ExpressO: From Express.js Implementation Code to OpenAPI Interface Descriptions

Souhaila Serbout(✉), Alessandro Romanelli, and Cesare Pautasso

Software Institute (USI), Lugano, Switzerland
{souhaila.serbout,alessandro.romanelli,cesare.pautasso}@usi.ch

Abstract. The current paper presents a novel Command Line Interface (CLI) tool called ExpressO. This tool is specifically developed for developers who seek to analyze Web APIs implemented using the Express.js framework. ExpressO can automatically extract a specification written in OpenAPI, which is a widely used interface description language. The extracted specification consists of all the implemented endpoints, response status codes, and path and query parameters. Additionally, apart from facilitating automatic documentation generation for the API, ExpressO can also automatically verify the conformity of the Web API interface to its implementation, based on the Express.js framework.

The tool has been released on the npm component registry as 'expresso-api', and can be globally installed using the command:
`npm install -g expresso-api`.

Keywords: OpenAPI Specification · REST API · Express.js · Documentation generation

1 Introduction

In Continuous Software Development (CSD), the usage of modern software architecture tooling [18] and executable documentation is highly recommended [28, 30]. Ensuring that documentation is continuously consistent with the implementation throughout the software development cycle is required in order to avoid informal communication and tacit knowledge sharing between the software development team members [21]. Web APIs are a particular type of software for which producing up-to-date documentation is a must because it is a crucial artifact to support the API's learnability by developers [24]. Documenting small systems may be trivial, however when scaling up the size of the backend, producing and maintaining the desired documentation can prove challenging and quite resource-intensive.

As an attempt to solve this problem, ExpressO is a tool that helps Express.js [29] developers to generate documentation for their APIs taking nothing as input other than the backend code they already wrote. The obtained documentation is compliant with the OpenAPI specification [1]. While the automatically generated artifacts can be manually augmented with natural language

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
T. Batista et al. (Eds.): ECSA 2022, LNCS 13928, pp. 29–44, 2023.
https://doi.org/10.1007/978-3-031-36889-9_4

descriptions, easing the rapid generation of API documentation, ExpressO can also check the consistency of the interface extracted from the implementation with the existing documentation, thus highlighting gaps between the interface documentation and the corresponding implementation code.

We target the Express.js framework due to its wide adoption and the lack of tools that can extract the OpenAPI description only based on the implementation code itself. Existing tools such as Express OpenAPI [27] or swagger-autogen [2] require additional code annotations or time-consuming configuration steps to produce similar results.

2 Background: OpenAPI

APIs can be described using natural language, informal models, or general-purpose modeling languages. There exist also machine-readable Domain Specific Languages [14] for describing them, such as RAML [3], WADL [17], WSDL [11], I/O Docs [4], and OpenAPI [1], which gained more importance in the five last years by being selected as a standard language for APIs description.

For what concerns our tool, OpenAPI describes an API as a set of endpoints $\mathcal{E}$, which may receive zero or more parameters $\mathcal{P}$ and produce one or more expected responses for each endpoint $\mathcal{R}$.

$$APIComponents = \{c, c \in \mathcal{E} \cup \mathcal{P} \cup \mathcal{R}\}$$

From now on we call the endpoints, the parameters and the responses: 'API Components'.

OpenAPI descriptions comprehend also metadata about the API and description fields with values written in natural language. They also contain detailed descriptions of operations' request and response bodies, which can be specified using JSON Schema when exchanging JSON message payloads.

There is a broad set of emerging tools and approaches centered around the OpenAPI standard [5]: test cases generation [12,22], API analytics tools [13,26], as well as implementation code for client skeletons and server stubs (e.g. [23]). ExpressO focuses on the opposite problem: generating interface descriptions starting from the implementation code.

In our study, we utilize OpenAPI [20] as the standard target language for documenting the extracted Web API. To generate a valid OpenAPI specification, it is essential to extract all the necessary information from the backend code to populate the required fields in the OpenAPI metamodel. An example of such a specification, produced by the OpenAPI Initiative [19], is presented in Listing 1.1. We have highlighted the API Components that are essential for a valid specification in green. The fields that are not mandatory but can enhance the specification's detail are marked in orange. The optional fields that are typically written in natural language to provide additional insights about the API Components are shaded in gray.

Listing 1.1. Example of OAS3.0. Properties highlighting: green → must-have, orange → nice-to-have, gray → out of scope / user defined.

```
1  {
2    "openapi": "3.0.0",
3    "info": {
4      "version": "1.0.0",
5      "title": "Swagger Petstore",
6      "license": {
7        "name": "MIT"
8      }
9    },
10   "servers": [
11     {
12       "url": "http://petstore.swagger.io/v1"
13     }
14   ],
15   "paths": {
16     "/pets": {
17       "get": {
18         "summary": "List all pets",
19         "operationId": "listPets",
20         "tags": [
21           "pets"
22         ],
23         "parameters": [
24           {
25             "name": "limit",
26             "in": "query",
27             "description": "How many items to return at one time (max 100)",
28             "required": false,
29             "schema": {
30               "type": "integer",
31               "format": "int32"
32             }
33           }
34         ],
35         "responses": {
36           "200": {
37             "description": "A paged array of pets",
38             "headers": {
39               "x-next": {
40                 "description": "A link to the next page of responses",
41                 "schema": {
42                   "type": "string"
43                 }
44               }
45             },
46             "content": {
47               "application/json": {
48                 "schema": {
49                   "$ref": "#/components/schemas/Pets"
50                 }
51               }
52             }
53           }
54         }
55       }
56     }
57 }
```

Omitting the human-readable fields, such as `description`, `summary`, and `tags`, which may require user input, our proposed approach discussed in Sect. 5 can generate a rudimentary specification that covers all the essential fields, describing an endpoint in terms of its **path**, **HTTP methods**, and **response**

codes. Additionally, by conducting a static analysis of the routes, ExpressO can obtain the relevant **parameter** details.

3 Related Work

While other methods have attempted to extract structured REST API documentation from unstructured sources [10], our work focuses on generating documentation directly from the source code, assuming that the API has been implemented using the Express.js framework.

Unlike Express OpenAPI [27], which requires the user to provide an OpenAPI description containing the API's metadata as input and explicitly annotate each Express.js route with the corresponding OpenAPI metadata, ExpressO does not necessitate any input beyond the Express.js backend code. In the former case, the developer can add human-readable descriptions to be included in the resulting API, but the interface and implementation specifications are mixed in the same source code. Developers must rewrite the information already present in the previously written endpoint code. Express OpenAPI then combines the pieces to produce a coherent document that is served statically.

On the other hand, swagger-autogen [2] can be run with the backend to generate documentation each time the backend is executed. These modules are independent of any backend framework and assume that the backend implements routes following the conventions of Express.js.

Table 1 provides an overview of the input requirements for ExpressO, Express OpenAPI, and swagger-autogen. The comparison results of the ExpressO's Comparator are more granular and detailed than the ones produced by similar tools. `OAS Diff` [15] only provides a count of the modified or added API Components. Instead, `OpenAPI Diff by Microsoft Azure` [16] has as main goal to detect breaking changes as it outputs a report that classifies the changes affecting each API Component. An in comparison with other JSON/YAML diff tools ExpressO's Comparator provides a more domain specific reports grouping the detected differences by API components: endpoints, parameters, and responses.

4 Use Cases

In the design of the tool we envisioned the following use cases:

Table 1. Inputs required by different OpenAPI generation tools

		Code annotations	Basic description	Config file	Backend code
Express OpenAPI	2015	Yes	Yes	No	Yes
Swagger-autogen	2020	No	No	Yes	Yes
ExpressO	2022	No	No	No	Yes

1. Helping developers to keep both the implementation code and the interface documentation continuously synchronized; For that, the user can use the `'expresso generate'` CLI command to generate the new specification corresponding to the current version of the implementation.
2. Helping API designers to verify whether the implementation matches the structure they modeled and track the progress of an API development project; This corresponds to the CLI command `'expresso compare'` which compares two given input specifications, or the `'expresso test'` which generates an OpenAPI specification for the backend then compares it to an input reference specification. The tool will generate both a human-readable and a machine-readable report about what has been matched and what is missing for each specification.
3. Making it easier for developers to detect breaking changes by using ExpressO to compare the OpenAPI description of the current version of the API with a previously generated specification, also using the `'expresso compare'`. The comparison report can be used as ground truth to perform Regression Testing on the new changes;
4. Supporting researchers who want to perform empirical studies on real-world APIs. Using ExpressO they can automatically extract well-formatted knowledge from the Express.js source code of a large set of projects, by simply running the `'expresso generate'` for each of the projects.

5 ExpressO

5.1 Approach

ExpressO is a command-line interface (CLI) tool designed to extract the essential components from the source code of an Express.js project and produce a valid OpenAPI specification that describes the REST API. The tool uses a combination of static and dynamic analyses. The dynamic analysis involves injecting a Proxy component, which replaces the `express` npm package in the input project. This Proxy is used to intercept calls that configure the API routing table and extract the code of the corresponding function handlers. The Analyzer then performs static analysis on this code to extract information about the request parameters and response status codes.

The only input that the system requires to generate a specification for a REST API is the original source code. ExpressO first identifies the application entry point file and creates an abstract syntax tree that is scanned for all usages of the Express app that listens on a port. Once all the source files have been analyzed, a data structure holds the properties of the respective endpoints, routers, and applications.

To avoid altering the source code, ExpressO replaces the express.js NPM package with an instrumented version of the same, called the Proxy. The Proxy acts as an intermediary between the system and the original express functionality, allowing access to the latter whilst keeping track of calls being made by the 'Express.Application' object. By intercepting every call made to the Express

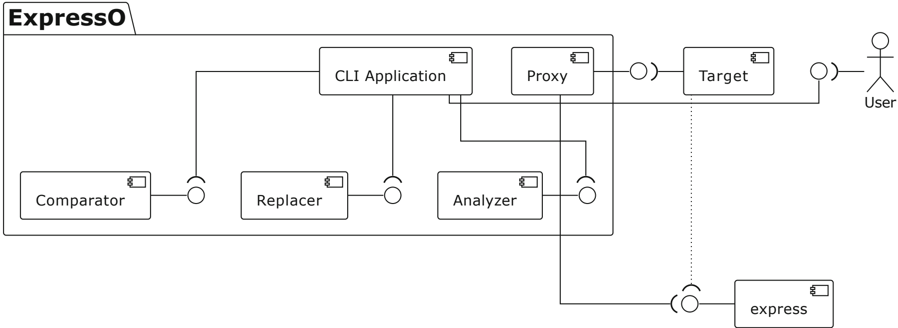


Fig. 1. Logical view of ExpressO

framework, the Proxy collects data in a data structure that is able to store all the information needed to reconstruct the API description. Specifically, the Proxy builds the API routing table by linking each endpoint path and method combination with the corresponding handler function. Then, by running a static analysis on the handler code, ExpressO retrieves the response codes and parameters of a given endpoint.

The hybrid approach employed by ExpressO, which combines both static and dynamic analysis techniques, enables the tool to extract the endpoints of deeply modular backends without having to statically analyze their entire code. While static analysis alone requires navigation through the imports and exports of several files, which can be a slow process when traversing large code-bases, the dynamic analysis component of ExpressO allows for more efficient extraction of endpoint information. By intercepting every call made to the Express framework, the Proxy component of ExpressO collects the data needed to reconstruct the API description without the need for time-consuming static analysis of all the backend code.

5.2 Architecture

We depict an overview of the logical view of the architecture of ExpressO in Fig. 1. In the rest of this section, we explain in detail the different software components part of the ExpressO tool, which will be separately demonstrated.

CLI Application. ExpressO utilizes the CLI Application interface to provide access to its functionalities. The available command lines are illustrated in Fig. 2. In the simplest case, where developers want to generate a specification without any customization, they can run the tool with the command `expresso generate`, as most parameters have default values. Additionally, other available commands are presented in Fig. 2.

Replacer. The Replacer module is responsible for generating a functioning copy of an Express.js backend that can be initiated by the CLI Application module as a Child Process. This module performs an additional task of substituting the original express package inside the `node_modules` directory with

the Proxy component. This replacement enables the working copy to utilize our Proxy module in the same way as the original express module, without requiring any modification to the input code. Moreover, the ExpressO module is retrieved from the `node_modules` of the npm global packages, which must be present to utilize the CLI Application. This approach eliminates the need to install `expresso-api` both locally and globally, reducing the burden on users to maintain both installations up-to-date.

```
[[~] expresso --help                                                          11:42:37 ]

 Usage: expresso [expresso-options] [command] [command-options]

 Available options:
  -H  --help     Prints to console command line commands and options
  -V  --version  Prints to console the current version of expresso

 Available commands:
  generate  Generates OpenAPI specification for the Express.js project in the current directory
  test      Generates OpenAPI specification and compares it with a user-provided ground truth
  compare   Compares two OpenAPI specifications regardless of version or format

 All commands can be further inspected with: expresso [command] [-H | --help]
```

Fig. 2. ExpressO Command Line Help

Proxy. The Proxy module plays a crucial role in intercepting all calls made to the Express framework and storing relevant information about the routes and corresponding request handler code. This information is then used to extract the API components necessary to generate a valid OpenAPI specification. It is important to note that the Proxy intercepts only the route configuration setup calls on the express framework itself and not the calls made to actual API endpoints by test clients. In other words, the Proxy operates at the backend start-up phase and captures the necessary information about the routes and handlers without actually executing them. While the command used to start the backend can be customized, the default command used in the experiments was `npm start` with no additional inputs.

Child Process. Upon invoking the Replacer and creating a working copy, the CLI Application generates a Child Process that executes the modified version of the project with the replaced module, including the Proxy. Our approach involves terminating the Child Process upon the first write-out to the intermediate model representation file.

Analyser. The analyzer module is responsible for parsing the intermediate model representation, which is stored as a JSON file, back into a working data structure. This allows the representation to be statically analyzed using the npm package abstract-syntax-tree [6].

Regarding parameters, for the first release of ExpressO, we limit the definition of parameters to two types: **Path** and **Query** parameters. Other parameter types, such as Cookies and Headers, are not supported in the furrent version of the tool.

When an endpoint is retrieved, the parameters used within that endpoint are detected by statically analyzing the route.

Comparator. The comparator module is designed to read and compare two OpenAPI descriptions (API_{source} and API_{target}), which can be written in either YAML or JSON format and may be in similar or different versions of OpenAPI. The comparison is done based on the criteria described in the rest of this section.

5.3 API Comparison and Coverage Report

While comparing the two API descriptions, the comparator computes and reports the following:

- **Matched**: the set of API Components present in both descriptions.

$$M = API_{source} \cap API_{target}$$

- **Partially matched**: when some API path parameters are present in both specifications, but their names do not exactly match.

$$PM = \{c, c \in API_{target} \wedge \exists c' \in API_{source} | c \approx c'\}$$

  The partial matching feature is implemented to address discrepancies in the naming of path parameters between the Express.js routes in the code and the corresponding paths in the target OpenAPI description being compared. This is because some developers may use different names for the same path parameter in the code and the OpenAPI description, even though they are semantically equivalent or referring to the same parameter. This can create issues during the comparison process.

  To address this, when performing the match, parameter names are not required since path parameters are positional and their name serves only to identify them as they are referenced from the code. Therefore, the names used in the target specifications are usually human-comprehensible, while the names generated from the implementation code are more direct.

  For example, consider a path p_i : `/users/userId` for an endpoint E_i in API_{source}, and a path p'_j : `/users/username` for an endpoint $E'j$ in API_{target}. Although the parameter names are different, these paths represent the same endpoint template with a fixed `/users/` segment followed by one parametric segment. Hence, they should be matched: $p_i \approx p'_j$. Consequently, we consider that the endpoints are partially matching: $E_i \approx E'_j$, irrespective of the path parameter names.

```
(base) ➜  backend-viajes git:(master) ✗ expresso compare expresso-openapi.json:swagger.yaml
Results for OpenAPI specification comparison between the following files:
  - expresso-openapi.json
  - swagger.yaml

Endpoints coverage ──────────────────────────── 83.33% (5/6 Endpoints)

Strict coverage (no partials) ──────────────── 50.00% (3/6 Endpoints)


Missing (1):
 GET  /

No extra entities detected

Matched (3):
 GET   /api/v1/travels
 POST  /api/v1/travels
 GET   /api/v1/travels/find

Partially matched (2):
 DELETE  /api/v1/travels/{id|_id}
 PATCH   /api/v1/travels/{id|_id}

Responses coverage ─────────────────────────── 20.00% (3/15 Responses)

Strict coverage (no partials) ──────────────── 20.00% (3/15 Responses)


Missing (12):
 200  GET     /
 403  GET     /api/v1/travels
 500  GET     /api/v1/travels
 500  POST    /api/v1/travels
 404  GET     /api/v1/travels/find
 500  GET     /api/v1/travels/find
 204  DELETE  /api/v1/travels/{id}
 404  DELETE  /api/v1/travels/{id}
 500  DELETE  /api/v1/travels/{id}
 201  PATCH   /api/v1/travels/{id}
 404  PATCH   /api/v1/travels/{id}
 500  PATCH   /api/v1/travels/{id}

Extra (7):
 400  GET     /api/v1/travels
 400  POST    /api/v1/travels
 400  GET     /api/v1/travels/find
 200  DELETE  /api/v1/travels/{_id}
 400  DELETE  /api/v1/travels/{_id}
 200  PATCH   /api/v1/travels/{_id}
 400  PATCH   /api/v1/travels/{_id}

Matched (3):
 200  GET   /api/v1/travels
 201  POST  /api/v1/travels
 200  GET   /api/v1/travels/find

No partially matched entities detected

Parameters coverage ────────────────────────── 100.00% (2/2 Parameters)

Strict coverage (no partials) ──────────────── 0.00% (0/2 Parameters)


No missing entities detected

Extra (3):
 x-user      GET  /api/v1/travels
 id_cliente  GET  /api/v1/travels/find
 estado      GET  /api/v1/travels/find

No matched entities detected

Partially matched (2):
 id|_id  DELETE  /api/v1/travels/{id|_id}
 id|_id  PATCH   /api/v1/travels/{id|_id}
```

Fig. 3. Snapshot of a human-readable coverage report example generated by ExpressO for an Express.js project found on GitHub [7]

– **Missing**: elements that are only present in the target specification;

$$MISS = \{c, c \in API_{target} \wedge c \notin API_{source}\}$$

– **Additional**: elements that are only present in the compared specification;

$$ADD = \{c, c \in API_{source} \wedge c \notin API_{target}\}$$

Although the comparator module can be utilized to compare any two HTTP APIs described using OpenAPI, its primary purpose is to assess the level of coverage achieved when comparing the generated description with a ground truth description. To achieve this, we have established two metrics to evaluate the coverage level of each API component:

– **Strict Coverage**: how many matched API Components over the total number of API_{source} Components;

$$C_{\text{strict}} = \frac{\text{size}(M)}{\text{size}(API_{source})}$$

– **Broad Coverage**: how many matched and partially matched API Components over the total number of API_{source} Components;

$$C_{\text{broad}} = \frac{\text{size}(M) + \text{size}(PM)}{\text{size}(API_{source})}$$

The objective of these metrics is to accurately quantify the degree to which generated documentation matches the ground truth (the specification found in the software repository). While manually-created documentation can be more exhaustive, our primary focus is on ensuring that we can accurately identify all endpoints along with their associated responses and parameters.

In addiction to printing the coverage metrics in machine-readable JSON file format, this module also serves as a reporting tool that outputs the comparison data in a human-readable format, producing a report in the terminal (as shown in Fig. 3).

6 Evaluation

In [25], we presented an extensive evaluation of ExpressO using a dataset of 91 Express.js projects collected from GitHub. These projects have been selected because they include the OpenAPI description of the corresponding API that we used to compute the coverage metrics, using the original specification shared in the software repository as a ground truth.

Web first selected projects that can be directly run with an `npm install && npm start` command, then filtered the ones containing an OpenAPI description file. We then remove from the dataset the projects that take more than 10s to start with an `npm start`. In Fig. 4 we show in details the dataset filtering decision tree, and in Table 2 we show the APIs size distribution computed from the OpenAPI descriptions found in each of the 91 remaining projects.

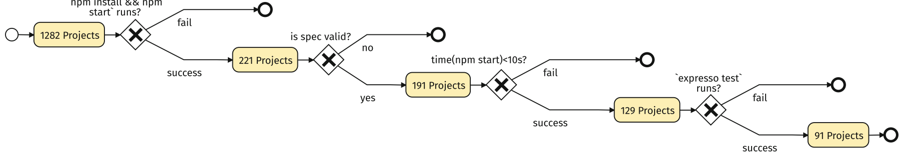
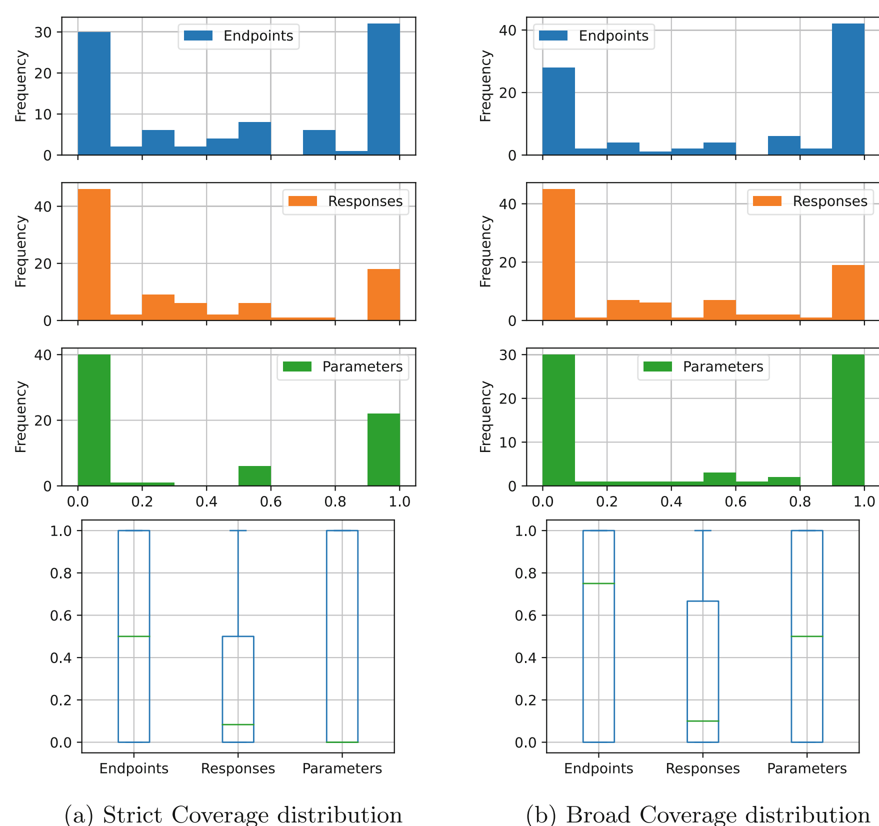


Fig. 4. The decision tree followed to filter the Express.js projects used in the evaluation of ExpressO

Table 2. Distribution of number of paths and operations metrics

	μ	σ	**Min**	**25%**	**50%**	**75%**	**Max**
#Paths	4.42	4.18	1	2	3	5	30
#Operations	6.38	6.92	1	2	5	7	53



(a) Strict Coverage distribution (b) Broad Coverage distribution

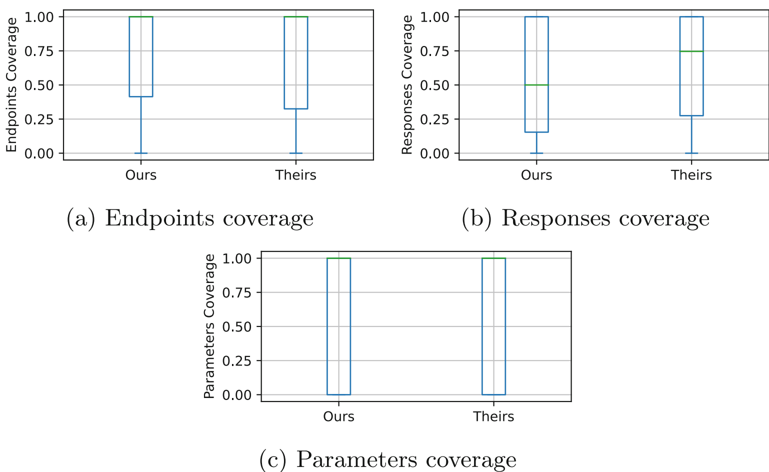
Fig. 5. C_{strict} and C_{broad} computed values distribution in 91 projects

In Fig. 5 it can be observed that the population of parameters seems to be smaller compared to that of endpoints and responses. Moreover, considering partial matches enables ExpressO to reach a higher coverage level.

As swagger-autogen requires a time-consuming manual configuration step for every new project, we restricted our study to a smaller subset of 23 working projects for comparative evaluation, after verifying the correctness of the produced output. Although ExpressO could generate valid specs for all 91 projects, it took us 33 attempts on different repositories to generate 23 usable specifications. In 30% of cases, swagger-autogen failed to produce valid documentation.

6.1 API Components Coverage

Fig. 6 depicts the computation of the C_{strict} metric by comparing the specifications generated by ExpressO and swagger-autogen against the specifications found in the projects which are considered a ground truth. A values of 1 indicates that all the features in the original specification were matched, while a value of 0 indicates that none of the features were matched.



(a) Endpoints coverage

(b) Responses coverage

(c) Parameters coverage

Fig. 6. Computed values of C_{strict} of ExpressO's vs. swagger-autogen

6.2 Performance

Time Taken by ExpressO. Because we are able to profile our system, when we ran it on the dataset of 20 repositories we were able to differentiate between time taken by the Replacer, Analyzer and Child Process components separately. This gives us the median execution time breakdown visualized in Fig. 7: Child Process 2017 ms; Analyzer 123 ms;

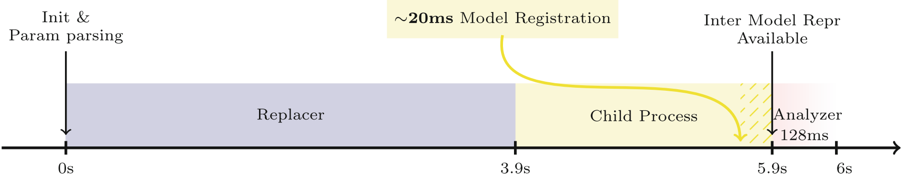


Fig. 7. Timeline of `expresso generate` command at the component level

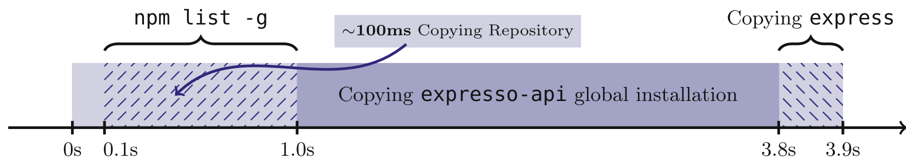


Fig. 8. Timeline of `Replacer` component

As we can see from Fig. 8, the repository size has a very minor effect on the overall time.

ExpressO vs. Swagger-Autogen. While in the case of ExpressO, the time that matters is the one it takes to analyze code, and generate the specification, in the case of swagger-autogen to produce a specification it is needed to keep in mind that it involves manual configuration. We distinguish: (1) Time To Start (TTS): the time elapsed from when a project is cloned and installed, to the moment that we are able to run our analysis; (2) Time To Run (TTR): the time taken to analyze the backend and produce the specification.

In the following results, our main focus was on TTR, particularly in comparing the performance of Analyzer with swagger-autogen. To evaluate the TTR of swagger-autogen accurately, we need to consider the time required to set up the swagger.js file by configuring it correctly. This is a manual activity that varies in duration depending on the user's experience with swagger-autogen and familiarity with the backend. In the initial run, we consider our system to have superior performance if a user cannot complete the necessary steps to use swagger-autogen within 5.8 s (Table 3). In subsequent runs, this manual step is no longer necessary to be fair.

Table 3. Performance Comparison (Average Execution Time)

	TTS	TTR	Total
`swagger-autogen`	>0	265 ms	>265 ms
`expresso`	5917 ms	123 ms	6040 ms

ExpressO is not affected by the total size of the input code in the same way as a completely static analysis, as it only needs to listen for calls made to the Express.js framework, instead of parsing all the project files and walking through its entire structure. This can be observed in Fig. 9 and Table 4, where we calculate the Pearson correlation between the express project size measured in lines of code (LOC) and the time taken by the tools (TTR), in order to establish if there is a statistical correlation between these two variables. The first row of the table shows that there is a strong correlation between the LOC of a project and the time taken by swagger-autogen, while the correlation between the API size (number of endpoints) and the TTR of ExpressO is medium.

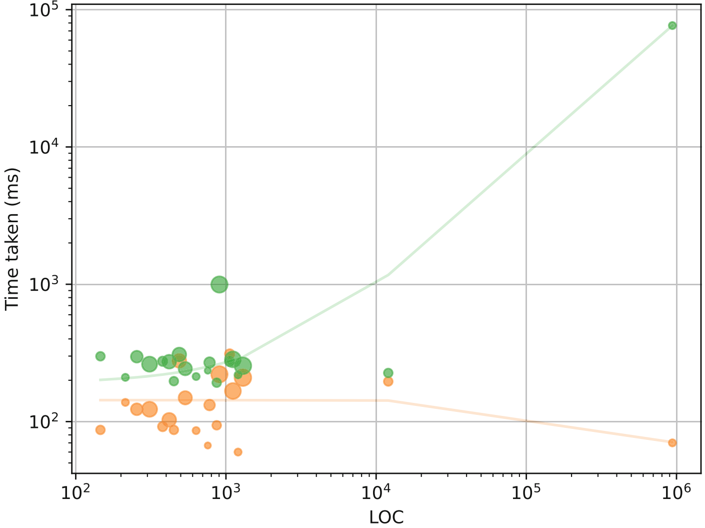


Fig. 9. Logarithmic scatter plot comparison between **Analyzer** (orange) and `swagger-autogen` (green) timings, with linear trendlines. The size of a datapoint relates to the number of declared endpoints

Table 4. Corr. of Time To Run (TTR) against implementation size (LOC) and output API size (Endpoints)

		Correlation	***p*-value**
LOC	`swagger-autogen`	0.9999	0.000
LOC	`expresso`	–0.2292	0.3310
Endpoints	`expresso`	0.4841	0.030
Endpoints	`swagger-autogen`	–0.220	0.3501

7 Conclusion

In this paper, we introduce ExpressO, a tool that automatically extracts a skeleton of OpenAPI descriptionsfrom the corresponding JavaScript implementation based on the Express.js framework. Unlike existing automatic generation tools such as swagger-autogen, ExpressO does not require any time-consuming manual configuration and can be immediately used on any Express.js compliant project. While most existing tools for extracting interfaces from implementation support a code-first approach to API development, where the implementation is manually annotated with metadata that should be extracted and published as part of the OpenAPI description, ExpressO supports an API-first approach [9]. It can compare the API description extracted from the code with a given OpenAPI description to verify that the paths, operations, response codes, and parameters are implemented as advertised. ExpressO instead supports an API-first approach [9], as it can compare the API description extracted from the code with a given OpenAPI description to check whether the paths, operations, response codes, and parameters are indeed implemented as advertised.

Although currently, the tool only supports Express.js backends, its hybrid approach combining static and dynamic analysis can be applied to other backend frameworks whose route configuration settings can be instrumented and intercepted similarly. We aim to extend the tool to support a broader range of inputs (APIs whose backend is implemented using other frameworks and other programming languages) and outputs (API descriptions conforming to other specifications, e.g. RAML). ExpressO is freely available on the npm registry under the "expresso-api" name [8].

Acknowledgements. This work was partially supported by the SNF with the API-ACE project number 184692.

References

1. OpenAPI Initiative. https://www.openapis.org/
2. https://github.com/davibaltar/swagger-autogen
3. RAML. https://raml.org/
4. I/O Docs. https://support.mashery.com/docs/read/IO_Docs
5. OpenAPI.Tools. https://openapi.tools/
6. https://www.npmjs.com/package/abstract-syntax-tree
7. https://github.com/FIS-Proyecto-Equipo1/backend-viajes
8. https://www.npmjs.com/package/expresso-api
9. Beaulieu, N., Dascalu, S.M., Hand, E.: API-first design: a survey of the state of academia and industry. In: Latifi, S. (ed.) 19th International Conference on Information Technology-New Generations, ITNG 2022. AISC, vol. 1421, pp. 73–79. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-97652-1_10
10. Cao, H., Falleri, J.-R., Blanc, X.: Automated generation of REST API specification from plain HTML documentation. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601, pp. 453–461. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69035-3_32

11. Christensen, E.: Web services description language (wsdl) 1.1. https://www.w3.org/TR/2001/NOTE-wsdl-20010315 (2001)
12. Corradini, D., Zampieri, A., Pasqua, M., Ceccato, M.: Restats: a test coverage tool for restful APIS. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 594–598. IEEE (2021)
13. Di Lauro, F., Serbout, S., Pautasso, C.: Towards large-scale empirical assessment of web APIs evolution. In: Brambilla, M., Chbeir, R., Frasincar, F., Manolescu, I. (eds.) ICWE 2021. LNCS, vol. 12706, pp. 124–138. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-74296-6_10
14. Fowler, M.: Domain-Specific Languages. Pearson Education, London (2010)
15. GitHub-Repository: Openapi diff. https://github.com/tufin/oasdiff
16. GitHub-repository: Openapi diff by microsoft azure. https://github.com/Azure/openapi-diff
17. Hadley, M.J.: Web application description language (wadl). Technical Report, USA (2006)
18. Hasselbring, W.: Software Architecture: Past, Present, Future. In: The Essence of Software Engineering, pp. 169–184. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73897-0_10
19. Initiative, O.: Oas v3.0 petstore example. https://github.com/OAI/OpenAPI-Specification/blob/main/examples/v3.0/petstore.json
20. Initiative, O.: Openapi v3.1.0 specification. https://spec.openapis.org/oas/v3.1.0
21. Jongeling, R., Fredriksson, J., Ciccozzi, F., Cicchetti, A., Carlson, J.: Towards consistency checking between a system model and its implementation. In: Babur, Ö., Denil, J., Vogel-Heuser, B. (eds.) ICSMM 2020. CCIS, vol. 1262, pp. 30–39. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58167-1_3
22. Karlsson, S., Čaušević, A., Sundmark, D.: Quickrest: property-based test generation of openapi-described restful APIS. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp. 131–141. IEEE (2020)
23. Koren, I., Klamma, R.: The exploitation of OpenAPI documentation for the generation of web frontends. In: Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW 2018, pp. 781–787 (2018)
24. Robillard, M.P.: What makes APIS hard to learn? answers from developers. IEEE Softw. **26**(6), 27–34 (2009)
25. Romanelli, A.: ExpressO. Master's thesis, Faculty of Informatics, University of Lugano (2022). https://thesis.bul.sbu.usi.ch/theses/2035-2122Romanelli/pdf?1674133800
26. Serbout, S., Di Lauro, F., Pautasso, C.: Web APIS structures and data models analysis. In: 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C), pp. 84–91. IEEE (2022)
27. Spencer, J.: Express openapi. https://github.com/kogosoftwarellc/open-api
28. Theunissen, T., van Heesch, U., Avgeriou, P.: A mapping study on documentation in continuous software development. Inf. Softw. Technol. **142**, 106733 (2022)
29. TJ Holowaychuk, S., et al.: Express.js documentation. https://expressjs.com/en/5x/api.html
30. Van Heesch, U., Theunissen, T., Zimmermann, O., Zdun, U.: Software specification and documentation in continuous software development: a focus group report. In: Proceedings of the 22nd European Conference on Pattern Languages of Programs, pp. 1–13 (2017)

View publication stats