# Migrating Integration from SOAP to REST

Can the Advantages of Migration Justify the Project?

Master of Science in Technology Thesis
University of Turku
Department of Computing
Software Engineering
May 2023
Roosa Virta

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin Originality Check service.

UNIVERSITY OF TURKU

Department of Computing

Roosa Virta: Migrating Integration from SOAP to REST

Master of Science in Technology Thesis, 78 p.

Software Engineering

May 2023

This thesis investigates the functional and conceptual differences between SOAP-based and RESTful web services and their implications in the context of a real-world migration project. The primary research questions addressed are:

- What are the key functional and conceptual differences between SOAP-based and RESTful web services?
- How can SOAP-based and RESTful service clients be implemented into a general client?
- Can developing a client to work with REST and SOAP be justified based on differences in performance and maintainability?

The thesis begins with a literature review of the core principles and features of SOAP and REST, highlighting their strengths, weaknesses, and suitability for different use cases. A detailed comparison table is provided to summarize the key differences between the two web services.

The thesis presents a case study of a migration project from Lemonsoft's web team, which involved adapting an existing integration to support SOAP-based and RESTful services. The project utilized design patterns and a general client implementation to achieve a unified solution compatible with both protocols.

In terms of performance, the evaluation showed that the general client led to faster execution times and reduced memory usage, enhancing the overall system efficiency. Additionally, improvements in maintainability were achieved by simplifying the codebase, using design patterns and object factories, adopting an interface-driven design, and promoting collaborative code reviews. These enhancements have not only resulted in a better user experience but also minimized future resource demands and maintenance costs.

In conclusion, this thesis provides valuable insights into the functional and conceptual differences between SOAP-based and RESTful web services, the challenges and best practices for implementing a general client, and the justification for resource usage in such a solution based on performance and maintainability improvements.

**Keywords**: SOAP, REST, web service, migration, communication protocol

# Table of Contents


**1 Introduction** .... **1**

**2 Web services** .... **3**

**2.1 Core concepts** .... **4**

2.1.1 Basic operations .... 4

2.1.2 Semantics for transactions .... 5

2.1.3 Exchanging data .... 5

**2.2 Technologies utilized with web services** .... **6**

2.2.1 XML .... 6

2.2.2 WSDL .... 7

2.2.3 UDDI .... 8

2.2.4 HTTP .... 8

2.2.5 JSON .... 10

2.2.6 Cache .... 11

2.2.7 Proxy .... 12

**3 SOAP** .... **13**

**3.1 Core concepts** .... **13**

3.1.1 Message handling .... 13

3.1.2 Remote Procedure Calls .... 14

3.1.3 Electronic Document Interchange .... 14

3.1.4 SOAP message structure .... 15

**3.2 Developing SOAP-based services** .... **15**

3.2.1 Top-down approach .... 16

3.2.2 Bottom-up approach .... 16

3.2.3 Generally .... 17

**3.3 Consuming SOAP-based services** .... **18**

**4 REST** .... **20**

**4.1 Core concepts** .... **21**

4.1.1 Constraints .... 21

4.1.2 Data elements .... 24

4.1.3 Resource methods .... 25

**4.2 Developing RESTful service** .... **25**

**4.3 Consuming RESTful service** ........ **26**

**5 Comparing SOAP-based and RESTful service** ........ **28**

**5.1 Strengths and weaknesses** ........ **28**

5.1.1 SOAP-based services ........ 28

5.1.2 RESTful services strengths and weaknesses ........ 30

**5.2 Comparison of SOAP-based and RESTful Web Services** ........ **32**

**6 Migration project** ........ **35**

**6.1 Background** ........ **35**

6.1.1 WordPress and WooCommerce ........ 35

6.1.2 Lemonsoft ERP ........ 36

6.1.3 Lemonsoft Integration ........ 37

**6.2 Project** ........ **38**

6.2.1 Planning and challenges ........ 38

6.2.2 Design patterns ........ 39

6.2.3 SOLID ........ 46

6.2.4 RESTful client implementation ........ 48

6.2.5 Controlled release ........ 49

**6.3 Retrospective of the project** ........ **49**

6.3.1 Development effort ........ 50

6.3.2 Evaluating the difference in performance ........ 50

6.3.3 Evaluating the difference in maintainability ........ 54

6.3.4 Justification of resource usage of the project ........ 55

**7 Conclusion** ........ **57**

**References** ........ **59**

**Attachments** ........ **62**

## Examples

Example 2.2.1-1 Simple XML document ..... 6
Example 2.2.1-2 Comprehensive example of XML document ..... 7
Example 2.2.4-1 HTTP GET request headers. The request body is empty. ..... 9
Example 2.2.4-2 HTTP GET response headers ..... 9
Example 2.2.4-3 HTTP GET response body. ..... 10
Example 2.2.5-1 Example of JSON message ..... 11
Example 3.1.3-1 SOAP Messages structure visually. ..... 15

## Figures

Figure 2.1.1-1 Operational model of web services ..... 4
Figure 6.2.2-1 Object factory in practise ..... 41
Figure 6.2.2-2 Singleton in practise ..... 42
Figure 6.2.2-3 Chain of Responsibility pattern in practise ..... 44
Figure 6.2.2-4 Builder pattern in practise ..... 46

## Tables

Table 5.1.2-1 Comparing SOAP-based and RESTful services ..... 33
Table 6.3.2-1 Aggregated results for the old SOAP client, 30 runs ..... 51
Table 6.3.2-2 Aggregated results for the new general client (SOAP), 30 runs ..... 52
Table 6.3.2-3 Aggregated results for the new general client (REST), 30 runs ..... 52

## Attachments

Attachment 1. SOAP-based service example ..... 62
Attachment 2. SOAP-based service's example WSDL ..... 63
Attachment 3. SOAP-based service's example unit tests ..... 64
Attachment 4. RESTful service example ..... 65
Attachment 5. RESTful service's example unit tests ..... 65
Attachment 6. RESTful client in Lemonsoft Integration ..... 66
Attachment 7. Performance test ..... 70

# 1 Introduction

Web services have become essential in modern distributed computing, enabling systems to communicate over a network using open standards and protocols such as HTTP. As businesses and organizations become increasingly dependent on software to manage their operations, the ability of these systems to exchange data and functionality becomes essential. Web services provide a standard method for distributed systems to interact with one another, regardless of the underlying technologies or platforms.

Simple Object Access Protocol (SOAP) and Representation State Transfer (REST) are two popular approaches for implementing a communication protocol for web services. [1]

SOAP is a communication protocol that relies heavily on XML for message formatting and transmission. SOAP is typically used for enterprise-level applications that require complex messaging and security features. SOAP messages are often larger and more complex than REST messages, leading to slower performance and higher resource usage. [2], [3]

REST, on the other hand, is an architectural style. Protocols using REST are formally known as RESTful protocols but are commonly referred to as REST protocols. RESTful protocols are lightweight and use standard HTTP methods like GET, POST, PUT, and DELETE to access and manipulate resources. REST is typically used for more straightforward applications that do not require the complexity and overhead of SOAP. REST messages generally are smaller and more efficient than SOAP messages, which can result in faster performance and lower resource usage. [4]

Although SOAP and REST share the goal of facilitating network communication between systems, they have different approaches and use different protocols and standards. For instance, REST is based on more general principles than SOAP, which is more structured and formal. Therefore, this thesis explores general implementations of SOAP-based and RESTful services, focusing on consuming these services and migrating from one to the other. [5], [6]

The thesis showcases a migration project from Lemonsoft's web team that involves adapting an existing integration to a new version of the same service with a different transaction protocol while providing support for older versions. The project utilizes design patterns, refactoring, and implementing a general client to support multiple protocols. Finally, the retrospective of the

project is determined if the migration project justified itself with differences in resource usage and maintainability.

The thesis is structured around three research questions:

**RQ1**: What are the key functional and conceptual differences between SOAP-based and RESTful web services?

**RQ2**: How can SOAP-based and RESTful service clients be implemented into a general client?

**RQ3**: Can developing a client to work with REST and SOAP be justified based on differences in performance and maintainability?

To answer these questions, the thesis begins with a general introduction to web services, followed by separate chapters on SOAP and REST, where the core concepts of each protocol are explained, and their implications for the development and consumption of a web service are discussed. These concepts are then compared, and **RQ1** is answered in **Section 5.2**.

The migration project is then introduced, and subsequent chapters focus on consuming the services and the implications of a change in protocol for integrations. The project is presented with background information, and the planning and upcoming challenges are detailed. These are then discussed separately, and the solutions implemented in the migration project are explained, answering **RQ2 in Section 6.2**.

Finally, the project's retrospective analysis of how the migration project and general client have affected the plugin and whether noticeable differences in performance and maintainability exist. The discussion answers **RQ3 in Section 6.3**, considering the earlier comparisons in the thesis.

The thesis concludes with a discussion of the implications of the findings and potential future research areas.

# 2 Web services

Web services have become an essential component of modern distributed computing, enabling seamless communication between different systems over a network without imposing limitations on technology and programming languages. These services use open standards and protocols like HTTP to ensure a common, platform-independent method of communication, as defined by the World Wide Web Consortium (W3C). [2, p. 6]

The W3C's definition states that a web service is a software system identified by a URI, with public interfaces and bindings defined and described using XML, which can be discovered by other software systems that may interact with the service as prescribed by its definition, using XML-based messages conveyed by internet protocols. [7]

Web services are prevalent in modern web and mobile applications, which rely on them to enable communication between different systems and expose functionality to external clients by running the web service on a remotely reachable server. While web services are typically hosted on servers for more effortless scalability and interoperability, a server environment is not a requirement, as web services were historically run on Local Area Networks (LAN)[3, p. 7-10].

Web services can be implemented using various programming languages and technologies and consumed by clients using multiple technologies. The communication between a web service and a client is typically implemented with HTTP as the transport protocol and XML and JSON as the messaging formats, differing based on the used communication protocol. [2]

The upcoming sections of this focus on the fundamental mechanics of web services, exploring their core concepts and the technologies often utilized with them. The core concepts discussed include the details of basic operations, transaction semantics, and their role in facilitating communication. In terms of often utilized technologies, different messaging formats such as XML and JSON are discussed, along with technologies varying from transport protocol to technologies providing performance and security.

By the conclusion of this chapter, readers will have gained a comprehensive introduction to the core concepts and technologies often utilized with web services.

## 2.1 Core concepts

### 2.1.1 Basic operations

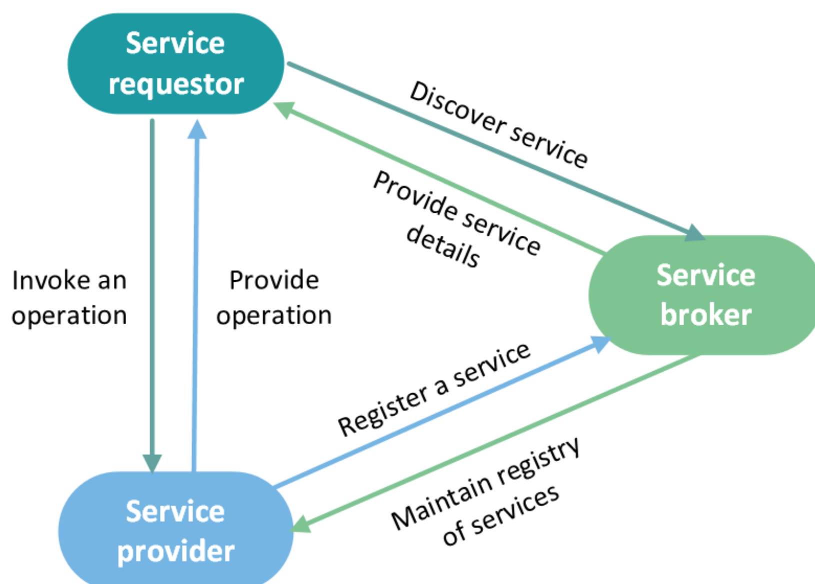



**Figure 2.1.1-1 Operational model of web services**

A simple operational model can represent web service operations, as illustrated in **Figure 2.1.1-1**. The model depicts a three-entity relationship that includes a service provider, service broker, and service requestor, with each entity playing a distinct role in implementing, discovering, and consuming web services. [8, Ch. 2]

Illustrated entities and their relationships are as follows: [8, Ch. 2]

**Service provider**: The service provider's role is to implement and provide access to the operations within the web service. The provider defines the operations and shares them with the service broker.

**Service broker**: The service broker is responsible for registration and web service discovery. The broker catalogues different service types, descriptions, and locations to assist service requesters in locating and consuming the services.

**Service requestor**: The service requestor is accountable for consuming the services. The requester identifies the web service using a service broker and requests the services, invoking the execution of an operation by the service provider.

### 2.1.2 Semantics for transactions

For web service transactions to be successful, all entities must understand the messages and processes involved. If one entity misinterprets a message's format or meaning, the entire transaction could fail or result in incorrect processing. For example, in an e-commerce transaction, the customer's order data needs to be precisely handled by the provider to ensure that the appropriate item is shipped, the right amount is charged, and the order is delivered on time. Any misinterpretation of the data could result in improper processing, such as the wrong item being shipped, the incorrect amount being charged, or the order not being processed. [7]

Web service transactions must be well-defined and standardized to handle data and messages between participating entities properly. This is where Web Service Definition (WSD) comes in, as it specifies the web service's processes in a machine-readable format. [7] The WSD can be found in a WSDL file, which defines the message format the service requestor should use, and where to find the appropriate processes from the service provider. The WSDL file promotes system interoperability by defining the web service's interface and message formatting. The service requestor can then use the information in the WSDL file to generate a client-side code that handles communication between the requestor and the service provider. [9]

### 2.1.3 Exchanging data

XML is the most prevalent format used in standard web service implementations for exchanging data. XML is a widely accepted standard for exchanging information between systems and programming languages. It is a human-readable, flexible, and extensible format, making it suitable for representing complex data structures. [10] [11]

When the service provider receives an XML message across the network, it converts it into a format that the connecting systems can interpret. The design of this conversion depends on which web service the client provides the transaction agent. For instance, software written in PHP could use PHP's SoapClient class to create a client to access the web service. However, the logic for handling the XML messages is not restricted to a single implementation model, meaning that the client for the web service can be created in multiple ways. [12]

The use of XML in web services has several advantages, including its platform-agnostic nature, flexibility in accommodating different data types, and human-readable format. XML also supports a wide range of validation options and is widely supported by web service

technologies. Additionally, JSON has become a popular alternative to XML due to its compactness, ease of parsing, and suitability for JavaScript-based clients. [13]

## 2.2 Technologies utilized with web services

Various communication technologies and protocols are used to enable interoperability in web services. In general, web services have transactions with semantics and functions, which can be found in a WSDL file. Furthermore, these transactions can use XML format for the messages, at least in the case of SOAP. Consequently, it is necessary to employ transaction protocols such as HTTP or HTTPS.

### 2.2.1 XML

XML stands for "The Extensible Markup Language" and is a standard for describing data in a structured, simple, and platform-independent format. It is neither a programming language nor a natural language but a metalanguage that enables the creation of markup languages. In other words, it allows the tagging of data with descriptive names so that both people and software can comprehend the meaning of the data contained between tags. Because it offers a more standard and flexible format to transfer messages with extensible data formats, it has become one of the many technologies often attributed to the success of web services. [7] [10]

Although XML has no predefined tags, its overall structure follows a pattern. While the author determines the tags and the structure of the final document, the tags often come in pairs with an opening and closing tag or combined if the tag does not contain any data.

```
<simple_xml_document/>
```

**Example 2.2.1-1 Simple XML document**

**Example 2.2.1-1** Simple XML document is a minimalistic XML document with only a single tag. As the tag does not contain any data, the tag itself, in addition to being the opening tag, is also the closing tag.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<BookShelf>
  <BookShelfIdentification>1672586745-7286599</BookShelfIdentification>
  <Category>Web Services</Category>
  <!-- optional -->
  <AmountOfBooks/>
  <Book>
    <Name>PHP web services</Name>
    <Author>L. J. Mitchell</Author>
    <ReleaseYear>2016</ReleaseYear>
    <BookMeta>
        <Tags>SOAP, Web Services</Tags>
    </BookMeta>
  </Book>
  <Book>
    <Name>Pro PHP XML and Web Services</Name>
    <Author>Robert Richards</Author>
    <ReleaseYear>2006</ReleaseYear>
    <BookMeta>
        <Tags>PHP, XML, Web Services</Tags>
    </BookMeta>
  </Book>
</BookShelf>
```

**Example 2.2.1-2 Comprehensive example of XML document**

**Example 2.2.1-2** depicts a more comprehensive XML document with multiple levels: The document begins with the information about the file, followed by structured data. In this example, the data depicts a bookshelf with two books and information about them.

### 2.2.2 WSDL

WSDL stands for "Web Service Description Language", which accurately represents the acronym since WSDL files are intended to specify procedures, data types, and parameters expected in the processes of described web service. The WSDL files are typically written with XML. While XML is often in a human-readable format, WSDL is generated and interpreted by machines, as they usually describe the web service and its processes as a whole and in a form to be processed efficiently by engines. [9]

WSDL could provide the before-mentioned semantics for provider and requester entities to pass on to the agents, allowing the interoperability of the web service to begin. The WSDL definitions are compatible with all programming languages, systems, and software, as the web service stays constant, with the agents having to meet the requirements set by the definitions. [7]

WSDL files are not requirements for web service but are commonly regarded as a good practice in development. As the WSDL files are machine-processed, the files provide means for a more accessible web service without an in-depth knowledge of the service. [9]

**Attachment 2** depicts a WSDL file defining a " UserService " service, describing operations for creating and retrieving users. As seen from the example, the file follows XML structure. While it can be read and understood, it is not made for the ease of readability from a human point of view.

Note the structural similarity between the above example and **Example 2.2.1-2**.

### 2.2.3 UDDI

The development of the Universal Description, Discovery, and Integration (UDDI) registries was a collaborative effort by Ariba, IBM, and Microsoft in 2000 to establish standards for accessing, describing, and utilizing web services. These registries were created to manage data on web service providers and their implementations, allowing providers to post and maintain information about their services while consumers can query the data to discover the benefits and determine their use cases. The UDDI specification was developed to standardize web services in the UDDI registry, defining what can be seen as an exemplary registry implementation. [14]

The UDDI registries can be accessed in various ways, including programmatically using Application Programming Interfaces (APIs). However, they are primarily utilized in private environments such as intranets and extranets. The use of UDDI has declined over the years due to the increasing use of other discovery mechanisms, such as search engines and social media platforms. Nonetheless, UDDI remains an essential component of the web services ecosystem, and its standards continue to influence the development of discovery technologies. [14]

### 2.2.4 HTTP

HTTP stands for Hypertext Transfer Protocol and is a widely used protocol for transferring resources between a client and a server, such as HTML pages and photos. It is a stateless protocol, meaning each request must contain all the necessary data to receive a response. [15] However, this also means that the server does not retain data from previous client queries, which may result in performance issues for some applications.

An HTTP transaction consists of a request and response that adhere to the same structure. The request and response include a request/response line, a header, a blank line, and an optional message body, such as a file. The header contains the request or response metadata, such as the content type and encoding. HTTP transactions are typically sent over the TCP/IP protocol suite and utilize ports 80 and 443 for unsecured and secured connections. [16]

Examples of HTTP transactions in this chapter are collected from the requests and responses when opening the University of Turku's webpage.

```
GET /fi HTTP/1.1
Host: www.utu.fi
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:101.0)
Gecko/20100101 Firefox/101.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/we
bp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

**Example 2.2.4-1 HTTP GET request headers. The request body is empty.**

**Example 2.2.4-1** specifies the HTTP method; the most common are GET and POST. In this case of requesting data (a webpage), we use the GET method. After specifying the method, the rest of the requests are headers, which define what we are looking for and what we accept. As the request uses GET, the request body does not need to contain anything.

```
<!DOCTYPE html>
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Connection: keep-alive
Cache-Control: max-age=16588800, public
Date: Sun, 26 Jun 2022 12:52:02 GMT
Content-language: fi
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Content-Security-Policy: upgrade-insecure-requests; default-src https:
data: 'unsafe-inline' 'unsafe-eval'
Content-Encoding: gzip
Age: 2442
```

**Example 2.2.4-2 HTTP GET response headers.**

In **Example 2.2.4-2**, the beginning has the state for the response "HTTP/1.1.-200 OK". This state indicates that the request has succeeded. In a case of an invalid status, the response code would change depending on the reason for the failure. After this, the headers indicate the

content type, encoding and other data the browser and the user could need. **Example 2.2.4-3** contains the queried HTML page.

```
<!DOCTYPE html>
<html lang="fi" dir="ltr">
  <head>
    <meta charset="utf-8" />
      <link rel="canonical" href="https://www.utu.fi/fi" />
      <link rel="shortlink" href="https://www.utu.fi/fi" />
      <meta name="description" content="Turun yliopisto on 25 000
      opiskelijan ja työntekijän aktiivinen akateeminen yhteisö.
      Tutkimme, opetamme ja teemme työtä paremman tul evaisuuden
      puolesta." />
      <meta property="og:title" content="Turun yliopisto" />
```

**...**

**Example 2.2.4-3 HTTP GET response body.**

### 2.2.5 JSON

JSON, or JavaScript Object Notation, is a lightweight data-interchange format designed for human readability and ease of parsing by machines. It is based on a variant of the JavaScript programming language standard, ECMA-262 3rd Edition - December 1999 [17], and is widely used due to its simplicity and flexibility. JSON is independent of any programming language but utilizes widely adopted standards of languages like C++, Java, JavaScript, and Python. This flexibility and compatibility with different programming languages make JSON an ideal data exchange format.

JSON consists of two main structures: an array of values and an object, an unordered collection of name-value pairs. These structures are widely used in programming languages, and their simplicity allows for easy data exchange across different systems. [18]

```json
{
  "BookShelf": {
    "BookShelfIdentification": "1672586745-7286599",
    "Category": "Web Services",
    "AmountOfBooks": null,
    "Books": [
      {
        "Name": "PHP web services",
        "Author": "L. J. Mitchell",
        "ReleaseYear": "2016",
        "BookMeta": {
          "Tags": "SOAP, Web Services"
        }
      },
      {
        "Name": "Pro PHP XML and Web Services",
        "Author": "Robert Richards",
        "ReleaseYear": "2006",
        "BookMeta": {
          "Tags": "PHP, XML, Web Services"
        }
      }
    ]
  }
```

**Example 2.2.5-1 Example of JSON message**

**Example 2.2.5-1** depicts JSON message with multiple levels of nesting, starting with the "BookShelf" key mapping to an object value representing the bookshelf. The subsequent object properties consist of key-value pairs that provide detailed information about the bookshelf and its books.

### 2.2.6 Cache

Caching is a fundamental optimization technique used to store frequently accessed resources or data, thereby reducing network requests and server load. Caching can be implemented at multiple levels of the web service architecture, including client-side with browser cache, server-side with caching frameworks such as Redis or Memcached, and network-level with CDN services. [19]

Client-side caching is a common technique that utilizes the browser cache to store frequently accessed data, reducing the number of requests to the server. On the other hand, server-side caching frameworks like Redis or Memcached can store frequently accessed data in memory, allowing faster retrieval times. [19] [20] Content Delivery Networks (CDNs) can also be used

to cache content at the network level, minimizing the distance between the user and server and reducing latency. [21]

However, caching can pose difficulties. It is essential to ensure that cached data is consistent with the original data source and to address cache expiration and invalidation issues. Appropriate caching mechanisms, such as HTTP caching headers or client-side caching libraries, should be employed to accomplish this. [19] [4]

### 2.2.7 Proxy

Proxy servers act as a middleman between the client and the server. They send requests from the client to the server and send the server's response back to the client. [22]

In web services, proxies can enhance security, performance, and scalability. A proxy can act as a firewall by filtering and blocking unauthorized requests from external clients. Additionally, it can act as a load balancer, distributing incoming requests across multiple servers to balance the load and improve performance. [22]

There are different types of proxies, such as forward and reverse proxies. Clients use forward proxies to access Internet resources, while servers use reverse proxies to deal with incoming client requests. Reverse proxies are often used in web applications to improve performance and scalability by distributing requests across multiple servers and caching frequently accessed resources. [15]

# 3 SOAP

SOAP, an acronym for Simple Object Access Protocol, is a lightweight protocol designed for exchanging structured information in a networked environment where systems are geographically dispersed and not under a single administrative control. [23] It uses HTTP and XML as the underlying transfer protocol and messaging format to transfer structured data across networks in the web service environment.[24, p. 5] With the web services definitions provided via WSDL files, SOAP offers interoperability without confining the interoperability to agents on specific systems.

SOAP provides a basic messaging framework for exchanging data, but extensions and modules can add additional functionality such as security, transactions, and reliability. [3] The SOAP client could be constructed with only the endpoint URL information and the web service port number. While the former is the minimum needed for creating the client, the web service's optional WSDL file is often used as a recommended way of communicating the semantics and definition of the web service between entities. [24, p. 6]

The upcoming sections of this chapter focus on the fundamental concepts of SOAP, offering both theoretical insights and practical perspectives. The theoretical exploration covers the mechanics of SOAP message handling, the dynamics of Remote Procedure Calls (RPC) and Electronic Document Interchange (EDI), as well as the structure of SOAP messages. From a practical standpoint, the chapter focuses on the development and utilization of SOAP-based services.

By the end of this chapter, readers will have gained a comprehensive introduction to SOAP, its core concepts, and practical applications within the realm of web services.

## 3.1 Core concepts

### 3.1.1 Message handling

SOAP messages are handled through Document Requests and Remote Procedure Calls (RPC). In RPC, an enveloped message is sent to invoke an operation, resulting in a response from the process. On the other hand, document requests, formally known as Electronic Document Interchange (EDI), are used to request and store documents rather than handling physical files. [7]

### 3.1.2 Remote Procedure Calls

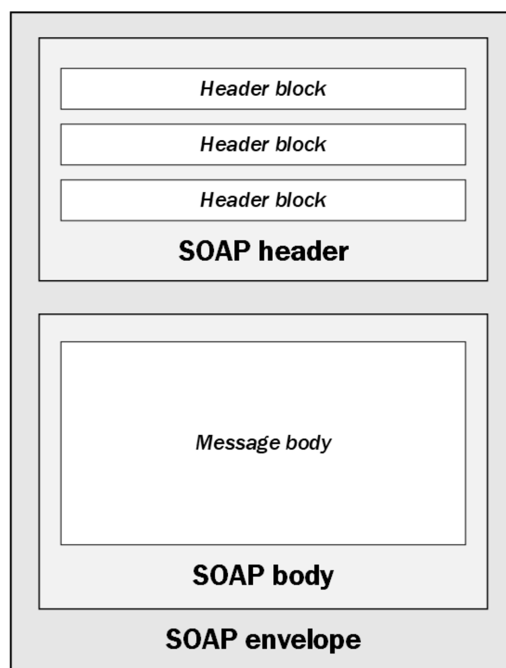
Remote Procedure Calls (RPC) are a method for executing procedures in different network locations. In most cases, RPCs are synchronous, which means that the calling process waits for a response from the server before continuing. [2]

### 3.1.3 Electronic Document Interchange

During data transfers between client and server via Electronic Document Interchange (EDI), the XML document is passed within the body of the SOAP message instead of forwarding it as a parameter. [25] The SOAP message contains the necessary information for the procedure triggered by the XML document. When the request reaches the server, it is processed, and after processing, a response with a new XML document is sent.

The response could contain queried data or status for the operation triggered by request. EDI imposes fewer restrictions than RPC, and XML schema alterations do not affect the service's structure or the agents relying on it. Additionally, EDI is usually processed asynchronously or in a queue, leading to better performance than RPC. [25]

### 3.1.4 SOAP message structure




**Example 3.1.3-1 SOAP Messages structure visually.**

A SOAP message is an XML document structured as shown in **Example 3.1.3-1.** The SOAP message comprises an envelope with an optional header and a mandatory body. The SOAP header encompasses data blocks (e.g., authorization, routing settings) pertinent to handling the SOAP message. The SOAP message sent is enclosed within the SOAP body as the message body. The message body can contain any content written in XML, including the parameters needed for a remote procedure call. [23]

## 3.2 Developing SOAP-based services

SOAP-based services are web services that use SOAP as a communication protocol and XML as a file format for exchanging data. SOAP is easy to implement and integrate into an application because it is well-defined and follows WSDL standards. Many programming languages, including PHP, have libraries that provide ready-made SOAP clients and servers. [16, Ch. 7]

Two major approaches to creating a SOAP-based web service are top-down (contract-first) and bottom-up (code-first). In the top-down approach, a WSDL file is created first to outline the web service, and it is best suited for projects where the web service must be added to an existing

system. The bottom-up approach is best for projects where the web service is the focus and needs to be set up before creating the WSDL.

### 3.2.1 Top-down approach

The top-down approach, contract-first, involves creating a WSDL file to outline the web service by describing the provided operations and the data types used in the mentioned operations. This approach is helpful when creating a web service based on an existing system, as the WSDL can be used to create the skeleton of the service logic. The skeleton can be created manually or with the help of a tool that generates code, which would generate the minimum amount of code the web service would need to perform specific tasks.[26]

Following are the usual steps of the top-down approach:

1. Creating the WSDL: Define the web service's operations and the data types utilized. Then, following the WSDL's rules and conventions, write the file clearly and precisely.
2. Generating the code: After creating the WSDL, it can generate the web service's code skeleton, which can be accomplished manually or with a code-generation tool. The generated code typically includes classes and methods for handling web service operations and necessary data types.
3. Implementing the service logic: The generated code skeleton provides a framework for the web service. The service logic is implemented by writing code to do what needs to be done and adding it to the generated code.
4. Deploying the web service: Upon implementing the service logic, the web service is deployed to be accessed by clients. Deploying includes setting up and configuring a web server to host the web service.

### 3.2.2 Bottom-up approach

The bottom-up approach, also known as the "code-first" approach, is one of the approaches for developing SOAP-based services. This approach involves implementing the web service before creating the WSDL, which gives developers more flexibility in creating the service structure. [27]

Following are the usual steps of the bottom-up approach:

1. Implementing the service: The web service is implemented in the bottom-up approach before creating the WSDL. Implementation includes writing code to do the necessary operations and making them available as web service operations.

2. Generating the WSDL: Optionally, once the web service has been implemented, the code can be used to create a WSDL.

3. Deploying the web service: The bottom-up and top-down approaches do not usually differ.

An example implementation of a SOAP-based service is showcased in **Attachment 1**. This example is of a bottom-up approach to creating a simple calculator service, with operations for summing and extracting two numbers given to it as parameters. The example is written in PHP and uses a NuSOAP library, with the actual service running on a local server built with WAMP.

The implementation starts by setting error reporting and including the NuSOAP library. A new instance of the nusoap_server class is then created, which is used to configure and register the web service's WSDL and methods. At this point, there is no existing WSDL file.

The script for WSDL creation configures it by specifying the web service's name and namespace and setting the schema's target namespace. After the web service's and WSDL's initialization, the script registers two methods, 'add' and 'subtract', which take two integers and return one integer, respectively. The script then defines the methods by creating two functions, 'add' and 'subtract', which perform the respective mathematical operations.

Finally, the script tries to invoke the service by using the request to call the 'service' method on the server object and passing in the input data. This operation creates a WSDL file from the registered data. Generated WSDL can be found in **Attachment 2**.

### 3.2.3 Generally

As mentioned in the previous chapter, two major approaches for creating a SOAP-based web service are top-down (contract-first) and bottom-up (code-first). In addition to choosing the appropriate implementation approach, several other vital considerations exist when developing a SOAP-based service. These include ensuring the WSDL is well-defined, handling data type conversions and marshalling/unmarshalling between the client and the web service, addressing security and testing the web service.

Ensuring a well-defined WSDL involves defining the web service's operations and data types. The top-down approach involves writing the WSDL in the correct format and following the Web Services Description Language conventions. The bottom-up approach consists in generating the WSDL from the code and ensuring that it accurately reflects the operations and data types provided by the web service. In addition, handling data type conversions and marshalling/unmarshalling between the client and the web service is necessary to ensure that the client and the web service can communicate effectively. [28]

Testing and security are optional but highly recommended for all web service developments. In the case of SOAP-based services, testing can be challenging for SOAP-based web services, as there are often multiple systems and data types that need to be integrated and tested. [28]. In developing the security for the service, encryption, authentication, and authorization measures should often be prioritized. [29]

## 3.3 Consuming SOAP-based services

Consuming SOAP-based services can be done using a variety of approaches. The most common methods are a SOAP client library, a proxy class, or raw SOAP messages. Code generation tools for creating a skeleton for requests are also available, but it is important to note that they may not provide an optimal solution. [14, Ch. 20]

A SOAP client library is one of the simplest ways to use a SOAP-based service. These libraries provide an API for interacting with a SOAP service and handle the details of constructing and parsing SOAP messages, allowing the developer to focus on the application logic. Various SOAP client libraries, such as SoapClient for PHP, are available on all platforms. [14, Ch. 18]

Another option for consuming a SOAP-based service is with a proxy class. A proxy class acts as an intermediary between the client and the service. It often provides the same functionality as a client library. This approach is commonly used in languages like Java and C#. [14, Ch. 18]

Consuming SOAP-based services can also be done using raw SOAP messages. The developer constructs the SOAP message manually and sends it to the service using the HTTP protocol. The service responds with a SOAP message, which the developer must parse manually. This approach requires a deep understanding of the SOAP specification and is considered more complex and error-prone than using a SOAP client library or proxy class. [30]

In addition to the methods discussed above, tools are available for generating code for consuming SOAP-based services. These tools take a WSDL file as input and generate code for interacting with the service. This method can be helpful when the only documentation for a service is the WSDL, but the code it creates is often hard to understand and complex.

The creation of a simple SOAP client for unit testing Calculator Service showcased in **Section 3.2** is found in **Attachment 3**. The attachment is written in PHP and uses a NuSOAP library for SOAP client and PHPUnit for unit testing.

The script starts by including dependencies and creating an instance of the `nusoap_client` class. The client is then used to call the 'add' and 'subtract' methods of the web service, located at the specified URL, and the expected results are compared to the actual results using the `assertEquals` method.

The test case is then added to a test suite and run. The output of the test suite is finally printed, showing whether the tests passed or failed.

# 4 REST

REST is a style of architecture created in the early 2000s as an alternative to the more complicated and rigid architectures used before. [31] REST uses the web's existing features and protocols to create a scalable and interoperable web service, which contributes to why RESTful web services have become an essential part of modern web applications. [24]

One of the main reasons REST was created was to fix some of the problems and limitations of earlier web service architectures like SOAP and WSDL, which relied heavily on XML. Moreover, these architectures were often complex and challenging to implement, making them less flexible and scalable. [24]

REST was made to solve these problems by using a lighter, more flexible architecture that relies on standard HTTP methods (like GET, POST, PUT, and DELETE) to access and change resources. RESTful web services use simple, human-readable representations of resources (typically in JSON or XML format), which are easy to work with and understand. [24]

Roy Fielding, one of the primary creators of HTTP, introduced the core concepts of REST in his PhD dissertation, "Architectural Styles and the Design of Network-Based Software Architectures." [31] The REST introduced a new principle of focusing on a system's resources instead of the data and functionalities in developing an interoperability solution in web service. Applying the before-mentioned principle was suggested to be addressed with interaction by HTTP. [32]

The upcoming sections of this chapter focus on the fundamental concepts of REST, examining it from both theoretical and practical perspectives. Theoretical exploration covers the principles of the REST architectural style, the constraints that define it, and the standard HTTP methods used in RESTful services. From a practical standpoint, the chapter discusses the development and usage of RESTful services.

By the end of this chapter, readers will have gained a comprehensive introduction to REST, its core concepts, and practical applications within the realm of web services.

## 4.1 Core concepts

### 4.1.1 Constraints

The REST architecture defines five required constraints and one optional. The constraints defined by Roy Fielding are Client-Server, Stateless, Cache, Uniform Interface, Layered System, and optional Code on Demand. [33, Ch. 5.1]

#### *4.1.1.1 Client-Server*

The client and server separation is essential; as stated by Fielding in his dissertation, the separation of concerns between client and server enables the server to evolve independently of the client and allows multiple clients to access the same resources without any coupling between the server and the clients. [33]

The client-server separation enhances the scalability of web services by reducing the complexity of the server. By delegating the responsibility of processing client requests to the client, the server can focus on providing resources and handling incoming requests. As a result, the server can handle more requests and serve more clients. [4]

Additionally, the client-server separation also allows for improved flexibility and reusability. By making the server's resources available to clients of different types, the server can be used for various purposes, and the same resources can be shared among multiple clients. Furthermore, separating concerns between the client and server enables both components to be developed independently, leading to faster development cycles and easier maintenance. [4] [33]

#### *4.1.1.2 Stateless*

"Statelessness means that every HTTP request happens in complete isolation. When the client makes an HTTP request, it includes all information necessary for the server to fulfil that request." [33, p. 86]

In the Client-Server architecture, the Stateless constraint requires that each request contains all the necessary data for processing, meaning that server resources are not maintained for individual client sessions. This constraint simplifies the architecture by decoupling the server from client-specific data, allowing concurrent processing and enhancing scalability. [33]

The constraint simplifies the development of a RESTful system by enabling server-side functionalities to be performed concurrently. Furthermore, the constraint increases the requests' stability, as the requests' success is not dependent on prior context. [4]

#### *4.1.1.3 Cache*

The final constraint regarding client-server communication is the requirement for responses to be classified as cacheable or non-cacheable. This classification reduces the number of requests the server receives, allowing clients to retain responses for additional context, improving efficiency and the system's performance. [33]

On the other hand, caching might decrease stability since using obsolete data opens the door to errors and unusable data. [19] [4]

#### *4.1.1.4 Uniform Interface*

The uniform interface constraint is implemented through a set of standards for communication between components, known as interface constraints, and it applies to both requests and responses. These standards provide a common language that all components in the system can use to communicate with one another and allow the system to be evolved and scale. [34]

The four key interface constraints of the uniform interface in a REST are: [33, p. 82] [34]

1. Resource identification in requests: Each resource is identified by a unique URL, which allows clients to access and manipulate the resource without accidentally modifying the wrong resource, as the resource can be queried with the unique identifier.

2. Resource representation in requests and responses: Standard data formats such as JSON or XML represent the state of resources in requests and responses. Representing resources allows for components to communicate with one another more efficiently.

3. Self-descriptive messages: Each request and response include metadata describing the message's content, allowing components to process the message without additional context or information.

4. Hypermedia as the engine of application state (HATEOAS): Hypermedia links are included in requests and responses to allow clients to discover and navigate the available resources in the system. Allowing for navigation through hypermedia with relative and

absolute paths enables the system to evolve and change over time without breaking existing clients.

Other interface constraints in a REST system may include using standard data formats such as JSON or XML and requiring all components to be stateless. By adhering to these interface constraints, components in a REST system can interact with one another predictably and consistently. [4]

#### *4.1.1.5 Layered system*

The last required constraint, the layered system, expands on Uniform Interface by restricting the client to only the outermost layer of the system, meaning that the client does not have information if it is communicating with a proxy or the actual server. As a result, a network can have multiple layers of components, with each component only interacting with its adjacent layer. In addition, this constraint separates business logic into different layers, each with a specific purpose, such as security or data validation. [33]

Layering ensures that clients can only access the outermost layer of the system and does not have information about intermediate layers, such as proxies or gateways, which can be added to provide additional functionalities in the system. Proxies can act as intermediaries, relaying requests between clients and servers and are used for load balancing and security checks. On the other hand, Gateways translate requests and responses between different protocols, allowing clients to access services that use protocols other than HTTP. [31] [35, Ch. 5]

#### *4.1.1.6 Code on Demand*

The only optional constraint in REST is Code on Demand, which indicates that the server can extend the client's functionalities at runtime by transferring executable code. This constraint allows for dynamic content or interactive features to be added to a web page without requiring the user to download additional software or updates. However, this constraint poses a security risk as it enables the server to execute code on the client's machine. As a result, some applications utilizing REST may choose to disable or limit code use on-demand to minimize security risks. [33]

A typical example of code on demand is when a web page loads and executes client-side code, such as JavaScript, sent from the server. This code can add new functionality to the page, such

as dynamic content or interactive features, without requiring the user to download any additional software or updates. [31]

### 4.1.2 Data elements

#### *4.1.2.1 Resource*

R. T Fielding defines the resource as follows:
"A resource is anything important enough to be referenced as a thing in itself" [33, p. 81]

R. T Fielding and R. N Taylor add to the definition of the resource followingly:
"Any information that can be named can be a resource: a document or image, a temporal service (e.g., "today's weather in Los Angeles"), a collection of other resources, a moniker for a non-virtual object (e.g., a person), and so on." [36, p. 3]

A resource is mapping a set of entities, not the entity itself. It can be static in that the mapping always corresponds to the same entity or dynamic with the entity or mapping changing—the only aspect of the resource, which must always be static, is the resource's semantics.

Resource definition was designed to be abstract to allow more leeway in RESTful implementations. [36]

#### *4.1.2.2 Resource identifier*

Each resource is described by at least a single URI (Uniform Resource Identifier), the resource's name and address to identify the particular resource uniquely. [36, p. 4]

#### *4.1.2.3 Representation*

For the resource to be usable, URI must allow access or refer to it remotely, as they are not directly interactable. The resource's state is saved to a neutral format, known as resource representation, to make it interactable.

R. T Fielding and R.N Taylor define the representation followingly:

"A representation is a sequence of bytes, plus representation metadata to describe those bytes. Other commonly used but less precise names for a representation include document, file, and HTTP message entity, instance, or variant." [36, p. 4]

#### *4.1.2.4 Self-descriptive messages*

Each request and response need to be understandable without prior context. This independency from context means that the requests are stateless, media types are defined with the requests, and the requests have a standard set of methods.[33, p. 98]

#### *4.1.2.5 Hypermedia as the engine of application state (HATEOAS)*

HATEOAS means allowing dynamic navigation between resources with the use of hypermedia links. The concept is comparable to travelling between web pages in websites using links. [6] [13]

### 4.1.3 Resource methods

Instead of employing a protocol that operates on the HTTP layer for interactions, REST's central concept is interactions between machines using HTTP methods. The following HTTP methods are generally used by RESTful protocols [36, pp. 7–9]:

1. POST, add or create a new resource.
2. GET, retrieve a resource using its URI.
3. PUT, update selected recourse.
4. DELETE and remove selected recourse.

These methods correspond with CRUD operations (Create-Read-Update-Delete).

## 4.2 Developing RESTful service

To develop a RESTful service, a web framework supporting REST principles is often used. For example, Django for Python, Restlet for Java, and Ruby on Rails [4, Ch. 12] make exposing resources to HTTP's uniform interface easy. In addition, these frameworks take care of how HTTP requests and responses are handled so that the developer can focus on the application's logic.

After a framework for the RESTful service has been chosen and the framework has been set up in an environment fulfilling the framework's dependencies, implementation focuses on implementing the services' resources, including creating the necessary classes and methods for

handling the HTTP requests and responses for each resource. In this implementation, the resources and methods used should adhere to the REST principles for the service to be RESTful. For example, a GET request gets a resource and a POST request to make a new resource. [37]

Developing a RESTful service includes testing and security implementations. For example, testing could consist of unit testing, which tests individual components of the service, and integration testing, which tests the entire service. On the security side, the minimum implementation should include implementing proper authentication and authorization mechanisms to protect the service and its resources, considering the potential for XSS and cross-site request forgery attacks, and taking steps to prevent them. [4, Ch. 7] [37]

An example implementation of RESTful service can be seen in **Attachment 4**. The example is in PHP, with the service running on a local server built with WAMP. The example mirrors the previous web service example found in **Section 3.2.**

The content type is set to JSON, and the superglobal variable, $_SERVER, is used to retrieve the HTTP method, path, and request body. Next, the input data in JSON is parsed from the request body, resulting in an associative array containing request data.

The required method is chosen for the data handling by checking the HTTP method and request URI.

If the HTTP method is 'GET' and the request URI contains the path "/add", the input data in array form is run through an operation, which adds the data to a response array.

Subtracting follows the same principle.

If the request does not match the expected conditions, the script sets the HTTP response code to 404 (not found), and the response array is empty.

Finally, the response array is converted to a JSON object using the `json_encode` function and printed, effectively returning the response to the client.

## 4.3 Consuming RESTful service

Consuming RESTful services involves sending an HTTP request to a specific endpoint and processing the response. The request may include parameters, headers, and a body, depending on the nature of the service and the desired action. The response typically has a status code,

headers, and a body containing the data requested. There are many ways to use RESTful services. For example, HTTP client libraries can make requests directly. [4, Ch. 2]

HTTP client libraries are one of the most common methods for consuming RESTful services. As with SOAP libraries, the HTTP client libraries make it easier to send requests and receive responses, often with extra features like authentication, compression and caching that make this easier. [4, Ch. 2]

Another way to consume RESTful services is through a REST client platform. These clients, such as Postman and Insomnia, are typically used for testing and development. In addition, REST API clients provide a graphical way to make requests and see responses. [34]

In addition to these methods, consuming RESTful services directly through programming languages such as Java or Python is possible. This can be done by making HTTP requests using the built-in language libraries and then parsing the responses.

Regardless of the method used for consumption, it is important to note that when consuming RESTful services, proper error handling should be implemented to handle any errors. For example, error handling could include handling HTTP status codes, parsing error responses, and logging any problems. [16, Ch. 8]

An example implementation of a client and unit testing for RESTful service can be seen in **Attachment 5**. The example is written in PHP, with the cURL library for the client and PHPUnit for unit testing.

The test case class defines two test methods, `testAdd` and `testSubtract`, which evaluate the functionality of the web service's add and subtract endpoints.

The `testAdd` method creates a cURL resource, sets the request's URL and options, and then sends a GET request with the parameters 5 and 3 to the add endpoint. It then stores the response in a variable and uses the `assertEquals` method to compare it with the expected result.

Similarly, the `testSubtract` method replicates the test with a different endpoint and expected result.

The test case is then added to a test suite, executed, and the result is printed to indicate whether the test passed.

# 5 Comparing SOAP-based and RESTful service

SOAP and REST have been developed to serve different purposes, as SOAP is a protocol, and REST is an architectural style that could be applied to a protocol. Therefore, when deciding on a protocol to use on a web service, there are many different aspects to consider, as there is no definitive" better protocol" between SOAP and protocol built with REST, which translates into SOAP-based and RESTful services also.

For example, SOAP-based services are better for applications requiring complex data structures or detailed error handling, while RESTful service may be a better choice for applications that need to be flexible and scalable.

The upcoming sections of this chapter focus on providing a comprehensive comparison of SOAP-based and RESTful services, beginning with introducing their respective strengths and weaknesses. This analysis will set the stage for a comparison table to provide an answer for **RQ1**: "What are the key functional and conceptual differences between SOAP-based and RESTful web services?".

By the end of this chapter, readers have been introduced to functional and conceptual differences of SOAP-based and RESTful, providing reasoning for moving from SOAP-based to RESTful web service.

## 5.1 Strengths and weaknesses

### 5.1.1 SOAP-based services

#### *5.1.1.1 Transport independence*

SOAP messages can be sent over any transport protocol, including synchronous and asynchronous protocols such as HTTP, SMTP, BEEP, and Jabber. This transport independence allows SOAP and web services to adapt to the evolving nature of the internet and continue to be used as protocols are replaced or upgraded. This quality is like how a physical letter can be carried by various forms of transportation to reach its destination. In addition, SOAP messages can be sent between any client and server, regardless of the infrastructure between them, as the message itself is independent. [13]

Transport independence translates into interoperability between different platforms and operating systems. This interoperability enables companies to connect to legacy systems and exchange data as SOAP messages, allowing previously incompatible applications to interoperate. In addition, this allows SOAP to be used as middleware between legacy systems, allowing different systems to communicate and access data from each other. [28]

#### *5.1.1.2 Standardization*

SOAP is a web standard that has been widely adopted and is supported by various tools and libraries that make it easy to implement and use.

The standardization of SOAP also ensures that different protocol implementations are interoperable, meaning they can exchange information and work together seamlessly. In addition, standardization enables services provided by other organizations or systems that organizations technologies in a distributed environment.

#### *5.1.1.3 Self-contained messages*

Each SOAP message includes all the information needed to process the request or response, including the data being exchanged, the procedure to perform on the data, and any necessary metadata. [23]

#### *5.1.1.4 Built-in error handling*

SOAP includes several standard fault codes and other mechanisms that can be used to indicate and diagnose errors that occur when processing SOAP messages.

For example, SOAP defines a `Fault` element that can indicate that an error has occurred when processing a request. The `Fault` element includes a `fault code` element that specifies the type of error that occurred and a `faultstring` element that provides a human-readable description of the error. [3], [23]

Using standard fault codes and other mechanisms in SOAP makes it easier to identify and diagnose errors that occur when processing requests or responses. Standard fault codes can be especially useful in a distributed environment, where services may be implemented by different organizations or systems using other technologies.

*5.1.1.5 Complexity*

SOAP is generally regarded as a complex protocol due to its age, verbose messaging, and use of many older technologies, making it hard to learn and begin developing. [28]

*5.1.1.6 Verbose messages*

The verbosity of SOAP messages can be a disadvantage in several ways. One drawback is that it can make messages larger and more difficult to transmit over the network. Since XML uses a lot of tags and other syntaxes to define the structure of the message, the resulting messages can be significant, making them slower to transmit and more challenging to handle. In high-volume or real-time traffic environments, the overhead of transmitting and processing large messages can negatively impact performance.

*5.1.1.7 Heavyweight*

SOAP is considered a heavyweight protocol because it uses a strict XML format for its messages, which makes it more verbose and less efficient than other protocols like JSON. In addition, SOAP messages are typically transmitted using HTTP, which adds even more overhead to the overall message size, making SOAP less well-suited for use in scenarios where bandwidth is limited, or performance is a concern. [13]

### 5.1.2 RESTful services strengths and weaknesses

*5.1.2.1 Simplicity*

RESTful web services are considered simple because they utilize well-known W3C/IETF standards, including HTTP, XML, URI, and MIME, which already have well-established infrastructure and tools for implementation in various programming languages, systems and software. [38] Reducing the planning and development effort needed to create a RESTful service client, as it can be tested using a standard web browser without needing specialized software.

Because the service can be tested using a standard web browser without specialized software, the amount of planning required to begin developing a client for a RESTful service is minimal. In addition, a RESTful web service's development process is like a dynamic website's development. [38]

#### *5.1.2.2 Flexibility*

RESTful services are flexible because they are based on the architecture of the World Wide Web, designed to be flexible and adaptable. The use of HTTP and URI enables RESTful services to interact with other systems and applications quickly, and different media types allow for data representation in various formats. In addition, RESTful services can be efficiently modified by adding new resources, operations, and functionality as needed. Because of this, RESTful services are easy to change to meet changing business needs and requirements.

#### *5.1.2.3 Scalability*

The idea of client and server separation is one of REST's key concepts, and as such, RESTful web services keep them separate, allowing for leveraging HTTP and the infrastructure of the web, including caching, load balancing, and content delivery networks, which are designed to support high levels of traffic and data transfer. [38]

RESTful web services use a stateless communication model, meaning each request is independent and does not rely on any shared state of the client and the server. Allowing for horizontal scaling, where additional resources can be easily added to the system to support increasing traffic without worrying about maintaining a state across multiple servers. [24], [39]

On a technical level, RESTful web services can be scaled to serve many clients, as REST has built-in caching, clustering, and load-balancing support. Also, messaging does not need optimization, as it is often very lightweight. The lightweight nature of messaging is due to the formats commonly used as messaging forms (such as JSON or plain text), which have special needs for system resources. [38]

#### *5.1.2.4 Lack of standardisation*

REST is an architectural style rather than a standard or specification, so it lacks standardization. As a result, no strict rules must be followed when implementing a RESTful web service, which can lead to flexibility and creativity in designing these services.

While having flexibility and the possibility for creative implementations can be generally considered a good thing, this lack of standardization can also be a weakness of REST. [13] It can lead to confusion and misunderstandings between different REST implementations, making

ensuring interoperability between various web services complex, which can be a problem when dealing with complex applications requiring multiple web services to work together. [38]

#### *5.1.2.5 Limited security*

RESTful web services do not provide built-in security mechanisms, so security measures must be implemented at the application level. Meaning that developers must ensure that proper authentication and authorization measures are in place to protect their RESTful web services from unauthorized access. [13]

#### *5.1.2.6 Limited support for transfer protocols*

RESTful services only support the use of HTTP as the transfer protocol, which can be a problem for some applications because HTTP is a synchronous protocol, meaning the client must wait for the response from the server before it can continue processing. Additionally, it can cause issues with performance and scalability, especially in applications requiring high concurrency levels or real-time data processing.

## 5.2 Comparison of SOAP-based and RESTful Web Services

For answering the **RQ1**: "What are the key functional and conceptual differences between SOAP-based and RESTful web services?" a comparison table, **Table 5.1.2-1**, is used. The following comparison table summarises the key functional and conceptual differences between SOAP-based and RESTful web services identified in the preceding strengths and weaknesses chapters. In addition, it provides a side-by-side comparison of essential features of the two types of web services, including the protocol, data format, standardization, messaging, transport independence, security, performance, scalability, flexibility, ease of use, and other relevant factors.

The table is intended to illustrate the trade-offs and advantages of each approach with a high-level comparison, as actual implementation and usage of each type of web service may vary depending on the specific context and use case.

**Table 5.1.2-1 Comparing SOAP-based and RESTful services**

| | ***SOAP-based service*** | ***RESTful service*** |
| --- | --- | --- |
| *Protocol* | Uses SOAP protocol for exchanging messages. | Uses HTTP protocol, primarily GET, POST, PUT, and DELETE methods. |
| *Data Format* | Uses XML for data exchange. | Supports multiple formats such as XML, JSON, and plain text. |
| *Standardization* | SOAP is a widely adopted web standard supported by various tools and libraries that ensure interoperability. | REST is an architectural style that leverages well-known W3C/IETF standards, including HTTP, XML, URI, and MIME. |
| *Messaging* | Uses a messaging protocol with an envelope, headers, and body for communication, which can result in complex message structures. | Uses a stateless request-response model, where each request is independent and can be sent with simple, lightweight headers. |
| *Transport Independence* | SOAP messages can be sent over any transport protocol, including synchronous and asynchronous protocols such as HTTP, SMTP, BEEP, and Jabber, allowing SOAP and web services to adapt to the evolving nature of the internet. | RESTful services only support the use of HTTP as the transfer protocol. |
| *Self-contained messages* | Each SOAP message includes all the information needed to process the request or response, including the data being exchanged, the procedure to perform on the data, and any necessary metadata. | RESTful services also include all the information needed in each request, but they are simpler and more lightweight than SOAP messages. |
| *Built-in error handling* | SOAP includes several standard fault codes and other mechanisms that can be used to indicate and diagnose errors that occur when processing SOAP messages. | RESTful web services require error handling to be implemented at the application level. |

| | | |
|---|---|---|
| *Security* | Offers built-in security features, including WS-Security, for encryption and authentication. | Requires security measures to be implemented at the application level. |
| *Performance* | Can be slower due to the overhead of the XML format and messaging protocol. | Typically, faster due to the use of lightweight formats and simpler architecture. |
| *Scalability* | Can be scaled to serve many clients but may require additional effort due to their complexity. | Designed to be scalable and adaptable, allowing for horizontal scaling and leveraging HTTP infrastructure. |
| *Flexibility* | Not as flexible as RESTful web services, as they require the entire message to be sent each time, even if only a small amount of data is needed. | More flexible as clients can request only the data they need and can be easily adapted to new requirements. |
| *Ease of Use* | Can be more complex due to the SOAP protocol and messaging structure, which requires additional knowledge and tools. | More straightforward to use, as they rely on standard HTTP methods, which are familiar to most developers, and have a simpler architecture. |
| *Complexity* | Generally regarded as a complex protocol due to its age, verbose messaging, and use of many older technologies, making it hard to learn and begin developing. | Considered more straightforward because they use well-known standards with well-established infrastructure and implementation tools. |
| *Verbose messages* | The verbosity of SOAP messages can make messages larger and more difficult to transmit over the network. | Simpler and more lightweight than SOAP messages, making them easier to transmit over the network. |
| *Heavyweight* | Considered a heavyweight protocol due to its strict XML format for its messages, making it more verbose and less efficient than other protocols like JSON. | Considered lightweight and efficient due to their use of simple formats and architecture. |

# 6 Migration project

To further build upon the preceding chapters, this chapter aims to answer the second research question, "How can SOAP-based and RESTful service clients be implemented into a general client?" by discussing the Lemonsoft webshop team's migration project. The project involved migrating from a SOAP-based client to a more modern RESTful client due to Lemonsoft ERP's decision only to provide security updates for its SOAP-based services. The migration required a complete overhaul of the existing Lemonsoft Integration WordPress plugin’s core files.

To address these issues, design patterns were employed, and an object factory was used to extract the API from the base plugin, creating a single-entry point for all requests and responses to the web service. This approach allowed for creation of necessary objects for each service based on the requested protocol, equipped with functionalities specified in interfaces. Doing so established a general client for each service with functions for responses and requests communicated with REST and SOAP protocols.

The result eliminated the need for web service client logic in other plugin areas, making it possible to use the same client code for different protocols. This approach offers a solution to the challenge of implementing SOAP-based and RESTful service clients into a general client.

The upcoming sections of this chapter focus on the migration project, by first providing background on the project, leading up to the project itself and the solutions employed within it and concluding with retrospective of the project. Answer for **RQ2**: “How can SOAP-based and RESTful service clients be implemented into a general client?” is discussed in **Section 6.2**, while answer for **RQ3**: “Can developing a client to work with REST and SOAP be justified based on differences in performance and maintainability?” is discussed in **Section 6.3**.

By the end of this chapter, readers have gained the general idea of the client developed in the migration project and understand the justification of the project.

## 6.1 Background

### 6.1.1 WordPress and WooCommerce

WordPress is a popular Content Management System (CMS) for creating and managing websites. It is based on PHP and uses a MySQL database, focusing on allowing for

modifications and extensibility. This is achieved through the use of themes and plugins, with themes allowing users to customize the appearance of their website while plugins extend the functionality of the CMS.[40]

WooCommerce is a plugin for WordPress that adds a suite of powerful tools for managing online stores, allowing the CMS to function as an e-commerce platform. These tools include product management, payment processing, calculating shipping and tax costs, and inventory management. With WooCommerce, users can create and manage online stores that sell physical and digital and subscription-based products and services.[41]

One of the key aspects of WordPress and WooCommerce is their extensibility, which allows for adding new features and connections to other services via plugins. This includes SOAP-based or RESTful services that import data from external systems, such as ERPs, into the CMS. This integration can automate workflows and improve inventory management, among other benefits. [40], [41]

### 6.1.2 Lemonsoft ERP

Lemonsoft is a Finnish SaaS company specializing in designing, developing, and selling Enterprise Resource Planning (ERP) software solutions, primarily catering to small and medium-sized enterprises. In this passage, we discuss their products, Lemonsoft ERP and Lemonsoft Integration plugin, and the recent transition from a SOAP-based web service to a RESTful service interface. [42]

A modern ERP system, like Lemonsoft's, streamlines and automates various day-to-day business activities, leading to tangible user benefits. By consolidating critical data in a centralized database, Lemonsoft ERP allows companies to track business operations in real-time, work more efficiently across departments, and make informed decisions. [43]

Lemonsoft ERP features a modular structure, improving scalability and enabling seamless integration of various business functions, such as inventory management, finance and accounting, human resources, customer relationship management, supply chain management, and more. In addition, the system is accessible online and maintained by a cloud service provider, ensuring that it is always up to date-and running smoothly. [43]

Recently, Lemonsoft has updated its service interface, moving away from the SOAP-based web service and adopting a new RESTful service interface. The changing needs of the industry

prompted the shift, as RESTful interfaces offer more flexibility and are considered more lightweight than SOAP-based services. As a result, the RESTful interface is now the primary service interface for Lemonsoft ERP, and no new clients are set up with the SOAP-based interface.

This transition to a RESTful service interface has several advantages, such as improved performance, easier integration with other web services, and more straightforward development and maintenance—however, the change presents challenges for existing integrations.

### 6.1.3 Lemonsoft Integration

The Lemonsoft Integration plugin is a versatile tool designed to extend the capabilities of WooCommerce and WordPress platforms, allowing integration with the Lemonsoft ERP system. By facilitating the transfer of product information, customer data, and orders from the web store to the ERP system, the plugin streamlines business operations and enhances the user experience on the website.

While the Lemonsoft Integration plugin was initially intended for Business-to-Customers (B2C) use, it was extended for Business-to-Business (B2B) service with the Lemonsoft B2B plugin.

One key feature of the Lemonsoft Integration plugin is its queue-based system for sending data to Lemonsoft. For example, this system allows new orders to be placed on the web store, added to a queue, and sent to Lemonsoft when it is their turn. Furthermore, this queue can be customized to prioritize specific data transfers when synchronizing the web store and ERP system.

The Lemonsoft Integration plugin is written in PHP, utilizes the typical plugin architecture in WordPress, and leverages the SoapClient-library to connect to the Lemonsoft ERP system. In addition to integration, the plugin offers several tools that enhance web store functionality, improve user experience, and add advanced features.

In response to the recent transition from SOAP-based web service to the RESTful service interface in Lemonsoft ERP, the development team is working on a RESTful client for the Lemonsoft Integration plugin.

## 6.2 Project

As part of the migration project, the existing Lemonsoft Integration plugin had to be changed to work with a new version of the same service that used a different transaction protocol. A SOAP-based client had to be changed to a RESTful client. This was required because Lemonsoft ERP released a more modern version of its RESTful services, eventually replacing the existing SOAP-based one.

As all integration plugin clients use SOAP-based services, a new RESTful client is built as part of the migration project to work seamlessly alongside the existing SOAP-based client. The aim is to create a general client to connect to each service without having logic for protocols or message formatting outside the scope of web service client logic in other plugin areas. This is hypothesized to speed up the integration process and improve the plugin's overall performance.

Besides the reasoning of necessity due to the upcoming deprecation of Lemonsoft ERP's SOAP-based service, there were several other reasons why it was decided to migrate to a RESTful client. Firstly, RESTful services are considered more lightweight than SOAP-based services and offer more flexibility in terms of API design. In addition, they provide a better performance, easier integration with other web services, and more straightforward development and maintenance.

### 6.2.1 Planning and challenges

The planning process for the migration project encountered several challenges due to the initial complexity of the plugin's logic and the scattered nature of the client logic. It was quickly realized that the original plan to replace only the client and map responses as needed was not feasible as it would require duplicating the logic everywhere to handle SOAP and REST separately. To overcome this challenge, the team spent a few hours brainstorming possible solutions and demonstrating their planned implementations.

The solution that emerged was to create a general client and refactor the plugin's core to separate the client logic from other functions. For example, this implementation would have only one entry point for requesting a service, with requests and responses sent and formatted as general objects. Interfaces were utilized to ensure that SOAP and REST had the same logic, and different design patterns and refactoring methods were used to build the general client. This

approach was expected to speed up the integration process and improve the plugin's performance.

Another challenge was encountered at the beginning of the migration project when it became apparent that the plugin's core needed to be refactored to separate the client logic from other functions. As the logic was scattered throughout the plugin, this would require changes to multiple files. All the new files were initially created in a refactored-core folder to ensure a smooth transition, with their namespaces having `Refactored/` prepended to them. Old namespaces were then prepended with `Refactored/`, allowing parts of the core to be slowly introduced to existing logic. Once the project is fully ready, the old files are deleted, the `Refactored/` prepend is removed, and the file structure is returned to normal.

Moreover, while refactoring the files to use the new logic, the team noticed that some methods handling the responses were extremely bloated and needed to be divided for better results. They addressed this issue by introducing a builder object to the refactored core, resulting in more concise and readable code.

To ensure the response format was more general, the team followed SOLID's Interface Segregation Principle (ISP) to the letter. As an example of `ProductObjects`, the ISP states that clients should not be forced to depend on interfaces they do not use. By separating the typical behaviour of `ProductObjectREST` and `ProductObjectSOAP` into a `ProductObjectInterface`, the code adheres to the ISP by allowing clients to depend only on the necessary interface than having to rely on the concrete implementation of `ProductObjectREST` or `ProductObjectSOAP`.

### 6.2.2 Design patterns

The Lemonsoft Integration project utilized creational and behavioural design patterns to create a client that works with SOAP-based and RESTful services. This chapter introduces and explains the patterns that were used in the project.

Design patterns are essential in software development as they provide solutions to common problems that have been solved in the past, enabling developers to create custom solutions that meet their program's needs. Instead of focusing on specific pieces of code, patterns provide broad ideas that can be used repeatedly.[44] [45, Ch. Introduction]

Software design has two main categories of design patterns: creational and behavioural. Creational patterns are used to instantiate objects or classes, while behavioural patterns are used to manage object interactions and responsibilities. [44]

It is essential to differentiate between patterns and algorithms. Algorithms provide instructions to accomplish a specific goal, while patterns provide high-level descriptions of solutions that may vary across different programs. [45, Ch. Introduction]

#### *6.2.2.1 Object Factory Pattern*

The Object Factory Pattern is a creational design pattern that provides a way to create objects in a superclass while allowing subclasses to change the type of objects made. Instead of creating objects directly with a constructor, the pattern uses a factory method that creates the object and sends it back to the client code.[44] [46, Ch. 4]

One of the primary advantages of this pattern is that it allows the creation of objects without exposing how they were made to the client code, which improves the modularity of the client code and makes it easier to maintain. Furthermore, the factory method can centralize object creation logic, ensuring that objects are created consistently.[46, Ch. 4]

The Object Factory Pattern is utilized as the entry point to the general client implementation of SOAP and REST. For instance, when a scheduled stock run needs to request information from Lemonsoft ERP, the job's handler method calls the `WebServiceFactory` class to get a service. The factory queries data from the database to initialize a client with the correct information and then determines the proper protocol based on that information.

With the protocol information, the factory class initializes the RESTful service class to the correct endpoint, which initializes the client and provides the interaction logic. The factory can build objects for SOAP and REST protocols and create multiple objects depending on the service endpoint (e.g., product, pricelist, customer). All these endpoints are found as classes using SOAP or REST as a protocol, and they implement interfaces to ensure a common logic.

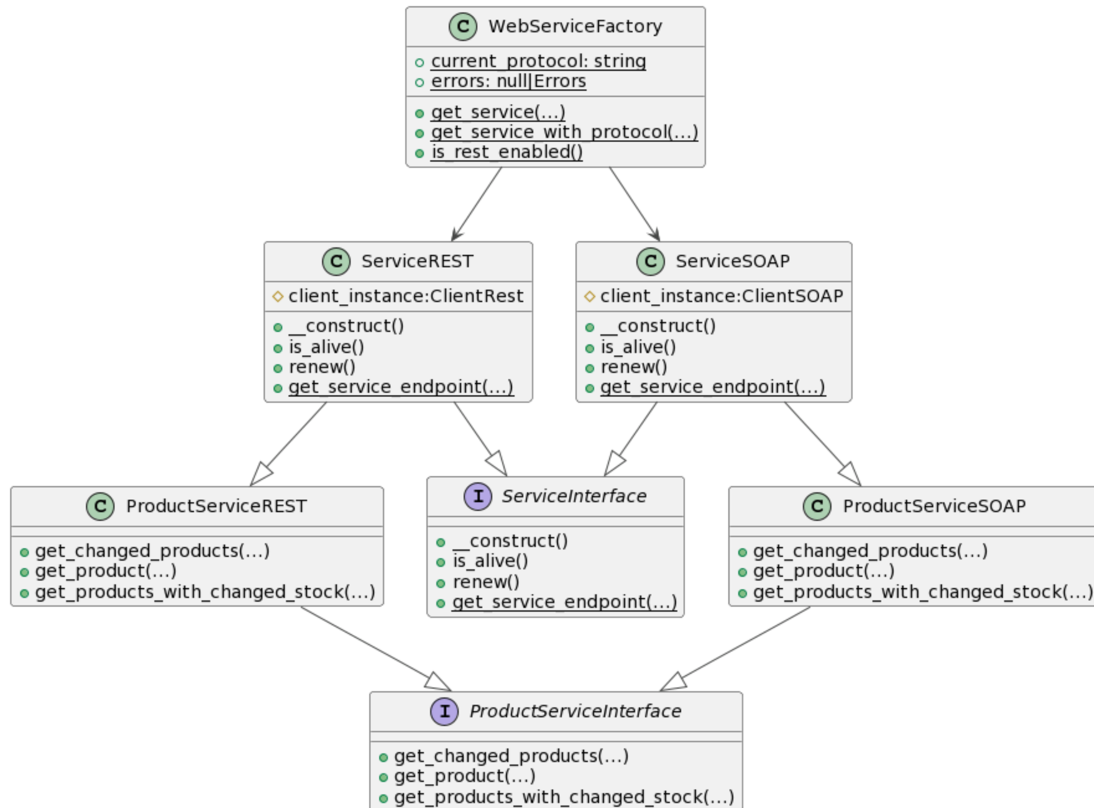



**Figure 6.2.2-1 Object factory in practise**

**Figure 6.2.2-1** illustrates a factory pattern implementation for Lemonsoft ERP web services. The `WebServiceFactory` class is the entry point for the pattern, providing methods for accessing services. It determines the correct protocol for a requested service endpoint and initializes a REST or SOAP service based on it.

The `ServiceInterface` is an interface class defining the common structure of the `ServiceREST` and `ServiceSOAP` classes. These classes allow communication with the ERP system through the specified protocol. Similarly, the `ProductServiceInterface` is an interface class that defines the common structure of the `ProductServiceREST` and `ProductServiceSOAP` classes. These classes provide functionality for accessing specific product-related information from the ERP system.

The diagram illustrates the relationships between these classes, including inheritance relationships between the `ServiceREST` and `ServiceSOAP` classes and the `ProductServiceREST` and `ProductServiceSOAP` classes with their respective interfaces.

*6.2.2.2 Singleton Pattern*

The Singleton pattern is a creational design pattern that ensures a class has only one instance and offers a global point of access to that instance. This pattern is beneficial when there is a need to manage the number of class instances that can be created and to ensure that there is only one class instance within the entire application. By restricting the number of class instances to one, the Singleton pattern helps ensure that resources are used efficiently, and code is simplified. [44] [46, Ch. 4]

The Singleton pattern is utilized in the `ClientSOAP` and `ClientREST` clients. This approach ensures that configuration for the client is not queried each time a new request is made, which would be a significant resource usage. For example, without Singleton, if we were to get multiple different products and a customer for each of them, we would need to create a new client each time we request, which would consume more resources and increase complexity.

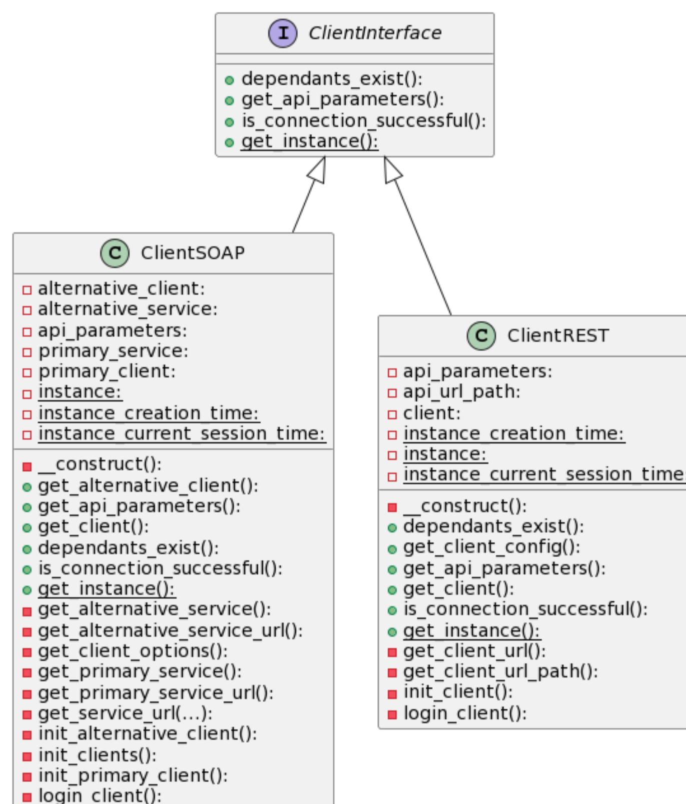



**Figure 6.2.2-2 Singleton in practise**

**Figure 6.2.2-2** illustrates the implementation of the Singleton Pattern in the `ClientSOAP` and `ClientREST` classes. Both classes implement `ClientInterface`, which defines the common structure and behaviour of the two classes. In addition, the Singleton pattern ensures that only

one instance of the `ClientSOAP` and `ClientREST` classes is created and can be accessed globally through the public static method `get_instance()`.

The `ClientSOAP` class has additional private variables and methods related to SOAP-based services, while the `ClientREST` class has variables and methods related to RESTful services. Both classes have methods to initialize the client, log in to the service, and check the status of the connection and any dependencies needed to create the client.

#### *6.2.2.3 Chain of Responsibility*

The Chain of Responsibility is a behaviour-based design pattern that allows the creation of a chain of objects to handle requests. Each object in the chain can handle the request or pass it on to the next object. [44] Instead of embedding the handling logic in each object, a chain of objects handles the logic by incrementally adding it. [46, Ch. 4]

One of the main advantages of the Chain of Responsibility pattern is that it allows adding or removing handlers efficiently without changing the client's code. Furthermore, this pattern promotes loose coupling, making maintaining and expanding the codebase easier.

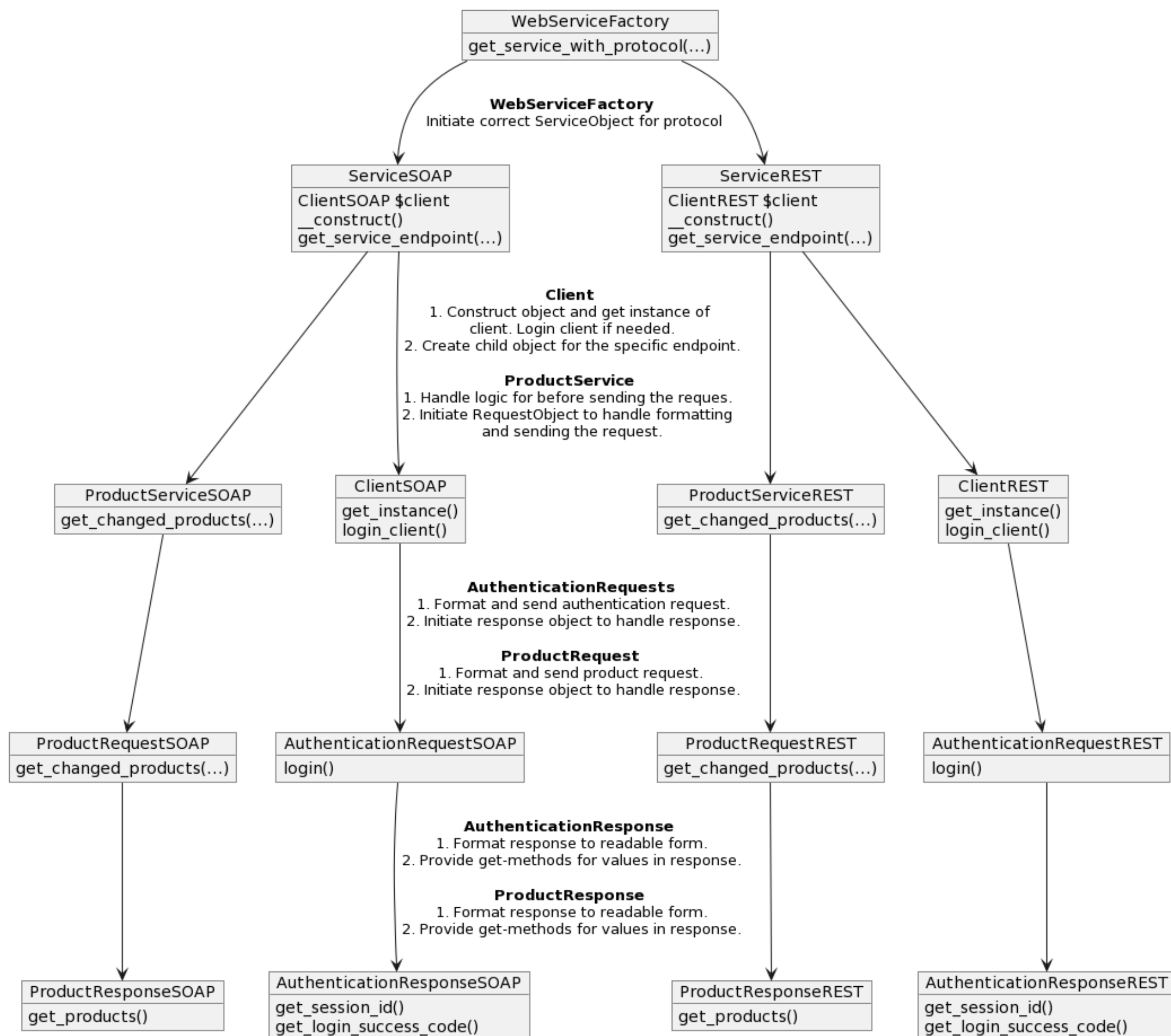



**Figure 6.2.2-3 Chain of Responsibility pattern in practise**

**Figure 6.2.2-3** illustrates the Chain of Responsibility pattern's implementation in the project. The responsibility is passed through the chain of objects for authentication requests and product requests, beginning at the `WebServiceFactory` and ending at the service responses.

The `WebServiceFactory` acts as the entry point and determines the correct protocol for a requested service, creating a `ServiceSOAP` or a `ServiceREST` object accordingly.

The `ServiceSOAP` and `ServiceREST` objects have associated client objects responsible for constructing the object and getting its instance. If required, they also log in to the client. In addition, each service object has child objects for specific endpoints, such as `ProductServiceSOAP` and `ProductServiceREST`, which handle the request logic before sending it.

The `ClientSOAP` and `ClientREST` objects are singletons that ensure only one instance of the object is created, and the configuration for the client does not need to be queried each time a new request is made.

The `AuthenticationRequestSOAP` and `AuthenticationRequestREST` objects are responsible for formatting and sending authentication requests to the corresponding endpoints. The `AuthenticationResponseSOAP` and `AuthenticationResponseREST` objects handle the response from the authentication requests, formatting it to a readable form and providing get-methods for the response's values.

Similarly, the `ProductRequestSOAP` and `ProductRequestREST` objects handle the formatting and sending of product requests to the corresponding endpoints. Finally, the `ProductResponseSOAP` and `ProductResponseREST` objects take the response from the product requests, formatting it to a readable form and providing get-methods for the values in the response.

#### *6.2.2.4 Builder*

The Builder Pattern is a creational design pattern that provides a flexible approach to creating objects with various configurations or properties, eliminating the need for multiple constructors or factory methods with different parameters. It separates the object creation process from its final presentation and allows for the step-by-step building of complex objects. A separate builder class is created to manage object creation and configuration to implement this pattern. [44][46, Ch. 4]

In the context of the project, after mapping the web service responses to objects with similar structure, it became necessary to change the logic in the scheduled job logics, which had "handler methods" that modified the SOAP response data according to a set of rules and had much additional logic. Due to the growth in these methods, they needed to be divided into smaller, more maintainable methods. However, since the methods relied on different values, using global values was not an option.

A builder class was implemented and initiated with the necessary arguments to solve this issue and then constructed step by step using smaller, more maintainable methods. In addition, the builder class allows for easier expansion and reusability of the code, and public methods can be added to allow additional data from paged runs while reusing existing methods. [46, Ch. 4]

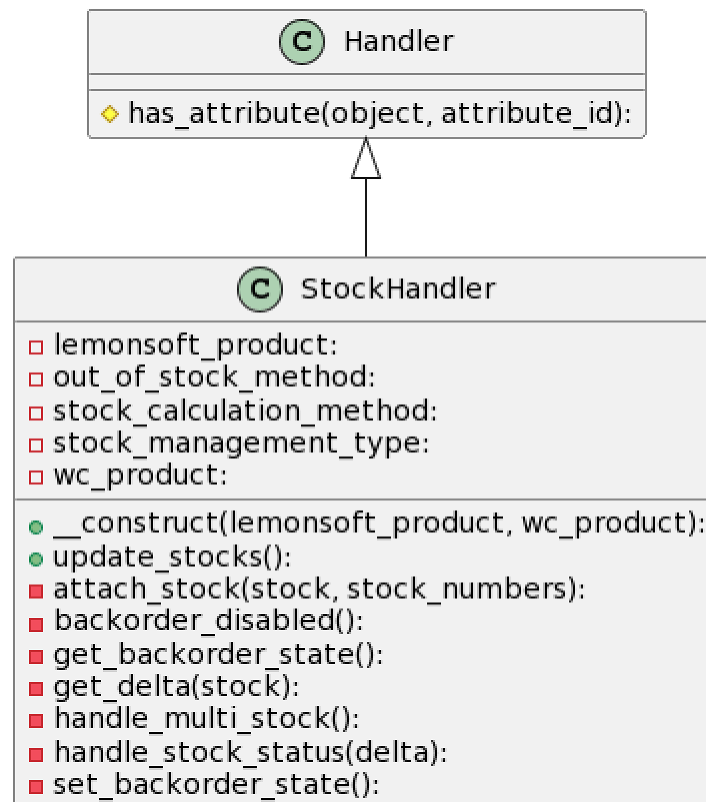



**Figure 6.2.2-4 Builder pattern in practise**

**Figure 6.2.2-4** illustrates the implementation of the builder pattern in the form of a `StockHandler` class that extends the `Handler` class. It contains several private variables and methods to handle the logic related to stock management, including updating the stock values of a product based on the data received from the web service. In addition, the `StockHandler` class implements various methods to calculate and manage the stock of a product based on the current stock values and the values received from the web service. This class aims to provide a flexible approach to handling the logic of stock management in the project.

### 6.2.3 SOLID

The SOLID principles are five design principles for object-oriented programming that aim to make the software more maintainable, flexible, and easy to extend. [47] The principles include:

Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should have only one responsibility or job to do. This principle helps to avoid coupling between different parts of the codebase. [47]

Open-Closed Principle (OCP): A class should be open for extension but closed for modification. This principle promotes using inheritance and composition to add new features to the codebase without modifying existing code. [48]

Liskov Substitution Principle (LSP): Subtypes should be substitutable for their base types. This principle ensures that objects of a derived class can be used in place of objects of the base class without affecting the correctness of the program. [48]

Interface Segregation Principle (ISP): A client should not be forced to depend on methods it does not use. This principle advocates using smaller, more focused interfaces instead of large, general-purpose ones. [47]

Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Instead, details should depend on abstractions. This principle promotes using interfaces and abstract classes to decouple code and reduce dependencies. [47] [48]

**Figure 6.2.2-3** demonstrates an instance of the SRP, OCP, and DIP in the Chain of Responsibility pattern. All objects in the figure have only one responsibility, and the architecture is open for extension and closed for modification. Additionally, high-level modules such as `Service` objects depend on abstraction rather than implementation details.

**Figure 6.2.2-1** demonstrates an instance of the ISP and LSP in the Object Factory pattern. ISP In the figure `ServiceInterface` and `ProductServiceInterface` interfaces are smaller, more focused interfaces instead of large, general-purpose ones. It also showcases that `ServiceREST` and `ServiceSOAP`, derived classes, implement the `ServiceInterface` interface and can be used in place of the `ServiceInterface` base class without affecting the correctness of the program. Similarly, `ProductServiceREST` and `ProductServiceSOAP`, derived classes, implement the `ProductServiceInterface` interface and can be used in place of the `ProductServiceInterface` base class without affecting the correctness of the program.

In the context of the migration project, the SOLID principles were applied to improve the quality and maintainability of the codebase.

### 6.2.4 RESTful client implementation

The RESTful client implementation is essential to the migration project as it allows communication with a RESTful web service. The client is implemented using the Guzzle library, which provides an easy-to-use interface for making HTTP requests.

The Client class is a singleton, meaning only one class instance can exist anytime. This is achieved by making the constructor private and providing a static `get_instance()` method that returns the existing instance or creates a new one if none exists.

The client's construction involves initializing the API parameters, checking that all the necessary dependencies exist, and ensuring the client can connect to the RESTful API. If any of these checks fail, the client is not created.

After the client is created, the `login_client()` method is called to log in to the RESTful API and retrieve a session ID. The session ID is then used in subsequent requests by setting it in the authorization header.

To ensure consistent behaviour between SOAP and REST services, interfaces and design patterns such as the Chain of Responsibility pattern are implemented. The benefits of using a RESTful client are improved performance and scalability.

The RESTful client implementation can be seen in **Attachment 6,** with the subsequent paragraphs elaborating more on the specifics of the client class.

The `dependants_exist()` function checks whether the `GuzzleHttp\Client` class exists. `is_connection_successful()` function ensures that the REST client can reach the API URL. `init_client()` function initializes the REST client with options that can be modified using the filter `'lemonsoft_integration_REST_client_config'.` `login_client()` function logs the client in for REST using an `AuthenticationRequest` object.

The `get_client_url()` function formats and returns the client URL, while the `get_client_config()` function returns the configuration options for the REST client. `get_client_url_path()` function returns the base URL for the client, which should point to the REST API.

The `get_instance()` function returns either an existing instance or creates a new one. If the current session lifetime is ten minutes or over, the instance is renewed. If renewal fails, a new instance is created.

### 6.2.5 Controlled release

Releasing a software project involves making it available for end-users or customers, making it crucial to manage the release process properly to minimize errors, downtime, and user dissatisfaction.

The migration project required updates for all scheduled actions, manual runs, and the new service endpoint logic creation. The project was released in stages to avoid the risk of an overwhelming amount of bug reports from users, one endpoint and base files relying on it at a time.

The project's file structure was designed to facilitate this approach, with the new `refactored-core` folder following the same relative structure as before and new namespaces only differing from the old ones by the `Refactored/` prefix. This allowed releasing the project one stage at a time, changing only the relevant imported namespaces and possibly minor changes to functions.

This controlled approach allowed for managing the risk of introducing errors or downtime and ensuring a smoother transition for end-users.

## 6.3 Retrospective of the project

**Section 6.2** answered the **RQ2**, "How can SOAP-based and RESTful service clients be implemented into a general client?" by providing an overview of the implantation project. First, the project developed a general client to communicate with SOAP-based and RESTful services to leverage the benefits of RESTful protocol's improved performance and scalability while maintaining compatibility with existing SOAP-based services. Furthermore, using design patterns and an object factory simplified the codebase and enhanced maintainability.

This chapter evaluates the efficiency, maintainability, and scalability differences between SOAP and REST services in the context of the Lemonsoft Integration plugin. It also considers the resource usage of developing a general client for both protocols concerning the benefits of performance and maintainability, providing the answer for the **RQ3**, " Can developing a client

to work with REST and SOAP be justified based on differences in performance and maintainability?".

### 6.3.1 Development effort

The development of a general client capable of handling both SOAP-based and RESTful services was not the initial approach. Initially, an attempt was made to create a mapper, which consumed several workdays before it was concluded to be unmaintainable. To reassess the situation, brainstorming sessions were conducted, involving all developers, which amounted to approximately one workday in total.

After reassessing, the decision was made to implement a general client that would provide compatibility, maintainability, and performance benefits. The foundation and architecture of the general client were established relatively quickly, taking only one week to complete. This rapid progress was due in part to the use of design patterns and an object factory, which helped streamline the codebase, reduce complexity, and facilitate future maintenance.

However, the development process for each endpoint proved to be more time-consuming, requiring one week of development time per endpoint. This was largely because many dependent files needed to be refactored and modified to ensure compatibility with the new general client architecture. After completing the development of each area, testing and pull request (PR) reviews took an additional couple of days.

Although the initial attempt at creating a mapper was unsuccessful, the subsequent development of a general client proved to be a worthwhile investment. The general client provided the desired compatibility, maintainability, and performance benefits, and the development effort can be justified considering the improvements it brought to the overall system.

### 6.3.2 Evaluating the difference in performance

To evaluate the performance differences between the old SOAP implementation and the new general client, a series of performance tests were carried out, with data collected from multiple runs. Each task was executed 30 times, and the results were aggregated for analysis. The primary task involved retrieving updated product information spanning 5 pages, with a limit of 50 products per page, followed by importing these products into the system. The underlying database for each test remained consistent, with the only variation being the service protocol and the client's handling of requests and responses.

The imported products served as updates to existing products rather than new additions. As a result, the state of the products and their contained data remained consistent after each test for both the web service and web shop. The old and new implementations performed the following tasks:

1. Send a request to the product endpoint in service.
2. Receive a response with the raw product data.
3. Format the raw product data
4. Update existing products with the formatted data

The test code for this can be seen in **Attachment 7**. The test code is set up as follows:

- The `run_tests()` function is executed when the WordPress `plugins_loaded` action is triggered. This is due to the test functions being housed within WordPress plugins, which depend on the WordPress ecosystem, including WooCommerce plugins. While it is possible to run the test from the command line, it would necessitate a headless instance of WordPress, potentially leading to unrealistic results since the plugins were not designed for this purpose.
- Service instances are generated, and requests for products are sent in a paginated format.
- Received product data is then formatted into a manageable form and saved to the database.

Each test was run 30 times, and collected data was aggregated and formatted with maximum of four decimal places.

**Table 6.3.2-1 Aggregated results for the old SOAP client, 30 runs**

***Old SOAP Client***

| | |
|---|---|
| *Runtime Average (s):* | 4.40545 |
| *Runtime Range (s):* | 3.86108 - 5.73695 |

| | |
|---|---|
| *Runtime Variance* | 3.55541 |
| *Runtime Std. Deviance* | 1.88558 |
| *CPU Time Average (s):* | 0.1282 |
| *CPU Time Range (s):* | 0.063 – 0.203 |

**Table 6.3.2-2 Aggregated results for the new general client (SOAP), 30 runs**

| ***new SOAP Client*** | |
|---|---|
| *Runtime Average (s):* | 4.35733 |
| *Runtime Range (s):* | 4.10365 - 4.93412 |
| *Runtime Variance* | 0.51964 |
| *Runtime Std. Deviance* | 0.72086 |
| *CPU Time Average (s):* | 0.2031 |
| *CPU Time Range (s):* | 0.1560 – 0.2970 |

**Table 6.3.2-3 Aggregated results for the new general client (REST), 30 runs**

| ***New REST Client*** | |
|---|---|
| *Runtime Average (s):* | 4.2436 |
| *Runtime Range (s):* | 3.9850 - 4.8804 |
| *Runtime Variance* | 0.3914 |

| | |
|---|---|
| *Runtime Std. Deviance* | 0.6299 |
| *CPU Time Average (s):* | 01903 |
| *CPU Time Range (s):* | 0.14 – 0.28 |

Based on the data collected, we can observe the following performance differences from **Table 6.3.2-1**, **Table 6.3.2-2:** and **Table 6.3.2-3**

*6.3.2.1 Runtime*

The runtime represents the total time required to complete a task, taking into account resource wait times and other processes involved in the execution. In the context of the performance tests conducted, each task consisted of retrieving updated product information across 5 pages, with a limit of 50 products per page, and importing these products into the system. The tests were executed 30 times, with the results aggregated for analysis.

When comparing the average runtimes, the new general client using REST demonstrates the best performance with an average runtime of 4.2436 seconds. The new SOAP client follows closely, with an average runtime of 4.35733 seconds. The old SOAP client lags behind, with an average runtime of 4.40545 seconds. These results indicate that the REST client is more efficient in terms of execution time, as it completes the tasks faster than both the old and new SOAP clients.

This improved efficiency in runtime can be attributed to various factors, such as the REST client's ability to better handle resource wait times and streamlined processes for requesting and receiving product data. As a result, the REST implementation allows for quicker completion of the tasks, potentially leading to better user experiences and increased system throughput.

*6.3.2.2 CPU Time*

CPU time refers to the actual duration the CPU spends processing a task, excluding factors such as resource wait times and other processes. When comparing the average CPU time for each client, the old SOAP client is the most efficient, with an average CPU time of 0.1282 seconds. Following this is the new general client using REST, with an average CPU time of 0.1903

seconds, and finally, the new SOAP client, with an average CPU time of 0.2031 seconds. The lower CPU time for the old SOAP client indicates better efficiency in terms of CPU usage.

However, it is essential to evaluate the consistency of the results in addition to the average CPU time. Both the new general clients using SOAP and REST display greater consistency in CPU times, as demonstrated by their lower standard deviations compared to the old SOAP client. This improved consistency in CPU time suggests more effective resource management and a more predictable performance overall.

#### *6.3.2.3 Conclusion*

When comparing the new general clients using SOAP and REST, it is evident that the REST client exhibits superior performance in terms of runtime efficiency, while the old SOAP client has better CPU usage efficiency. However, both new general clients, SOAP and REST, demonstrate more consistent CPU times, which implies better resource management and more predictable performance. These results highlight the potential advantages of adopting REST as the protocol for web services, with its improved execution time and consistent CPU usage.

To further optimize the REST client's performance, it is essential to address areas such as memory usage and the number of function calls. Strategies like implementing effective caching mechanisms, sending requests asynchronously, or optimizing resource usage could lead to significant enhancements in the overall performance of the REST client. By addressing these concerns, the REST client's efficiency and reliability could be further improved, making it an even more compelling choice for web service implementations.

### 6.3.3 Evaluating the difference in maintainability

The maintainability of the code has significantly improved throughout the project, resulting in noticeable differences between the old and new implementations.

The new implementation has been meticulously documented, and the code has been appropriately commented on according to WordPress's code quality guidelines. This was not the case with the initial implementation, which lacked proper documentation and comments. Adhering to best practices makes the new implementation easier to understand, modify, and maintain, especially for new developers in the project.

The new implementation employs interfaces that enforce consistent public functions for REST and SOAP clients to enhance maintainability and prevent errors due to missing functions. In addition, this guarantees compatible access to the client across different communication protocols, making it easier to maintain and extend the codebase.

The new implementation uses the same approach to handle responses from SOAP and REST clients. By passing the responses through tailored object constructors implementing the same interface, the resulting objects have identical getter methods, allowing developers to query data from the response without accounting for differences in message formats or protocol. This abstraction simplifies the code and reduces the likelihood of errors due to format inconsistencies.

The new code has undergone multiple pull request (PR) processes and team meetings to ensure that all stakeholders understand the desired functionality and have opportunities to contribute to the improvements. This collaborative approach has led to a more robust and maintainable codebase.

Finally, the new implementation is characterized by a well-organized and modular code structure, unlike the initial version, which had scattered functionality. The separation of concerns in the new code makes navigating, maintaining, and extending the codebase easier.

### 6.3.4 Justification of resource usage of the project

The response for **RQ3**: "Can developing a client to work with REST and SOAP be justified based on differences in performance and maintainability?" is that in the context of this migration project, the resource usage has been justified by the new general client's significant benefits. However, it is crucial to note that this justification cannot be universally applied, as the resource usage and benefits of each project should be assessed on a case-by-case basis.

To provide concrete examples of how the mentioned improvements are worth the cost of development:

1. Enhanced performance: The REST implementation's faster runtimes and consistent CPU times lead to a better user experience and increased system throughput. The capacity to handle more requests in the same amount of time could potentially boost customer satisfaction and revenue.

2. Simplified codebase: Unifying the handling of responses from both SOAP and REST clients streamlines the codebase, allowing developers to write less code when implementing new features or fixing bugs. This reduction in complexity saves development time and decreases the likelihood of introducing new issues.

3. Efficient development process: The interface-driven design and modular code structure enable developers to work on different parts of the codebase in parallel without creating conflicts or dependencies. This approach allows for a more efficient development process, quicker integration of new features or bug fixes, reduced time-to-market for improvements, and lowered overall development costs.

4. Improved onboarding: Enhanced documentation and adherence to coding standards make it easier for new team members to understand and contribute to the project. This ease of onboarding reduces the time and resources required to train new developers, leading to cost savings and improved productivity.

5. Greater maintainability: The new implementation's maintainability improvements reduce the potential for bugs and decrease the time needed for future updates. A more stable and well-organized codebase allows developers to more easily identify and fix issues, enhancing overall system reliability and user experience. Furthermore, streamlined code and a modular structure facilitate the integration of future updates or new features, ensuring the project remains up-to-date and adaptable to changing requirements.

The investment in development efforts for the migration project has effectively addressed the technical debt associated with the old implementation. In addition, by replacing the outdated SOAP implementation with a more efficient and maintainable general client, the project has minimized the risks and costs associated with maintaining an outdated and less efficient system.

As the new implementation has noticeable improvements in maintainability, future updates and modifications to the codebase are expected to require fewer resources. This improved maintainability, resulting from better code quality, interface-driven design, unified response handling, and a modular code structure, ensures that developers can efficiently modify and maintain the code as needed. Consequently, the project is better equipped to adapt to evolving requirements and changes, reducing long-term maintenance costs and resource demands.

# 7 Conclusion

Although SOAP and REST share the goal of facilitating network communication between systems, they adopt different approaches and use different protocols and standards. Furthermore, SOAP is more structured and formal, while REST is based on more general principles. Therefore, this thesis explored general implementations of SOAP-based and RESTful services, focusing on consuming these services and migrating from one to the other.

The thesis showcases a migration project from Lemonsoft's web team, which involved adapting an existing integration to a new version of the same service with a different transaction protocol while providing support for older versions. The project employed design patterns, refactoring, and the implementation of a general client to support multiple protocols. Finally, a retrospective analysis was conducted to determine if the migration project justified itself in terms of resource usage and maintainability.

This thesis addressed three research questions:

**RQ1**: What are the key functional and conceptual differences between SOAP-based and RESTful web services?

**RQ2**: How can SOAP-based and RESTful service clients be implemented into a general client?

**RQ3**: Can developing a client to work with REST and SOAP be justified based on differences in performance and maintainability?

The answer to **RQ1** was provided through a comparison table, **Table 5.1.2-1** in **Section 5.2**, which summarized the key functional and conceptual differences between SOAP-based and RESTful web services. This side-by-side comparison highlighted the two web services' essential features, strengths, and weaknesses.

**Section 6.2** addressed **RQ2** by providing an overview of the implementation project. The project developed a general client to communicate with SOAP-based and RESTful services, leveraging the benefits of the RESTful protocol's improved performance and scalability while maintaining compatibility with existing SOAP-based services. Furthermore, using design patterns and an object factory simplified the codebase and enhanced maintainability.

The evaluation of efficiency, maintainability, and scalability differences between SOAP and REST services, in the context of the Lemonsoft Integration plugin, answered **RQ3 in Section 6.3**.

The new REST client's superior performance is demonstrated by the faster runtime and consistent CPU time. This improvement enables the application to handle more requests in the same amount of time, thus potentially increasing customer satisfaction and revenue. Additionally, the investment in resources for the migration project effectively addressed the technical debt of the old implementation. The project minimized the risks and costs of maintaining an outdated and less efficient system by replacing the outdated SOAP implementation with a more efficient and maintainable general client.

As the new implementation showed significant improvements in maintainability, future updates and modifications to the codebase are expected to require fewer resources. This enhanced maintainability, resulting from better code quality, interface-driven design, unified response handling, collaborative code review, and a modular code structure, ensures that developers can efficiently modify and maintain the code as needed. Consequently, the project is better equipped to adapt to evolving requirements and changes, reducing long-term maintenance costs and resource demands.

In conclusion, this thesis provides insights into the key differences between SOAP and REST services and highlights the benefits of implementing a general client capable of communicating with both protocols.

# References


[1] 'W3C Glossary and Dictionary', *W3*, May 31, 2010. https://www.w3.org/2003/glossary/ (accessed Sep. 25, 2022).

[2] J. Snell, D. Tidwell, and P. Kulchenko, *Programming Web services with SOAP*, 1st ed. Sebastopol, CA: O'Reilly & Associates, 2002.

[3] D. Box *et al.*, 'Simple object access protocol (SOAP) 1.1', Jan. 2000.

[4] L. Richardson and S. Ruby, *RESTful web services: web services for the real world*. in Web services for the real world. Beijing Köln: O'Reilly, 2007.

[5] F. Halili and E. Ramadani, 'Web Services: A Comparison of Soap and Rest Services', *MAS*, vol. 12, no. 3, p. 175, Feb. 2018, doi: 10.5539/mas.v12n3p175.

[6] S. Mumbaikar and P. Padiya, 'Web Services Based On SOAP and REST Principles', 2013.

[7] 'Web Services Architecture', *W3*, Feb. 11, 2004. https://www.w3.org/TR/ws-arch/ (accessed Oct. 13, 2022).

[8] R. Nagappan, *Developing Java Web services: architecting and developing secure Web services using Java*. Indianapolis, Ind: Wiley, 2003.

[9] R. Chinnici, J. Moreau, A. Ryman, and S. Weerawarana, 'Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language', *W3C working draft*, vol. 26, Jan. 2004.

[10] Liam R. E. Quin, 'XML Essentials', *W3*, May 31, 2010. https://www.w3.org/standards/xml/core (accessed Jan. 11, 2023).

[11] T. O'Reilly, *What is Web 2.0*. O'Reilly Media, 2009.

[12] Rick Strahl, 'Using XML for Messaging in Distributed Applications (Part 1)', *CODE Magazine: 2000 - Spring*.

[13] Dr. K. Wagh and R. Thool, 'A Comparative study of SOAP vs REST web services provisioning techniques for mobile host', *Journal of Information Engineering and Applications*, vol. 2, pp. 12–16, Jul. 2012.

[14] R. Richards, *Pro PHP XML and web services: master working with XML and Web services using PHP ; covers PHP versions 5 and the forthcoming 6*. Berkeley, Calif: Apress, 2006.

[15] R. T. Fielding, M. Nottingham, and J. Reschke, 'HTTP Semantics', no. 9110. in Request for Comments. RFC Editor, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9110

[16] L. J. Mitchell, *PHP web services: APIs for the modern web*, Second edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2016.

[17] Lokesh Gupta, 'REST Architectural Constraints', *REST API Tutorial*, Sep. 03, 2022. https://restfulapi.net/rest-architectural-constraints/#uniform-interface (accessed Oct. 28, 2022).

[18] ECMA-404, 'The JSON Data Interchange Format', *Json*, 2013. ttps://www.json.org/json-en.html (accessed Oct. 21, 2022).

[19] Jeff Posnick and Ilya Grigorik, 'Prevent unnecessary network requests with the HTTP Cache', *web.dev*, Apr. 17, 2020. https://web.dev/http-cache/ (accessed Mar. 27, 2023).

[20] 'Introduction to Redis'. 2023. Accessed: Mar. 27, 2023. [Online]. Available: https://redis.io/docs/about/

[21] 'What is a Content Delivery Network (CDN)? | How do CDNs work?' Cloudflare, 2023. Accessed: Mar. 27, 2023. [Online]. Available: https://www.cloudflare.com/learning/cdn/what-is-a-cdn/

[22] R. Fielding *et al.*, ‘RFC 2616, Hypertext Transfer Protocol – HTTP/1.1’. 1999. [Online]. Available: http://www.rfc.net/rfc2616.html

[23] Martin Gudgin *et al.*, ‘SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)’, *W3*, May 31, 2010. https://www.w3.org/TR/soap12-part1/ (accessed Oct. 26, 2022).

[24] A. Rodriquez, ‘IBM Developer’, *ibm*, 2013. https://developer.ibm.com/articles/ws-restful/ (accessed Oct. 21, 2022).

[25] IBM, ‘What is EDI?’, *IBM*, May 31, 2010. https://www.w3.org/2003/glossary/ (accessed Jan. 10, 2023).

[26] IBM, ‘Creating top-down Web services’, *IBM*, May 31, 2010. https://www.ibm.com/docs/en/rsm/7.5.0?topic=overview-creating-top-down-web-services (accessed Jan. 10, 2023).

[27] IBM, ‘Creating bottom-up Web services’, *IBM*, May 31, 2010. https://www.ibm.com/docs/en/rsm/7.5.0?topic=overview-creating-bottom-up-web-services' (accessed Jan. 10, 2023).

[28] J. Tihomirovs and J. Grabis, ‘Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics’, *Information Technology and Management Science*, vol. 19, no. 1, Jan. 2016, doi: 10.1515/itms-2016-0017.

[29] F. Belqasmi, J. Singh, S. Y. Bani Melhem, and R. H. Glitho, ‘SOAP-Based vs. RESTful Web Services: A Case Study for Multimedia Conferencing’, *IEEE Internet Comput.*, vol. 16, no. 4, pp. 54–63, Jul. 2012, doi: 10.1109/MIC.2012.62.

[30] J. Hündling and M. Weske, ‘Web Services: Foundation and Composition’, *Electronic Markets*, vol. 13, no. 2, pp. 108–119, Jan. 2003, doi: 10.1080/1019678032000067226.

[31] Lauren Long, ‘What RESTful actually means’, *Code Words*, Feb. 28, 2020. https://codewords.recurse.com/issues/five/what-restful-actually-means (accessed Dec. 12, 2022).

[32] J. Lloret Mauri, Institute of Electrical and Electronics Engineers, IEEE Communications Society, and IEEE Systems, Man, and Cybernetics Society, Eds., *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI 2015): Kochi, [Kerala], India, 10 - 13 August 2015 ; [including co-affiliated symposia]*. Piscataway, NJ: IEEE, 2015.

[33] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, California: UNIVERSITY OF CALIFORNIA, 2000.

[34] Omar Ismail, ‘Principles & Best practices of REST API Design’, *Principles & Best practices of REST API Design*, Nov. 28, 2021. https://www.linkedin.com/pulse/principles-best-practices-rest-api-design-omar-ismail (accessed Dec. 12, 2022).

[35] T. Erl, P. Merson, and R. Stoffers, *Service-oriented architecture: analysis and design for services and microservices*, Second edition. in The Prentice Hall service technology series from Thomas Erl. Boston: Prentice Hall, 2017.

[36] R. T. Fielding and R. N. Taylor, ‘Principled design of the modern Web architecture’, *ACM Trans. Internet Technol.*, vol. 2, no. 2, pp. 115–150, May 2002, doi: 10.1145/514183.514185.

[37] I. Zuzak and S. Schreier, ‘ArRESTed Development: Guidelines for Designing REST Frameworks’, *IEEE Internet Comput.*, vol. 16, no. 4, pp. 26–35, Jul. 2012, doi: 10.1109/MIC.2012.60.

[38] C. Pautasso, O. Zimmermann, and F. Leymann, ‘Restful web services vs. “big”’ web services: making the right architectural decision’, in *Proceeding of the 17th international conference on World Wide Web - WWW ’08*, Beijing, China: ACM Press, 2008, p. 805. doi: 10.1145/1367497.1367606.

[39] A. P. Ciganek, M. N. Haines, and W. Haseman, ‘Horizontal and Vertical Factors Influencing the Adoption of Web Services’, in *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS’06)*, Kauia, HI, USA: IEEE, 2006, pp. 109c–109c. doi: 10.1109/HICSS.2006.202.

[40] ‘Learn about WordPress origins and version history’, *WordPress*, Oct. 16, 2018. https://wordpress.org/documentation/article/learn-about-wordpress-and-version-history/

[41] ‘WooCommerce Documentation’, *WooCommerce*, Jan. 01, 1970. https://developer.woocommerce.com

[42] Lemonsoft, ‘Lemonsoft for investors’, *Investors Lemonsoft*, 2022. https://investors.lemonsoft.fi/frontpage/ (accessed Mar. 28, 2023).

[43] Lemonsoft, ‘Lemonsoft’, *Lemonsoft*, 2022. https://lemonsoft.fi (accessed Mar. 28, 2023).

[44] E. Gamma, Ed., *Design patterns: elements of reusable object-oriented software*. in Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995.

[45] M. Fowler, *Patterns of enterprise application architecture*. in The Addison-Wesley signature series. Boston: Addison-Wesley, 2003.

[46] E. Freeman, E. Robson, K. Sierra, and B. Bates, Eds., *Head First design patterns*. Sebastopol, CA: O’Reilly, 2004.

[47] Mariam Jaludi, ‘Software Design Principles: SOLID’, *Medium*, Mar. 29, 2023. https://mariam-jaludi.medium.com/software-design-principles-solid-ef0c3d5b008a (accessed Jul. 06, 2019).

[48] Paul Knulst, ‘Why SOLID Design Matters: Avoid Code Smells and Write Maintainable Code’, *pauls dev blog*, Dec. 09, 2022. https://www.paulsblog.dev/why-solid-design-matters-avoid-code-smells-and-write-maintainable-code/ (accessed Mar. 29, 2023).

# Attachments

## Attachment 1. SOAP-based service example

```php
<?php

error_reporting(E_ALL ^ E_NOTICE);
// include the NuSOAP library
require_once('lib/nusoap/nusoap.php');

// create a new soap server
$server = new nusoap_server;

// configure the WSDL
$server->configureWSDL('calculator', 'urn:calculator');
$server->wsdl->schemaTargetNamespace = 'urn:calculator';


// register two methods: add and subtract
/** @link add() */
$server->register(
    'add', // method name
    array('a' => 'xsd:int', 'b' => 'xsd:int'), // input parameters
    array('return' => 'xsd:int'), // output parameters
    'urn:calculator', // namespace
    'urn:calculator#add', // soapaction
    'rpc', // style
    'encoded', // use
    'Adds two integers' // documentation
);

/** @link subtract() */
$server->register(
    'subtract', // method name
    array('a' => 'xsd:int', 'b' => 'xsd:int'), // input parameters
    array('return' => 'xsd:int'), // output parameters
    'urn:calculator', // namespace
    'urn:calculator#subtract', // soapaction
    'rpc', // style
    'encoded', // use
    'Subtracts two integers' // documentation
);

// create the methods
function add($a, $b)
{
    return $a + $b;
}

function subtract($a, $b)
{
    return $a - $b;
}

// use the request to (try to) invoke the service
$server->service(file_get_contents('php://input'));
```

# Attachment 2. SOAP-based service's example WSDL

```
<definitions targetNamespace="urn:calculator">
    <types>
        <xsd:schema targetNamespace="urn:calculator">
            <xsd:import
namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
            <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/"/>
        </xsd:schema>
    </types>
    <message name="addRequest">
        <part name="a" type="xsd:int"/>
        <part name="b" type="xsd:int"/>
    </message>
    <message name="addResponse">
        <part name="return" type="xsd:int"/>
    </message>
    <message name="subtractRequest">
        <part name="a" type="xsd:int"/>
        <part name="b" type="xsd:int"/>
    </message>
    <message name="subtractResponse">
        <part name="return" type="xsd:int"/>
    </message>
    <portType name="calculatorPortType">
        <operation name="add">
            <documentation>Adds two integers</documentation>
            <input message="tns:addRequest"/>
            <output message="tns:addResponse"/>
        </operation>
        <operation name="subtract">
            <documentation>Subtracts two integers</documentation>
            <input message="tns:subtractRequest"/>
            <output message="tns:subtractResponse"/>
        </operation>
    </portType>
    <binding name="calculatorBinding" type="tns:calculatorPortType">
        <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="add">
            <soap:operation soapAction="urn:calculator#add" style="rpc"/>
            <input>
                <soap:body use="encoded" namespace="urn:calculator"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
                <soap:body use="encoded" namespace="urn:calculator"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
        <operation name="subtract">
            <soap:operation soapAction="urn:calculator#subtract"
style="rpc"/>
            <input>
                <soap:body use="encoded" namespace="urn:calculator"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
```

```
                <soap:body use="encoded" namespace="urn:calculator"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
    </binding>
    <service name="calculator">
        <port name="calculatorPort" binding="tns:calculatorBinding">
            <soap:address
location="http://example.com:8000/SOAP/calculator.php"/>
        </port>
    </service>
</definitions>
```

## Attachment 3. SOAP-based service's example unit tests

```
<?php

// include the NuSOAP library
require_once('lib/nusoap/nusoap.php');

// create a new soap client

use PHPUnit\Framework\TestCase;
use PHPUnit\Framework\TestSuite;

class CalculatorTest extends TestCase
{
    // test the add method
    function testAdd()
    {
        $client = new
nusoap_client('http://example.com:8000/SOAP/calculator.php?wsdl');
        $result = $client->call('add', array(5, 3));
        $expected = 8;
        $this->assertEquals($expected, $result);
    }

    // test the subtract method
    function testSubtract()
    {
        $client = new
nusoap_client('http://example.com:8000/SOAP/calculator.php?wsdl');
        $result = $client->call('subtract', array(5, 3));
        $expected = 2;
        $this->assertEquals($expected, $result);
    }
}

// create a test suite
$testSuite = new TestSuite('CalculatorTest');

// add the test cases to the test suite
$testSuite->addTest(new CalculatorTest('testAdd'));
$testSuite->addTest(new CalculatorTest('testSubtract'));

// run the test suite
$result = $testSuite->run();

// print the result
```

```
print_r($result->passed());
```

# Attachment 4. RESTful service example

```
<?php
// set the content type to JSON
header('Content-Type: application/json');

// get the HTTP method, path, and body of the request
$method = $_SERVER['REQUEST_METHOD'];
$request = explode('/', trim($_SERVER['REQUEST_URI'], '/'));
$input = json_decode(file_get_contents('php://input'), true);

// create a simple array for the response
$response = array();

// check the HTTP method and the request
if ($method == 'GET' && count($request) == 5 && $request[2] == 'add') {
    // add the two numbers and return the result
    $response['result'] = $request[3] + $request[4];
} else {
    if ($method == 'GET' && count($request) == 5 && $request[2] ==
'subtract') {
        // subtract the two numbers and return the result
        $response['result'] = $request[3] - $request[4];
    } else {
        // set the response code to 404 (not found)
        http_response_code(404);
    }
}
// convert the array to JSON and print it
echo json_encode($response);
```

# Attachment 5. RESTful service's example unit tests

```
<?php

use PHPUnit\Framework\TestCase;
use PHPUnit\Framework\TestSuite;

class CalculatorTest extends TestCase
{
// test the add method
    function testAdd()
    {
        $url = 'http://example.com:8000/REST/calculator.php/add/5/3';
        $expected = '{"result":8}';
        $ch = curl_init();

        // set the URL and options
        curl_setopt($ch, CURLOPT_URL, $url);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

        // execute the request and store the response
        $result = curl_exec($ch);

        // close the cURL resource
```

```
        curl_close($ch);

        $this->assertEquals($expected, $result);
    }

    // test the subtract method
    function testSubtract()
    {
        $url = 'http://example.com:8000/REST/calculator.php/subtract/5/3';
        $expected = '{"result":2}';

        // create a new cURL resource
        $ch = curl_init();

        // set the URL and options
        curl_setopt($ch, CURLOPT_URL, $url);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

        // execute the request and store the response
        $result = curl_exec($ch);

        // close the cURL resource
        curl_close($ch);

        $this->assertEquals($expected, $result);
    }

}

// create a test suite
$testSuite = new TestSuite('CalculatorTest');

// add the test cases to the test suite
$testSuite->addTest(new CalculatorTest('testAdd'));
$testSuite->addTest(new CalculatorTest('testSubtract'));

// run the test suite
$result = $testSuite->run();

// print the result
print_r($result->passed());
```

# Attachment 6. RESTful client in Lemonsoft Integration

```
<?php

namespace Lemonsoft\Data\Api\WebService\REST;

use Exception;
use GuzzleHttp\Client as GuzzleClient;
use Lemonsoft\Data\Api\Tools\ApiParameters;
use Lemonsoft\Data\Api\WebService\ClientInterface;
use Lemonsoft\Data\Api\WebService\REST\Request\AuthenticationRequest;
use Lemonsoft\Data\Api\WebService\WebServiceFactory;
use Lemonsoft\Utilities\Log;

/**
```

```
 * Singleton Client class for REST. Initializes the Guzzle Client and
fetches a valid session id.
 * Afterwards, provides the client with requests.
 */
class Client implements ClientInterface
{
    private static ?Client $instance = null;
    private ?ApiParameters $api_parameters;
    private ?GuzzleClient $client = null;
    private string $api_url_path = 'api';
    private static int $instance_creation_time;
    private static int $instance_current_session_time;

    /**
     * Construction is private to enforce singleton pattern.
     *
     * @throws Exception
     * @see get_instance()
     */
    private function __construct()
    {
        // Initialize API parameters for REST calls
        $this->api_parameters = new ApiParameters();

        // Check that Guzzle exists, all api_parameters are valid, and the
API URL can be reached. Otherwise, abort.
        if (!$this->dependants_exist() || !$this->api_parameters-
>get_valid() || !$this->is_connection_successful()) {
            return null;
        }

        // Initialize and set clients, then login to them. If login is
successful, set the instance creation and current session time and return
the instance.
        if ($this->init_client() && $this->login_client()) {
            self::$instance_creation_time = time();
            self::$instance_current_session_time = time();

            return $this;
        }

        return null;
    }

    /**
     * Checks whether the GuzzleHttp\Client class exists or not and adds a
warning if it does not.
     *
     * @return bool
     */
    public function dependants_exist(): bool
    {
        if (!class_exists('GuzzleHttp\Client')) {
            WebServiceFactory::$errors->add_error(
                'No Guzzle',
                'Please contact your administrator to install Guzzle.'
            );

            return false;
        }
```

```
        return true;
    }

    /**
     * Ensure the REST client can reach the API URL.
     *
     * @return bool
     */
    public function is_connection_successful(): bool
    {
        $url = $this->get_client_url();

        return $this->api_parameters->is_connection_successful($url);
    }

    /**
     * Initialize the REST client. Options for the client can be modified
with filter 'lemonsoft_integration_REST_client_config'
     *
     * @return bool
     */
    private function init_client(): bool
    {
        $config = apply_filters('lemonsoft_integration_REST_client_config',
$this->get_client_config());

        try {
            $this->client = new GuzzleClient($config);

            return true;
        } catch (Exception $e) {
            WebServiceFactory::$errors->add_error(
                'Unable to initialize client',
                $e->getMessage()
            );
            $this->client = null;
        }

        return false;
    }

    /**
     * Login to the client for REST.
     *
     * @return bool
     * @throws Exception
     * @noinspection PhpMemberCanBePulledUpInspection
     */
    private function login_client(): bool
    {
        $authentication_response = (new AuthenticationRequest($this))-
>login();
        if ($authentication_response && $authentication_response-
>get_login_success()) {
            $session_id = $authentication_response->get_session_id();
            $this->api_parameters->set_session_id($session_id);
            $config = $this->get_client_config();
            $config['headers']['Session-Id'] = $session_id;
            $this->client = new GuzzleClient($config);

            return true;
```

```
        }
        $session_error = $authentication_response ?
$authentication_response->get_login_success_code() : 'No response';
        WebServiceFactory::$errors->add_error($session_error, 'Login was
unsuccessful.');

        return false;
    }

    /**
     * Get the client URL and format it.
     * @return string
     */
    private function get_client_url(): string
    {
        return $this->api_parameters->get_api_url() . $this-
>get_client_url_path() . '/';
    }

    /**
     * Get config for a REST client.
     *
     * @return array
     */
    public function get_client_config(): array
    {
        return array(
            'base_uri' => $this->get_client_url(),
            'headers' => array(
                'Accept' => 'application/json',
            ),
        );
    }

    /**
     * Get the base URL for the client. The base URL should point to REST
API.
     *
     * @return string
     */
    private function get_client_url_path(): string
    {
        return $this->api_url_path;
    }

    /**
     * Client class is a singleton. This method returns either an existing
instance or creates a new one.
     * If the current session lifetime is ten minutes or over, we renew the
instance. If this fails, we create a new instance.
     *
     * @return Client
     * @throws Exception
     */
    public static function get_instance(): Client
    {
        if (self::$instance === null ||
empty(self::$instance_creation_time)) {
            self::$instance = new self();
        } else {
            $session_age = time() - self::$instance_current_session_time;
```

```
            if ($session_age >= 10 * MINUTE_IN_SECONDS && !(new Service())-
>renew()) {
                $instance_age = time() - self::$instance_creation_time;
                Log::debug(__CLASS__, __FUNCTION__, "Renewal for the REST
instance failed. Instance age in Unix time was {$instance_age}. Creating a
new instance.");
                self::$instance = new self();
            }
            self::$instance_current_session_time = time();
            Log::debug(__CLASS__, __FUNCTION__, 'Renewed REST client
instance.');
        }

        return self::$instance;
    }

    /**
     * Get API parameters for the current client
     * @return null|ApiParameters
     */
    public function get_api_parameters(): ?ApiParameters
    {
        return $this->api_parameters;
    }

    /**
     * @return null|GuzzleClient
     */
    public function get_client(): ?GuzzleClient
    {
        return $this->client;
    }

}
```

# Attachment 7. Performance test

```
<?php
use Lemonsoft\Enums\LemonsoftServiceEndpoints;
use Lemonsoft\Utilities\Api;
use Refactored\Lemonsoft\Data\Api\Product;
/** @link run_rest_test() */
add_action('plugins_loaded', 'run_tests', 1000);
/**
 * Runs the tests for old SOAP implementation and the general client
implementation with both SOAP and REST.
 * After running the tests, the link for the aggregated test report is echoed.
 * @throws Exception
 */
function run_tests()
{
    $updated_after = date('Y-m-d', strtotime("-365 days"));
    perform_test('old', $updated_after, 'SOAP');
    perform_test('new', $updated_after, 'SOAP');
    perform_test('new', $updated_after, 'REST');
    exit;
}
/**
 * Calculate CPU usage
```

```
 * @param $ru
 * @param $rus
 * @param $index
 *
 * @return float|int
 */
function rutime($ru, $rus, $index){
    return ($ru["ru_$index.tv_sec"]*1000 +
intval($ru["ru_$index.tv_usec"]/1000))
           -  ($rus["ru_$index.tv_sec"]*1000 +
intval($rus["ru_$index.tv_usec"]/1000));
}
/**
 * Calculate variance.
 *
 * @param array $array_of_numbers
 * @return float
 */
 function get_variance(array $array_of_numbers): float
 {
    $variance = 0.0;
    $total_elem_in_array = count($array_of_numbers);
    // Calc Mean.
    $avg_value = array_sum($array_of_numbers) / $total_elem_in_array;

    foreach ($array_of_numbers as $item) {
        $variance += pow(abs($item - $avg_value), 2);
    }

    return $variance;
}
/**
 * Simple deviation.
 *
 * @param float $variance
 *
 * @return float
 */
 function get_std_deviation(float $variance): float
 {
    return sqrt($variance);
}
/**
 * @param $title
 * @param $arr
 *
 * @return void
 */
function print_stats($title, $arr){
     $variance = get_variance($arr);
     $std_dev = get_std_deviation($variance);
     $avg = array_sum($arr)/count($arr);
     $min = min($arr);
     $max = max($arr);
    echo '<div>';
    echo "<h2>-------$title-------</h2>";
```

```
    echo '<table>';
    echo '<tr><th>Variance</th></tr>';
    echo "<tr><td>$variance</td></tr>";

    echo '<tr><th>Std Deviation</th></tr>';
    echo "<tr><td>$std_dev</td></tr>";

    echo '<tr><th>Average</th></tr>';
    echo "<tr><td>$avg</td></tr>";

    echo '<tr><th>Min</th></tr>';
    echo "<tr><td>$min</td></tr>";

    echo '<tr><th>Max</th></tr>';
    echo "<tr><td>$max</td></tr>";
    echo '</table></div>';
}
/**
 * Performs the tests. The test does the following:
 * 1. Calls for an instance of a function which provides access to the client.
 * 2. Uses the client to send a request to the web server.
 * 3. Receives the response from the web server.
 * The test is run 30 times and the results are aggregated in the test report,
by providing the ids of the runs to the report address.
 *
 * @param $prefix
 * @param $updated_after
 * @param string $protocol
 *
 */
function perform_test($prefix, $updated_after, string $protocol = '')
{
    $service_function = $prefix . '_get_product_service';

    $iterations = 10; // Number of times to run each function
    $time = array();
    $cpu = array();


    for ($i = 0; $i < $iterations; $i++) {
        $page = 5;
        $cpu_before = getrusage();
        $start = microtime(true);
        $api = $service_function($protocol);
        $index = 1;
        while( $index <= $page) {
            $return_data = $api->get_paged_list($updated_after, $index, 50);
            $index++;
            echo $return_data['handled'] . '<br>';
        }

        $time[] = (microtime(true) - $start);
        $cpu_after = getrusage();
        $cpu[] = rutime($cpu_after, $cpu_before, 'utime');
    }
    print_stats('Time (s)', $time);
```

```
    print_stats('CPU (s)', $cpu);
}
/**
 * Ask for Api instance to be initialized if it does not exist and then gets
the api.
 * Returns a Product api instance.
 *
 * @return mixed
 */
function old_get_product_service($protocol = '')
{
    $api = (Api::instance())::get_api();

    return $api->Product();
}
/**
 * Asks for a factory object to provide an instance fitting for communicating
with product endpoint. Can either give SOAP or REST, depending on the
specified protocol.
 *
 * @param $protocol
 *
 * @return Product
 */
function new_get_product_service($protocol)
{
    $service =
WebServiceFactory::get_service_with_protocol(LemonsoftServiceEndpoints::PRODUC
T_SERVICE, $protocol);

    return new Product($service);
}
```