body parser for the web server.
                                                    [boolean] [default: false]
  -n, --natsUri           NATS server URI used by the nats adapter.
                                  [string] [default: ["nats://localhost:4222"]]
  -w, --useWebsocket      Use WebSocket server and message forwarding gateway
                                                    [boolean] [default: false]
  -i, --inbound           Comma separated list of inbound NATS topics to forward
                          through websocket                [string] [default: ""]
  -o, --outbound          Comma separated list of outbound NATS topics to forward
                          towards from websocket           [string] [default: ""]
  -m, --enableMocking     Enable the server to use examples data defined in
                          swagger files as mock responses.
                                                    [boolean] [default: false]
      --help              Show help                                    [boolean]
```

1.3.3 Set the port

```
easer -p 8081
```

1.3.4 Set the log level and format

Use the following parameters:

```
easer -logLevel <log-level> -logFormat <log-format>
```

or

```
easer -l <log-level> -t <log-format>
```

The valid log-level values are:

- `error`: 0,
- `warn`: 1,

- `info`: 2 (default),
- `verbose`: 3,
- debug: 4,
- `silly`: 5.

The log-format value is one of `plainText` (default) or `json`.

For example the `info` level looks like this with `plainText` format:

```
$ easer -l info

2019-08-04T12:44:42.905Z [easer@4.0.0] info: Start up webServer
2019-08-04T12:44:42.917Z [easer@4.0.0] info: Express server listening on port 30
2019-08-04T12:44:42.918Z [easer@4.0.0] info: App runs the jobs...
```

in json format:

```
$ easer -l info -t json

{"message":"Start up webServer","level":"info","label":"easer@4.0.0","timestamp"
08-04T12:45:28.789Z"}
{"message":"Express server listening on port 3007","level":"info","label":"easer
08-04T12:45:28.801Z"}
{"message":"App runs the jobs...","level":"info","label":"easer@4.0.0","timestam
08-04T12:45:28.802Z"}
```

And the debug level in `plainText`:

```
$ easer -l debug

2019-08-04T12:46:27.703Z [easer@4.0.0] debug: webServer config:{"app":{"name":"e
static":{"contentPath":"/home/tombenke/topics../website/static-
tutorial","config":{"dotfiles":"allow","index":true}},"responses":{"200":{"descr
tutorial","ignoreApiOperationIds":true,"enableMocking":false,"basePath":"/","oas
tutorial","dumpConfig":false}
2019-08-04T12:46:27.705Z [easer@4.0.0] info: Start up webServer
2019-08-04T12:46:27.712Z [easer@4.0.0] debug: Bind /home/tombenke/topics../websi
tutorial to / as static content service
```

```
2019-08-04T12:46:27.713Z [easer@4.0.0] debug: restapi.setEndpoints/endpointMap [
2019-08-04T12:46:27.716Z [easer@4.0.0] info: Express server listening on port 30
2019-08-04T12:46:27.717Z [easer@4.0.0] info: App runs the jobs...
```

1.3.5 Static Content Server

By default, the `easer` server works as a static content server, that provides the content of the current working directory. For example, let's suppose your current working directory is the root of the easer repository, then you start the server:

```
$ easer

2019-08-04T13:08:42.398Z [easer@4.0.0] info: Start up webServer
2019-08-04T13:08:42.409Z [easer@4.0.0] info: Express server listening on port 30
2019-08-04T13:08:42.411Z [easer@4.0.0] info: App runs the jobs...
```

When the server started, you can open the http://localhost:3007/ URL with a browser, then you will something like this:

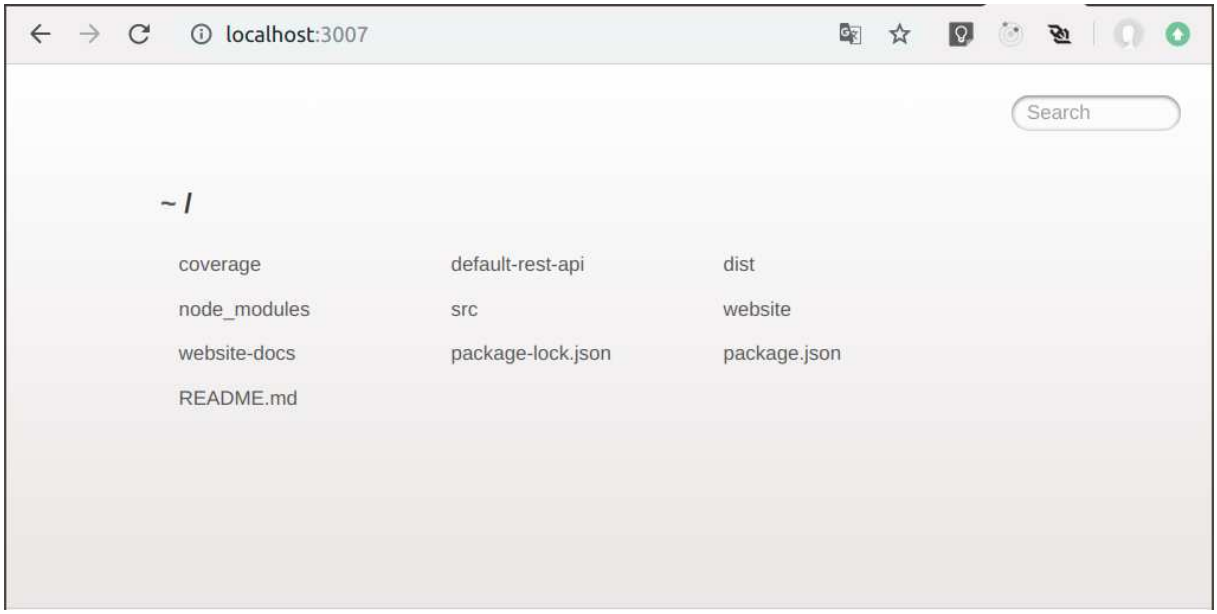

Figure 2: Static Content Example

2 Guides

2.1 Configuration

2.1.1 Overview

`easer` can be configured via:

- configuration file,
- environment variables,
- command line arguments,
- the combination of these above.

The preferred way of configuring the server is via environment, especially in the production environment, however it makes sense to use the other two methods, or even you need to combine them, for example during development. since there are many configuration parameters, it is not trivial to find out what is the current, effective parameter setup.

You can use the following CLI parameter to make the `easer` dump out its actual configuration:

```
easer -d
```

This is an example for the output:

```
{
  "webServer": {
    "logBlackList": [],
    "port": 3007,
    "useCompression": false,
    "useResponseTime": false,
    "useMessaging": false,
    "middlewares": {
      "preRouting": [],
      "postRouting": []
    },
    "restApiPath": "/home/tombenke/topics../website/static",
    "staticContentBasePath": "/home/tombenke/topics../website/static",
    "topicPrefix": "easer",
```

```
    "ignoreApiOperationIds": true,
    "enableMocking": false,
    "basePath": "/",
    "oasConfig": {
      "parse": {
        "yaml": {
          "allowEmpty": false
        },
        "resolve": {
          "file": true
        }
      }
    },
    "bodyParser": {
      "raw": true,
      "json": false,
      "xml": false,
      "urlencoded": false
    }
  },
  "nats": {
    "servers": ["nats://localhost:4222"],
    "timeout": 2000
  },
  "wsServer": {
    "topics": {
      "inbound": [],
      "outbound": []
    }
  },
  "app": {
    "name": "easer",
    "version": "5.0.2"
  },
  "configFileName": "config.yml",
  "useWebsocket": false,
  "logger": {
    "level": "info",
```

```
    "transports": {
      "console": {
        "format": "plainText"
      }
    }
  },
  "installDir": "/home/tombenke/topics../website/static",
  "dumpConfig": true
}
```

2.1.2 Overview of the config parameters

2.1.2.1 General server parameters

Dump the effective configuration object, before start:

- CLI parameter: `-d [true]`, or `--dumpConfig [true]`.

Set the port where the server will listen:

- CLI parameter: `-p 8081` or `--port 8081`.
- Environment: `WEBSERVER_PORT`.
- Config object property: `webServer.port`
- Default value: `3007`.

Define the REST API, using swagger or OpenApi descriptor(s):

- CLI parameter: `-r /app/rest-api/api.yml`, or `--restApiPath /app/rest-api/api.yml`.
- Environment: `WEBSERVER_RESTAPIPATH`.
- Config object property: `webServer.restApiPath`
- Default value: the current working directory.

Define the base-path (prefix) for the REST API endpoints:

- CLI parameter: `-b /base/path`, or `--basePath /base/path`.
- Environment: `WEBSERVER_BASEPATH`.
- Config object property: `webServer.basePath`
- Default value: `/`.

Enable Mocking. The server will response the first example found in the `examples` array of endpoint descriptor if there is any. For proper working, it requires the `ignoreApiOperationIds` config parameter to be `true` in case the `operationIds` of the endpoints are defined. The easer set this parameter to `true` by default:

- CLI parameter: `--enableMocking`, or `-m`.
- Environment: `WEBSERVER_ENABLE_MOCKING`.
- Config object property: `webServer.enableMocking`.
- Default value: `true`.

Ignore the `operationId` property of the API endpoint descriptor:

- CLI parameter: N.A.
- Environment: `WEBSERVER_IGNORE_API_OPERATION_IDS`.
- Config object property: `webServer.ignoreApiOperationIds`.
- Default value: `true`.

Set the base path of the endpoints that provide static content:

- CLI parameter: N.A.
- Environment: `WEBSERVER_STATIC_CONTENT_BASEPATH`.
- Config object property: `webServer.staticContentBasePath`.
- Default value: the current working directory.

Compress response bodies for all request:

- CLI parameter: `--useCompression [true]`, or `-s [true]`.
- Environment: `WEBSERVER_USE_COMPRESSION`.
- Config object property: `webServer.useCompression`.
- Default value: `false`.

API calls return with response time header:

- CLI parameter: N.A.
- Environment: `WEBSERVER_USE_RESPONSE_TIME`.
- Config object property: `webServer.useResponseTime`.
- Default value: `false`.

Enable the raw body parser for the web server:

- CLI parameter: `--parseRaw <boolean>`.
- Environment: `WEBSERVER_PARSE_RAW_BODY`.
- Config object property: `webServer.bodyParser.raw`.
- Default value: `true`.

Enable the JSON body parser for the web server:

- CLI parameter: `--parseJson <boolean>`.
- Environment: `WEBSERVER_PARSE_JSON_BODY`.
- Config object property: `webServer.bodyParser.json`.
- Default value: `false`.

Enable the XML body parser for the web server:

- CLI parameter: `--parseXml <boolean>`.
- Environment: `WEBSERVER_PARSE_XML_BODY`.
- Config object property: `webServer.bodyParser.xml`.
- Default value: `false`.

Enable the URL Encoded body parser for the web server:

- CLI parameter: `--parseUrlencoded <boolean>`.
- Environment: `WEBSERVER_PARSE_URL_ENCODED_BODY`.
- Config object property: `webServer.bodyParser.urlencoded`.
- Default value: `false`.

2.1.2.2 Logging

Set the log level of the server and its internal components:

- CLI parameter: `-l <level>`, or `logLevel <level>`
- Environment: `EASER_LOG_LEVEL`.
- Config object property: `logger.level`.
- Possible values: `info`, `debug`, `warn`, `error`.
- Default value: `info`.

Set the log format of the server and its internal components:

- CLI parameter: `-t <format>`, or `--logFormat <format>`.
- Environment: `EASER_LOG_FORMAT`.
- Config object property: `logger.transports.console.format`.
- Possible values: `plainText`, `json`.
- Default value: `plainText`.

2.1.2.3 MESSAGING (NATS) Gateway

Use messaging middleware to forward REST API calls:

- CLI parameter: `-u [true]`, or `--useMessaging [true]`.
- Environment: `WEBSERVER_USE_MESSAGING`
- Config object property: `webServer.useMessaging`.
- Default value: `false`.

The topic prefix for messaging based forwarding of REST API calls

- CLI parameter: `--topicPrefix <prefix-string>`.
- Config object property: `webServer.topicPrefix`.
- Default value: “easer”.

Define the URI of the NATS server used by the nats adapter:

- CLI parameter: `-n <nats-uri>`, or `--natsUri <nats-uri>`.
- Environment: `NATS_SERVERS`.
- Config object parameter: `nats.servers`.
- Default value: `["nats://demo.nats.io:4222"]`.

Define the NATS timeout value:

- CLI parameter: N.A.
- Environment: `WEBSERVER_MESSAGING_REQUEST_TIMEOUT`.
- Config object property: `webServer.messagingRequestTimeout`.
- Default value: `2000`.

See npac-nats-adapter for further details.

2.1.2.4 WebSocket Gateway

Use WebSocket server and message forwarding gateway:

- CLI parameter: `--useWebsocket [true]`, or `-w [true]`.
- Environment: `EASER_USE_WEBSOCKET`.
- Config object property: `useWebsocket`.
- Default value: `false`.

Define the inbound NATS topics as a comma-separated list that will be forwarded towards websocket:

- CLI parameter: `--inbound <list-of-topics>`, `-i <list-of-topics>`.
- Environment: `WSGW_INBOUND_TOPICS`.
- Config object property: `wsServer.topics.bound`.
- Default value: `""`.

Define the outbound NATS topics as a comma separated list that will be forwarded from websocket towards NATS topics:

- CLI parameter: `--outbound <list-of-topics>`, `-o <list-of-topics>`.
- Environment: `WSGW_OUTBOUND_TOPICS`.
- Config object property: `wsServer.topics.outbound`.
- Default value: `""`.

2.2 REST API Specification

2.2.1 Define the REST API Endpoints

The `easer` server needs to have the service endpoints defined via the standard OpenAPI Specification endpoint descriptors. These descriptors can be written either in OAS 2.0 (formerly called Swagger) or OAS 3.0 format.

The endpoints can be placed into a single JSON or YAML file, but also they can be organized into several files, under a directory structure, having a single root file, that holds a base references to the other files.

When we start `easer`, we have to refer to this sigle file, or this root file, using the `-r` or `--restApiPath` CLI parameter.

The person-rest-api repository holds a quite simple, but complete project, that demonstrate how to define the REST API endpoints. Beside the OAS endpoint descriptors, in this repository you will

find some scripts, that makes possible the validation of the descriptors, as well as that generates human-readeable formats of the specification: - in swagger-ui format - in redoc format

Just clone this repo, and read the README to learn more about how to use it as a template for your REST API specifications:

```
git clone git@github.com:tombenke/person-rest-api.git
```

2.2.2 Define Paths for Static Content

On top of the OAS standard, there is an extension to the Path Item object that should be used, if you want to define paths that the server should provide via its `static` middleware.

Let's suppose we want to make the `swagger.json` file available via the server, at the `/api-docs` path, using the GET method. Then we need to create a path definition similar to this:

```
#/api-docs:
get:
  tags:
    - 'swagger'
  summary: |
    Responses the files from the directory
    defined by the contentPath property
  x-static:
    contentPath: ./docs/
    config:
      dotfiles: allow
      index: true
  produces:
    - application/json
  responses:
    '200':
      description: OK
  deprecated: false
```

The key here is the `x-static` object, which holds the configuration properties for the `static` middleware.

See the specification of the `express.static` middleware for further details.

2.2.3 Define Examples for Mocking

Both OAS 2.0 and OAS 3.0 make possible to define examples to the several methods and content types of the endpoints. The examples added to the endpoint specifications, can be used for static mocking, as it is described in the Mock Server section.

2.3 Static Web Server

We can define the service endpoints to the `easer` server via standard endpoint descriptors, as the following Figure shows:

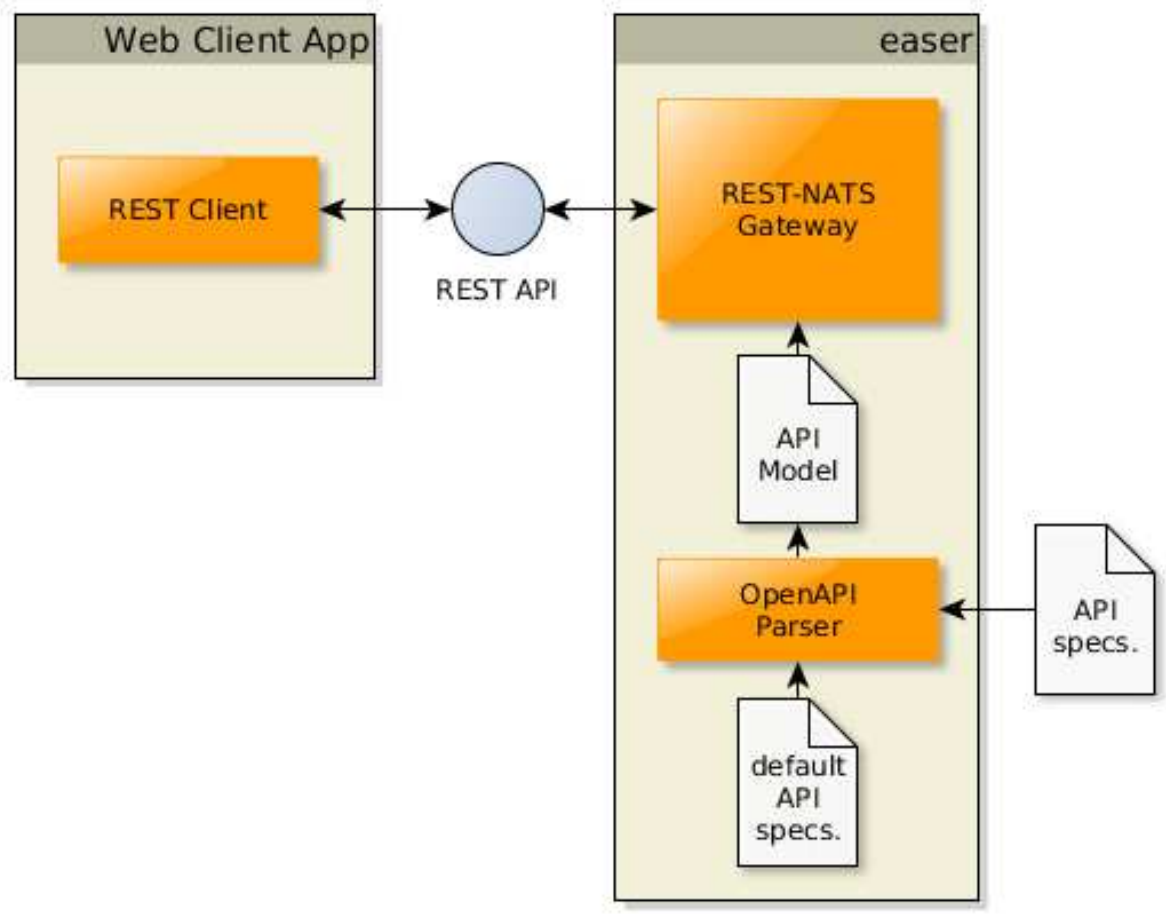

Figure 3: Default Static Webserver Architecture

However, in case we do not create endpoint descriptors, neither give it to easer, then the server uses its built-in descriptor, that looks like this:

```
{
  "info": {
    "title": "An API that provides the current directory as static content",
    "version": "1.0"
  },
  "paths": {
    "/": {
      "get": {
        "responses": {
          "200": {
            "description": "OK"
          }
        },
        "x-static": {
          "config": {
            "dotfiles": "allow",
            "index": true
          },
          "contentPath": "<the-current-working-directory>"
        }
      }
    }
  },
  "swagger": "2.0"
}
```

It defines one path that is `/`, and uses the `x-static` extensional property to tell the server that this path has to be forwarded to the static-middleware. This property also holds the configuration parameters of this middleware.

So step into a folder (for example the root of the easer repository) that contains the web content you want to observe, then start the server:

```
$ easer

2019-08-04T13:08:42.398Z [easer@4.0.0] info: Start up webServer
2019-08-04T13:08:42.409Z [easer@4.0.0] info: Express server listening on port 30
2019-08-04T13:08:42.411Z [easer@4.0.0] info: App runs the jobs...
```

Open the http://localhost:3007/ URL with a browser, then you will something like this:

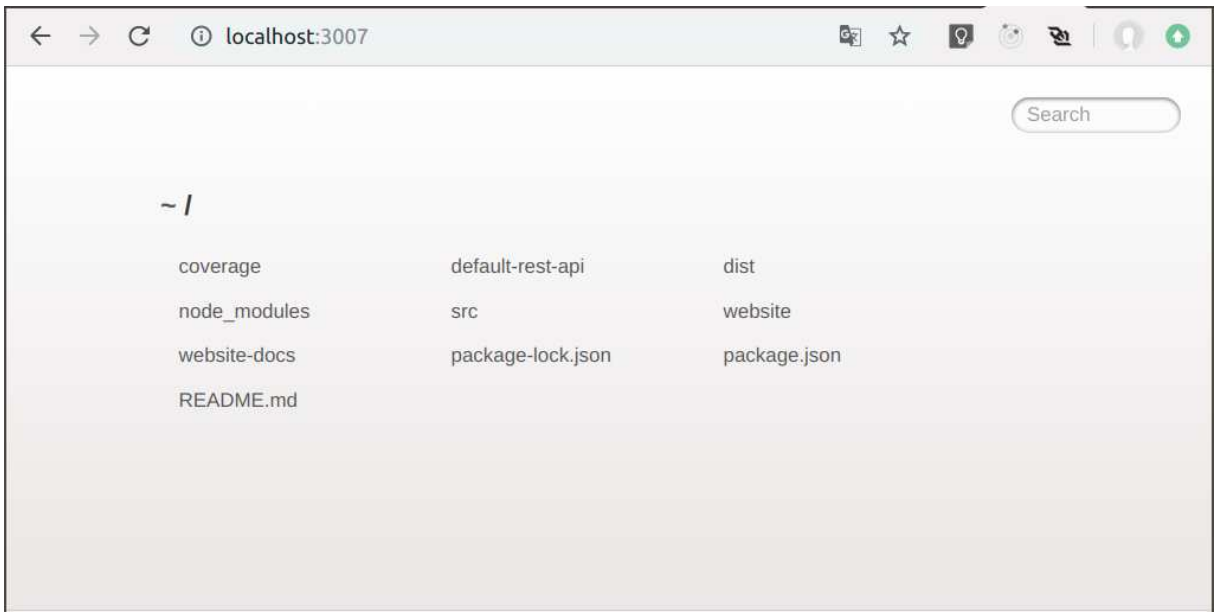

Figure 4: Static Content Example

2.4 REST API / NATS Gateway

This section describes how we can use the `easer` server as a REST API Proxy that delegates the incoming REST API calls to microservices, that are implemented in a separate, standalone application.

We will use the following example projects for demonstration:

- The person-rest-api is the specification of REST API endpoints of a simple person service;
- The person-service-js is the JavaScript implementation of REST API endpoints of a simple person service.

First clone these repositories:

```
git clone git@github.com:tombenke/person-rest-api.git
git clone git@github.com:tombenke/person-service-js.git
```

The `easer` REST API Proxy loads the endpoint descriptors from OAS format files. After loading the descriptors, it immediately is able to respond to the incoming REST calls.

Start the easer, with the REST API:

```
$ easer -r person-rest-api/rest-api/api.yml
2019-08-05T04:03:03.117Z [easer@4.0.0] info: Load endpoints from /home/tombenke/
tutorial/person-rest-api/rest-api/api.yml
2019-08-05T04:03:03.187Z [easer@4.0.0] info: Start up webServer
2019-08-05T04:03:03.200Z [easer@4.0.0] info: Express server listening on port 30
2019-08-05T04:03:03.201Z [easer@4.0.0] info: App runs the jobs...
```

Now the server is running and the REST endpoints are available, so try to call them:

```
$ curl http://localhost:3007/persons -v

*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 3007 (#0)
> GET /persons HTTP/1.1
> Host: localhost:3007
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 501 Not Implemented
< X-Powered-By: Express
< Content-Type: application/json; charset=utf-8
< Content-Length: 78
< ETag: W/"4e-mVxPm4tUvCVz9epJEB6Bbkm4UMA"
< Date: Mon, 05 Aug 2019 04:03:21 GMT
< Connection: keep-alive
<
* Connection #0 to host localhost left intact

{"error":"The endpoint is either not implemented or `operationId` is ignored"}
```

We got `501` error in the response, and the body says that the endpoint is not implemented. This is because the `easer` is responsible for the REST API at the edge, and not for the implementation of them, so the responses will be errors in all cases.

In order to bind the incoming endpoint calls with the service functions, the `easer` uses a messaging middleware, and synchronous, topic based, RPC-like calls to the service functions. The requests are passed towards the service implementations in JSON format messages through a previously agreed topic.

The service implementations are independently running, standalone applications.

The following figure shows the architecture of the REST API / NATS Gateway Mode

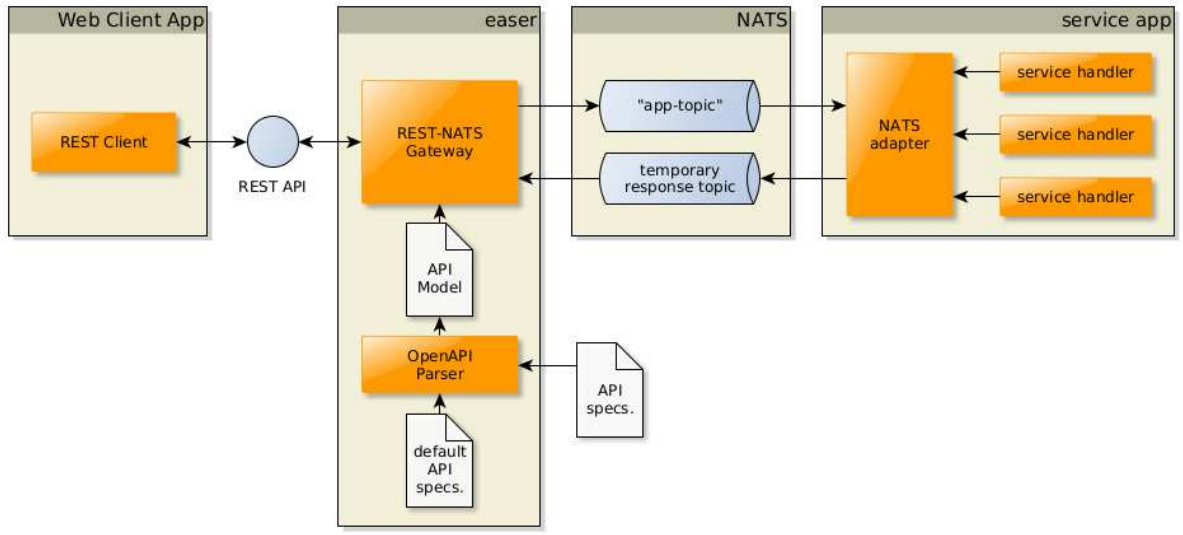

Figure 5: The Architecture of the REST API / NATS Gateway Mode

in order to have a completely working system, we need to start all the three main components: - The NATS middleware, - The application, that implements the service endpoints, - The easer, which is configured to connect to the NATS server, and to forward the traffic towards the service implementations.

First, let's start the NATS server, using Docker:

```
$ docker run -it --rm --network=host -p 4222:4222 -p 6222:6222 -
p 8222:8222 --name nats-main nats

[1] 2019/08/05 04:21:42.975522 [INF] Starting nats-server version 2.0.2
[1] 2019/08/05 04:21:42.975579 [INF] Git commit [6a40503]
[1] 2019/08/05 04:21:42.975687 [INF] Starting http monitor on 0.0.0.0:8222
[1] 2019/08/05 04:21:42.975760 [INF] Listening for client connections on 0.0.0.0
[1] 2019/08/05 04:21:42.975782 [INF] Server id is NBW6XLRZ4SJRIRYTRDLVHUTBY55DLH
[1] 2019/08/05 04:21:42.975791 [INF] Server is ready
[1] 2019/08/05 04:21:42.976324 [INF] Listening for route connections on 0.0.0.0:
```

The NATS server will provide its services on the `nats://localhost:4222` URI.

Then starts the service application, in a new terminal:

```
$ node person-service-js/index.js
```

Finally restart the easer server with the following parameters:

```
$ easer -r person-rest-api/rest-api/api.yml -n nats://localhost:4222 -
-topicPrefix person-demo -u

2019-08-05T04:43:04.362Z [easer@4.0.0] info: Start up nats
2019-08-05T04:43:04.384Z [easer@4.0.0] info: Load endpoints from /home/tombenke/
tutorial/person-rest-api/rest-api/api.yml
2019-08-05T04:43:04.449Z [easer@4.0.0] info: Start up webServer
2019-08-05T04:43:04.461Z [easer@4.0.0] info: Express server listening on port 30
2019-08-05T04:43:04.462Z [easer@4.0.0] info: App runs the jobs...
```

Where:

- `-n nats://localhost:4222`: defines the URI of the NATS server;
- `--topicPrefix person-demo`: defines the prefix for the name of the topic to use. The topic name is generated by the following pattern: `<topicPrefix.<endpoint.method>_<endpoint.uri>`;
- `-u`: Enables `easer` to forward the incoming calls as messages to the NATS topic.

If we try to call again the REST API, it will successfully serve the request:

```
$ curl http://localhost:3007/persons -v
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 3007 (#0)
> GET /persons HTTP/1.1
> Host: localhost:3007
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: application/json; charset=utf-8
< Content-Length: 2
< ETag: W/"2-l9Fw4VUO7kr8CvBlt4zaMCqXZ0w"
< Date: Mon, 05 Aug 2019 04:47:38 GMT
< Connection: keep-alive
```

```
<
* Connection #0 to host localhost left intact
[]
```

The response is an empty array, since there is no `persons` uploaded to the services yet.

We can use other endpoints to make sure, the API is working correctly:

```
$ curl -X PUT http://localhost:3007/persons/leia -H "Content-type: application/j
d '{"id":"leia","familyName":"Organa","givenName":"Leia"}'
{"id":"leia","familyName":"Organa","givenName":"Leia"}

$ curl http://localhost:3007/persons
[{"id":"leia","familyName":"Organa","givenName":"Leia"}]
```

The messages that the `easer` sends to the NATS topic contain the most relevant parts of the incoming request, such as: the query and path parameters, the headers, the method, the path, and the body. Beside the request, this message also contains the minimized version of the service descriptor itself, to help the service implementation to successfully do its job.

The RPC-like topic call works on a way, that the messaging system creates a temporary response topic, that the service implementation can use for sending the response back to the `easer` server. So the service endpoint will send back another message, that has to contain all the information which is needed to create a well formed REST response, such as: the status code, the headers, and the body, if there is any.

This is an example for an incoming `GET /persons` request that is sent to the `get_/persons` topic:

```
{
  "topic": "easer.put_/persons/{personId}",
  "method": "put",
  "uri": "/persons/{personId}",
  "endpointDesc": {
    "uri": "/persons/{personId}",
    "jsfUri": "/persons/:personId",
    "method": "put",
    "operationId": null,
    "consumes": [],
    "produces": [
```

```
      "application/json"
    ],
    "responses": {
      "200": {
        "status": "200",
        "headers": {},
        "examples": {
          "application/json": {
            "noname": {
              "mimeType": "application/json",
              "value": {
                "id": "2a1152ee-4d77-4ff4-a811-598555937625",
                "familyName": "Skywalker",
                "givenName": "Luke"
              }
            }
          }
        }
      },
      "400": {
        "status": "400",
        "headers": {},
        "examples": {
          "application/json": {}
        }
      }
    }
  },
  "request": {
    "cookies": {},
    "headers": {
      "host": "localhost:3007",
      "user-agent": "curl/7.81.0",
      "accept": "*/*",
      "content-type": "application/json",
      "content-length": "67"
    },
    "parameters": {
```

```
      "query": {},
      "uri": {
        "personId": "luke-skywalker"
      }
    },
    "body": "{\"id\":\"luke-skywalker\",\"familyName\":\"Skywalker\",\"givenName
  }
}
```

The topic name is generated via the following pattern from the endpoint definition: `<topicPrefix>.<method>_<uri>`, that is `easer.get_/persons` in this specific case.

At the NATS message level the easer also sends some headers. These are the followings:

```
{
  "content-type":"application/json",
  "message-type":"rpc/request"
}
```

It informs the responders about the representation of the message (`the content-type`), and the type of the payload (`message-type`).

The service implementations are independent applications, that can run in a distributed environment. For the same endpoint, there may be several implementations exist at the same time, so how will the messages find the right service implementation?

So the `easer` passes the incoming requests towards these services. The service handlers sends back the responses to `easer` that finally forwards the responses towards the client.

In this case the `getPersonsServiceHandler()` service handler function will respond with two things: the NATS message headers, and the payload in stringified form, as the examples show below:

The response JSON object (which will be sent to the gateway in stringified format):

```
{
  "status": 200,
  "headers": {
    "Content-type": "application/json"
  },
  "body": []
}
```

```
And the payload (the stringified version of the JSON response object):
"{\"status\":200,\"headers\":{\"Content-type\":\"application/json\"},\"body\":{\
skywalker\",\"familyName\":\"Skywalker\",\"givenName\":\"Luke\"}}"
```

The response headers in the NATS message:

```
{
  "content-type":"application/json",
  "message-type":"rpc/response"
}
```

2.5 Mock Server

Beside the normal function of a REST API Gateway server, it may play a special role too, in the period of development of the system. A REST API is an interface, where two system components meet, as well as the two teams meet who develop the frontend application from one hand side, and the backend services from the other hand side.

Even in case the two teams are the same, or it is only same individual person, it worth to take into consideration this point, because the parallel development of the components, that are situated at the opposite side of the interface depends on each other. From this perspective the REST API specification can be taken into consideration as a contract, that both parties needs to be conform with. It is not enough to declare the conformance, but that has to be proven too, otherwise the interface will be a kind of Pandora's box, and the bugs occur at any of its sides will be a good reason for the teams for fingerpointing, and to blame each other. in order to avoid this trap, we need a tool that makes possible for both parties to develop independently from the other, and at the same time, helps to prove the conformance with the agreement. This tool is the mock server, which can create a kind of Demilitarized Zone for the cooperating teams.

In order to switch `easer` to mock server mode, we need to use the `--enableMocking` or, simply the `-ms`switch, then the server starts responding to the calls from the REST API model loaded from the OAS files, as you can see on the following Figure:

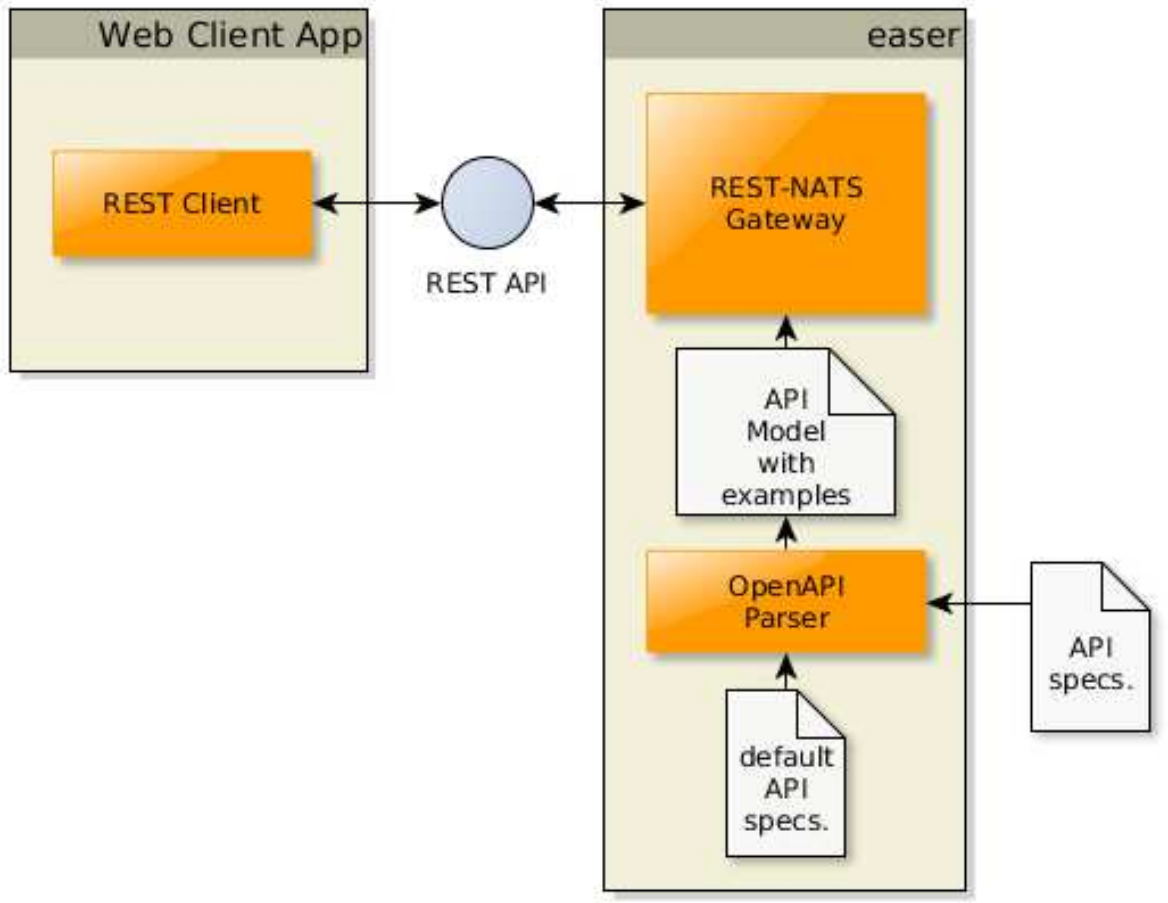

Figure 6: The Architecture of the Static Mocking Mode

Let's start the `easer` to act as a static mocking server for the person-rest-api:

```
$ easer -r person-rest-api/rest-api/api.yml -m
2019-08-05T06:28:57.646Z [easer@4.0.0] info: Load endpoints from /home/tombenke/
tutorial/person-rest-api/rest-api/api.yml
2019-08-05T06:28:57.715Z [easer@4.0.0] info: Start up webServer
2019-08-05T06:28:57.728Z [easer@4.0.0] info: Express server listening on port 30
2019-08-05T06:28:57.729Z [easer@4.0.0] info: App runs the jobs...
2019-08-05T06:29:01.108Z [easer@4.0.0] info: HTTP GET /persons
```

Then check the endpoints, if they really respond to the requests:

```
$ curl http://localhost:3007/persons
```

```
[{"id":"2a1152ee-4d77-4ff4-a811-598555937625","familyName":"Skywalker","giveName
397f-4923-bdf2-16334a76c29f","familyName":"Skywalker","givenName":"Anakin"}]
```

```
$ curl http://localhost:3007/persons/anakin
```

```
{"id":"2a1152ee-4d77-4ff4-a811-598555937625","familyName":"Skywalker","givenName
```

The responses are coming from the examples defined to the specific endpoint responses. The server takes into consideration of the `Accept` header, so examples can be defined for multiple mime-types.

There are cases, when the static mocking is not satisfying. We need to implement some intelligence to the mocked service. Another challenge is, when we have implemented some of the backing services, but some of them are not completed, and we need to mock a combination of static and dynamic mocking, as you can see on the next Figure, below:

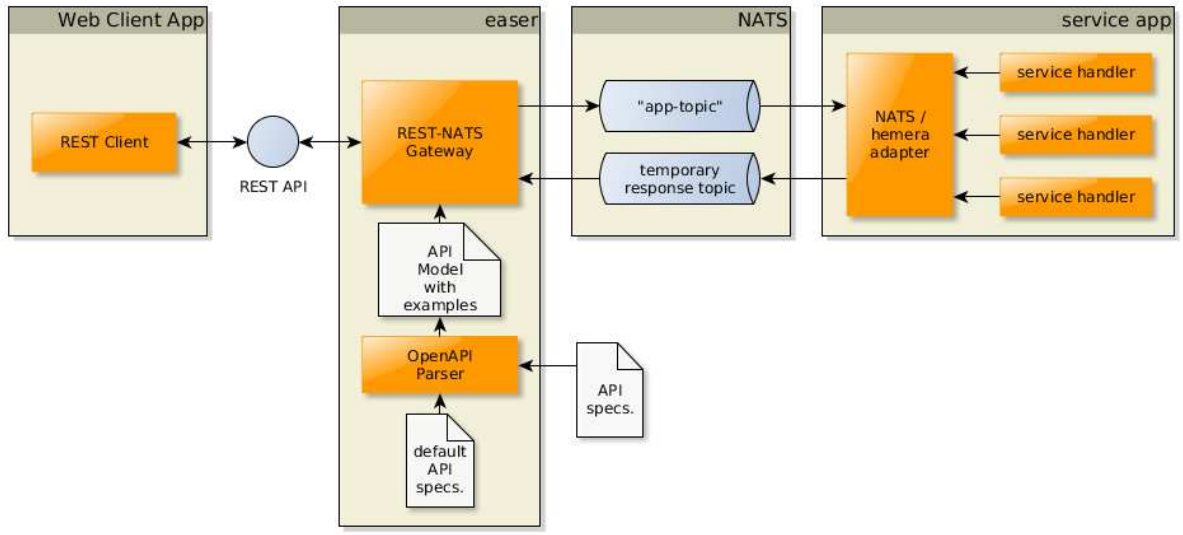

Figure 7: The Architecture of the Combined Mocking Mode

In order to combine the two modes, we need to switch on the NATS middleware and the message forwarding, but at the same time we also can enable the mocking as well:

So lets start the NATS:

```
$ docker run -it --rm --network=host -p 4222:4222 -p 6222:6222 -
p 8222:8222 --name nats-main nats
```

start, the service implementation:

```
$ node person-service-js/index.js
```

and start the `easer` with mocking and forwarding enabled:

```
$ easer -r person-rest-api/rest-api/api.yml -n nats://localhost:4222 -
-topicPrefix person-demo -u -m
```

Where: `--n nats://localhost:4222`: defines the URI of the NATS server; `---topicPrefix person-demo`: defines the prefix for the name of the topic to use. The topic name is generated by the following pattern: `<topicPrefix.<endpoint.method>_<endpoint.uri>`; `- -u`: Enables `easer` to forward the incoming calls as messages to the NATS topic; `- -m`: Enables the mocking (both static and dynamic as well if MESSAGING forwarding is enabled (`-m`)).

Now we will see that those endpoints, that has backing service handler, will work:

```
$ curl http://localhost:3007/persons
[]

$ curl -X PUT http://localhost:3007/persons/leia -H "Content-type: application/j
d '{"id":"leia","familyName":"Organa","givenName":"Leia"}'

{"id":"leia","familyName":"Organa","givenName":"Leia"}

$ curl http://localhost:3007/persons
[{"id":"leia","familyName":"Organa","givenName":"Leia"}]
```

What you can see above is that the same as in case of the normal working of REST / NATS gateway mode, which means that the service handlers are responding, however the mocking feature is also switched on. But when will the mocking happen? It happens, if there is an incoming call in relation to an endpoint, that has no registered service handler, but has examples in the swagger descriptor. In such a situation, the `easer` will detect that the service handler does not respond (`ERR {"name":"PatternNotFound","message":"No action found for this pattern"...}`) and the NATS connection is timed out, so it finds an example and sends back this static mock data to the client.

In the following example you can see that there is a request to the `POST /persons` endpoint, that has no registered handler function (it is commented out in the source code), so the static example is sent back:

```
$ curl -X POST http://localhost:3007/persons

{"id":"2a1152ee-4d77-4ff4-a811-598555937625","familyName":"Skywalker","givenName
```

2.6 Websocket / NATS Gateway

2.6.1 About WebSocket-NATS-Gateway feature of easer

WebSocket Server and Gateway to NATS topics using Pattern Driven Micro Service calls and asynchronous data pipelines. It makes possible the passing of messages among websocket clients and/or NATS clients.

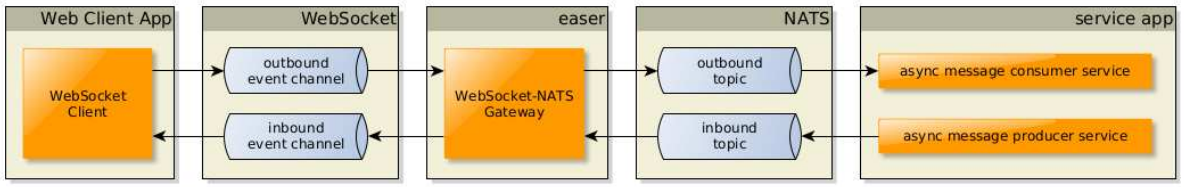

Figure 8: The WebSocket / NATS Gateway Mode Architecture

The WebSocket uses event handlers to manage the receiving and sending of messages. The websocket clients can subscribe to event names, that they observe, and act in case an incoming message arrives.

The messaging middlewares typically use the topic to name the channels through which the messages are transferred.

The messages can be forwarded back-and-forth between websocket event channels and messaging topics using their names to associate them.

Important Note: In the followings we will use the terms of inbound and outbound message channels. It refers to the grouping of event channels to which the messages will be send or received from by the websocket client. So They meant to be **inbound and outbound from the viewpoint of the websocket client** (or UI frontend application).

The functioning of the websocket gateway is quite simple:

1. We define the list of inbound and outbound event channel names.
2. The gateway will forward the messages coming in the outbound event channels to the NATS topics with the same name.
3. The gateway will also forward the messages coming in the NATS topics toward the inbound event channels.

2.6.2 Configure the websocket gateway

In order to use the websocket-nats gateway, we need to enable the MESSAGING mode (`-u`, `--useMessaging`) that switch on messaging in general, and the usage of websocket server (`-w`, `--useWebsocket`).

We also need to define the inbound (`-i`, `--inbound`) and outbound (`-o`, `--outbound`) event channels. We can define zero to many inbound and outbound names, separated by comma, for example: `-i "update,data,notification", or` -o "feedback,accept"', etc..

The following command makes easer to enable the usage of websocket gateway using the `IN` inbound and OUT outbound channels:

```
$ easer -u -w -i "IN" -o "OUT"

2022-03-08T09:16:06.373Z [easer@5.0.2] info: nats: Start up
2022-03-08T09:16:06.384Z [easer@5.0.2] info: nats: Connected to NATS
2022-03-08T09:16:06.395Z [easer@5.0.2] info: Start up webServer
2022-03-08T09:16:06.400Z [easer@5.0.2] info: Express server listening on por
2022-03-08T09:16:06.401Z [easer@5.0.2] info: wsServer: Start up wsServer ada
2022-03-08T09:16:06.402Z [easer@5.0.2] info: App runs the jobs...
```

As the server is configured, now we can use the gateway feature to forward messages either from websocket clients (e.g. from browser) to NATS clients (e.g. backend microservices), or vice versa.

2.6.3 Send messages from NATS clients to websocket client

We can connect to the server with a websocket client built-in to a frontend application running in a browser, but it is also possible to test the configuration without having a frontend.

We can test the working of the gateway with the wsgw tool, using its `wsgw producer` command to send messages from one side, and the `wsgw consumer` command to consume at the other side of the server.

For example, in one terminal window start receiving messages at the websocket side with the consumer:

```
wsgw consumer -u http://localhost:3007 -t "IN"

2022-03-08T09:15:19.704Z [wsgw@1.8.6] info: App runs the jobs...
```

```
2022-03-08T09:15:19.705Z [wsgw@1.8.6] info: wsgw client {"channelType":"WS",
2022-03-08T09:15:19.705Z [wsgw@1.8.6] info: Start listening to messages on W
```

Then send some message from the NATS side with the producer in another terminal:

```
$ wsgw producer -u nats://localhost:4222 -t "IN" -m '{"notes":"Some text..."
```

```
2022-03-08T09:16:16.292Z [wsgw@1.8.6] info: nats: Start up
2022-03-08T09:16:16.306Z [wsgw@1.8.6] info: nats: Connected to NATS
2022-03-08T09:16:16.306Z [wsgw@1.8.6] info: App runs the jobs...
2022-03-08T09:16:16.309Z [wsgw@1.8.6] info: {"notes":"Some text..."} >> [IN]
2022-03-08T09:16:16.310Z [wsgw@1.8.6] info: Successfully completed.
2022-03-08T09:16:16.310Z [wsgw@1.8.6] info: App starts the shutdown process.
2022-03-08T09:16:16.311Z [wsgw@1.8.6] info: nats: Shutting down
2022-03-08T09:16:16.311Z [wsgw@1.8.6] info: Shutdown process successfully fi
```

on the console, running the consumer, you should see something like this as a result:

```
...
2022-03-08T09:16:16.312Z [wsgw@1.8.6] info: WS[IN] >> "{\"notes\":\"Some tex
```

Important Note: Be careful, and pay attention on the URLs used within the commands! We use the same command to consume, and/or publish messages to the gateway, only the URL makes difference to determine which side the client will communicate with:

- When we want to connect to the websocket side, we use the `-u http://localhost:3007` argument,
- when we want to connect to the NATS side, we use the `-u nats://localhost:4222` argument.

2.6.4 Send messages from websocket client to NATS clients

We can send messages from the websocket side as well, using the same tools.

In one terminal window start receiving messages at the NATS side with the consumer:

```
wsgw consumer -u nats://localhost:4222 -t "OUT"
```

```
2022-03-08T09:38:30.367Z [wsgw@1.8.6] info: nats: Start up
2022-03-08T09:38:30.381Z [wsgw@1.8.6] info: nats: Connected to NATS
2022-03-08T09:38:30.382Z [wsgw@1.8.6] info: App runs the jobs...
2022-03-08T09:38:30.382Z [wsgw@1.8.6] info: wsgw client {"channelType":"NATS
2022-03-08T09:38:30.382Z [wsgw@1.8.6] info: Start listening to messages on N
```

Then send some message from the websocket with the producer in another terminal:

```
$ wsgw producer -u http://localhost:3007 -t "OUT" -m '{"notes":"Some text...

2022-03-08T09:39:30.325Z [wsgw@1.8.6] info: App runs the jobs...
2022-03-08T09:39:30.334Z [wsgw@1.8.6] info: {"notes":"Some text..."} >> [OUT
2022-03-08T09:39:30.355Z [wsgw@1.8.6] info: Successfully completed.
2022-03-08T09:39:30.356Z [wsgw@1.8.6] info: App starts the shutdown process.
2022-03-08T09:39:30.357Z [wsgw@1.8.6] info: Shutdown process successfully fi
```

on the console, running the consumer, you should see something like this as a result:

```
...
2022-03-08T09:39:30.355Z [wsgw@1.8.6] info: NATS[OUT] >> "{\"notes\":\"Some
```

For further details on how the websocket-NATS gatway is working, and on the usage of the `wsgw` commands, read the documentation of the `wsgw` tool.

2.7 Easer Internals

In order to have all the basic functions a cloud ready component should have, `easer` is built-upon the npac architecture, which is a lightweight Ports and Adapters Container for applications running on Node.js platform.

To act as MESSAGING Gateway, `easer` uses the built-in npac-webserver-adapter.

Note: There are two ways of implementing service modules with the npac-webserver-adapter:

1. Service implementations are built-into the server. in this case you need to make a standalone npac based server, using directly the npac-wsgw-adapters module, and integrate the endpoint implementations into this server. In this case the endpoint implementations have to be referred in the swagger files via the `operationId` properties of the endpoint descriptors.

2. The `easer` way: You implement a standalone service module, that listens to NATS topic (defined by the endpoint URI and method), define the API via swagger, and start the following system components: the NATS server, the service implementation module, and the `easer` server configured with the API descriptors.

With `easer` it possible to create two-way asynchronous communication between the frontend and the backing services. The frontend uses websocket and the `easer` forwards the messages towards NATS topics. it also works in the opposite direction, `easer` can subscibe to NATS topics and the received messages are forwarded towards the frontend via websocked. This feature is build upon the npac-wsgw-adapters module. There is helper tool called wsgw, that makes possible to publish to and subscribe for topics. This tool can send and recive messages through both NATS and websocket topics. See the README files of the mentioned modules and tools for details.