U.P.B. Sci. Bull., Series C, Vol. 86, Iss. 3, 2024 ISSN 2286-3540

A WEB SERVICE IMPLEMENTATION – SOAP VS REST

Tudor-Alin NIȚESCU[1], Andreea-Iulia CONCEA-PRISĂCARU[1], Valentin SGÂRCIU[2]

Nowadays, the usage of web services has increased significantly, therefore choosing a proper architecture is a really important step when designing a web application. The software architecture can determine the overall structure of the application, its components and how they interact between themselves. The most popular software architectural patterns are SOAP and REST, as they are spread all over the software development market. The main goal of this paper is to highlight the importance of adopting a proper architecture when designing a web service, as well as to compare the two architectural patterns SOAP and REST. To achieve this, we are going to showcase and apply some of the most important concepts of this field via a demo web application.

Keywords: microservices, web services, web architecture, SOAP, REST.

1. Introduction

In recent years, the software development market is experiencing continuous growth, driven by the increasing demand for web services across the software domain. The primary quality of web services is interoperability, enabling various systems/components to communicate with each other, regardless of the programming language, framework or platform used.

To ensure that web services are scalable, secured and reliable, it is essential to select a proper architectural pattern. REST (Representational State Transfer) and SOAP (Simple Object Access Protocol) are the most representative architectural patterns used for web services, each one having its own strengths and weaknesses. The decision to choose one of them should be based on the specific demands and requirements of each system. What can fold a certain system may not fold for another, thus in order to make the best decision an extensive analysis should be performed beforehand.

One notable trend in web service development is the shift towards more lightweight and flexible architectural patterns. While SOAP has traditionally been

[1] PhD student., Dept. of Automation and Industrial Informatics, University POLITEHNICA of Bucharest, Romania, e-mail: tudor_alin.nitescu@stud.acs.upb.ro.
[1] PhD student., Dept. of Automation and Industrial Informatics, University POLITEHNICA of Bucharest, Romania, e-mail: andreea.concea@stud.acs.upb.ro.
[2] Prof., Dept.of Automation and Industrial Informatics, University POLITEHNICA of Bucharest, Romania, e-mail: valentin.sgarciu@upb.ro.

the preferred option for constructing web services, the emergence of REST architecture has sparked considerable debate within the web software development area regarding the superior architectural choice. RESTful architecture has gained interest in the past years due to its simplicity, scalability and compatibility with the HTTP protocol.

On the opposite, SOAP-based systems offer robust features for security and reliability but are facing challenges mostly related to complexity due to the verbosity of SOAP messages and the strict adherence to XML schemas. Therefore, these can lower performance metrics and hinder interoperability in a web application.

Additionally, the integration of modern technologies such as microservices, containerization and cloud computing has further reshaped the landscape of web service development, focusing on agility, scalability and cost-effectiveness in deploying and managing web services.

This paperwork aims to perform a comparative study of SOAP and REST architectures, evaluating their respective strengths and weaknesses considering the demands of modern web service development area. This was achieved by implementing a software application using both architectural patterns and following the same requirements. The implementation and subsequent comparison involved an evaluation of various factors such as performance, scalability and ease of implementation.

2. State of the art

In this chapter we are going to explore the findings of other researchers from both academical and businesses domains. This comprehensive overview will provide a better understanding of the current state of knowledge in this field, offering a solid foundation for our future research and analysis.

A group of researchers from Riga Technical University compared the two software architectures, SOAP and REST, having as guidelines the following criteria: costs, code length, speed and reliability. Their results were not concluded, as the decision of adopting a certain architecture should be based on the requirements of each system. However, they are recommending the usage of REST for simpler systems and the usage of SOAP for more complex systems consisting in more components, as SOAP is offering additional security layers [1].

A group of Macedonian researchers also studied the differences between the two architectural models. Their findings reveal that SOAP is more restricted in code level, while REST is more permissive, allowing free format and focusing on modular code. On top of that, developing applications based on SOAP is more challenging compared to those based on REST, primarily due to SOAP being an older software architecture, while REST represents a more modern approach. On the other hand, REST is still lacking standards on the security policies side, while SOAP has better support on this end [2].

Another study has brought light over this topic by presenting the individualization of each architectural model features. Their findings provided more concrete conclusions compared to previous studies, indicating that REST outperforms SOAP in terms of speed and memory efficiency, whereas SOAP excels in terms of security and reliability [3].

A thesis studying the migration efforts from SOAP to REST revealed that the superior performance of REST client through faster runtime and consistent CPU usage enhances the application's capability to handle more requests within the same timeframe. Moreover, replacing the SOAP implementation with a more efficient and maintainable service architecture such as REST helps in minimizing the risks and costs for maintaining an outdated system, since future updates and improvements to the codebase will require fewer resources. Factors like better code quality, interface-driven design, unified response handling and modular code structure ensure that developers can efficiently modify and maintain the code as needed, helping the software project to adapt to evolving requirements and changes, while reducing maintenance costs and resource demands [4].

In a recent study employing an experimental approach, ten students were divided into two groups to assess the maintainability of both REST and SOAP services. Each group had individual tasks to modify and improve the web services of already existing applications, in both server and client side. Notably, despite the similar number of lines of code required for the REST and SOAP providers, REST clients required twice as many lines of code as SOAP clients. However, this had no impact on the cyclomatic complexity, which was the same in both implementations. The study's findings indicate that designing the application based on MVC (Model-View-Controller) architecture helps in lowering the maintenance costs, regardless of whether REST or SOAP protocols are being used, due to its inherent loose coupling. However, the conclusions suggest that providing web services implies lower costs using REST, while consuming web services is associated with lower maintenance costs when using SOAP [5].

These results have shown that both architectural patterns have advantages and disadvantages, and their adoption is disputable and should be based on some KPIs (Key Performance Indicators), reporting them to each system context.

3. Methodology

This section will introduce the main concepts utilized in our implementation which serves as a practical comparison between the two architectures.

3.1. Concepts

3.1.1. Web services and microservices

The main key concepts used on a wide scale in the field of software development are web services and microservices.

Web services represent a group of software applications that can be created using a variety of programming languages, involving the usage of standardized protocols, such as HTTP, and facilitating the communication between different types of devices across the internet.

Microservices, on the other hand, refer to a particular architectural style for creating, designing and delivering applications. This concept involves breaking down applications into smaller, independent services (called microservices) that can communicate effectively and utilize lightweight mechanisms for specific business needs. Each component of the main web service is created and deployed as a separate service. As a result, services can be upgraded, scaled, or even replaced without affecting the entire program, providing flexibility. Furthermore, the scalability of microservices leads to a lower cost of development when comparing with other technologies [6].

3.1.2. REST API

REST is a type of API architecture that enables communication between a client and a server in web applications using the HTTP protocol. It offers flexibility and isn't bound to a specific transfer protocol, making the implementation straightforward. The main components of REST include addressability, a uniform interface, and statelessness. REST functions similarly to CRUD operations (Create, Read, Update, Delete), which map to popular database operations like INSERT, SELECT, UPDATE, and DELETE in SQL. [7]

The goal of REST is to create a set of guidelines for designing distributed systems that offer optimal performance, scalability, and simplicity. These architectural qualities are achieved by imposing specific restrictions on components, interfaces, and data elements.

REST operates within the client-server model and uses a request-response communication flow. A client initiates an action on a specific resource by sending a request from a Web Application, API or any component that makes an API call. This request must contain the identifier of the resource and the action to be performed on it. Depending on the action, the request and response messages may have additional meta-data elements, which can be classified into resource data, resource meta-data, representation data, representation meta-data, and control data [8]. The request is then processed and based on the action specified by the HTTP request method, the data is either fetched, created, modified or removed from the database. The high-level design of the REST architecture can be found below, in Fig. 1.

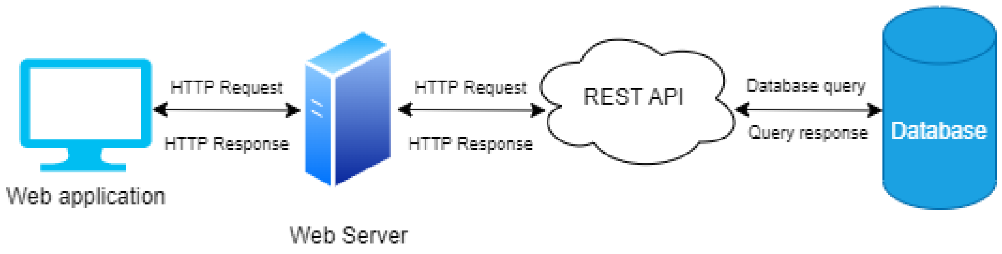


Fig. 1. REST architecture

3.1.3. SOAP API

Simple Object Access Protocol (SOAP) is a messaging protocol that utilizes XML messaging format for communication across networks. It is a critical component in Service-Oriented Architecture (SOA) and its related web services needs. The main goal of SOAP is to transmit data over the network, using HTTP to transfer information across the internet.

SOAP Binding refers to the method of exchanging messages over the Transport layer. There are two different binding styles for messaging requests in SOAP: Remote Procedural Call (RPC) and Document Style:

- Remote Procedural Call (RPC) – This concept entails communication between a client and a server based on a request and a response, using XML as the format for both. The root element, Envelope, determines the overall structure of the message. RPC uses a specific design for the request and response messages. Communication in RPC is synchronous, meaning that a response is received only after the message request has been sent. This style is simpler and less versatile, making it more suitable for sending smaller messages.
- Document Style - It is a more sophisticated and complex style, often referred to as message-oriented. In this style, XML data is passed as a body rather than as parameters. It allows rich content and can handle complex data structures [3].

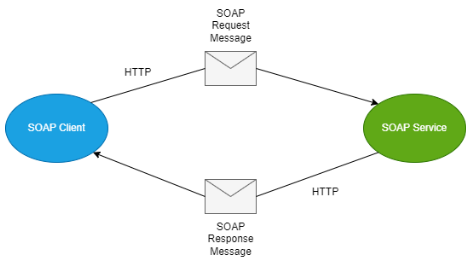


Fig. 2. SOAP architecture

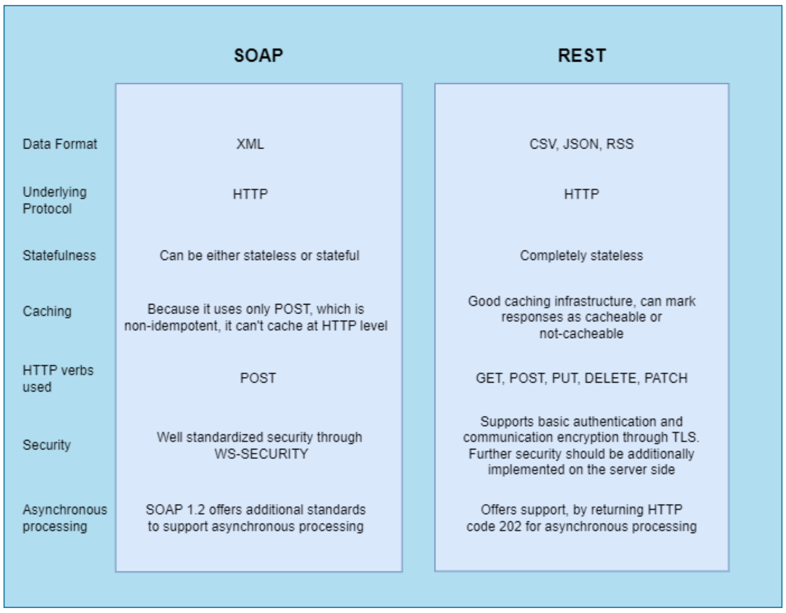
3.1.4. REST API vs SOAP API

Both REST and SOAP frameworks offer support for Service Oriented Architecture (SOA) applications. The choice between them depends on the business requirements and the architecture of the entire system. SOAP is an appropriate choice for applications that require a high level of security, reliability, and transaction management, and for applications that need to exchange complex data over the network. REST is the better choice for applications that need to be lightweight, flexible, and scalable. The best approach depends on the specific requirements of an application.

A comparison between the two architectural styles is provided below in Fig 3., which was made based on the following categories:

- **Data Format**: while SOAP is very strict regarding the data format, allowing XML only, REST is more permissive, allowing formats such as CSV, JSON and RSS.
- **Underlying Protocol**: both use the HTTP protocol.
- **Statefulness**: while SOAP can be either stateless (does not retain information about the state of a client between requests to the server) or stateful (each request from a client to the server is treated as an independent transaction; the server does not store any client-specific data between requests), REST is completely stateless.
- **Caching**: SOAP can use only POST requests, which are non-idempotent (can provide different results when repeating the same operation). Therefore, it can't cache at HTTP level. On the other hand, REST has an entire caching infrastructure, being able to mark responses as cacheable or not-cacheable.
- **HTTP verbs used**: SOAP is strictly tied to POST, while REST can use GET, POST, PUT, DELETE and PATCH.
- **Security**: SOAP provides well standardized security through WS-SECURITY (Web Services Security), which includes specifications for message integrity, confidentiality, authentication and authorization. On top of that, it allows encryption and signing SOAP messages to ensure their integrity and confidentiality, as well as the authentication of both the sender and the receiver. On the opposite side, REST supports basic authentication and communication encryption through TLS (Transport Layer Security) and it requires further implementation on the server side to enhance the security level of the application.

- **Asynchronous processing**: Both REST and SOAP support asynchronous processing.



	SOAP	REST
Data Format	XML	CSV, JSON, RSS
Underlying Protocol	HTTP	HTTP
Statefulness	Can be either stateless or stateful	Completely stateless
Caching	Because it uses only POST, which is non-idempotent, it can't cache at HTTP level	Good caching infrastructure, can mark responses as cacheable or not-cacheable
HTTP verbs used	POST	GET, POST, PUT, DELETE, PATCH
Security	Well standardized security through WS-SECURITY	Supports basic authentication and communication encryption through TLS. Further security should be additionally implemented on the server side
Asynchronous processing	SOAP 1.2 offers additional standards to support asynchronous processing	Offers support, by returning HTTP code 202 for asynchronous processing

Fig. 3. SOAP vs REST

3.2. Tools

For the implementation, we have used the following tools: IntelliJ Enterprise Edition for the coding part (written in Java) and Postman for the code testing through API calls.

3.2.1. IntelliJ

IntelliJ IDEA is an integrated development environment (IDE) created by JetBrains. It is a Java-based software that provides a comprehensive set of tools and features to help developers write, test, and debug code efficiently. IntelliJ IDEA's intelligent code completion and error analysis capabilities help developers to write code quickly and accurately. Additionally, its integrated debugging tools simplify the process of identifying and resolving errors. The IDE offers advanced refactoring capabilities, facilitating the reorganization and restructuring of code. This ensures that the code remains well-structured, maintainable, and readable [9].

3.2.2. Postman

Postman is a user-friendly interface designed for sending HTTP requests, receiving responses, and visualizing the data returned from APIs. With Postman, developers can easily test and debug their APIs. It provides a large variety of features, including the ability to save and organize collections of API requests, automate repetitive tasks, generate code snippets in various programming languages, and perform advanced tasks such as setting up and managing environments and variables [10].

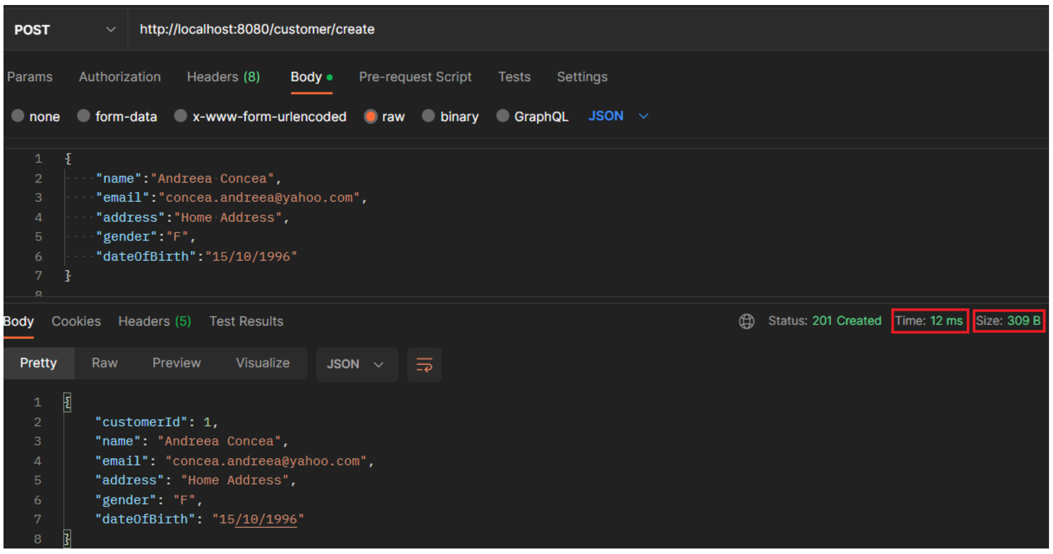
4. Implementation

This section showcases the implementation of a web services application, adhering to both REST and SOAP architecture principles, for the purpose of conducting a comparative analysis.

The implemented application provides functionalities for managing customers and accounts, enabling a range of CRUD operations (create, read, update and delete) for both customers and accounts. Upon creation in the database, each customer has the capability to be associated with one or multiple accounts, representing a one-to-many relationship between customers and accounts tables in the database. Additionally, the application ensures data integrity and consistency by implementing input validation mechanisms and enforcing referential integrity constraints between customers and their associated accounts within the database schema.

4.1. REST implementation

For the REST implementation, we have split the code in various code packages. In the controller package we have implemented the logic for handling incoming HTTP requests; in the entity package we have defined the model structure (the attributes and validations for Account and Customer objects); in the repository package we have defined the database operations (create, read, update and delete); in the service package we have implemented the methods which are called by the endpoints to perform the database operations. We have used SQLite, which is an in-memory database (it stores the data in computer’s main memory, eliminating the need to access disk storage). In the figures from below, the results of CRUD operations (Create, Read, Delete and Update) performed using Postman HTTP requests are illustrated, with the response times and sizes highlighted with red in response section.



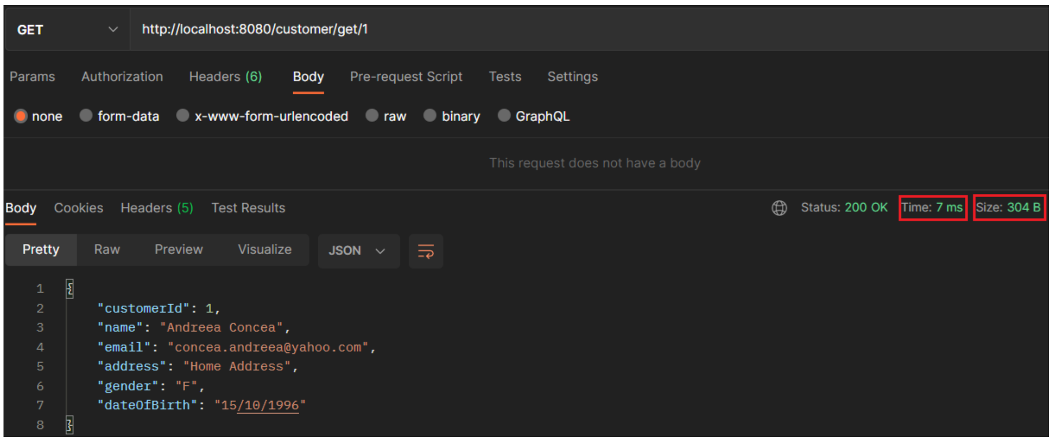
```
POST http://localhost:8080/customer/create
```

```
{
    "name":"Andreea Concea",
    "email":"concea.andreea@yahoo.com",
    "address":"Home Address",
    "gender":"F",
    "dateOfBirth":"15/10/1996"
}
```

Status: 201 Created Time: 12 ms Size: 309 B

```
{
    "customerId": 1,
    "name": "Andreea Concea",
    "email": "concea.andreea@yahoo.com",
    "address": "Home Address",
    "gender": "F",
    "dateOfBirth": "15/10/1996"
}
```

Fig. 4. POST (Create) example for REST API



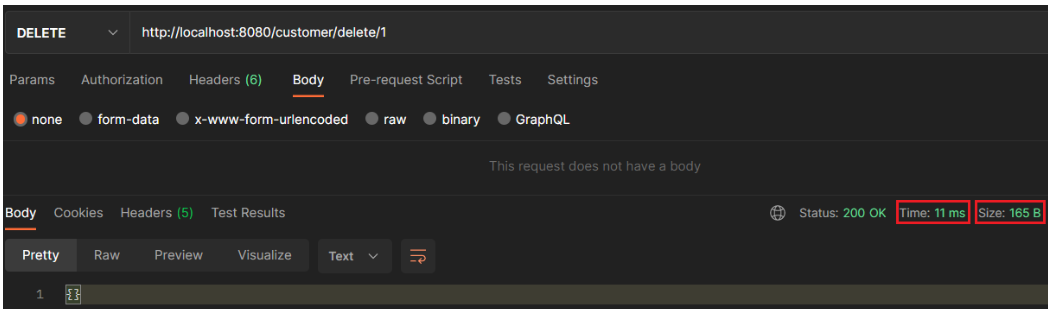
```
GET http://localhost:8080/customer/get/1
```

This request does not have a body

Status: 200 OK Time: 7 ms Size: 304 B

```
{
    "customerId": 1,
    "name": "Andreea Concea",
    "email": "concea.andreea@yahoo.com",
    "address": "Home Address",
    "gender": "F",
    "dateOfBirth": "15/10/1996"
}
```

Fig. 5. GET (Read) example for REST API



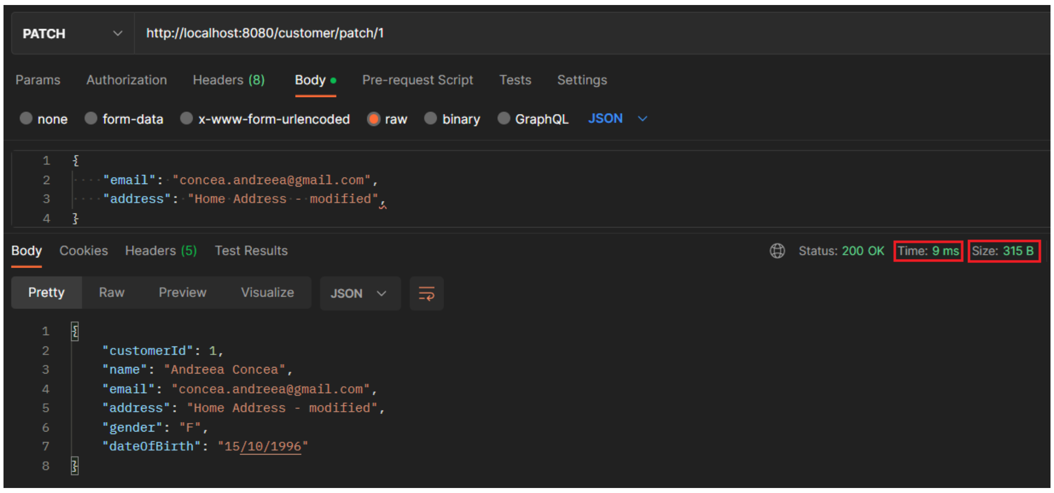
```
DELETE http://localhost:8080/customer/delete/1
```

This request does not have a body

Status: 200 OK Time: 11 ms Size: 165 B

```
{}
```

Fig. 6. DELETE example for REST API



http://localhost:8080/customer/patch/1

```
{
    "email": "concea.andreea@gmail.com",
    "address": "Home Address - modified",
}
```

Status: 200 OK Time: 9 ms Size: 315 B

```
{
    "customerId": 1,
    "name": "Andreea Concea",
    "email": "concea.andreea@gmail.com",
    "address": "Home Address - modified",
    "gender": "F",
    "dateOfBirth": "15/10/1996"
}
```

Fig. 7. PATCH (Update) example for REST API

The average response time of our sample requests was 9.7 milliseconds per call, this low response time being influenced by the usage of an in-memory database, which removes the need to connect to a database server.

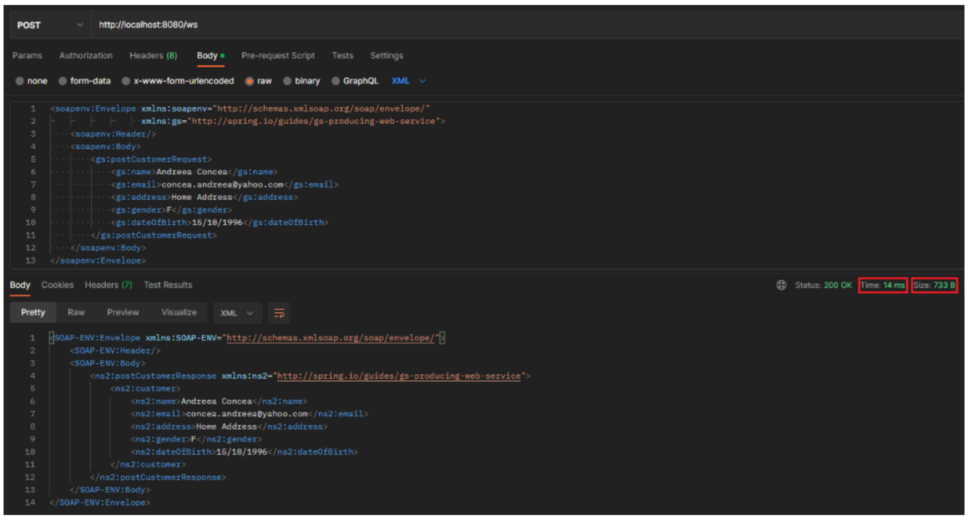
The lowest response size is on DELETE, since this endpoint returns just the status 200 OK for a successful response, with an empty body. The average response size is 273.3 bytes. This result is influenced by the JSON format, which is lightweight and does not use a lot of memory.

For the local implementation we have used HTTP protocol, but on a production-like application, the HTTPS protocol should be used for encrypting the data using SSL or TLS.

All the requests we made were synchronous, and we had 201 Created and 200 OK as response statuses.

4.2. SOAP implementation

The same application was implemented following the SOAP standards to compare the results obtained from both approaches. This approach was less modular, using just two packages: repository, which contains the data initialization logic and the methods used for the database access, and the endpoint package, containing the implementation of the API endpoints business logic for customer and account management. In the figures from below, an illustration of the same CRUD (Create, Read, Update and Delete) operations performed using Postman POST requests is shown:



POST http://localhost:8080/ws

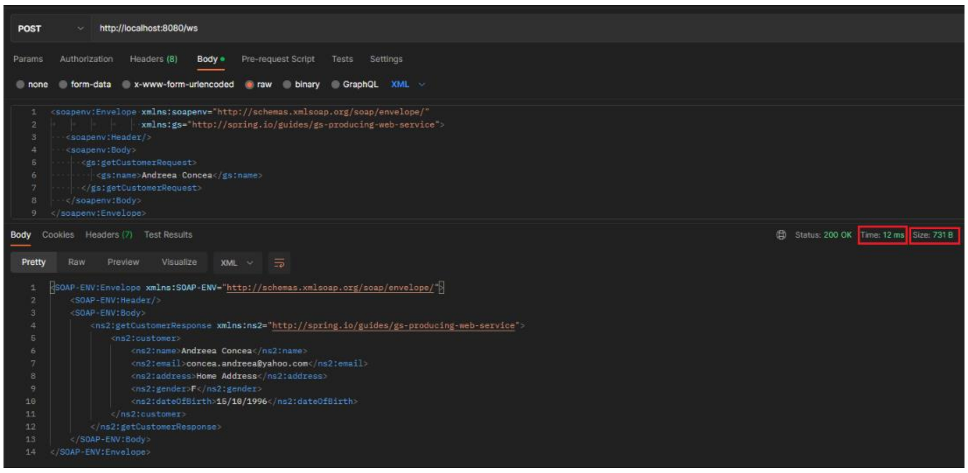
Body: raw XML

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:gs="http://spring.io/guides/gs-producing-web-service">
   <soapenv:Header/>
   <soapenv:Body>
      <gs:postCustomerRequest>
         <gs:name>Andreea Concea</gs:name>
         <gs:email>concea.andreea@yahoo.com</gs:email>
         <gs:address>Home Address</gs:address>
         <gs:gender>F</gs:gender>
         <gs:dateOfBirth>15/10/1996</gs:dateOfBirth>
      </gs:postCustomerRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Status: 200 OK Time: 14 ms Size: 733 B

```xml
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <ns2:postCustomerResponse xmlns:ns2="http://spring.io/guides/gs-producing-web-service">
            <ns2:customer>
                <ns2:name>Andreea Concea</ns2:name>
                <ns2:email>concea.andreea@yahoo.com</ns2:email>
                <ns2:address>Home Address</ns2:address>
                <ns2:gender>F</ns2:gender>
                <ns2:dateOfBirth>15/10/1996</ns2:dateOfBirth>
            </ns2:customer>
        </ns2:postCustomerResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 8. Create example for SOAP API



POST http://localhost:8080/ws

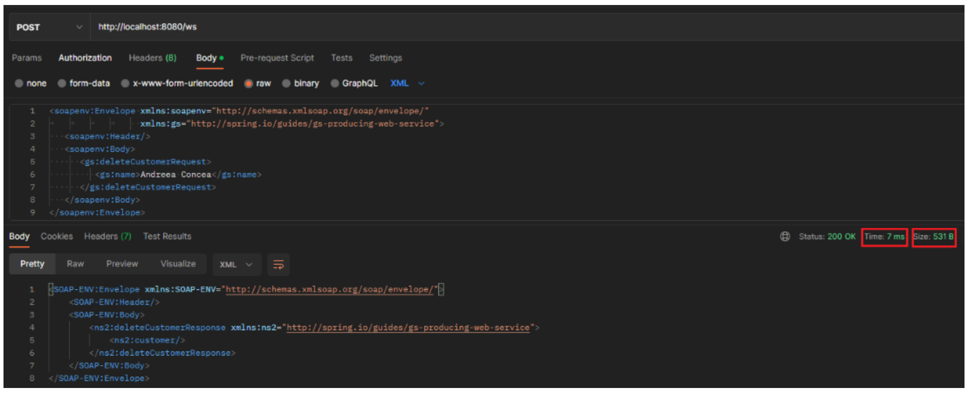
Body: raw XML

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:gs="http://spring.io/guides/gs-producing-web-service">
   <soapenv:Header/>
   <soapenv:Body>
      <gs:getCustomerRequest>
         <gs:name>Andreea Concea</gs:name>
      </gs:getCustomerRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Status: 200 OK Time: 12 ms Size: 731 B

```xml
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <ns2:getCustomerResponse xmlns:ns2="http://spring.io/guides/gs-producing-web-service">
            <ns2:customer>
                <ns2:name>Andreea Concea</ns2:name>
                <ns2:email>concea.andreea@yahoo.com</ns2:email>
                <ns2:address>Home Address</ns2:address>
                <ns2:gender>F</ns2:gender>
                <ns2:dateOfBirth>15/10/1996</ns2:dateOfBirth>
            </ns2:customer>
        </ns2:getCustomerResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 9. Read example for SOAP API



POST http://localhost:8080/ws

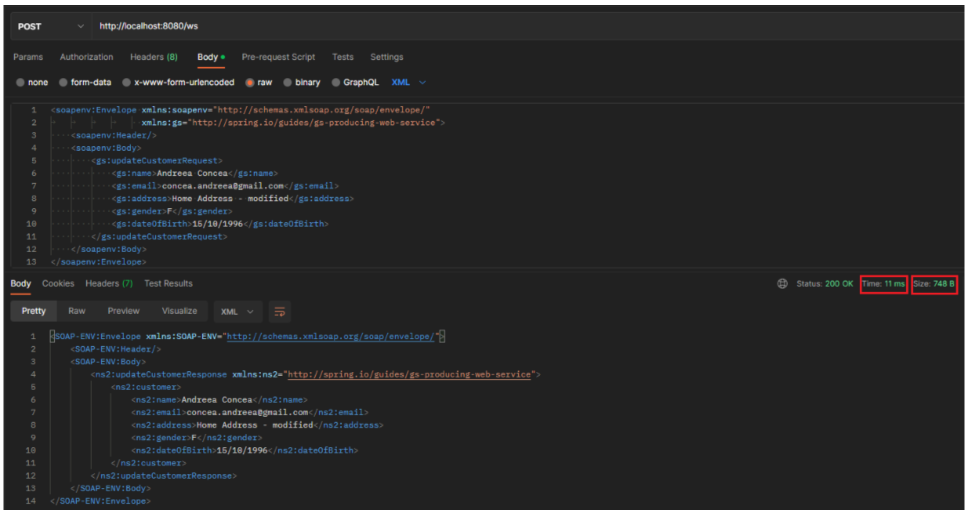
Body: raw XML

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:gs="http://spring.io/guides/gs-producing-web-service">
   <soapenv:Header/>
   <soapenv:Body>
      <gs:deleteCustomerRequest>
         <gs:name>Andreea Concea</gs:name>
      </gs:deleteCustomerRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Status: 200 OK Time: 7 ms Size: 531 B

```xml
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <ns2:deleteCustomerResponse xmlns:ns2="http://spring.io/guides/gs-producing-web-service">
            <ns2:customer/>
        </ns2:deleteCustomerResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 10. Delete example for SOAP API



http://localhost:8080/ws

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:gs="http://spring.io/guides/gs-producing-web-service">
    <soapenv:Header/>
    <soapenv:Body>
        <gs:updateCustomerRequest>
            <gs:name>Andreea Concea</gs:name>
            <gs:email>concea.andreea@gmail.com</gs:email>
            <gs:address>Home Address - modified</gs:address>
            <gs:gender>F</gs:gender>
            <gs:dateOfBirth>15/10/1996</gs:dateOfBirth>
        </gs:updateCustomerRequest>
    </soapenv:Body>
</soapenv:Envelope>
```

Status: 200 OK Time: 11 ms Size: 748 B

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <ns2:updateCustomerResponse xmlns:ns2="http://spring.io/guides/gs-producing-web-service">
            <ns2:customer>
                <ns2:name>Andreea Concea</ns2:name>
                <ns2:email>concea.andreea@gmail.com</ns2:email>
                <ns2:address>Home Address - modified</ns2:address>
                <ns2:gender>F</ns2:gender>
                <ns2:dateOfBirth>15/10/1996</ns2:dateOfBirth>
            </ns2:customer>
        </ns2:updateCustomerResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 11. Update example for SOAP API

The average response time of the sample requests in this case was 11 milliseconds per call, the higher response time being influenced by the larger amount of data sent using the XML format.

The lowest response size was on the delete operation in the SOAP case as well, since this endpoint returns an empty body for a successful deletion. The average response size was 685.7 bytes. This higher result was again influenced by the XML format, which is richer in information.

For the SOAP implementation, the requests we performed were also synchronous, having the HTTP code 200 OK for all the response statuses. The HTTP method POST was used for all the requests, as stated in the guidelines.

5. Results

In our implementation, the REST approach has proven to be more effective in terms of response time and size, mainly due to the lightweight nature of JSON, which consumes less memory. On the other hand, the inferior outcomes observed in the SOAP case can be attributed to the use of XML format, which, although being richer in information, tends to increase response time and size.

Another winning point for the REST implementation comes with the compatibility with multiple HTTP methods, POST, PUT, PATCH and DELETE, each one being allocated to its specific CRUD (Create, Read, Update, Delete) operation. From security perspective, the clear distinction between operations can help in restricting certain sensitive methods such as DELETE or PATCH, which can result in significant risks when used improperly. Additional security mechanisms or authentication requirements can be selectively applied based on the

HTTP method used in a request. In the SOAP case, all operations must be performed using POST requests, which can raise idempotency and access control problems.

From the ease of implementation perspective, REST is again the preferable choice, being a more modern architectural style with widespread support and extensive documentation. This abundance of resource helps in solving the code implementation impediments, as developers can easily access a wide variety of tutorials, examples and community forums. In contrast, SOAP, being an outdated architectural style, presents greater challenges in debugging and solving coding problems due to its complex nature and relatively limited online resources available. Besides that, REST has a better approach in terms of modularity, which facilitates the implementation of loosely coupled components.

Comparing the results obtained in this paperwork with the ones described in the state-of-the-art chapter, the tendency towards REST is shown in all cases in terms of modularity, faster runtime and lower memory usage. Moreover, the ease of implementation factor should also be considered, being easier to maintain and develop new features for an application that is designed with scalability in mind.

6. Conclusions

Our study showcased the main concepts of web services and some of the most popular architectural types, SOAP and REST.

We presented the main concepts regarding SOAP and REST architectural patterns, as well as a comparison between the two of them, which helped us further in our implementation process.

Our main contribution was the parallel implementation of an API, using both standards. The application was a proof of concept which offered basic functionalities for customer and account management, which can be used in the financial domain. We have implemented all CRUD (Create, Read, Update, Delete) operations and we have analyzed and compared the results.

We obtained better results for the REST implementation, having as KPIs (Key Performance Indicators) the following criteria: ease of implementation, runtime and memory efficiency. The average results for REST were the following: a response time of 9.7 milliseconds per call and a response size of 273.3 bytes. In comparison, the results obtained for SOAP were: a response time of 11 milliseconds per call and a response size of 685.7 bytes. In this regard it should be mentioned that REST is using JSON as data format, while SOAP is using XML, which is richer in information. Moreover, a common point is the fact that both architectural styles are using HTTP as underlying protocol, REST using all HTTP methods for the database operations, while SOAP is using only POST. Once again, this observation favors REST over SOAP. Additionally, SOAP, relying solely on the POST method, requires extra configuration in the XML files, further impacting both speed and memory size indicators.

Furthermore, speaking from our own experience, the widespread popularity of REST and the abundance of documentation in this regard facilitated the first part of our implementation, proving to be highly user-friendly, enabling straightforward development and maintenance, while in the SOAP case we faced challenges due to the lack of comprehensive resources, which complicated the implementation process.

As future directions of this paperwork we would also like to analyze SOAP vs REST from security perspective, in order to see which one is better from this point of view, as security is a very important aspect in the SDLC (Software Development Lifecycle) of a project.

R E F E R E N C E S

[1] *J. Tihomirovs, J. Grabis*, "Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics," Information Technology and Management Science, vol. 19, pp. 92–97, December 2016.

[2] *F. Halili, E. Ramadani*, "Web Services: A Comparison of Soap and Rest Services", Modern Applied Science, vol. 12, no. 3, 2018.

[3] *A. Soni, V. Ranga*, "API features individualizing of web services: REST and SOAP", Journal of Innovative Technology and Exploring Engineering, vol. 8, pp. 664-671, July 2019.

[4] *R. Virta*, "Migrating Integration from SOAP to REST", Master of Science in Technology Thesis, University of Turku, Department of Computing, May 2023.

[5] *S. Ahmad, S. Ali, N. Waqar, N. S. Naz, M. H. Mehmood*, "Comparative evaluation of the maintainability of RESTful and SOAP-WSDL web services", pp. 1-9, IEEE, 2023.

[6] *F. Dahri, A. M. Elhanafi, D. Handoko, N. Wulan*, "Implementation of Microservices Architecturein Learning Management System E-Course Using Web Service Method", Sinkron: Jurnal dan Penelitian Teknik Informatika, vol. 7, no. 1, January 2022.

[7] *A.A. Prayogi, M. Niswar, I. Amirullah, M. Rijal*, "Design and Implementation of REST API for Academic Information System" IOP Conference Series Materials Science and Enginering, July 2020.

[8] *L. L. Iacono, H. V. Nguyen, P. L. Gorski*, "On the Need for General REST-Security Framework", Future Internet – MDPI, December 2019.

[9] *K. Jarosław*, "IntelliJ IDEA Essentials", Packt Publishing Ltd, 2014.

[10] *D. Westerveld*, "API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing", Packt Publishing Ltd, 2021.