ExpressO: From Express.js implementation code to OpenAPI interface descriptions

Alessandro Romanelli, Souhaila Serbout, and Cesare Pautasso
alessandro.romanelli@usi.ch, souhaila.serbout@usi.ch, c.pautasso@ieee.org

Software Institute (USI), Lugano, Switzerland

Abstract. This tool demo paper brings forward a new CLI tool called ExpressO for developers who need to analyze a Web API implemented using the Express.js framework, and automatically extract a specification written in OpenAPI, a standard interface description language. The generated specification includes all of the implemented endpoints along with their response status codes and path and query parameters. In addition to automatic API documentation generation, developers can also use it to automatically determine whether the interface of a Web API matches 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 [16] and executable documentation is highly recommended [25, 27]. 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 [17]. 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 [20]. 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 [26] 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 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 [24] or swagger-autogen [6] 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 [12] for describing them, such as RAML [2], WADL [15], WSDL [9], I/O Docs [3], 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: test cases generation [10, 18] , API analytics tools [23, 11], as well as implementation code for client skeletons and server stubs (e.g. [19]). ExpressO focuses on the opposite problem: generating interface descriptions starting from the implementation code.

3 Related work

While other approaches tried to extract structured REST API documentation from non-structured resources [8], in this work we focus on documentation generation directly from the source code, under the assumption that the API has been implemented using the Express.js framework.

While ExpressO does not require the user to feed it with any input other than the Express.js backend code, Express OpenAPI [24] requires the user to give as input an OpenAPI description containing the API's metadata. Moreover, the tool requires developers to explicitly annotate each Express.js routes with the corresponding OpenAPI metadata. In this case, the developer can also include natural language descriptions that will be included in the resulting API.

However, the specification of the interface and the implementation are mixed in the same source code.

swagger-autogen [6] allows the user to run it together with the backend to generate documentation each time the backend is run. The modules are completely independent of any backend framework and make the only assumption of having the backend implement routes following the Express.js conventions.

In Table 1 we summarize the different types of inputs required by 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` [13] only provides a count of the modified or added API Components. Instead, `OpenAPI Diff by Microsoft Azure` [14] has as main goal to detect breaking changes as it outputs a report that classifies the changes affecting each API Component.

4 Use Cases

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

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.

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

5 ExpressO

5.1 Approach

ExpressO is a CLI-based tool that extracts from the source code of an Express.js project the necessary components to describe the REST API in a valid OpenAPI specification. The tool combines both static and dynamic analysis. The dynamic part of the analysis consists of the injection of a Proxy. It is a component that substitutes the `express` npm package in the input project, in order to intercept the calls to configure the API routing table and extract the code of the corresponding function handlers. Then the Analyzer statically analyzes such code to gather request parameters and response status codes.

The only input that the system requires to generate a specification is the original source code. The tool does not alter the source code, as it simply automatically replaces the express.js NPM package with an instrumented version of the same: the Proxy.

Our proxy acts as intermediary, whilst providing access to the original express functionality, but also keeping track of calls being made by the 'Express.Application' object. By intercepting every call made to the Express framework, the tool collects a data structure within the Proxy that is able to store all the information needed to reconstruct the API description. In particular, the Proxy builds the API routing table linking each endpoint path and method combination with the corresponding handler function. And, by running a static analysis on the handler code we retrieve the response codes, parameters of a given endpoint.

This hybrid approach mixing static and dynamic analysis makes it possible to retrieve the endpoints of deeply modular backends, without having to statically analyze their entire code.

5.2 Architecture

We depict an overview of the logical view of the architecture of ExpressO in Figure 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 The interface that ExpressO uses to expose the system's functionalities is the CLI Application. The supported command lines are shown in Figure 2. Most parameters have default values so that in the simplest case developers can run the tool with `expresso generate`. The other commands are shown in Figure 2.

Replacer The replacer module is responsible for creating a working copy of an Express.js backend that can be started by the CLI Application module as a Child Process. It also overrides the express package by replacing it inside the `node_modules` folder with the Proxy component.

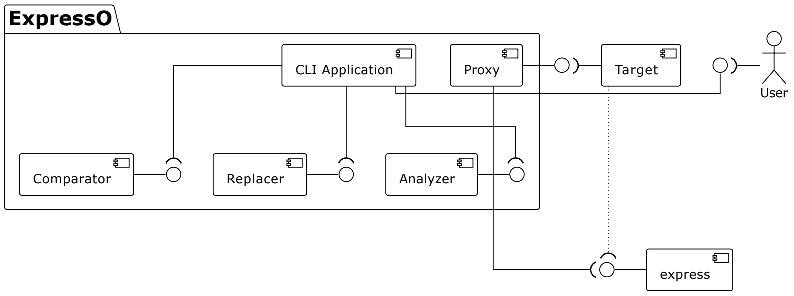
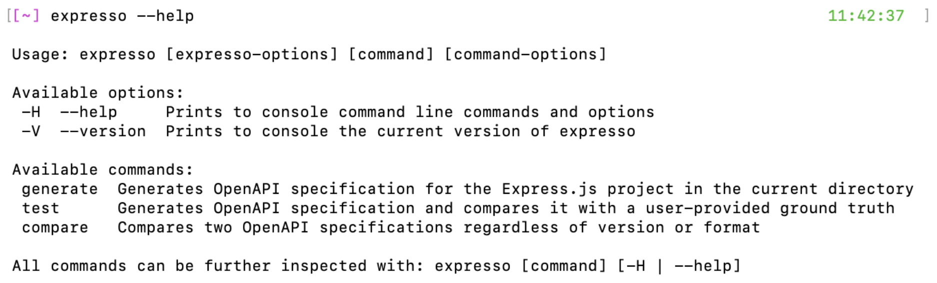


Fig. 1: Logical view of ExpressO



```
[[~] 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 takes care of intercepting all calls to the express framework and storing all the information representing the routes and the corresponding request handler code that is used to extract the API components used to later generate a valid OpenAPI specification.

It is worth noting that no calls to the actual API endpoints need to be performed by any test client, as the Proxy is meant to intercept the route configuration setup calls on the express framework itself, which are usually performed when the API backend starts. While the command used to start the backend can be customized, in our experiments we used the basic `npm start` command with no extra input.

Analyser The analyzer module is in charge of parsing the intermediate model representation – stored as JSON – back into a working data structure so that it can be statically analyzed using the abstract-syntax-tree [5] npm package.

Comparator The comparator module reads and compares two given OpenAPI descriptions ($API_{source}$and API_{target}), which can be written in similar or different versions of OpenAPI, in either YAML or JSON format, according to 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.

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

  This partial matching helps to reconcile differences in which developers may name path parameters in the Express.js routes (found in the code) and in the corresponding paths of the target OpenAPI description being compared. 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. As a consequence, within the target specifications, they are usually human-comprehensible names, whereas the name obtained when we generate our specification comes directly from the implementation code.
  For instance: p_i : `/users/{userId}` a path of an endpoint E_i in API_{source} and p'_j : `/users/{username}` a path of E'_j in API_{target}. These two paths represent the same endpoint template, with a fixed `/users/` segment followed by one parametric segment. Therefore they can and should be matched: $p_i \approx p'_j$. Thus, we consider that the endpoints are partially matching: $E_i \approx E'_j$ independently of the path parameter names.
- **Missing**: elements that are only present in the target specification;

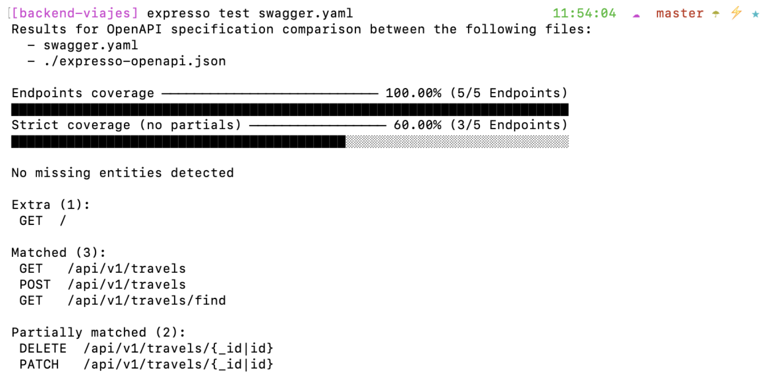
$$MISS = \{c, c \in API_{target} \land c \notin API_{source}\}$$

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

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

While the comparator module can be also used to compare the structure of any two HTTP APIs described using OpenAPI, it is mainly designed to measure the level of coverage that has been reached when comparing the generated description to a ground truth description. For that, we define two metrics to asses the level of coverage on the level of each API Component:

- **Strict Coverage**: how many matched API Components over the total number of API Components; $C_{\text{strict}} = \frac{\text{size}(M)}{\text{total}}$
- **Broad Coverage**: how many matched and partially matched API Components over the total number of API Components; $C_{\text{broad}} = \frac{\text{size}(M)+\text{size}(PM)}{\text{total}}$



```
[backend-viajes] expresso test swagger.yaml                       11:54:04  master
Results for OpenAPI specification comparison between the following files:
  - swagger.yaml
  - ./expresso-openapi.json

Endpoints coverage ─────────────────────────── 100.00% (5/5 Endpoints)
██████████████████████████████████████████████████████████████████████
Strict coverage (no partials) ───────────────── 60.00% (3/5 Endpoints)
██████████████████████████████████████████░░░░░░░░░░░░░░░░░░░░░░░░░░░░

No missing entities detected

Extra (1):
 GET  /

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}
```

Fig. 3: Snapshot of a portion of the human readable coverage report generated by ExpressO for an Express.js project found in GitHub [4]

The goal of these metrics is to concretely measure how close the generated documentation is to the ground truth. Even though human-generated documentation can be more detailed, we are mostly concerned with making sure that we can correctly list all the endpoints with their relative responses and parameters.

Besides printing the coverage metrics of the API Components in a machine-readable file format (JSON). This module also acts as a reporting tool that outputs the comparison data to a human-readable output, generating a report in the terminal (Figure 3).

6 Performance Evaluation

In [22], 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.

As swagger-autogen requires a time-consuming manual configuration step every time it needs to be used with a new project, in our study – after checking the correctness of the produced output – for the comparative evaluation, we further restricted the sample to a smaller subset of 23 working projects.

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.

To the TTR of swagger-autogen, we should add the time required to set up the swagger.js file by configuring it correctly. This is a manual activity that

Table 2: Performance Comparison (Average Execution Time)

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

Table 3: 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

requires different times depending on the user's experience with swagger-autogen and familiarity with the backend. During the first run, we may consider our system to have a superior performance if a user is unable to perform the required steps to use swagger-autogen within 5.8 seconds (Table 2). To be fair, in the following executions, such a manual step is no longer required.

In Table 3, we compute the Pearson correlation between the express project size measure in lines of code (LOC) and the TTR time taken by the tools to try to establish whether there is a statistical correlation between these two variables. In the first row, we can see that there is a high 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.

7 Conclusion

In this tool demo paper, we have introduced ExpressO, a tool for automatically extracting a skeleton of OpenAPI descriptions from the corresponding JavaScript implementation based on the Express.js framework. As opposed to existing automatic generation tools such as swagger-autogen, ExpressO does not require any time-consuming manual configuration as it can be used immediately on any Express.js compliant project. Most existing tools for extracting interfaces from implementation are meant to 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 instead supports an API-first approach [7], 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.

While the tool currently 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 in a similar way. We are working on extending the tool to support a broader set 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 [21].

References

1. OpenAPI Initiative. https://www.openapis.org/, accessed: 2021-06-01
2. RAML. https://raml.org/, accessed: 2021-06-01
3. I/O Docs. https://support.mashery.com/docs/read/IO_Docs, accessed: 2021-06-01
4. backend-viajes. https://github.com/FIS-Proyecto-Equipo1/backend-viajes, accessed: 2022-07-01
5. Ajdyna, E.: abstract-syntax-tree, https://www.npmjs.com/package/abstract-syntax-tree
6. Baltar, D.: swagger-autogen, https://github.com/davibaltar/swagger-autogen
7. Beaulieu, N., Dascalu, S.M., Hand, E.: Api-first design: A survey of the state of academia and industry. In: ITNG 2022 19th International Conference on Information Technology-New Generations. pp. 73–79. Springer (2022)
8. Cao, H., Falleri, J.R., Blanc, X.: Automated generation of rest api specification from plain html documentation. In: International Conference on Service-Oriented Computing. pp. 453–461. Springer (2017)
9. Christensen, E.: Web services description language (wsdl) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315 (2001)
10. 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)
11. Di Lauro, F., Serbout, S., Pautasso, C.: Towards Large-Scale Empirical Assessment of Web APIs Evolution. In: Proc. International Conference on Web Engineering (ICWE 2021), pp. 124–138 (2021)
12. Fowler, M.: Domain-specific languages. Pearson Education (2010)
13. GitHub-Repository: Openapi diff, https://github.com/tufin/oasdiff
14. GitHub-Repository: Openapi diff by microsoft azure, https://github.com/Azure/openapi-diff
15. Hadley, M.J.: Web application description language (wadl). Tech. rep., USA (2006)
16. Hasselbring, W.: Software architecture: Past, present, future. In: The Essence of Software Engineering, pp. 169–184. Springer, Cham (2018)
17. Jongeling, R., Fredriksson, J., Ciccozzi, F., Cicchetti, A., Carlson, J.: Towards consistency checking between a system model and its implementation. In: International Conference on Systems Modelling and Management. pp. 30–39. Springer (2020)
18. 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)
19. 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 '18. pp. 781–787 (2018)
20. Robillard, M.P.: What makes apis hard to learn? answers from developers. IEEE software **26**(6), 27–34 (2009)
21. Romanelli, A.: expresso-api, https://www.npmjs.com/package/expresso-api
22. Romanelli, A.: ExpressO. Master's thesis, Faculty of Informatics, University of Lugano (2022)
23. 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)

24. Spencer, J.: Express openapi, https://github.com/kogosoftwarellc/open-api
25. Theunissen, T., van Heesch, U., Avgeriou, P.: A mapping study on documentation in continuous software development. Information and Software Technology **142**, 106733 (2022)
26. TJ Holowaychuk, S., et al.: Express.js documentation, https://expressjs.com/en/5x/api.html
27. 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)