Everything You Always Wanted to Know About JSON Schema (But Were Afraid to Ask)

Mohamed-Amine Baazizi
Sorbonne Université, LIP6 UMR 7606
France
baazizi@ia.lip6.fr

Dario Colazzo
Université Paris-Dauphine, PSL - PRAIRIE PSAI
France
dario.colazzo@dauphine.fr

Giorgio Ghelli
Dipartimento di Informatica
Università di Pisa
Pisa, Italy
ghelli@di.unipi.it

Carlo Sartiani
DiING
Università della Basilicata
Potenza, Italy
carlo.sartiani@unibas.it

Stefanie Scherzinger
University of Passau
Passau, Germany
stefanie.scherzinger@uni-passau.de

ABSTRACT

The last few years have seen the ubiquitous diffusion of JSON as one of the most widely used formats for publishing and interchanging data, as it combines the flexibility of semistructured data models with well-known data structures like records and arrays. While various schema languages for describing JSON data have been proposed in the past, e.g., JSound and Joi, JSON Schema established itself as de-facto standard schema language for JSON data.

The main aim of this tutorial is to provide the audience with the basic notions for exploiting JSON Schema while processing and manipulating JSON data. This tutorial focuses on four main aspects: (1) we first describe Classical JSON Schema and introduce the features that are shared with the latest versions of the specification; (2) we introduce, then, Modern JSON Schema, explain why it differs from Classical JSON Schema, and discuss its novel evaluation model; (3) we analyze tools that support or exploit JSON Schema, like, for example, validators and data generators; and (4) we highlight open research challenges and opportunities related to JSON Schema.

1 INTRODUCTION

JSON Schema is a schema language for imposing constraints on the structure and the admissible values of a family of JSON documents. In particular, JSON Schema allows users to impose constraints on objects, arrays, and primitive values, such as numbers and strings; these constraints can be richly combined through traditional logical combiners (e.g., and, or, not, and exclusive or), and recursive data structures can be specified through recursive references.

Many versions have been defined for this language, notably Draft-03 of November 2010, Draft-04 of February 2013, Draft-06 of April 2017 [23], Draft 2019-09 of September 2019 [20], and Draft 2020-12 of December 2020 [21]. Draft 2019-09 introduces two important novelties to the evaluation model: annotation-dependent validation and dynamic recursive references, which have been generalized as general-purpose dynamic references in Draft 2020-12. According to the terminology introduced by Henry Andrews in [5], due to these modifications to the evaluation model, Draft 2019-09 is the first draft that defines *Modern JSON Schema*, while the previous drafts define variations of *Classical JSON Schema*.

JSON Schema currently represents the *de facto* standard schema language to describe JSON data, and its adoption is almost ubiquitous: for instance, OpenAPI 2.0, 3.0, and 3.1 [1] adopt JSON Schema to describe both input and allowed operations on web services. Although widely adopted, JSON Schema remains a complex language, with a nontrivial semantics and complicated interplays among different operators. Furthermore, novel features of Modern JSON Schema have been interpreted in several different ways by different developers, leading to the development of a plethora of tools showing slightly different behavior on the same inputs.

In this tutorial proposal, we will present and discuss the most important and controversial features of Classical and Modern JSON Schema; we will also analyze several classes of JSON Schema tools, ranging from validators to data generators and static analysis tools. The main aim of this tutorial is to provide the audience and developers with the basic and fundamental concepts to proficiently use JSON Schema and its vast ecosystem of related tools.

Outline. This tutorial is split into five main parts:

(1) **JSON primer**. In this very introductory part of the tutorial, we review the basic notions about JSON and we will also introduce common notions that we will use throughout the tutorial.
(2) **Classical JSON Schema**. In this part of the tutorial we introduce the basic concepts about JSON Schema and describe in detail Classical JSON Schema. In particular, we focus on boolean operators, keywords, and assertions, as well as static references. We also discuss the implicative semantics of JSON Schema and present the most important complexity results on validation, satisfiability, and inclusion checking.
(3) **Modern JSON Schema**. In this part of the tutorial we shift our attention towards Modern JSON Schema and, in particular, discuss Draft 2020-12. We begin by introducing annotations and their impact on validation. We then focus on `"unevaluatedItems"`, `"unevaluatedProperties"`, and dynamic references, the most complex, innovative, but also controversial features of Modern JSON Schema. We finish this part by discussing the existing complexity results for Modern JSON Schema.
(4) **Tools and Applications**. In this part of the tutorial, we analyze tools that support the use of JSON Schema, both

Classical and Modern, with particular emphasis on validators, data generators, and static analysis tools.

(5) **Future Opportunities**. Finally, we outline open research problems as potential directions for new research in this area.

In what follows, we describe at a very high level the technical content covered in each of the last four aforementioned parts.

2 CLASSICAL JSON SCHEMA

In this part of the tutorial we will focus our attention on Classical JSON Schema, with particular emphasis on the latest *classical* draft (Draft 7 [22]), but we will also describe the common foundations shared by both Classical and Modern JSON Schema.

In JSON Schema, a schema is a logical combination of assertions that must be satisfied by a JSON value. JSON Schema supports a rich set of boolean operators, comprising explicit conjunction (i.e., `"allOf"`), implicit conjunction (i.e., {}), disjunction (i.e., `"anyOf"`), exclusive disjunction (i.e., `"oneOf"`), and negation (i.e., `"not"`).

Object values are described through multiple assertions, comprising `"properties"`, `"patternProperties"`, `"required"`, and further `"additionalProperties"`. Similarly, arrays are described by `"items"`, `"additionalItems"`, `"contains"`, and `"uniqueItems"`.

`"minProperties"`/`"maxProperties"`, `"minItems"`/`"maxItems"` can be further used to define lower and upper bounds on the number of properties and items, respectively.

A distinctive feature of most JSON Schema assertions is their *implicative* semantics. Consider, for example, the following schema:

$$\{\texttt{"properties"} : \{\texttt{"name"} : \{\texttt{"type"} : \texttt{"string"}\}\}\}$$

This schema states that, if value J is an object and has a property `"name"`, then the value of this property must be a string; if J is not an object or if it is an object lacking the property `"name"`, then the schema is trivially satisfied.

Classical JSON Schema supports recursive definitions through *static* references. These references can be used to refer to definitions inside the same schema or inside an external schema. References rely on JSON Pointer [13], a rather simple language to traverse a JSON document and locate a single element. Identifiers can be associated to schemas to simplify their use in external schemas.

This part of the tutorial will end with a discussion about the complexity of validation and satisfiability. Validation for Classical JSON Schema has been proved to be PTIME-complete by Pezoa et al. in [19] and by Bourhis et al. in [12], while satisfiability is EXPTIME-hard and in 2-EXPTIME [12]. Bourhis et al. [12] also proved that satisfiability is EXPTIME-complete when `"uniqueItems"` is omitted. Since satisfiability and inclusion for Classical JSON Schema are equivalent, these results also apply to inclusion and equivalence. These results are summarized in Table 1.

3 MODERN JSON SCHEMA

Modern JSON Schema represents not only an evolution of Classical JSON Schema but also, to some extent, a departure from some of the foundations of the language. Given the significance of the changes and the unconventional nature of some of the features that have been introduced, in this part of the tutorial we will discuss in detail the new draft, already adopted in OpenAPI 3.1 [16], and analyze the differences with Classical JSON Schema.

The first and most important change wrt Classical JSON Schema is represented by the introduction of *annotations*. They are generated during validation and, in a nutshell, list the properties and items that have been "evaluated" during the validation process. As a consequence, the result of validating a JSON value J against a schema S is no longer a boolean value, but instead a pair $< \tau, k >$, where τ is a boolean value, and k is a list of annotations.

The introduction of annotations has a very important consequence that we will discuss in detail in this part of the tutorial: common De Morgan equivalences are no longer valid, and, in particular, S is no longer equivalent to $\{\texttt{"not"} : \{\texttt{"not"} : S\}\}$. Annotations drive the behavior of two novel assertions that have been introduced in Modern JSON Schema to overcome some of the limitations of `"additionalProperties"` and `"additionalItems"`: `"unevaluatedProperties"` and `"unevaluatedItems"`.

Dynamic references have been added in Modern JSON Schema as an extension mechanism, allowing one to first define a base form of a data structure and then to refine it, much in the spirit of "self" refinement in OO languages.

For example, in the following schema, the reference

$$\texttt{"\$dynamicRef"} : \texttt{"http://mjs.ex/simple-tree\#tree"}$$

is *dynamic*, meaning that it may change its meaning when this schema is "accessed" through a different context.

```
{ "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/simple-tree",
  "$dynamicAnchor": "tree",
  "type": "object",
  "properties": {
    "data": true,
    "children": {
       "type": "array",
       "items": {
            "$dynamicRef": "https://example.com/simple-tree#tree" }
       }
    }
}
```

Here we have an example of such a different context: in the schema below, the tree of the previous schema is refined, and the meaning of the dynamic reference `"$dynamicRef" : ". . . #tree"` that is found inside `"http://mjs.ex/simple-tree"` is dynamically modified as a reference to `"http://mjs.ex/number-tree#tree"`. In the tutorial we will explain how this happens.

```
{   "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/number-tree",
    "$dynamicAnchor": "tree",
     "properties": {
         "data": { "type" : "number" }
      },
     "$ref": "https://example.com/simple-tree#tree"
}
```

Similarly to the previous part, this part of the tutorial will also end with a discussion about complexity results.

4 TOOLS AND APPLICATIONS

In this part of the tutorial we will present several tools for JSON Schema. We will focus on validators, data generators, and static analysis tools.

4.1 Validators

There exist several validators for Classical JSON Schema and slightly fewer for Modern JSON Schema. As of this day, the JSON Schema web site [17] lists at least 66 different validators for Classical JSON Schema and at least 39 for Modern JSON Schema, offering APIs for languages ranging from Java to JavaScript, C++, and many others, as summarized in Table 2. Some of these

Schema features		Validation	Satisfiability
No recursion	**No** `"uniqueItems"`	**PTIME-complete**	**PSPACE-complete**
	`"uniqueItems"`	**PTIME-complete**	PSPACE-hard In EXPSPACE
Recursion	**No** `"uniqueItems"`	**PTIME-complete**	**EXPTIME-complete**
	`"uniqueItems"`	**PTIME-complete**	EXPTIME-hard In 2EXPTIME

Table 1: Complexity results for Classical JSON Schema.

Language	Classical JSON Schema	Modern JSON Schema
.NET	4	3
C++	5	2
Go	3	2
Java	9	7
JavaScript	8	6
Kotlin	4	1
Perl	4	2
PHP	3	1
Python	4	3
Ruby	2	1
Rust	2	2
Scala	0	1
Swift	1	1

Table 2: Validators per programming language, as listed on the JSON Schema website (as of February 2025).

```
{   "description": "The test case description",
    "schema": { "type": "string" },
    "tests": [
        {   "description": "a test with a valid instance",
            "data": "a string",
            "valid": true  },
      {   "description": "a test with an invalid instance",
            "data": 15,
            "valid": false }
    ]
}
```

Figure 1: A sample test case from the JSON Schema Test Suite.

validators also offer a web-based GUI and helpful companion tools.

Given the wide number of validators available, it is of paramount importance to assess their quality and their conformance to the specification. To this end, the JSON Schema community designed the JSON Schema Test Suite [18] and developed the tool Bowtie [9].

The JSON Schema Test Suite comprises a set of *test cases*, each addressing a specific feature or operator of the specification. A test case is a JSON file containing a schema and an array of tests, where each test is formed by a JSON value to be validated against the schema and by a boolean value describing the expected validation result, as shown in Figure 1. The JSON Schema Test Suite supports all versions of the specification.

The JSON Schema Test Suite has been designed with the aim of helping developers implement and debug validators. However, a user willing to find the proper validator for their specific application context should manually run the tests on all the validators of interest, which can be cumbersome and time consuming. The Bowtie tool, developed by Julian Berman [9], tries to overcome these issues by providing an integrated environment to run tests on multiple validators; developers willing to integrate their own tool can use a specific API and must provide an OCI container image. Bowtie coordinates the execution of tests and collects and summarizes their results. The Bowtie website [10] reports on the results of almost daily runs of the JSON Schema Test Suite.

4.2 Data Generators

There are several data generators for JSON Schema. They usually take as input a schema, possibly annotated with further information, and return a JSON document that might satisfy the input schema. A few tools, like [11] and [4] for example, are based on a trial-and-error approach. In particular, the JSON Schema Faker [4] takes as input a schema S and generates a JSON instance J by considering the information in S; if J is valid wrt S, then generation terminates, otherwise the tool retries. This repeats until a valid data instance is generated or the maximum number of iterations has been reached; in the latter case, the last generated JSON value is returned, although it may be invalid.

Both [11] and [4] generate JSON instances where all the leaves are random values, and, hence, they cannot generate instances with realistic values, which may represent a significant obstacle in some application context. Furthermore, these tools cannot enforce schema coverage or branch coverage policies. Another class of tools, that compile a schema into a generation plan, may overcome these limitations. Hypothesis-jsonschema [3] is an extension of the popular property-based test generator Hypothesis [2]. This library takes as input a schema and returns an hypothesis strategy, e.g., a data generation plan for Hypothesis. This strategy can be further modified by the user to customize its behavior. This library has been further extended in schemathesis [15], a tool for generating fuzzers for OpenAPI.

4.3 Static Analysis Tools

In [14], Habib et al. describe a tool for Classical JSON Schema containment checking. The tool, written in Python and used within the Lale ML library, takes as input two schemas S_1 and S_2, and checks whether S_1 is contained in S_2. The tool exploits a classical rule-based approach; however, the set of rules being used is incomplete and, therefore, the tool may not be able to verify the containment.

In [6], Attouche et al. describe a witness generator for Classical JSON Schema. The tool takes as input a schema S, and returns a value J satisfying S if and only if S is not empty; an error message is returned in case S is unsatisfiable. Since in Classical JSON Schema $S_1 \subseteq S_2$ if and only if $S_1 \wedge \neg S_2 \neq \emptyset$, this tool can also be used for containment checking.

To the best of our knowledge there are no static analysis tools for Modern JSON Schema yet.

5 FUTURE OPPORTUNITIES

We finally discuss several open challenges and research opportunities related to JSON Schema, including the following.

Formalization and Static Analysis for Modern JSON Schema. Classical JSON Schema has been studied in detail, and its semantics has been formalized in several papers [8, 12, 19]. However, little has been done with regard to Modern JSON Schema, partly because of its novelty. The only known formalization for Modern JSON Schema has been described by Attouche et al. in [7]. Hence, a promising research direction is to understand how annotations and dynamic references impact the complexity of important decision problems like inclusion and equivalence.

Data Generation. While there exist several data generators for JSON Schema, they may produce low-quality output or require heavy human intervention to improve the quality of the data instances generated. A major research opportunity is to design data generation tools that are capable of generating realistic and high-quality data with little or no human intervention, possibly by exploiting LLMs to incorporate some world and linguistic knowledge.

6 DIFFERENCES WITH OTHER VERSIONS OF THE TUTORIAL

At EDBT 2019 and SIGMOD 2019, we presented two tutorials on schema languages and tools for JSON data. In those versions JSON Schema was a very minor topic and much more emphasis was placed on other languages, including Joi and TypeScript.

7 BIOGRAPHICAL SKETCHES

Mohamed-Amine Baazizi (Ph.D.) is Assistant Professor at Sorbonne Université. He received his PhD from Université of Paris-Sud and completed his postdoctoral studies in Télécom Paristech. His research focuses on exploiting schema information for optimizing the processing of semi-structured data.

Dario Colazzo (Ph.D.) is Full Professor in Computer Science at LAMSADE - Université Paris-Dauphine, as well as PRAIRIE Fellow. He received his PhD from Università di Pisa, and he completed his postdoctoral studies at Università di Venezia and Université Paris Sud. His main research activities focus on static analysis techniques for large scale data management.

Giorgio Ghelli (Ph.D.) is Full Professor in Computer Science, at Università di Pisa. He was Visiting Professor at École Normale Supérieure Paris, at Microsoft Research Center, Cambridge (UK), at Microsoft Co. (Redmond, USA), and at LAMSADE - Université Paris-Dauphine. He worked on database programming languages and type systems for these languages, especially in the fields of object oriented and semistructured data models.

Carlo Sartiani (Ph.D.) is Associate Professor in Computer Science at Università della Basilicata. He received his PhD from Università di Pisa, and he completed his postdoctoral studies at Università di Pisa. He worked on database programming languages and data integration systems, and his current research activities focus on semistructured and big data.

Stefanie Scherzinger (Ph.D.) is Full Professor in Computer Science at the University of Passau, where she chairs the Scalable Database Systems group. Drawing on her earlier experience as a software developer at IBM and Google, her research focuses on schema management tasks, including schema analysis, integration, discovery, and evolution.

ACKNOWLEDGMENTS

This work has been partially supported by the EU H2020 programme under the funding schemes ERC-2018-ADG G.A. 834756 "XAI: Science and technology for the eXplanation of AI decision making" and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – grant #385808805. We also acknowledge the support of the PRIN Project "BioConceptum" (2022AEEKXS), under the NRRP MUR program funded by the NextGenerationEU.

REFERENCES

[1] 2024. OpenAPI Initiative. Available online at https://www.openapis.org.
[2] 2025. Hypothesis. Available at https://hypothesis.readthedocs.io/en/latest/. Retrieved 13 February 2025.
[3] 2025. hypothesis-jsonschema. Available on GitHub at https://github.com/python-jsonschema/hypothesis-jsonschema. Retrieved 13 Febriary 2025.
[4] 2025. JSON Schema Faker. Available on GitHub at https://github.com/json-schema-faker/json-schema-faker and as an interactive tool at https://json-schema-faker.js.org. Retrieved 13 February 2025.
[5] Henry Andrews. 2023. Modern JSON Schema. Available online at https://modern-json-schema.com/.
[6] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2022. Witness Generation for JSON Schema. *Proc. VLDB Endow.* 15, 13 (2022), 4002–4014. https://www.vldb.org/pvldb/vol15/p4002-sartiani.pdf
[7] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2024. Validation of Modern JSON Schema: Formalization and Complexity. *Proc. ACM Program. Lang.* 8, POPL (2024), 1451–1481. https://doi.org/10.1145/3632891
[8] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2023. Negation-closure for JSON Schema. *Theor. Comput. Sci.* 955 (2023), 113823. https://doi.org/10.1016/J.TCS.2023.113823
[9] Julian Berman. 2024. Bowtie JSON Schema Meta Validator. https://github.com/bowtie-json-schema/bowtie Online tool. Version 2024.11.7.
[10] Julian Berman. 2025. Bowtie Report. Available at https://bowtie.report.
[11] Jim Blackler. 2022. JSON Generator. Available at https://github.com/jimblackler/jsongenerator. Retrieved 13 February 2025.
[12] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, Query languages and Schema specification. In *Proc. PODS*. 123–135. https://doi.org/10.1145/3034786.3056120
[13] P. Bryan, K. Zyp, and M. Nottingham. Aprile 2013. *JavaScript Object Notation (JSON) Pointer.* Technical Report. Internet Engineering Task Force. https://www.rfc-editor.org/info/rfc6901
[14] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2021. Finding Data Compatibility Bugs with JSON Subschema Checking. In *Proc. ISSTA*. 620–632. https://doi.org/10.1145/3460319.3464796
[15] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving Semantics-Aware Fuzzers from Web API Schemas. In *International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022*. 345–346. https://doi.org/10.1145/3510454.3528637
[16] Darrel Miller, Jeremy Whitlock, Marsh Gardiner, Mike Ralphson, Ron Ratovsky, and Uri Sarid. 2021. OpenAPI Specification v3.1.0. Available at https://spec.openapis.org/oas/v3.1.0.
[17] JSON Schema Org. 2025. JSON Schema. Available at https://json-schema.org.
[18] JSON Schema Org. 2025. JSON Schema Test Suite. https://github.com/json-schema-org/JSON-Schema-Test-Suite. Retrieved 13 February 2025.
[19] Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, and Domagoj Vrgoc. 2016. Foundations of JSON Schema. In *Proc. WWW*. 263–273.
[20] A. Wright, H. Andrews, and B. Hutton. 2019. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-handrews-json-schema-validation-02.* Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-handrews-json-schema-validation-02
[21] A. Wright, H. Andrews, and B. Hutton. 2020. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-bhutton-json-schema-validation-00.* Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-bhutton-json-schema-validation-00
[22] A. Wright, H. Andrews, and G. Luff. 2018. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-handrews-json-schema-validation-01.* Technical Report. Internet Engineering Task Force. https://json-schema.org/draft-07/draft-handrews-json-schema-validation-01
[23] A. Wright, G. Luff, and H. Andrews. 2017. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-wright-json-schema-validation-01.* Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-wright-json-schema-validation-01