REST in Pieces: RESTful Design Rule Violations in Student-Built Web Apps *

Sergio Di Meglio[1], Valeria Pontillo[2,3], and Luigi Libero Lucio Starace[1]

[1] Department of Electrical Engineering and Information Technology, University of
Naples Federico II, Italy
sergio.dimeglio@unina.it, luigiliberolucio.starace@unina.it
[2] Software Languages (SOFT) Lab, Vrije Universiteit Brussel, Belgium
[3] Gran Sasso Science Institute (GSSI), L'Aquila, Italy
valeria.pontillo@gssi.it

Abstract. In Computer Science Bachelor's programs, software quality
is often underemphasized due to limited time and a focus on foundational
skills, leaving many students unprepared for industry expectations. To
better understand the typical quality of student code and inform both
education and hiring practices, we analyze 40 full-stack web applications
developed in a third-year Web Technologies course. Using an automated
static analysis pipeline, we assess adherence to REST API design rules.
Results reveal frequent violations of foundational conventions, such as
missing hyphens in endpoint paths (98%), incorrect pluralization (88%),
and misuse of HTTP methods (83%). These findings highlight the need
for more focused instruction on API design and support the adoption of
automated tools to improve code quality in student projects.

Keywords: Web Applications· Software Quality· Static Analysis· REST

1 Introduction

In Computer Science Bachelor's programs, software development education of-
ten emphasizes functional application building over software quality, consistent
with ACM guidelines recommending limited early focus on quality aspects [4].
Time constraints and difficult pedagogical trade-offs further limit the compre-
hensive teaching of software quality principles [5]. Moreover, empirical data on
the structural and architectural quality of realistic student projects—especially
full-stack applications—are scarce, hindering informed curriculum decisions and
impacting industry readiness [21].

This gap has significant industry implications: graduates lacking solid skills in
clean, maintainable code shift the retraining burden to employers, who must in-
vest resources to bring junior developers up to production standards [16]. Bridg-
ing this academia-industry divide requires a better understanding of student
software quality.

To address this gap, we conducted an empirical investigation on 40 full-stack web applications developed by students enrolled in a third-year Web Technologies course within a Computer Science Bachelor's program. This course, typically taken in the final semester before graduation, requires students to design and implement complete web systems that integrate both front-end and back-end components. To this end, we designed a semi-automated analysis pipeline that leverages RESTRULER, a tool to identify REST API design rule violations, an indicator of structural degradation and design decay, commonly associated with the accumulation of social debt in collaborative software projects. Our findings indicate that while students delivered functional full-stack applications, they frequently violated key REST design conventions—most notably the use of hyphens in endpoint paths (98% of projects), plural nouns for collections (88%), and correct use of GET methods (83%).

These insights can inform teaching strategies to better prepare students for professional development. A replication package with all analysis artifacts is available to support further research and validation [14].

2   Background and Related Works

In the following, we introduce key concepts related to software quality, with a focus on design-level aspects specific to RESTful APIs. We also review existing tools for assessing REST API design rule violations and summarize prior research on the quality of student-developed software systems.

Software quality refers not only to the absence of defects but also to architectural and structural design, which supports maintainability, usability, and long-term sustainability [13]. In web applications, where systems interact through Web APIs, API design plays a crucial role: poor design can lead to misuse and increase maintenance costs [7, 13].

The Representational State Transfer (REST) architectural style, introduced by Fielding [9], has become the dominant paradigm for designing modern Web APIs. REST is not a protocol or a standard, but an architectural style defined by a set of constraints: statelessness, client-server separation, cacheability, a uniform interface, a layered system, and optional code-on-demand. APIs that fully adhere to these constraints are called REST APIs, and services that implement them are often referred to as RESTful [12].

A REST API is composed of interlinked resources, and this assembly of resources forms the REST API resource model. Each resource is uniquely addressable via URIs and manipulated through a set of well-defined operations using standard HTTP methods (e.g., `GET`, `POST`, `PUT`, `DELETE`). In a RESTful system, the structure and semantics of URIs, HTTP methods, response codes, and payload formats should follow clear and consistent patterns [19].

Designing a REST API, however, is not always straightforward. Although some guidelines are implied by the HTTP standard itself, others have emerged as best practices adopted by the community over time. These range from URI naming conventions to error handling and response formatting. Examples in-

Table 1: The 14 rules implemented to RESTRULER [2].

Rule Description	Identifier	Category
`401 ("Unauthorized")` must be used when there is a problem with the client's credential	RC401	HTTP Status Codes
A plural noun should be used for collection or store names	PluralNoun	URI Design
A singular noun should be used for document names	SingularNoun	URI Design
A trailing forward slash (/) should not be included in URIs	NoTrailingSlash	URI Design
A verb or verb phrase should be used for controller names	VerbController	URI Design
CRUD function names should not be used in URIs	NoCRUDNames	URI Design
Content-Type must be used	ContentType	Metadata Design
Description of request should match with the type of the request	DescriptionType	Metadata Design
Forward slash separator (/) must be used to indicate a hierarchical relationship	ForwardSlash	URI Design
`GET` and `POST` must not be used to tunnel other request methods	NoTunnel	Request Methods
`GET` must be used to retrieve a representation of a resource	GETRetrieve	Request Methods
Hyphens (-) should be used to improve the readability of URIs	Hyphens	URI Design
Lowercase letters should be preferred in URI paths	Lowercase	URI Design
Underscores (_) should not be used in URI	NoUnserscores	URI Design

clude placing verbs in URIs (e.g., `/createUser` instead of `/users`), misusing HTTP methods (e.g., using `GET` to delete a resource), or inconsistently applying status codes. As a result, REST API design is often viewed as a craft—balancing technical correctness, developer ergonomics, and consistency [3].

To reduce ambiguity and promote consistency in RESTful API design, numerous studies have attempted to formalize the abstract principles of REST into standardized design rules and best practices. These include both academic contributions, such as those by Petrillo et al. [20], Palma et al. [18], and Massè et al. [12]. These works provide actionable guidance on resource naming, proper use of HTTP methods, status codes, and error handling, serving as foundational references for REST API design.

Building on foundational guidelines, several works have focused on auto-matically detecting REST API design rule violations that reduce usability and maintainability. Tools like SARA [17] and SOFA [15] use rule-based and seman-tic analysis to uncover issues in URI structure, HTTP verb usage, and resource modeling. More recently, Bogner et al. introduced RESTRuler [2], a Java-based open-source tool that leverages static analysis to detect violations of 14 REST design rules, as shown in Table 1. Other empirical studies have analyzed open-source APIs to assess real-world adherence to REST principles, revealing frequent deviations despite claims of RESTfulness [3, 11].

Despite the importance of adhering to REST design principles, little attention has been paid to how these guidelines are understood and applied in educational settings. To date, no comprehensive studies have investigated the extent to which students are aware of these rules, nor how frequently they violate them when designing Web APIs. This lack of insight limits our ability to improve REST education and address misunderstandings at their root.

3   Study Design

The *goal* of this study is to evaluate the quality of RESTful back-end components in full-stack web applications developed by students enrolled in a third-year Web Technologies course within a Computer Science Bachelor's Degree program. As one of the final courses before graduation, it requires students to build complete web systems that integrate both front-end and back-end components.

This setting offers an opportunity to assess the technical competencies students acquire by the end of their academic journey. In particular, we focus on the design of REST APIs to identify recurring violations of established design principles. To this end, we pose the following research question:

> **RQ$_1$.** *What are the most frequent REST API design rule violations in student-developed web applications?*

The results of this study are intended to help educators refine course content by highlighting specific design aspects that require greater emphasis. Fur-thermore, the findings offer insight into the level of API design maturity that companies can expect from recent graduates entering the job market.

3.1   Web Technologies Course

The projects analyzed in this study were developed as part of Web Technologies, a 6 ECTS third-year course in the B.Sc. in Computer Science at the University of Naples Federico II, taught by the last author of the paper. The course builds on prior knowledge of object-oriented programming, software engineering and computer networks, and covers the basis of modern web development. Involved students are expected to be familiar with object-oriented programming and core software engineering concepts such as the software development lifecycle, soft-ware architecture and design, and code quality.

The course covers both client- and server-side web development, starting with foundational topics such as HTTP, modern HTML and CSS (including Flexbox, Grid, and responsive design), and advanced JavaScript (ES6+), focusing on DOM manipulation and asynchronous programming patterns. Server-side development progresses from traditional approaches like CGI and PHP to modern Node.js applications using Express, middleware, templating, and ORMs like Sequelize. REST APIs are a key focus, with dedicated lectures on authentication via JWT and design best practices [12].

The course further explores modern front-end development with tooling, SPA architecture, TypeScript, and detailed coverage of the Angular framework, emphasizing components, dependency injection, and HTTP interceptors. Security and web testing are strongly emphasized, addressing vulnerabilities such as XSS, CSRF, and SQL injection, alongside testing strategies including end-to-end tests with tools like Playwright [6, 8].

Assessment combines a written exam and a practical project requiring students to design and develop an SPA with a RESTful backend supporting authentication and CRUD operations. The students were instructed that the project work would be graded primarily based on its functional correctness, but also taking into account overall software quality and adherence to established REST guidelines. Students may select technologies from the course or alternative frameworks, provided they respect SPA architectural principles.

3.2   Collected Projects

The web applications analyzed in this study were developed by students enrolled in the Spring 2024 edition of the Web Technologies course. From the initial submission of 52 projects, we conducted a manual validation process in which the first and third authors verified the satisfaction of dependencies, compilation, and execution capabilities. This quality control phase resulted in the exclusion of 12 projects, yielding a final dataset of 40 functionally complete and manually verified applications.

The technological composition of these projects demonstrates consistency in backend implementation; specifically, 39 projects (97.5%) adopted JavaScript with the Express framework (consistent with the examples shown during the course), while one project (2.5%) used Java with the Spring framework. Front-end implementations exhibited greater diversity, with Angular, which was also discussed in detail during the course, dominating as the framework of choice (33 projects, 82.5%), followed by React (6 projects, 15%), and a single instance using Vue (2.5%). We have also observed notable differences between front-end and backend modules. The front-end modules demonstrate substantially larger codebases, averaging 2,099 lines of code (NCLOC) (Standard Deviation = 727) compared to 847 NCLOC (Standard Deviation = 603) for backend modules. A similar disparity can be observed for file organization, with front-end comprising an average of 66 files (Standard Deviation = 19), while the average for back-ends is 20 files (Standard Deviation = 11).
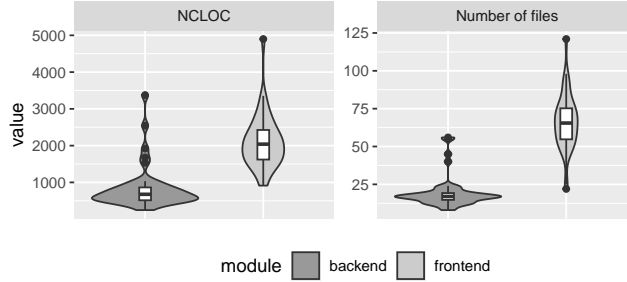
Fig. 1: Project Statistics.

The distribution of these metrics is reported in Figure 1. The Figure shows that while backend sizes cluster relatively tightly around the mean, front-end sizes exhibit greater variability, particularly in NCLOC. This pattern likely reflects both the inherent complexity of client-side development and the varying approaches students take in developing the front-end architecture.

3.3   Automated Pipeline

To evaluate the student submissions, we adopted an automated quality analysis pipeline inspired by the approach proposed by De Luca et al. [5]. The pipeline starts by extracting OpenAPI specifications from the back-end components of each project. For Express-based applications, we used `express-oas-generator` [1], while for Spring Boot projects, we employed `springdoc-openapi` [10]. These tools automatically inspect routing and controller logic to generate OpenAPI-compliant specifications. Once generated, the OpenAPI files were analyzed using RESTRULER [2], a tool that detects violations of widely accepted REST API design rules. We executed RESTRULER via its standalone JAR interface, producing a structured report for each project detailing the types and frequency of rule violations identified.

4   Results for RQ₁: What are the most frequent REST API design rule violations in the student-developed web applications?

The aggregated results on the detected REST design rule violations are reported in Table 2. These results show that notable patterns exist in how frequently certain best practices and design rules were overlooked across student projects.

The most widespread issue was the *missed use of hyphens to enhance readability* in endpoint paths, with the *Hyphens* rule violated in 98% of projects (39 out of 40). Similarly, pluralized nouns were not used for collection resource paths (*PluralNoun*) in 88% of projects (35 out of 40), while the *GETRetrieve*

Table 2: Overview of rule violations in REST APIs, including occurrence count, affected projects, and percentage.

Rule Identifier	Occurrences	#Projects	(%)
RC401	52	5	13
PluralNoun	179	35	88
SingularNoun	48	12	30
NoTrailingSlash	5	5	13
VerbController	1	1	3
NoCRUDNames	61	11	28
ContentType	322	14	35
DescriptionType	10	9	23
ForwardSlash	3	3	8
NoTunnel	28	4	10
GETRetrieve	201	33	83
Hyphens	170	39	98
Lowercase	35	5	13
NoUnderscores	5	3	8

rule, which ensured `GET` methods are used only for retrieval of resource representations, was violated in 83% of projects (33 out of 40). These high percentages suggest that foundational REST conventions related to URI structure and HTTP method semantics were either not fully internalized or deliberately disregarded by the majority of students.

Other rules exhibited moderate prevalence. The *ContentType* rule, which mandates explicit content-type headers, was violated in 35% of projects (14 out of 40), indicating that a significant subset of students overlooked this aspect of API communication. Similarly, *NoCRUDNames* (avoiding CRUD terms in URIs) and *SingularNoun* (using singular nouns for document names) were violated in 28% and 30% of projects, respectively. These findings suggest that while some students adhered to these conventions, a substantial minority did not, potentially reflecting inconsistent emphasis during instruction or varying interpretations of REST guidelines.

Less common but still noteworthy were violations of rules like *VerbController* (3% of projects) and *ForwardSlash* (8% of projects). Their low prevalence implies that these were either more intuitive for students or less prone to ambiguity in teaching materials. However, even minor violations, such as *NoTrailingSlash* and *Lowercase* (each appearing in 13% of projects), highlight areas where targeted instruction could further improve adherence.

Collectively, these results underscore that while students generally grasped high-level REST concepts, specific design principles, particularly those related to URI formatting and HTTP method usage, were frequently misapplied. The prevalence of violations in these areas suggests opportunities to refine pedagogical focus, perhaps through more explicit examples or automated linting tools during development.

5   Threats to validity

5.1   Construct Validity

RESTRuler's rule violations are based on a predefined set of REST design principles that may not encompass all real-world API design considerations. Additionally, since some projects required manual instrumentation to generate OpenAPI specifications for RESTRuler, inconsistencies in this process could introduce bias in the REST violation analysis.

5.2   Internal Validity

Manual steps involved in preparing projects for analysis—such as configuring or correcting OpenAPI generation—may have introduced variability in how RE-STRuler was applied. Incomplete or improperly generated OpenAPI specifications might have caused some violations to be missed, affecting the completeness and comparability of results across projects.

5.3   External Validity

Our findings are based on a single cohort of students from a single university who followed a Web Technologies course. As such, the generalizability of the results to other educational contexts or professional settings may be limited. Additionally, the projects were developed under academic constraints (e.g., deadlines, grading criteria), which may not reflect real-world development practices.

6   Conclusion and Future work

Our comprehensive analysis of 40 student-developed full-stack web applications revealed several patterns in REST API design violations that have important implications for computer science education. The findings demonstrate that while students generally master the functional aspects of web development, they struggle with quality-related concerns. These findings suggest that architectural design rules deserve more explicit attention in undergraduate computer science curricula. Specifically, educators should consider integrating automated quality checks into course projects, providing targeted feedback on REST API design, and emphasizing the industrial relevance of these quality aspects.

For future works, we plan to extend our analysis to include software quality aspects such as code smells and maintainability issues by integrating tools like SonarQube into the evaluation process. Furthermore, we intend to investigate the effectiveness of various pedagogical interventions in improving students' software quality. This will involve comparing project quality before and after introducing dedicated lectures on software quality, incorporating continuous static analysis during development, and providing REST API design templates or checklists.

References

1. express-oas-generator — npmjs.com. `https://www.npmjs.com/package/express-oas-generator`. [Accessed 04-05-2025].

2. Justus Bogner, Sebastian Kotstein, Daniel Abajirov, Timothy Ernst, and Manuel Merkel. Restruler: Towards automatically identifying violations of restful design rules in web apis. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, pages 123–134, 2024.

3. Justus Bogner, Sebastian Kotstein, and Timo Pfaff. Do restful api design rules have an impact on the understandability of web apis? *Empirical software engineering*, 28(6):132, 2023.

4. Alison Clear, Allen S Parrish, John Impagliazzo, and Ming Zhang. Computing curricula 2020: introduction and community engagement. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 653–654, 2019.

5. Marco De Luca, Sergio Di Meglio, Anna Rita Fasolino, Luigi Libero Lucio Starace, and Porfirio Tramontana. Automatic assessment of architectural anti-patterns and code smells in student software projects. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 565–569, 2024.

6. Sergio Di Meglio, Luigi Libero Lucio Starace, and Sergio Di Martino. E2e-loader: A tool to generate performance tests from end-to-end gui-level tests. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 747–751, 2025.

7. Sergio Di Meglio, Luigi Libero Lucio Starace, and Sergio Di Martino. Starting a new rest api project? a performance benchmark of frameworks and execution environments. In *IWSM-Mensura*, 2023.

8. Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Ruben Opdebeeck, Coen De Roover, and Sergio Di Martino. E2egit: A dataset of end-to-end web tests in open source projects. In *Proceedings of the 22nd International Conference on Mining Software Repositories*, pages 836–840, 2025.

9. Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

10. Library for OpenAPI 3 with spring-boot By Badr NASS LAHSEN. OpenAPI 3 Library for spring-boot — springdoc.org. `https://springdoc.org/`. [Accessed 04-05-2025].

11. Sebastian Kotstein and Justus Bogner. Which restful api design rules are important and how do they improve software quality? a delphi study with industry experts. In *Service-Oriented Computing: 15th Symposium and Summer School, SummerSOC 2021, Virtual Event, September 13–17, 2021, Proceedings 15*, pages 154–173. Springer, 2021.

12. Mark Masse. *REST API design rulebook: designing consistent RESTful web service interfaces*. " O'Reilly Media, Inc.", 2011.

13. Sergio Di Meglio and Luigi Libero Lucio Starace. Evaluating performance and resource consumption of rest frameworks and execution environments: Insights and guidelines for developers and companies. *IEEE Access*, 12:161649–161669, 2024.

14. Sergio Di Meglio, Valeria Pontillo, and Luigi Libero Lucio Starace. Rest in pieces: Code quality issues and restful design rule violations in student-built web apps — online appendix. `https://zenodo.org/records/15492462`, May 2025.

15. Naouel Moha, Francis Palma, Mathieu Nayrolles, Benjamin Joyen Conseil, Yann-Gaël Guéhéneuc, Benoit Baudry, and Jean-Marc Jézéquel. Specification and detection of soa antipatterns. In *Service-Oriented Computing: 10th International Conference, ICSOC 2012, Shanghai, China, November 12-15, 2012. Proceedings 10*, pages 1–16. Springer, 2012.

16. Arthur-Jozsef Molnar, Simona Motogna, Diana Cristea, and Diana-Florina Sotropa. Exploring complexity issues in junior developer code using static analysis and fca. In *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 407–414, 2024.

17. Francis Palma, Javier Gonzalez-Huerta, Mohamed Founi, Naouel Moha, Guy Tremblay, and Yann-Gaël Guéhéneuc. Semantic analysis of restful apis for the detection of linguistic patterns and antipatterns. *International Journal of Cooperative Information Systems*, 26(02):1742001, 2017.

18. Francis Palma, Tobias Olsson, Anna Wingkvist, and Javier Gonzalez-Huerta. Assessing the linguistic quality of rest apis for iot applications. *Journal of Systems and Software*, 191:111369, 2022.

19. Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs." big"'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814, 2008.

20. Fabio Petrillo, Philippe Merle, Naouel Moha, and Yann-Gaël Guéhéneuc. Are rest apis for cloud computing well-designed? an exploratory study. In *Service-Oriented Computing: 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings 14*, pages 157–170. Springer, 2016.

21. Rafał Włodarski, Jean-Remy Falleri, and Corinne Parvéry. Assessment of a hybrid software development process for student projects: a controlled experiment. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 289–299. IEEE, 2021.