Aalto University
School of Science
Bachelor's Programme in Science and Technology

Survey of web API definition languages

Bachelor's Thesis

25. maaliskuuta 2025

Niklas Saarikoski

Aalto University
School of Science
Bachelor's Programme in Science and Technology

ABSTRACT OF
BACHELOR'S THESIS

Author:	Niklas Saarikoski
Title of thesis:	Survey of web API definition languages
Date:	25th of March, 2025
Pages:	30
Major:	Computer Science and Engineering
Code:	SCI3027
Supervisor:	Professor Lauri Savioja
Instructor:	Doctoral Researcher Juho Vepsäläinen (Department of Computer Science Engineering)

Web Application Programming Interfaces (APIs) are the backbone of modern software development, enabling communication between diverse applications and platforms.To support the different needs, a variety of web API technologies have emerged, each with its own architecture, data transmission protocols, and design philosophies.

This bachelor's thesis introduces existing web APIs, examines applications developed for them, and evaluates their advantages and disadvantages. The comparison considers the popularity, efficiency, and architectural security of these APIs. The research has been conducted as a literature review, exploring existing materials to identify previous studies and potential gaps in the research.

The comparison focuses on four web API technologies: RPC (Remote Procedure Call), SOAP (Simple Object Access Protocol), REST (Representational State Transfer), and GraphQL (Graph Query Language). These APIs represent the vast majority of web APIs in use today, and provide a good look in to the different limitations brought on with certain design choices

Currently, the development of entirely new web API technologies seems unlikely. Although existing solutions have clear weaknesses, they are often offset by the strengths of other technologies. Nevertheless, in the future, new use cases might emerge that demand processes beyond the capabilities of current web API technologies.

Keywords:	API, REST, GraphQL, RPC, SOAP, web
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Teknistieteellinen kandidaattiohjelma

KANDIDAATINTYÖN
TIIVISTELMÄ

Tekijä:	Niklas Saarikoski
Työn nimi:	Survey of web API definition languages
Päiväys:	25. maaliskuuta 2025
Sivumäärä:	30
Pääaine:	Computer Science and Engineering
Koodi:	SCI3027
Vastuuopettaja:	Professori Lauri Savioja
Työn ohjaaja(t):	Tohtorikoulutettava Juho Vepsäläinen (Tietotekniikan laitos)

Verkon ohjelmointirajapinnat (API) ovat nykyaikaisen ohjelmistokehityksen selkäranka, jotka mahdollistavat viestinnän erilaisten sovellusten ja alustojen välillä. Tämä integraatiokyky on tehnyt verkko-API:ista välttämättömiä skaalautuvien, reagoivien ja vuorovaikutteisten järjestelmien rakentamisessa, erityisesti verkko- ja mobiilisovellusten alueilla. Näiden tarpeiden tukemiseksi on syntynyt useita erilaisia verkko-API-teknologioita, joilla on omat arkkitehtuurinsa, tiedonsiirtoprotokollansa ja suunnittelufilosofiansa.

Tässä kandidaatintutkielmassa esitellään olemassaolevia verkko-API:a, tutkin niitä varten tehtyjä sovelluksia ja näiden tuomia etuja sekä haittoja. Vertailussa otetaan huomioon API:en suosiota, tehokkuutta ja arkkitehtuurista tietoturvallisuutta. Tutkimus on tehty kirjallisuustutkielmana, jossa olemassaolevasta materiaalista etsitään tehtyjä tutkimuksia ja mahdollisia aukkoja tutkimuksissa.

Vertailussa on ollut neljä API-teknologiaa: RPC (Remote Procedure Call), SOAP (Simple Object Access Protocol), REST (REpresentational State Transfer) ja GraphQL (Graph Query Language). Niiden eroavaisuudet johtuvat pitkälti kehitysajankohdista ja sen ajan teknologioista.

Uuden verkko-API:n kehitys ei tällä hetkellä vaikuta todennäköiseltä. Vaikka nykyisissä ratkaisuissa on selkeitä heikkouksia, niin toinen olemassa oleva API pystyy paikkaamaan sitä. Tulevaisuudessa uusien käyttötarpeiden nousu voisi kuitenkin vaatia API:lta erilaisia prosesseja, joihin nykyiset verkko-API teknologiat eivät olisi riittäviä.

Avainsanat:	API, REST, GraphQL, RPC, SOAP, web
Kieli:	Englanti

Contents

1 Introduction

Web APIs have become integral in modern web development, playing a pivotal role in enabling communication between diverse applications and platforms. They facilitate the integration of heterogeneous systems, allowing data and functionality to be shared seamlessly across different environments [1]. This capability is especially crucial in today's interconnected digital landscape, where the demand for scalability, responsiveness, and interactivity is paramount. Web APIs are foundational in the creation of robust web and mobile applications, cloud-based services, and Internet of Things (IoT) systems. Their widespread adoption has revolutionized how developers build and deploy software, making them indispensable in both enterprise-level solutions and consumer-focused technologies.

1.1 Existing research

Maleshkova et al. [2] researches the evolution of API usability, versioning practices, and lifecycle management, particularly as APIs evolve to meet the demands of dynamic, distributed systems .

Security has also emerged as a critical area of investigation. As an example Bora and Bezboruah [3] studies the use of SOAP API in the medical field, while Tihomirovs and Grabis [4] compares the benefits of using the more secure SOAP to REST. Additionally, the rise of GraphQL has prompted research, such as the one done by Brito and Valente [5], into its query efficiency, performance optimization, and security implications compared to traditional REST APIs.

To address the growing complexity of web APIs, a variety of definition languages and frameworks have been developed. These technologies are tailored to meet the specific requirements of API consumers and providers, with distinctions in their underlying architectures, data transmission protocols, and design philosophies. Understanding these differences is critical for developers and organizations to select the most suitable tools for their projects, optimizing performance, security, and usability.

1.2 Research approach

In this bachelor's thesis, I will explore the landscape of web API definition languages, introducing the major existing standards, and investigating the most widely used services built on these APIs. The research questions in this thesis are: What web API definition languages exist? what kind of tools are available for them? And how do they compare to each other based on their performance, security features, developer experience, and ease of implementation? These aspects are essential for assessing the practical benefits and

trade-offs of adopting specific API definition languages.

The research has been conducted as a literature review, using existing studies and industry reports to identify key areas of comparison between APIs. By analyzing recent advancements in API definition languages and related technologies, this study will also consider potential future developments in the field. These predictions may offer insights into how web APIs will continue to evolve to meet emerging challenges, such as greater demands for scalability, enhanced security, and improved interoperability. This thesis aims to contribute to the broader understanding of web API technologies and their role in shaping the future of software development.

1.3 Structure of the thesis

In Section 2, I introduce what web APIs are, which APIs I'll be focusing on further, and how I'll be evaluating them. In Section 3, Section 4, Section 5, and Section 6, I focus on a single web API, introduce some of the larger services available for them, and then evaluate them in the aspects introduced in earlier chapters. In Section 7, I use the evaluations done on the APIs to more directly compare them with each other, and speculate on the suture developments of web APIs based on their history. In Section 8, I go over the main points of my research, summarize my main findings, and discuss further points of research.

2 Web API definition languages

Early websites existed largely as static content, with no dynamic or interactive content. Early interactivity was allowed by the introduction of Common Gateway Interfaces (CGI). CGI allowed users to interact through form submissions, but each request created a server process to execute the program [6]. This was resource intensive and didn't allow scaling for larger systems.

Web API definition languages are formalized frameworks used to describe the structure, operations, and data exchange mechanisms of web APIs. These languages enable developers to define APIs in a standardized and machine-readable format, fostering interoperability, consistency, and automation in the development and integration of web services [7].

2.1 Main features of web APIs

By providing clear specifications for endpoints, data types, authentication mechanisms, and error handling, API definition languages facilitate seamless communication between distributed systems. They also support various stages of the API lifecycle, including design, documentation, testing, and implementation. Prominent examples include OpenAPI [8], which is widely used for REST APIs, and Protocol Buffers (Protobuf) [9], often employed in gRPC for its compact serialization and cross-platform compatibility.

2.2 Available web API standards

Web APIs are typically used in web development to enable integration between web services, mobile apps, and third-party applications. They are commonly built using HTTP's standard communication operations: `GET`, `POST`, `PUT`, and `DELETE`. Web APIs can be implemented with different technologies, each with its own strengths and weaknesses.

There are multiple levels of abstraction that provide different perspectives on web APIs. At the definition language level, the focus is on defining the structure and possible languages used for the API. The framework level builds on specific definition languages and provides a collection of technologies to implement APIs [10]. Finally, at the platform level, services are offered to aid in building and managing web APIs. Platforms often support multiple API definition languages and integrate APIs as part of larger web service ecosystems.

These levels of abstraction help categorize and analyze web APIs by their purpose and functionality, making it easier to assess their strengths and limitations [7]. Early challenges in API design, such as tight coupling, hidden network communication, and

issues with parallelism and interoperability, have been addressed in modern APIs with innovations like managed APIs, which often require authentication and offer enhanced security and scalability.

The evolution of web APIs, including changes in API specification languages and frameworks, reflects their growing role in software development. Di Lauro et al. [10] investigate API services over time to understand trends, such as how APIs grow or shrink, their stability, and how frequently they update specification versions. These insights by Di Lauro et al. [10] reveal the dynamics of API development and maintenance, providing a basis for predicting future advancements in the field.

2.3 Evaluation of web APIs

To choose which web APIs to evaluate, there are a couple of different things to consider. The popularity of the definition language currently and it's part in the historical development of web APIs are the two major factors I am taking into consideration. RPC, SOAP and GraphQL are clear choices as web APIs to evaluate as they have clear frameworks, are often used for web APIs and represent different landscapes during development. RESTfull apis are included because of their overwhelming popularity in web APIs, even though it is a set of architectural restrictions for building a web API, rather than a strict framework [11].

To properly compare these different APIs, I have chosen four different apects: performance, ease of implementation, developer experience and security. An APIs performance can be evaluated in a few different aspects, such as speed, accuracy of calls and package size. Ease of implementation and developer experience both inspect the APIs from a developers perspective. Ease of implementation is concerned on creating an api service to which calls are made to, while developer experience is interested in creating API calls to an existing service. Security is largely concerned with the inbuilt security protocols for web APIs. A secondary concern is the architectural clarity that allows for building APIs with knowledge of available endpoints.

2.4 Summary

Web API definition languages provide standardized, machine-readable formats for defining the structure and functionality of web APIs, ensuring interoperability and consistency. Notable examples include OpenAPI for REST APIs and Protocol Buffers for gRPC. Web APIs operate at different abstraction levels: definition languages specify structure, frameworks provide implementation tools, and platforms offer API management services. Four of the most popular APIs (RPC, SOAP, GraphQL,

and REST) are compared based on performance, ease of implementation, developer experience, and security.

3 RPC

Remote Procedure Call (RPC) enables applications to communicate over a network by invoking procedures, or functions, on remote systems as if they were local [12]. In RPC, a client can execute code on a server without worrying about the details of network communication, making it possible for distributed systems to operate efficiently and cohesively [7]. First introduced in the 1980s, RPC remains foundational in distributed computing, providing the basis for various frameworks that allow seamless cross-network interaction among services.

3.1 Definition of RPC

RPC abstracts the complexities of network interactions by enabling a client application to invoke remote methods directly through a ”stub” or ”proxy” object, which manages the process of serializing arguments, sending requests over the network, and deserializing the server's response [13]. Traditional RPC protocols were synchronous, meaning the client would wait for the server to complete the call and return results. However, many modern implementations also support asynchronous calls, allowing non-blocking communication, which is essential for high-performance applications.

RPC has evolved significantly, leading to the development of numerous RPC frameworks and protocols, such as gRPC and XML-RPC. Each framework provides different approaches and optimizations for cross-platform communication, support for various data serialization formats, and compatibility with different programming languages. These frameworks are widely used in microservice architectures, real-time systems, and Internet of Things (IoT) applications [13], where efficient, low-latency communication is crucial.

In Figure 1 is an example of a basic gRPC query in Java. In it, a location (point) is asked for, and the check feature function passes the location to the other server. If the location is found, it will return the locations latitude and longitude. This is the most simple form of a gRPC query, but it does show the general structure of many more complex functions.

3.2 RPC-based services - XML-RPC

XML-RPC (Extensible Markup Language-Remote Procedure Call) is a protocol that facilitates communication between distributed systems by enabling remote execution of

```
public void getFeature(
Point request, StreamObserver<Feature> responseObserver) {
  responseObserver.onNext(checkFeature(request));
  responseObserver.onCompleted();
}

...

private Feature checkFeature(Point location) {
  for (Feature feature : features) {
    if (feature.getLocation().getLatitude() ==
    location.getLatitude()
        && feature.getLocation().getLongitude() ==
        location.getLongitude()) {
      return feature;
    }
  }

  // No feature was found, return an unnamed feature.
  return Feature.newBuilder().setName("")
  .setLocation(location).build();
}
```

Figure 1: Example of a gRPC query [14]

procedures across networks. It was introduced as a lightweight and platform-independent mechanism for exchanging information between applications [12]. The protocol uses XML to encode its messages and relies on HTTP as the underlying transport mechanism.

The primary advantage of XML-RPC lies in its simplicity and interoperability. By leveraging XML for message formatting, XML-RPC ensures compatibility across diverse programming environments [15]. HTTP, as a universally supported transport protocol, enables seamless communication even in restrictive network configurations. XML-RPC supports a variety of data types, including integers, strings, arrays, and structs, making it suitable for various use cases such as remote service calls and cross-platform integration.

Although XML-RPC has been largely superseded by more modern standards such as RESTful APIs and JSON-RPC, its historical significance in shaping early web services and distributed computing cannot be understated. It serves as a foundational technology for understanding the evolution of remote procedure call mechanisms in networked systems.

3.3 RPC-based services gRPC

Google Remote Procedure Call (gRPC) is an open-source RPC framework developed by Google in 2016 [16]. It facilitates efficient communication between distributed systems by enabling clients to directly call functions on remote servers as though they were local, using HTTP/2 for transport, Protocol Buffers (protobuf) for efficient data serialization, and built-in support for bidirectional streaming [17].

Designed for speed and scalability, gRPC is ideal for micro-service architectures and real-time applications, offering features such as load balancing, retry logic, and deadline propagation. gRPC supports various programming languages, making it versatile for cross-language environments. Additionally, gRPC can perform synchronous and asynchronous calls, and it offers ”streaming” capabilities, allowing continuous data transfer between client and server [17]. Overall, gRPC’s efficiency and cross-platform compatibility make it a popular choice for modern, high-performance networked applications.

3.4 Evaluation

Performance RPC achieves high performance by focusing on lightweight communication and efficient data serialization. Protocols like gRPC leverage binary serialization formats such as Protobuf, which are compact and faster to parse than text-based formats like JSON or XML. RPC’s procedural nature allows for low-latency interactions, making it suitable for high-performance, real-time systems. However, its tightly coupled client-

server interactions may introduce latency in distributed environments where network conditions fluctuate. Proper implementation of batching and streaming features in frameworks like gRPC can further enhance performance in appropriate use cases. Kiraly and Szekely [12] compare the performance of different RPC implementation, specifically gRPC, XML-RPC and jsonRPC with servers and clients using different coding languages. gRPC performed 25-50 percent faster in handling requests compared to the other two, even though all three of them are modern RPC architectures.

Ease of implementation RPC occupies an intermediate position in terms of ease of implementation. It is relatively straightforward for simple use cases, as developers can define functions that are directly accessible over a network. Frameworks like gRPC facilitate the development process by automating code generation from `.proto` files. However, RPC can become complex when dealing with non-trivial data serialization or multi-language environments, necessitating a deeper understanding of protocols such as Protobuf. This added complexity makes it less accessible for beginners but highly efficient for performance-critical applications. Zhang et al. [18] conducted a large scale analysis of RPC in industrial environments, focused on errors caused by faulty requests or faulty implementation. They highlight that RPC's in industrial settings are used in microservices, creating a web of dependencies that are difficult to test in isolation.

Developer experience RPC delivers an efficient but somewhat constrained developer experience. For simple use cases, its procedural nature allows developers to call functions across networks with minimal overhead. Frameworks like gRPC enhance productivity by automating code generation from *.proto* files, fostering consistency across languages. However, debugging binary protocols such as Protobuf can be less intuitive compared to the text-based formats of REST and GraphQL. Additionally, RPC's tightly coupled nature between client and server can reduce flexibility and complicate changes during the development lifecycle. Berg and Mebrahtu Redi [17] studied REST vs. grpc based on server-side implementation. They indicate that when conducting multiple requests to multiple clients gRPC performance time improves in comparison to RESTful services. On the other hand REST performs better than RPC on single on single requests, probably caused by gRPC requiring a connection to be established seperately.

Security RPC protocols, such as gRPC, present a mixed security profile. Modern implementations like gRPC support built-in encryption and authentication mechanisms, leveraging TLS for secure communication and pluggable authentication modules for flexibility. However, the tight coupling between client and server and the procedural nature of RPC can make it vulnerable to replay attacks and injection if not properly

sanitized. Binary protocols like Protobuf may obscure payload data from attackers but do not inherently prevent attacks, emphasizing the need for robust access controls and validation mechanisms. Berg and Mebrahtu Redi [17] note, that securing RPC requires considered effort from developers to create secure services. Centralized API distribution in a large-scale network does enhance security in comparison with REST.

3.5 Summary

Evaluating RPC frameworks from various perspectives highlights both their strengths and challenges. RPC generally provides a lightweight, high performance API that requires carefull structural planning in larger systems. Modern developments, specifically gRPC, provide structured service definitions and cross-platform support, though debugging binary protocols can be more challenging than text-based alternatives. Security in RPC implementations requires careful consideration. While modern frameworks like gRPC offer encryption and authentication mechanisms, the tightly coupled nature of RPC can introduce security vulnerabilities if not properly managed. Proper access controls, secure API distribution, and validation mechanisms are essential for ensuring robust security in large-scale distributed networks.

4 SOAP

SOAP (Simple Object Access Protocol) is a protocol used to exchange structured information in web services. SOAP APIs requires the use of XML for the message format and then utilizes various standard transport protocols (such as HTTP, SMTP, and TCP) for messaging over networks [19].

4.1 Definition of SOAP

SOAP is a protocol standard for enabling communication between distributed systems. Originally developed by Microsoft in collaboration with the World Wide Web Consortium (W3C) in 2000, SOAP provides a formalized framework for exchanging structured information via XML-based messages over a variety of network protocols, most commonly HTTP and SMTP [20].

The protocol is characterized by its extensibility and robustness [20] . SOAP defines a strict messaging structure, which includes an envelope for specifying the message framework, a header for additional metadata, and a body for the actual payload. This structure allows SOAP to support advanced features such as security, reliability, and transaction management [3].

SOAP is platform and language agnostic, making it an ideal choice for enterprise-level applications that require interoperability among heterogeneous systems [21]. However, its complexity and verbosity have led to the rise of alternative protocols, such as REST, that prioritize simplicity and lightweight communication. Despite this, SOAP remains a cornerstone technology in contexts requiring high levels of reliability and compliance with strict standards, such as financial services and healthcare. It can also be used in combination with other APIs to supplement its disadvantages [21]. In Figure 2 I show an example of a simple SOAP query in HTML. The beginning of the code section after envelope defines the message as a SOAP message, and provides the specific encoding style. While this basic example is simple, all soap queries require an envelope, and without addition tools the different parameters required for an envelope makes writing multiple different queries time consuming.

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body>
  <m:GetBook xmlns:m="https://www.SiteWeQuery.com/books">
    <m:Item>BookName</m:Item>
  </m:GetAuthor>
</soap:Body>

</soap:Envelope>
```

Figure 2: Example of a SOAP query [22]

4.2 Evaluation

Performance SOAP typically performs poorly compared to the other paradigms [4] due to its verbose XML-based messaging format and reliance on complex processing requirements such as parsing WSDL definitions and applying WS-Security standards. SOAP's inherent overhead makes it less suitable for scenarios requiring low-latency communication or high throughput. However, its synchronous and asynchronous messaging capabilities allow for reliable communication in distributed systems. SOAP's performance can be improved with optimized XML parsing techniques and compression, though these efforts increase implementation complexity. Tihomirovs and Grabis [4] offers

a study on SOAPs performance through evaluating Cost, Effort,Lines of code, Execution speed, Memory and Errors. Interestingly, SOAPs response times with querying messages were found to be 4-5 times higher than even an XML styled REST query.

Ease of implementation SOAP is the most challenging to implement among the four definition languages. Its reliance on XML-based messaging and WSDL (Web Services Description Language) to define service interfaces adds significant overhead [20]. While libraries such as Apache CXF and .NET WCF provide abstractions, SOAP's verbose nature and rigid structure make it cumbersome compared to other approaches. Moreover, the integration of advanced features, such as WS-Security, further complicates its implementation, often necessitating specialized expertise.

Developer experience SOAP offers a robust yet cumbersome developer experience. SOAP's rigid structure, reliance on XML, and verbose payloads increase the cognitive and technical load for developers [20]. Tools such as SOAP UI and Integrated Development Environments (IDEs) with WSDL support mitigate some challenges but do not eliminate the inherent complexity. While SOAP's built-in features, such as WS-Security, cater to enterprise use cases, they add further layers of complexity, often requiring specialized expertise. This makes SOAP less developer-friendly compared to the other paradigms, particularly for those new to the protocol.

Security SOAP stands out as the most security-focused paradigm due to its comprehensive, built-in standards. Features like WS-Security provide message-level encryption, digital signatures, and robust authentication, ensuring secure communication even in intermediary scenarios where transport-level security may not suffice [4]. Additionally, SOAP's use of WSDL (Web Services Description Language) enables detailed access control definitions. However, SOAP's complexity can introduce misconfigurations, and its verbose XML payloads may expose systems to XML-based attacks such as XML external entity injection. Properly implementing SOAP's security features often requires specialized expertise, increasing development effort but resulting in a robust security posture.

4.3 Summary

SOAP is a powerful yet complex protocol that excels in secure and reliable communication across distributed systems. While its performance and ease of use may not match modern alternatives, its robustness and compliance with strict security and interoperability standards ensure its continued relevance in specific domains. The utility in using SOAP as a web API depends heavily on the level of security that the system requires. For light-

security systems, SOAP is often low performing and hard to develop for in comparison to other web APIs.

5 REST

Representational State Transfer (REST) is an architectural style for designing distributed systems. REST emphasizes a stateless, client-server communication model that uses standard HTTP methods (GET, POST, PUT, DELETE) [7] to interact with resources identified by Uniform Resource Identifiers (URIs).

5.1 Principles of REST

RESTful systems adhere to a set of guiding principles, including statelessness, uniform interface, and resource representation via standard formats such as JSON or XML. These principles enable REST to be lightweight and scalable while promoting loose coupling between client and server. The stateless nature of REST ensures that each request from a client to a server contains all the information necessary to process the request, thereby simplifying interactions and improving scalability [11].

According to Rodríguez et al. [23] the design principles of REST APIs are: resource addressability, resource representations, uniform interface, statelessness and Hypermedia as the engine of state. Due to its simplicity, flexibility, and compatibility with web standards, REST has become the dominant design paradigm for web APIs and distributed systems. It is widely adopted in domains ranging from social media to cloud computing [10], providing a foundation for modern web services that prioritize ease of use and interoperability.

5.2 REST-based services OpenAPI

OpenAPI is a specification for defining and documenting RESTful APIs in a machine-readable format. Originally known as the Swagger Specification, OpenAPI provides a structured way to describe the endpoints, request/response parameters, data formats, authentication methods, and other key details of an API [24].

The OpenAPI Specification (OAS) uses JSON or YAML to describe APIs, making it both human-readable and compatible with automated tools. These tools can generate interactive documentation, client libraries, and server stubs, streamlining the development process [8]. Additionally, OpenAPI facilitates testing and validation, ensuring that APIs meet specified requirements and function reliably.

As a vendor-neutral and language-agnostic standard, OpenAPI is widely used across industries to improve collaboration between teams and support interoperability in distributed systems. Its role in fostering standardization and automation has made it a cornerstone in modern API development.

5.3 REST-based services JSON:API

JSON:API is a specification for building APIs that standardizes how resources are defined and exchanged using JSON (JavaScript Object Notation). It aims to streamline API development by providing a consistent structure for client-server interactions, focusing on efficiency, simplicity, and interoperability. The specification prescribes conventions for resource identification, relationships, metadata, and error handling, ensuring uniformity across implementations [25].

Key features of JSON:API include support for compound documents to reduce the number of network requests, standardized pagination, filtering, and sorting mechanisms, and a consistent approach to handling errors. By adhering to the JSON:API specification, developers can reduce the need for custom client-side code and facilitate seamless integration between distributed systems.

JSON:API has gained significant popularity as a specification for building APIs, particularly in scenarios requiring consistent and efficient data exchange. Its adoption is fueled by its ability to reduce redundancy in client-side code and its focus on standardization, which simplifies integration across distributed systems. JSON:API is widely used in modern application development, especially within microservices architectures and applications needing reliable client-server communication, such as single-page applications and mobile apps. While JSON:API is not as popular as OpenAPI, it does demonstrate a desire for new API Services, as it published its first stable version only two years ago in 2022.

5.4 REST-based services RAML

RAML (RESTful API Modeling Language) is a specification designed to facilitate the design, documentation, and implementation of RESTful APIs [26]. It provides a structured and human-readable format, typically written in YAML, for defining API resources, endpoints, data models, and behaviors [27]. RAML emphasizes simplicity and clarity, making it a tool that bridges communication between developers and stakeholders while promoting consistency in API design.

One of RAML's key features is its modularity, allowing reusable components such as traits, resource types, and libraries. This modularity not only enhances efficiency in

API development but also ensures scalability and maintainability, particularly for large systems. RAML also integrates with various tools for generating documentation, testing APIs, and creating client or server code.

By providing a design-first approach to API development, RAML enables developers to focus on creating robust and user-friendly APIs while adhering to RESTful principles [26]. Its emphasis on standardization and readability has made it a popular choice for teams prioritizing collaboration and streamlined workflows in API development.

5.5 Evaluation

Performance REST exhibits moderate performance characteristics due to its reliance on HTTP and the stateless, resource-based architecture. While REST's simplicity ensures compatibility and scalability, it can suffer from inefficiencies such as over-fetching (retrieving more data than necessary) or under-fetching (making multiple requests to gather related data) [28]. The text-based JSON payloads typically used in REST APIs are easy to parse but can be larger in size compared to binary formats, potentially increasing bandwidth usage. Caching mechanisms like HTTP caching can mitigate some performance bottlenecks, particularly for read-heavy operations. Rodríguez et al. [23] examines the usage of best practices in REST APIs and the effect of ignoring best practices for performance.

Ease of implementation REST is often regarded as the most straightforward API style to implement [23]. Its reliance on HTTP methods such as GET, POST, PUT, and DELETE to map creation, reading, update and deletion (CRUD) operations simplifies its design. Additionally, the extensive availability of frameworks, including Flask, Express, and Django REST Framework, reduces the development effort significantly. REST's alignment with HTTP's inherent features, such as caching and statelessness, further enhances its ease of implementation, particularly for developers with a basic understanding of web protocols.

Developer experience REST provides a favorable developer experience due to its simplicity and alignment with HTTP protocols. Its use of standard HTTP methods for CRUD operations makes it intuitive for developers familiar with web technologies [29]. REST APIs are well-supported by tools like Postman, Swagger, and a variety of frameworks, which streamline both development and debugging processes. However, REST can lead to challenges such as over-fetching or under-fetching of data, particularly in scenarios with complex relationships, potentially requiring additional effort to optimize endpoints.

Security REST offers a basic security model that is heavily dependent on the underlying HTTP protocol. Common mechanisms such as HTTPS, OAuth, API keys, and token-based authentication (e.g., Json Web Token) are widely used to secure REST APIs [30]. However, REST itself does not provide a standardized framework for security, leaving it up to developers to implement robust measures. While REST benefits from mature, well-documented practices, its reliance on endpoints for specific resources can increase the attack surface, making it vulnerable to threats like endpoint enumeration and injection attacks if improperly secured.

5.6 Summary

REST remains a foundational paradigm in API development due to its simplicity, scalability, and broad support across frameworks and tools. Emerging RESTful frameworks and evolving best practices continue to refine REST-based services, addressing its limitations and ensuring more efficient, secure, and maintainable implementations in modern web and cloud-based applications.

6 GraphQL

GraphQL is a query language developed by Facebook in 2012 and released as an open-source specification in 2015 [31]. Unlike REST APIs, which expose a set of predefined endpoints to retrieve fixed data structures, GraphQL enables clients to request precisely the data they need in a single, flexible query. This capability allows developers to query multiple resources in a single request, tailoring responses to specific application requirements and reducing the need for multiple network calls.

6.1 Definition of GraphQL

At its core, GraphQL operates with a single endpoint and uses a strongly typed schema that defines the data types, relationships, and operations available in the API [32]. Clients submit queries to this endpoint in a structured, hierarchical format, specifying the exact fields and relationships required, and the server responds accordingly. This granularity in data selection improves efficiency by avoiding over-fetching (retrieving too much data) and under-fetching (retrieving insufficient data), making GraphQL particularly useful for applications that consume complex and varied data sources.

In Figure 3, I show what a GraphQL query looks like. The query would return the title, and the authors name and bio of the book with id 1234. The queried object could have

a lot more information attached to it, but with GraphQL it will only return the info on fields that are specified.

```
Query {
  book(id: "1234") {
    title
    author {
      name
      bio
    }   }    }
```

Figure 3: Example of a GraphQL query [33]

6.2 GraphQL-based services

GraphQL tools are considerably less varied than those available for RESTful APIs. Prominent tools in the GraphQL ecosystem include GraphiQL, Codegen and GraphQL Shield. Additionally, Apollo offers features such as schema registry, query performance analytics, and error tracking, making it a comprehensive solution for managing GraphQL APIs [34]. These tools are centralized, and while they do offer services that can be found under REST APIs, they're future development is relying on metas continued support.

6.3 Evaluation

Performance GraphQL, designed to optimize data fetching, offers performance advantages in scenarios requiring precise and flexible data retrieval. By allowing developers to specify exactly what data they need, GraphQL avoids the over-fetching common in REST. However, its performance can degrade with complex queries, deeply nested relationships, or inefficient resolver implementations on the server. Unlike REST, GraphQL lacks native caching mechanisms, requiring developers to implement custom caching strategies or use libraries such as Apollo Client for client-side caching. The single endpoint model simplifies routing but places greater processing demands on the server, especially for queries requiring extensive backend computations.

Ease of implementation GraphQL, while more flexible than REST, presents a moderate level of complexity during initial implementation [35]. Unlike REST, GraphQL requires defining a schema and resolvers, which introduces additional design considerations. Frameworks like Apollo Server and Graphene offer tools to streamline development,

but a solid grasp of GraphQL's query language and execution process is necessary. This increases the learning curve, particularly for teams unfamiliar with the paradigm. Despite this, GraphQL's dynamic querying capabilities justify the additional complexity in scenarios requiring efficient data retrieval and manipulation.

Developer experience GraphQL offers a highly flexible developer experience, particularly in scenarios requiring precise data retrieval. Its query-based approach allows developers to request only the data they need [32], reducing inefficiencies common in REST APIs. The self-documenting nature of GraphQL schemas enhances discoverability and collaboration between frontend and backend teams. However, GraphQL's initial setup, involving schema and resolver definitions, introduces complexity [5]. Developers may face a steeper learning curve, particularly if they are unfamiliar with its query language or execution model. Despite these challenges, tools such as Apollo Client and GraphiQL significantly enhance developer experience by simplifying testing and monitoring.

Security GraphQL, with its dynamic query capabilities, introduces unique security considerations. Its ability to allow clients to request specific data makes it inherently susceptible to query-based attacks, such as query batching or Denial of Service (DoS) through complex or deeply nested queries. Mitigating these threats requires additional tools and measures, such as query complexity analysis, query depth limits, and server-side throttling [35]. On the other hand, GraphQL's single endpoint model reduces exposure to endpoint-specific attacks. Standard authentication mechanisms can be used, although the absence of an integrated security framework places the burden of implementation on developers.

6.4 Summary

GraphQL, developed by Facebook in 2012 and open-sourced in 2015, is a query language that allows clients to request exactly the data they need in a single, flexible query [32]. GraphQL operates through a single endpoint with a SQL styled query, reducing over- and under-fetching of data. GraphQL optimizes data fetching but can suffer performance issues with deeply nested queries or inefficient resolvers. Its flexibility introduces implementation complexity, necessitating schema and resolver definitions. The ability to make large and deeply nested queries makes GraphQL more vulnerable to DoS attacks. The GraphQL ecosystem includes tools to aid in the implementation and managment of GraphQL. However, as these are services are centralized they rely on continued support from Meta.

7 Discussion

In this section I will use the prior analysis of web APIs to make a more direct comparison of the existing web APIs. After comparing the APIs with each other, I will highlight what areas the different solutions are most suited for, and potential ways to mitigate their shortcoming. One of the challenges in comparing existing APIs, is that many of the evaluation criteria, such as developer experience, are not objective. Other factors that make direct comparison between APIs challenging is that the theoretical optimal API implementation to take advantage of the chosen system is going to differ from actual implementations. These difficulties are considered in the "ease of implementation" consideration earlier.

7.1 Comparison of web API definition languages

I have compiled prior analysis of web API definition languages to compare them with each other in Table 1. An important factor to consider in comparing web API architectures is that the data gathered depends on the implementations, not the potential of a web API architecture. The popularity of an API is likely going to effect the results. Developers are going to be more proficient with a popular API, and less likely to fall into known pitfalls.

The existing API definition languages are rather varied, especially when you take into consideration the additional tools that are available for them. REST APIs are unique among these as they are more of an architectural structure, that the service should follow. Tools like openAPI create proper limitations on the system, that somewhat ensure that the built system follows the recommendations of REST APIs. This limitation makes both maintaining the systems and creating queries for it more unified, and that in turn helps in clarifying how to work with those systems.

Security considerations are one possible area of development in future APIs. SOAP API currently has the most robust inbuilt security architecture, but it also has worse performance in comparison to more lightweight solutions. This creates potential areas for future development, but developing a new API, and having it widely adopted, requires considerable resources. It seems more likely that tools will be created for SOAP to limit its downsides, or aid in the creation of hybrid API solutions.

7.2 Hybrid API Solutions

The different APIs are not exclusionary, so systems can be built with different APIs in different sections. This can be done to use the strengths of, for example, SOAPs security in an area that requires it, while allowing for the use of more lightweight APIs

Table 1: Comparison of Web Service Protocols

	Performance	**Ease of implementation**	**Developer experience**	**Security**
RPC	Quick speed, and low latency	Requires architectural consideration to implement	Modern tools simplify query generation, old implementations require coding knowledge	No built-in security, modern tools implement addition security options
SOAP	Worst performance due to data size	Restrictions on XML makes implementation cumbersome	Unusual request structure requires getting used to	Data format and built-in tools creates robust security
REST	Reliable in small implementations, large scale is prone to over-fetching	Relies on known programming, has a lot of tools and guides	Prone to faulty requests when systems get larger	No built-in security protocols, up to developers
GraphQL	Accurate queries reduce load especially in large query contexts	Upfront effort to develop, structure makes maintenance easier	Queries are more verbose, but more specified	No built-in security protocols, has unique security concerns

in other sections. This flexibility allows for the optimization of performance, security, and efficiency based on specific needs, making it easier to tailor solutions for different use cases.

While these hybrid solutions can solve some issues, it does mean that the documentation for the service has to be clear what queries use which APIs, or risk having the queries become considerably more cumbersome to write. Additionally, managing multiple API styles increases the complexity of troubleshooting and maintenance, as developers must be familiar with different API formats.

7.3 Summary

Having an API definition language with all the strengths of the existing ones is not going to fix the issues caused by hybrid solutions, as functionality is not the only reason they

exist. Many systems are built from pre-existing web APIs that can be modified with little issues. Using easy to implement solutions will often be preferred over a more reliable long term solution. The possible areas for completely new API definition languages, or new tools for existing APIs, to emerge are probably going to be uses where the hindrances of existing definition languages become more pronounced. With gRPC, it became advantageous to design something new for IoT solutions, as they had different requirements to traditional web services.

8 Conclusion

In this thesis, I sought the answer to the following research questions: What web API definition languages exist? what kind of tools are available for them? And how do they compare to each other based on their performance, security features, developer experience, and ease of implementation? The web API definition languages that I chose to focus on were RPC, SOAP, REST and GraphQL. These four APIs represent a diverse range of approaches and paradigms in API design, each shaped by the technological requirements and trends of its time. By analyzing the research done on them, I aim to provide insight into the current landscape and future trajectory of web API definition languages.

8.1 Main findings

The development of new APIs is seemingly centered on fixing small issues in old systems that have become more troublesome due to a change in the industry. The existing popularity, and reliance on standard HTTP requests makes REST an easy API to implement for projects on the side.

REST, while not the best in any particular point of comparison, is good enough to be preferable in many situations. SOAP has been preferred in more rigid, security based implementations, where the trade-off of bandwidth usage and slower response times is an acceptable trade-off. GraphQL appears to be comparable on with REST on ease of implementation. However, the benefits gained from GraphQL's query precision is negligible for services with relatively small volumes of API queries.

For larger platforms like Facebook, GraphQL does provide considerable improvements. RPC has found a new niche, where further development on a seemingly outdated API method is warranted. The emergence of micro-services and IoT-devices brought on old problems of bandwidth efficiency and realtime capabilities that weren't optimal with REST. Currently, hybrid implementations of web APIs can supplement the weaknesses of each definition language, by using them in specifically in areas where they excel.

8.2 Open research questions

In looking at existing research, some gaps became clear. A lot of comparison exists between different REST API solutions, and REST with other APIs. However there is a clear lack of comparing non-REST APIs with each other. Relatively recent developments like gRPC and GraphQL create their own gaps in research. Research on the tools existing for GraphQL, how they can be implemented into hybrid solution and how their API endpoints degrade overtime are of specific interest.

”AI” services could cause new requirements for web APIs, in similar ways as IoT has. The needs of ”AI” web services is an emerging area, and is the most obvious current development that might create need for new API services. The emergence of potential new requirements and strains also leads to new focuses for research

References

[1] Pascal Giessler, Michael Gebhart, Dmitrij Sarancin, Roland Steinegger, and Sebastian Abeck. Best practices for the design of restful web services. In *International conferences of software advances (ICSEA)*, pages 392–397, 2015.

[2] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating web apis on the world wide web. In *2010 eighth ieee european conference on web services*, pages 107–114. IEEE, 2010.

[3] Abhijit Bora and Tulshi Bezboruah. Testing and evaluation of a hierarchical soap based medical web service. *International Journal of Database Theory and Application*, 7(5):145–160, 2014.

[4] Juris Tihomirovs and Jānis Grabis. Comparison of soap and rest based web services using software evaluation metrics. *Information technology and management science*, 19(1):92–97, 2016.

[5] Gleison Brito and Marco Tulio Valente. REST vs GraphQL: A controlled experiment. In *2020 IEEE international conference on software architecture (ICSA)*, pages 81–91. IEEE, 2020.

[6] Robert H Morrow and Adam J Mckee. Cgi scripts: A strategy for between-subjects experimental group assignment on the world-wide web. *Behavior Research Methods, Instruments, & Computers*, 30:306–308, 1998.

[7] Jacek Kopeckỳ, Paul Fremantle, and Rich Boakes. A history and future of web apis. *it-Information Technology*, 56(3):90–97, 2014.

[8] Open API Initiative. Open api specification v3.1, 2024. URL `https://spec.openapis.org/oas/v3.1.1.html`. [Accessed 08-12-2024].

[9] Google LLC. protobuf documentation, 2024. URL `https://protobuf.dev/overview/`. [Accessed 08-12-2024].

[10] Fabio Di Lauro, Souhaila Serbout, and Cesare Pautasso. A large-scale empirical assessment of web api size evolution. *Journal of web engineering*, 21(6):1937–1979, 2022.

[11] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[12] Sandor Kiraly and Szilveszter Szekely. Analysing rpc and testing the performance of solutions. *Informatica*, 42(4), 2018.

[13] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. Grpc: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Operating Systems Review*, 27(3): 75–86, 1993.

[14] Jonathan Stoikovich. grpc basics, 2021. URL `https://grpc.io/docs/languages/java/basics/`. [Accessed 08-12-2024].

[15] Mark Allman. An evaluation of xml-rpc. *ACM sigmetrics performance evaluation review*, 30(4):2–11, 2003.

[16] gRPC Authors. About grpc, 2024. URL `https://grpc.io/about/`. [Accessed 08-12-2024].

[17] Johan Berg and Daniel Mebrahtu Redi. Benchmarking the request throughput of conventional api calls and grpc: A comparative study of rest and grpc, 2023.

[18] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. White-box fuzzing rpc-based apis with evomaster: An industrial case study. *ACM Transactions on Software Engineering and Methodology*, 32(5):1–38, 2023.

[19] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. Soap version 1.2 part 1: Messaging framework (second edition), 2007. URL `https://www.w3.org/TR/soap12-part1/`. [Accessed 08-12-2024].

[20] Festim Halili, Erenis Ramadani, et al. Web services: a comparison of soap and rest services. *Modern Applied Science*, 12(3):175, 2018.

[21] Jonathan Lee, Shin-Jie Lee, and Ping-Feng Wang. A framework for composing soap, non-soap and non-web services. *IEEE Transactions on Services Computing*, 8(2): 240–250, 2014.

[22] W3 schools. Xml soap, 2003. URL `https://www.w3schools.com/xml/xml_soap.asp`. [Accessed 08-12-2024].

[23] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. Rest apis: A large-scale analysis of compliance with principles and best practices. In *Web Engineering: 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings 16*, pages 21–39. Springer, 2016.

[24] SmartBear Software. swagger spicification. URL `https://swagger.io/specification/`. [Accessed 12-03-2025].

[25] Yehuda Katz. Latest specification (v1.1), 2022. URL https://jsonapi.org/format/. [Accessed 08-12-2024].

[26] Vijay Surwase. Rest api modeling languages-a developer's perspective. *Int. J. Sci. Technol. Eng*, 2(10):634–637, 2016.

[27] Jonathan Stoikovich. Raml version 1.0: Restful api modeling language, 2021. URL https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/. [Accessed 08-12-2024].

[28] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology*, 33(1):1–41, 2023.

[29] Joel L Fernandes, Ivo C Lopes, Joel JPC Rodrigues, and Sana Ullah. Performance evaluation of restful web services and amqp protocol. In *2013 Fifth international conference on ubiquitous and future networks (ICUFN)*, pages 810–815. IEEE, 2013.

[30] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. An analysis of public rest web service apis. *IEEE Transactions on Services Computing*, 14(4):957–970, 2018.

[31] Armin Lawi, Benny LE Panggabean, and Takaichi Yoshida. Evaluating graphql and rest api services performance in a massive and intensive accessible information system. *Computers*, 10(11):138, 2021.

[32] Patrick Stünkel, Ole von Bargen, Adrian Rutle, and Yngve Lamo. Graphql federation: A model-based approach. *Journal of Object Technology*, 2020.

[33] GraphQL. Queries, 2021. URL https://graphql.org/learn/queries/. [Accessed 08-12-2024].

[34] Runjie Jin, Robert Cordingly, Dongfang Zhao, and Wes Lloyd. Graphql vs. rest: A performance and cost investigation for serverless applications. In *Proceedings of the 10th International Workshop on Serverless Computing*, pages 37–42, 2024.

[35] Gleison Brito, Thais Mombach, and Marco Tulio Valente. Migrating to graphql: A practical assessment. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 140–150. IEEE, 2019.