Formalising Solutions to REST API Practices as Design (Anti)patterns

Van Tuan Tran

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

September 2021

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Van Tuan Tran**

Entitled: **Formalising Solutions to REST API Practices as Design (Anti)patterns**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Joey Paquet

_____ Examiner
Dr. Brigitte Jaumard

_____ Supervisor
Dr. Yann-Gaël Guéhéneuc

Approved by _____
Lata Narayanan, Chair
Department of Computer Science and Software Engineering

August 10th, 2021 _____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Formalising Solutions to REST API Practices as Design (Anti)patterns

Van Tuan Tran

REST APIs are nowadays the de-facto standard for Web applications. However, as more systems and services adopt the REST architectural style, many problems arise regularly. To avoid these repetitive problems, developers should follow good practices and avoid bad ones. Thus, research on best and bad practices and how to design simple but effective REST APIs is important. Yet, to the best of our knowledge, only a few recurring REST API practices have been codified in the form of design (anti)patterns, which include a name, an intent, a context, and some solutions. There are works on defining or detecting some practices, but not on codifying these practices as patterns. In this thesis, we present the most up-to-date list of REST API practices and formalise them in the form of design (anti)patterns to ease their use by developers. During this formalisation, we also devise and formalise solutions to these practices. Finally, we validate our design (anti)patterns with a survey and interviews of 55 developers, which confirm that our solutions are acceptable by practitioners and reflect real problems and solutions.

Acknowledgments

This research work is funded by the Natural Science and Engineering Research Council (NSERC) and Concordia University. My sincere thank you to all academic personnel who made this research possible.

I want to give a special thank you to my supervisor, Professor Yann-Gaël Guéhéneuc, for the guidance, support, and advice he has provided throughout my time as his student. In addition, I would like to thank Dr. Manel Abdellatif for her help with this research.

I want to express my gratitude to my committee members, Dr. Joey Paquet and Dr. Brigitte Jaumard, for their time and guidance through the thesis review.

A warm thank you to those who helped with the surveys and interviews that validate this research.

I would also like to thank my colleagues in the Ptidej lab for their friendship, guidance, and weekly discussion that sparkled new ideas and questions.

Finally, I would like to thank my family and friends, who always supported me while I worked on this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decade, the information presented on the Internet moved from simple static Web pages to sophisticated interactive Web applications that can be customized by and react to user actions. Users expect to find in their Web browsers the same applications that they run on their local computers, making these Web applications more complicated than ever.

More and more Web applications use the REpresentational State Transfer (REST) architectural style, which separates the concerns of the server (store, process, and serve resources) with that of the client application (present information). Simple Object Access Protocol (SOAP) used to be used to expose services to clients. However, starting from the 2000s, many companies and organizations migrated their services from SOAP to REST to ease the development of Web applications and simplify developers' access to data. For example, in 2006, Google deprecated SOAP for its Search API and moved to REST[1]. Another evidence of this trend is that the number of REST APIs increases every year, from 445 APIs in 2007 to over 24,000 in 2021[2].

As with any other architectural style, REST APIs can be more or less "well" defined and used. They are subjects to good and bad practices. To evaluate how good a REST system is, Richardson[3] proposed a maturity model for REST APIs. Other researchers also proposed good practices to make REST APIs more understandable and reusable (Masse, 2011; Petrillo, Merle, Moha, & Guéhéneuc, 2016).

[1] https://groups.google.com/g/google.public.web-apis/c/YOHPWSqcFBA
[2] https://www.programmableweb.com/apis/directory
[3] http://martinfowler.com/articles/richardsonMaturityModel.html

The academic and gray literature report 14 problems related to REST APIs and their uses. For example, Rodríguez et al. (2016) reported that only a few Web services reach maturity level 3, which is "Hypermedia as the engine of state". In another example, cloud providers reached a low but acceptable level of maturity in REST API design thanks to strict and precise specifications (Petrillo et al., 2016).

Still, the literature has so far not systematically described these good and bad practices in the form of design (anti)patterns: either a context, a problem, and its solution or a context, a recurring design with bad consequences, and an alternative design with more positive outcomes.

In practice, we examine two popular web frameworks in the industry: ASP.NET core[4] and Java Spring[5]. These web frameworks are open source with huge community contributions and discussions. Both of them are mature and in active development. Multiple good practices are already supported, or partially supported, as we will discuss in Chapter 3.

In general, we propose three contributions:

(1) We review the academic and gray literature related to REST APIs and identify eight good practices and 12 bad practices.

(2) We propose practical solutions to these problems and formalize them in the form of REST API design (anti)patterns.

(3) We validate our solutions via a survey of 55 participants and four interviews, which shows that our solutions and resulting patterns are generally well accepted by developers.

The rest of the paper is as follows. Chapter 2 summarises the related work. Chapter 3.1 describes our approach. Chapter 3 discusses REST API practices and their concrete solutions. Chapter 4 explains how we evaluated our patterns by surveying developers. Chapter 5 discusses threats to the validity of our patterns as well as our approach in general. Chapter 6 concludes with future work. Each chapter includes a summary section at the end for quick review of the thesis.

Part of this thesis has been submitted for publication to the **19th International Conference on Service-Oriented Computing (ICSOC)**, the top conference on the topic of services. We submitted

[4] https://dotnet.microsoft.com/apps/aspnet
[5] https://spring.io/projects

the first draft and received positive comments. Therefore, we will submit a revised version in Mid August 2021.

Unrelated to the topic of this thesis, we also published a short paper in the New Ideas and Emerging Results track of the **9^{th} Working conference on Software Visualisation (VISSOFT)**.

Chapter 2

Related Work

To search for related works, we start with the list of REST API practices. Then, we look into its original definition and references for each practice to see if there is already a solution. Next, we search Google Scholar with the practice's name as the keyword to find any paper and publication that provides a solution. Finally, we use a search engine to search for any technical blog, tutorial, frameworks documentation to see if any solution existed but was not published as papers or publications.

We summarise the few works that identified/defined good and bad practices in the development of REST APIs in Section 2.1. Then we describe the solutions proposed for some of these practices in Section 2.2. Thus, we show that many practices have no solutions or that their solutions have not been validated. We also show that practices and solutions are scattered and argue that developers would benefit from explicit design (anti)patterns that formalize these practices and their solutions.

2.1 Practices

Few academic works attempted to identify and formalize good and bad practices for the design of REST APIs.

Rodriguez, Crasso, Zunino, and Campo (2010) proposed the "Content negotiation" good practice, which suggests that servers should serve different formats of the same resources on request. It was one of the first good REST API practices studied in the literature.

Masse (2011), in the book "REST API Design Rulebook", defined 84 rules to design a consistent REST API, some of which became *de facto* standard, e.g., "Amorphous URIs" or "CRUD function name should not be used in URIs" (also called "CRUDy URIs"). The good practice suggests the four verbs and their variances ***CREATE***, ***READ***, ***UPDATE*** and ***DELETE*** should not be presented in the URI. Evdemon provides examples of the "CRUDy URIs" bad practice, where CRUD verbs are included in the URI[1].

In addition, we could combine multiple rules to produce a single good practice. For example, conforming to all the rules related to "Request Methods" could become a good practice "Use the correct HTTP Verbs". On the other hand, violations of these rules could lead to the bad practice of "Use the wrong HTTP Verbs". Another example, violations of the rules related to "Message Body Format" could become a bad practice "Ignoring MIME type".

Fredrich (2012) defined three bad practices, including "Context-less resource name", "Non-hierarchical nodes", "Singularized and Pluralized Nodes". He also gave two good practices, which are "List pagination" and "API Versioning". "Context-less resource name" is when URIs are composed of nodes that are not in the same semantic context. "Non-hierarchical nodes" is when the nodes are not hierarchically ordered. "Singularized and Pluralized Nodes" are a set of rules for naming the URI's nodes. "List pagination" suggests that all lists should be paginated, no matter how short the list is. Finally, "API Versioning" suggests the REST APIs should be versioned, even if this is the first version.

Palma et al. (2017) define the "Non-pertinent documentation" bad practice when the documentation is not matching the existing REST APIs.

Tilkov (Tilkov, 2008) defines and explains seven REST API bad practices, including "Breaking self-descriptiveness", "Forgetting Hypermedia" (bad practice of "Entity Linking"), "Ignoring MIME type" (bad practice of "Content negotiation"), "Ignoring status code", and "Misusing cookies". For the bad practice "Tunnel everything through GET" and "Tunnel everything through POST", we combine them with other misuses of HTTP Verbs as "Use the wrong HTTP Verbs" for simplicity. "Breaking self-descriptiveness" means developers ignore standardized headers, formats, protocols and using non-standard ones. "Ignoring status code" happens when a server does

[1] https://bit.ly/3i5CIsc

not use status codes or use the wrong ones; "Misusing cookies" when a server store the session's state or cookies, breaking the statelessness of REST APIs.

In addition to these practices, we propose two new good practices: "Server Timeout" and "POST-PUT-PATCH Return", which we discuss in Section 3. Table 2.1 list out all the good and bad practices in both academia and gray literature.

Table 2.1: List of good and bad practices in REST API systems

Good practices		Content negotiation (Rodriguez et al., 2010)
		End-point redirection
		Entity linking
		Response caching
		List Pagination (Fredrich, 2012)
		API Versioning (Fredrich, 2012)
Bad practices	**Linguistic**	Context-less resource name (Fredrich, 2012)
		Non-hierarchical nodes (Fredrich, 2012)
		Amorphous URIs (Masse, 2011)
		CRUDy URIs (Evdemon, 2016; Masse, 2011)
		Singularized and Pluralized Nodes (Fredrich, 2012)
		Non-pertinent documentation (Palma et al., 2017)
	REST	Ignoring MIME type (Tilkov, 2008) (bad practice of Content negotiation)
		Ignoring status code (Tilkov, 2008)
		Forgetting Hypermedia (Tilkov, 2008) (bad practice of Entity Linking)
		Misusing cookies (Tilkov, 2008)
		Use the wrong HTTP verbs (Masse, 2011; Tilkov, 2008)
		Breaking self-descriptiveness (Tilkov, 2008)
		Server Timeout
		POST-PUT-PATCH Return

2.2 Solutions to Good and Bad Practices

Researchers proposed solutions for some good practices, which are "Content Negotiation", "Entity Linking", "API Versioning", "Server Timeout", and "Response Caching". Popular web frameworks also often provide some support for some practices, which are "Content Negotiation", "Response Caching", "POST-PUT-PATCH Return", and "List Pagination". In the following section, we will discuss each of these solutions.

For **"Content negotiation"**, Lemlouma and Layaïda (2001) proposed a "Negotiation and adaptation core" (NAC) that works as a proxy between media servers and consuming clients. Based on the clients' profiles, the NAC converts responses into appropriate formats. However, NAC is a general architecture, and the authors do not discuss any concrete implementation. Butler (2001) using `if else` conditional constructs to perform content negotiation, which is obviously not scalable. In addition, the author uses an XML file for configurations of the matched nodes and properties. The file can be extended later to be compatible with new conditions. They also admitted that the matching algorithms are too simple and may need a more complex algorithm and more complex ways of grouping attributes. The examined web frameworks have built-in supports for content negotiation. We will discuss these built-in solutions in Section 3.2.

For **"Endpoint Redirection (URL Redirection)"**, we could not find any academic solution. The gray literature only explains the concept of URL redirection and how to set them up in some servers. Popular servers, like Microsoft IIS[2] or Apache Tomcat[3], implement URL redirection with some configurations. These built-in solutions solve problems like redirecting HTTP to HTTPS, removing or adding the "www" prefix to the root URL, but not for a complex resource mapping mechanism.

For **"Entity Linking"** (or its corresponding bad practice "Forgetting Hypermedia"), Liskin, Singer, and Schneider (2011) describe using a wrapper module to convert a normal response to a response that conforms to "Entity Linking". This approach improves old REST systems not supporting Entity Linking and reaching level three in Richardson's Maturity Model.

For the **"POST-PUT-PATCH return"** good practice, user "iswinky" on StackExchange asked the question on the Software Engineering sub-forum[4]. The answers and discussion reached a general acceptance that the server should return the created/updated object or the object's ID. Developers can also return the URL to request the object, conforming to the "Entity Linking" good practice. The same related questions were asked on StackOverflow and had the same answers[5].

"API Versioning" is a common problem of REST API systems. Kaminski, Litoiu, and Müller

[2] https://docs.microsoft.com/en-us/iis/extensions/url-rewrite-module/using-the-url-rewrite-module

[3] https://tomcat.apache.org/tomcat-10.0-doc/rewrite.html

[4] https://softwareengineering.stackexchange.com/q/314066/243384

[5] https://stackoverflow.com/a/1829913/4506315

(2006) proposed the "Chain of Adapters" design technique. The technique uses an adapter to adapt the previous version to the next version. Suppose there are three versions, and the latest is version three; it will need two adapters; first, to adapt version one to version two, then to adapt it again to version three, creating a "Chain of Adapters". However, the more versions the REST API system support, the slower and more complex it gets to use the adapters. Leitner, Michlmayr, Rosenberg, and Dustdar (2008) present an approach that puts an additional service between a client and a REST API. In this service, developers implement selection strategies to redirect the client request to the correct REST API version of the server support. However, this approach needs a stand-alone service deployed, not integrated with the REST API source code. We will discuss our solution in comparison with these solutions in Section 3.5.

For **"Server Timeout"**, Eastbury et al. proposed a design[6], which we extend to maximize its benefit for REST API developers in Section 3.6. Also, this good practice could solve the problems mentioned by Dai, He, Gu, and Lu (2018), which includes misused timeout, improper timeout handling, unnecessary timeout, and clock drifting in various cloud systems.

"Response caching" is a common good practice. Rabinovich and Spatscheck (2002) explained the basic concept and architectural solution of Web caching. Wessels (2001) explained the caching mechanism using a proxy server. Both of the examined Web frameworks have multiple built-in caching techniques, which we discuss in Section 3.8.

For **"List Pagination"**, both Google[7] and Microsoft[8] suggest that REST APIs returning lists should use pagination. Masse (2011) also states that collections should be returned in chunk. Murphy, Alliyu, Macvean, Kery, and Myers (2017) shows that pagination was proposed in 24 over 32 REST API company guidelines. Both of the examined Web frameworks have partial support for pagination, which we discuss in Section 3.9.

[6] https://docs.microsoft.com/en-us/azure/architecture/patterns/async-request-reply

[7] https://cloud.google.com/apis/design/design_patterns#list_pagination

[8] https://github.com/microsoft/api-guidelines/blob/vNext/Guidelines.md#98-pagination

2.3 Summary

In general, previous work primarily defined and detected good and bad REST API practices. Only a few authors proposed and validated solutions to these practices, with some limitations. For each practice, we summarise and compare these solutions with our own. In addition, we also provide concrete implementations in two popular frameworks (ASP.NET and Java Spring). Finally, we provide sample implementations and supplements for "Response Caching" and "List Pagination" good practice, supported by the frameworks.

Chapter 3

REST API Anti-patterns

3.1 Categories of Good and Bad Practices of REST APIs

We categorize practices into two categories: technical and non-technical. The technical category includes practices that can be solved by or made conformed via a programmatic solution or some Web framework's features. The non-technical category includes practices that require developers' effort to conform. Usually, these practices are domain- or business-specific. For example, the URI structures represent the relationships between the nodes to avoid the "Non-hierarchical Nodes" bad practice. Yet, companies disagree on constructing URIs showing this relationship and even discourage nesting structures. Another example, for the "Using the wrong HTTP Verbs" bad practice, IBM only mentions the verbs `GET` and `POST` while Google uses `GET`, `POST`, `PUT`, `DELETE` and invents some new verbs, like `LIST` and `MOVE` (Murphy et al., 2017). Therefore, we focus only on technical practices.

For each practice in the technical category, we propose an architectural solution in Section 3, which should be simple but guarantee conformance to the good practice and require little effort to implement.

Table 3.1 shows the division between technical and non-technical practices. The practices with (+) have built-in (partial) solutions in the examined frameworks, which we discuss and compare to our solutions in Section 3.

Table 3.1: Categorizing REST API practices

Technical	Non-technical
Content negotiation (+)	Entity Endpoint
Endpoint redirection	Contextless Resource name
Entity Linking	Non-hierarchical Nodes
Response caching (+)	Amorphous URIs
API Versioning	CRUDy URIs
Server Timeout	Singularized & Pluralized Nodes
POST-PUT-PATCH Return (+)	Non-pertinent Documentation
List Pagination (+)	Breaking Self-descriptiveness
	Ignoring status code
	Using the wrong HTTP Verbs
	Misusing Cookies

Besides existing solutions, we examined 23 software design patterns for object-oriented programming (Gamma, Helm, Johnson, & Vlissides, 1993) to reuse their practical implementations for each of the eight good practices in the technical category when possible. In some REST API practices, we need to modify the selected design pattern to fit with the REST API frameworks. If no design pattern could solve the problems, we extend the search to other resources and the gray literature (i.e., blog posts, technical tutorials, StackOverflow, etc.).

The examined Web Frameworks already have built-in features for "Response Caching" and "List Pagination". Therefore, we only present these solutions and provide sample usages. For "Content negotiation", we compare our solutions with the built-in features. Developers can choose to use the solutions that fit with their projects based on their advantages and disadvantages.

We present each of the practices following the same pattern:

(1) Practice name

(2) Problem statement

(3) Expected result/output

(4) Solution

(5) Sample implementation/source code

Visit the GitHub repository[1] to explore the sample source code implementations.

[1] https://github.com/huntertran/restapi-practices-impl

3.2 Content Negotiation

3.2.1 Problem

The client can only process and manipulate the resources in some formats. An example is the JSON and XML format that the server should support. JSON is faster to parse and smaller to transport over the internet, but XML format supports namespaces, comments, and metadata.

3.2.2 Expected result

We expect:

- The resources of the same type should be served in various formats (e.g., JSON and XML, image file formats and Base64 encoded).
- The server should set a default format if the client does not specify the requested format.
- The implementation of each data format should be easily modifiable. Developers can add a new implementation for a new data format easily.

3.2.3 Solution

Based on the request header, the server prepares the data in the requested format and then returns the data in the response body. We could use factory design patterns (Gamma et al., 1993) to initiate the object in the required format. For each set of required formats, there should be a corresponding concrete Factory (i.e., XML and JSON will have a concrete factory. PNG, JPG and Base64 will have another concrete factory). Figure 3.1 illustrates this solution.

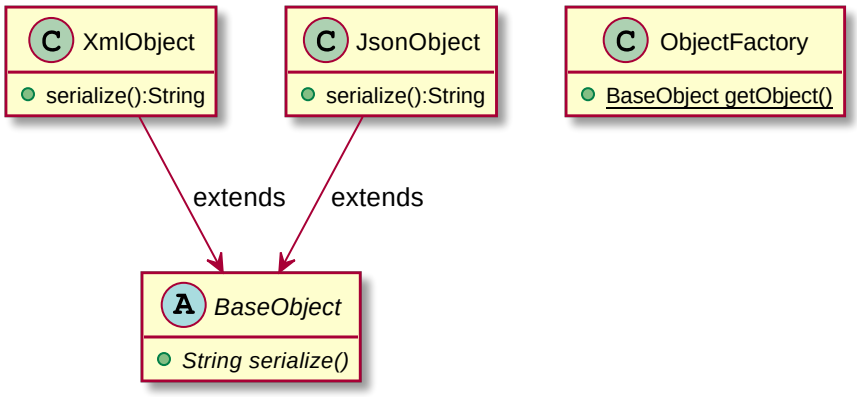

Figure 3.1: Class diagram for Content Negotiation solution

In the `ObjectFactory` class, developer could set the default format by using the `default:` clause of the `switch` statement.

3.2.4 Source code

Abstract class for all formats of object

```
public abstract class BaseObject {
    public abstract serialize();
}
```

Classes for the object with the preferred format:

```
import BaseObject;

public class XmlObject extends BaseObject {
    @Override
    public String serialize() {
        String result = "";
        // XML serialization logic
        return result;
    }
}

public class JsonObject extends BaseObject {
    @Override
    public String serialize() {
        String result = "";
        // JSON serialization logic
        return result;
    }
}
```

Factory class:

```
public class ObjectFactory {
    public static BaseObject getObject(String type) {
        switch(type) {
            case "application/xml": {
                return new XmlObject();
            }
            case "application/json":
            default: {
                return new JsonObject();
            }
        }
    }
}
```

Usage:

```
import ObjectFactory;
import BaseObject;

public class Controller {
    public String requestHandler(String requestedMIMEType) {
        BaseObject result = ObjectFactory.getObject(requestedMIMEType);
        // compute the result
        return result.serialize();
    }
}
```

Both Java Spring and ASP.NET core support content negotiation with the default set to `JSON` format. Table 3.2 compares content negotiation implementation in popular frameworks.

	Java Spring	**ASP.NET core**	**Our approach**
Common media types	Yes	Yes	Yes
Customizable serializer	No	Yes	Yes
Do not require data annotation on model	No	Yes	Yes
Built-in support ignorable	Yes	Yes	N/A

Table 3.2: Content Negotiation feature comparison

ASP.NET Core has the most flexible support for Content Negotiation. Developers can override the format serializer to better fit with their business. For Java Spring, developers cannot override the serializer, but they can combine multiple approaches to achieve the desired effect. The XML format in Java requires data annotation to be added to the model class to work correctly. This limitation sometimes changes the designed data structure.

3.3 Endpoint redirection

3.3.1 Problem

When the data structure changes or developers refactor the URI structure, resources can be moved to new locations. However, a client could request the resource using the old URIs. Then the server should return these requests with HTTP code 3xx and the new location.

The most obvious way to resolve this problem is to modify the old controller class to return the new location with a 3xx status code. This action will violate the *open-closed principle* (Martin, 2002). In addition, developers cannot roll back to the previous version of the API quickly if they do not have backups like a source control system or a container ready to be deployed.

3.3.2 Expected result

Most of the REST APIs frameworks use the Model-View-Controller (MVC) pattern. The solution should be on top of or integrated with the MVC pattern. In addition, the solution should satisfy the following:

- There should be a class that handles redirection logic, separated from other classes or controllers.
- The redirection logic class should have the same interface as the old controller class.
- The new controller class should extend or include the redirection logic but still conforming to the *Single Responsibility Principle*.

3.3.3 Solution

There are two possible solutions for this good practice. Each solution will have advantages and disadvantages.

Solution 1: Extend from a `Redirector` class

In this solution, the old controller and the redirector will implement the same interface. The new controller will extend the redirector class. The methods in the new controller and the redirector

will be exposed as API endpoints to clients. The solution is illustrated in Figure 3.2.

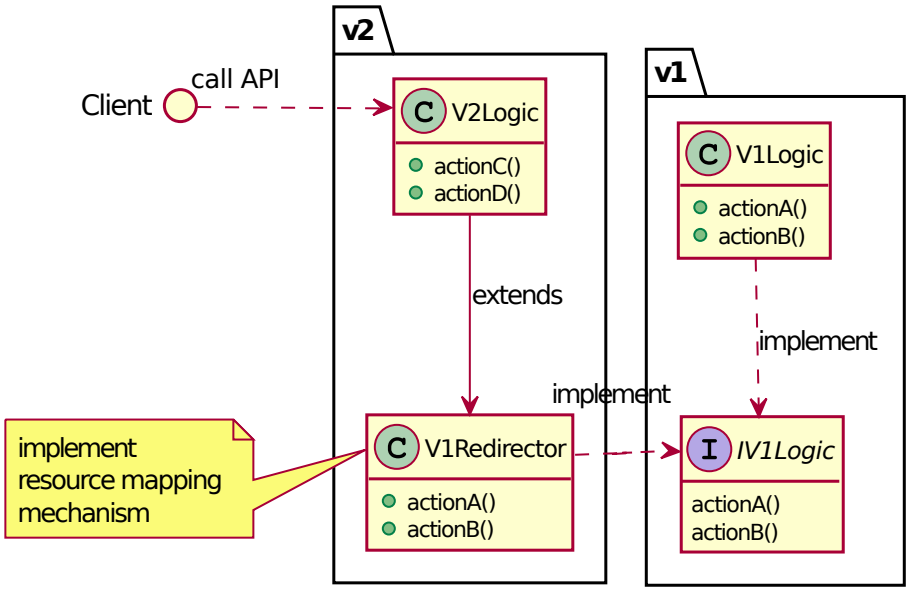

Figure 3.2: Class diagram for Extend from a Redirector class

Solution 2: Nested class inside the new controller

The redirection logic is still separated into a stand-alone class, implementing the interface of the old controller class. The difference is that the redirection logic class is a nested class inside the new controller. This solution allows the redirector to access methods and variables in the new controller.

The new controller implements the interface of the old controller and contains a private instance of the redirector class. The interface implementation makes sure all the old methods were handled correctly. The private instance works as a proxy to the actual logic of the redirector class. The solution is illustrated in Figure 3.3.

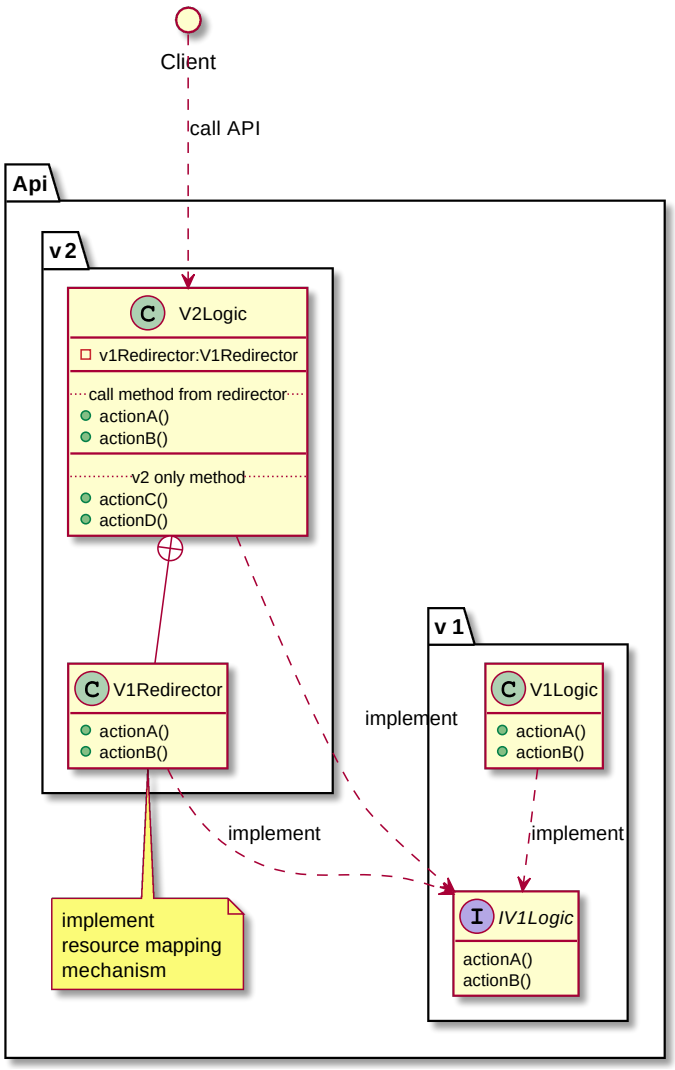

Figure 3.3: Class diagram for Nested redirector

Comparison between both solutions

Table 3.3 depicts the comparison for both solutions. Developers can choose to implement either one of them based on the business of the system.

	Extend	Nested
Pros	Redirection logic were separated in class and file	Allow the outer class (the main class) be inherited from another class
	Easier to implementation. Can be implemented in multiple languages.	The Redirector has access to resources in the main class
Cons	The Redirector cannot access resources in the main class	Redirection logics are coded inside the main class
	The main class cannot inherit from other class (if the language did not support multiple inheritance)	Not all languages support nested classes

Table 3.3: Comparison of two solutions for Endpoint Redirection good practice

3.4 Entity Linking

3.4.1 Problem

Entity Linking good practice has a corresponding bad practice called Forgetting Hypermedia (Palma, Dubois, Moha, & Guéhéneuc, 2014; Tilkov, 2008). Developers need to find out programmatically the link to the resources that are related to the current request. For example, when designing service for the content management system (CMS), after sending a `GET` request to retrieve a post, in case of conforming Entity Linking good practice, the server should return the post details, including the link to comment and likes.

```
{
  "post": {
    "title": "Lorem ipsum",
    "content": "Lorem ipsum",
    "links": [
      {
        "rel": "comment",
        "method": "post",
        "uri": "/post/123/comment"
      },
      {
        "rel": "like",
        "method": "get",
        "uri": "/post/123/like"
      }
    ]}}
```

By examining the response above, developers know that that they can make a request to post a new comment or to get the like of the post. In case of `Forgetting Hypermedia` bad practice, the object `links` in the response is not available. Therefore, developers do not know if they can post a new comment or get the like of the post or not. They have to make these requests to the server to find out. If they cannot due to some reason, e.g., the client was not authenticated, the server will refuse the request with 4xx HTTP codes.

3.4.2 Expected result

The solution should allow the system to do the following:

- The current action should have access to the instance of other controllers that contain the related resources so that the system can check the availability of these resources.
- The current action should have access to class and method information (e.g., name, annotation, public and private variable). This is required to construct the URI since popular REST API frameworks use naming conventions or annotation to construct the URIs of the API.

3.4.3 Solution

The method that handles the current request will have a list of related classes containing the related resources. To loop through the list, all these classes should be an implementation of an interface. Accessing methods implementing the same interfaces in different classes, based on the type of the class (i.e., dynamic biding), is called *double-dispatch* (Paepcke, 1993). Unfortunately, mainstream programming languages that were used by REST API frameworks did not support this mechanism. To address this problem, we could use a modified version of Visitor pattern (Martin, 2002) with language reflection features to access data annotation.

All the controllers that are intended to be used for populating links for the related resources will be the implementation of the `LinkedResource` interface. This interface has a method `accept()` that accept a visitor instance of type `ResourceVisitor`. For each logic to select the resource to be included, a separated concrete `Visitor` will be created. These visitors could share

the common logic in abstract class `CommonResourceVisitor`. Developers can hide the complexity of language refection logic here. Figure 3.4 illustrates the solution in UML class diagram.

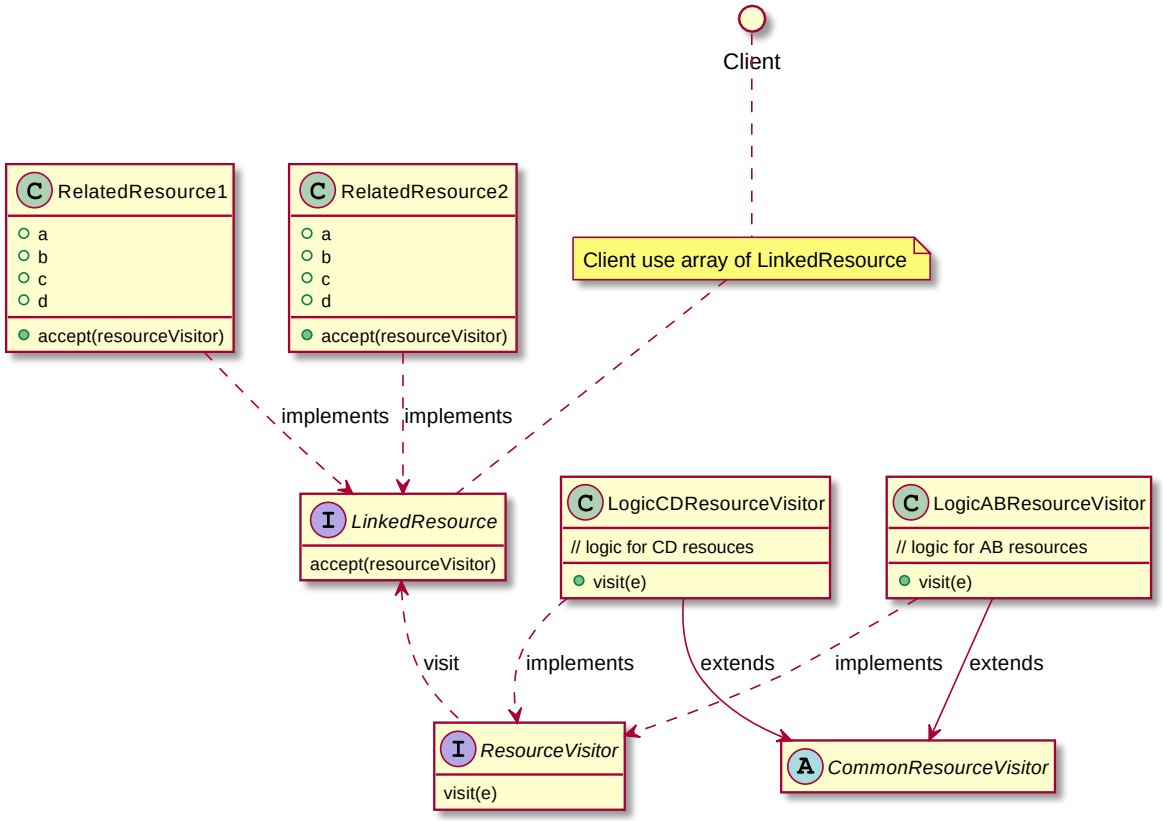

Figure 3.4: Class diagram for Entity Linking solution

However, this solution is only applicable with programming languages that support reflection. If a language does not support reflection, then, based on the way Web frameworks construct URIs, developers must find a way to programmatically extract the URIs to satisfy the expected results, as mentioned in Section 3.4.2

The sample implementation with reflection is available for Java Spring and ASP.NET Core (see `https://git.io/JGRWW` and `https://git.io/JGRW8`).

3.5 API Versioning

3.5.1 Problem

REST API systems, or more generally software, evolve. In traditional software, the developers can distribute the new version while the old ones continue working well. However, when changes were deployed to a REST API system, they may break the client applications. The client application developers may not be aware of it nor have enough time to adapt to the changes.

In addition to communicating and advertising the new version of the API, the REST API developers need to support the old API for a time in parallel with the new one. It allows the coexistence of multiple versions of the API. To separate the old API from the new one, the developers can choose one of these approaches:

- Include the version of the API in the URI:
    - Version the API globally for all resources
    - Version the API separately for each type of the resources (e.g., a school API, the student resources can be at version 3, while the teacher resources are in version 2)
- Include the version in the header(s). It could be the `accept` header or a custom header invented by the REST API developer.

There are some challenges when evolving a REST API system, such as transformation between JSON and XML, M to N Mapping, delete method, authorization protocol change, rate limit, and authorization of API access (Li, Xiong, Liu, & Zhang, 2013)

3.5.2 Expected result

We expect that:

- The URIs of a version should not change over time. The client application will always have the expected result when requesting a resource during the lifetime of that version.
- Client applications can access multiple versions of the API. In other words, the REST API system can support multiple versions at the same time.

- The REST API system can use a single database. If there are breaking changes in the database, there should be a mechanism to translate the changes to an understandable object for the old version.

3.5.3 Solution

Kaminski et al. (2006) proposed the "Chain of Adapters" design technique to address this problem, as we mentioned in Section 2.2. We present our solution in the following.

We propose using the Proxy design pattern to redirect/convert the request to the old version of the API. For each version of the exposed API, there should be a corresponding interface. The interface does not serve a purpose in the current version but will be the contract for the next version. The class will contain a private instance of each old API class that supports this next version. Thus, all the API versions share some common logic that won't change over time. This logic can be separated into an abstract class. The API controllers inherit from it. Figure 3.5 illustrate this solution.

The REST API developers could use this solution in multiple scenarios. For example, when there is a security fix that was implemented in the new version. The developers can capture the request to the old version and convert the parameters to be compatible with the latest version. They then use these converted parameters to execute the fixed method in the new version and return the result. Another scenario is that when most of the logic of the latest version is the same as the old one. There are only a few changes that were implemented in new methods. Developers could call the methods in the old version by using the private instance of the old class declared in the new class.

One advantage of our solution is that the new version only implements the interfaces of the old version if that version needs to be supported. For example, if there is version three, and this version should support both versions two and one, then controllers of the version three class will implement the interfaces of versions two and one. This approach eliminates the need of "chain of adapter" and allow developers to support versions that are not ordered in time, e.g., version four can choose to support version one and three, remove the support for version two, compared to the solution proposed in (Kaminski et al., 2006). Furthermore, our solution is implemented inside the application, differently from the solution proposed in (Leitner et al., 2008).

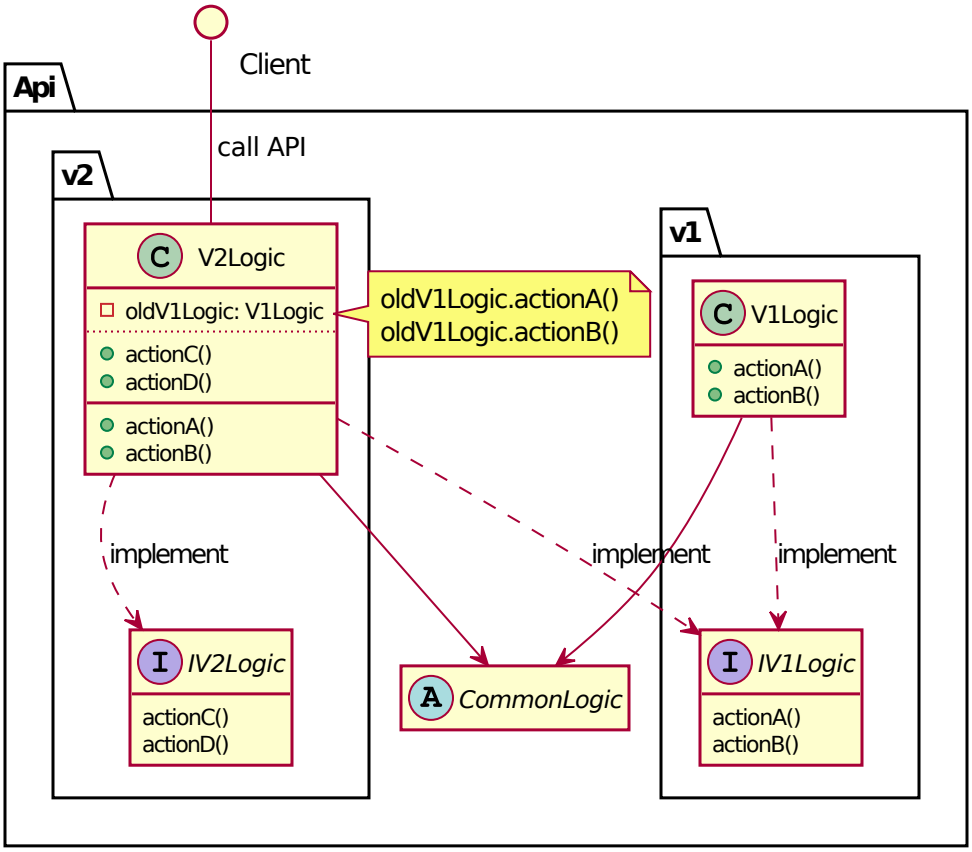

Figure 3.5: Class diagram API Versioning solution

3.6 Server Timeout

3.6.1 Problem

When there is a long-running operation on the REST API server, the client needs to wait to receive a response. In a traditional system, the communication between the running operation and the object that consumes the operation result is permanent. However, in a REST API system, the servers communicate with the consumers (the client applications) through the network. This introduces some potential problems: if the client and the server disconnected, the operation continues to run on the server, but the result is no longer needed. The same if the client cancels or abandons the request. These problems will waste the server's resources.

3.6.2 Expected result

The long-running operation should have a mechanism to cancel itself if the result was not needed. To indicate the cancellation, the server could use a timeout or the client request to cancel the operation. By doing so, the server could release the resources held by the process.

3.6.3 Solutions

Two solutions could solve the problems. The combined implementation of both solutions produces the best result but with additional complexity.

Solution 1: Timeout

The long-running operation will have a timeout predefined by the developer. When the time is up, the operation will be canceled. However, the developers need to set the time for the operation to run. In some cases where there are unpredictable factors, this is not practical. For example, if the operation depended on the data retrieved through the network, but the speed is not stable, it will be challenging to set a fixed time.

In Java, developer could use `ExecutorService` to start a new single thread executor. Then they can submit a long running operation wrapped in a `Callable`, since Java 1.5. In ASP.NET Core, developers could use `TimeOutAfter`[2] or `CancellationTokenSource.CancelAfter`[3] in .NET 5.0 or later

Solution 2: Asynchronous Request-Reply pattern (HTTP Polling)

This solution was proposed by Eastbury (2019). The solution will require some extra work from both server and client developers. Developers need to implement an operation status checker and a cache system to store the operation result on the server. On the client side, the developers need to implement a polling mechanism to periodically get the operation status and finally get the response from the `Resource Endpoint`. Figure 3.6 illustrates this process.

[2] https://git.io/JGRWB
[3] https://bit.ly/3vDwdB1

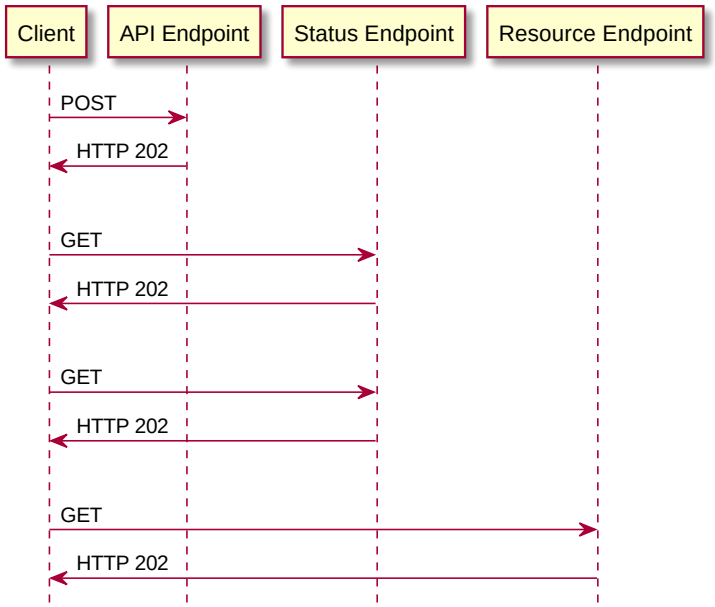

Figure 3.6: Asynchronous Request-Reply process

(1) When the client sends the request that triggers a long-running operation, the server will respond as quickly as possible with `202 Accepted` HTTP code.

(2) The server continues its long-running operation while the client is periodically polling for the result.

(3) When the result is available, the polling will return with `302 Found` HTTP code with the URI for the result.

(4) The client sends a `GET` request to get the result.

Comparisons between the two solutions

Each solution has some advantages and disadvantages. These points were listed in table 3.4

Table 3.4: Comparison between Timeout and Asynchronous Request-Reply

	Timeout	**Async request-reply**
Pros	No requirement for client side	Client and server can re-establish connection after disconnected
	Easier to implementation	The same result can be served instantly
	Single system involved	
Cons	Risk of incorrect predefined timeout value	Need implementation in both client/server
		Need a mechanism to store calculated data
	If the client disconnect early, the server still wasting resources until timeout	

Combination of both solutions

Developers can choose to combine both approaches to maximize the benefits, especially when they choose to follow the Asynchronous Request-Reply pattern since they only need to add a bit extra work to combine both solutions. There are three stages:

(1) Initiate.

(2) Polling.

(3) Termination.

In the Initiate stage, the client sends a request to trigger the long-running operation. In addition to the HTTP Polling process, the server registers the operation status. In the Polling stage, the client periodically polls the long-running operation. With each polling request, the server resets the timeout counter. In case of disconnection between client and server, the server timeouts and aborts the long-running operation and releases any resources. Figure 3.7 illustrates this process.

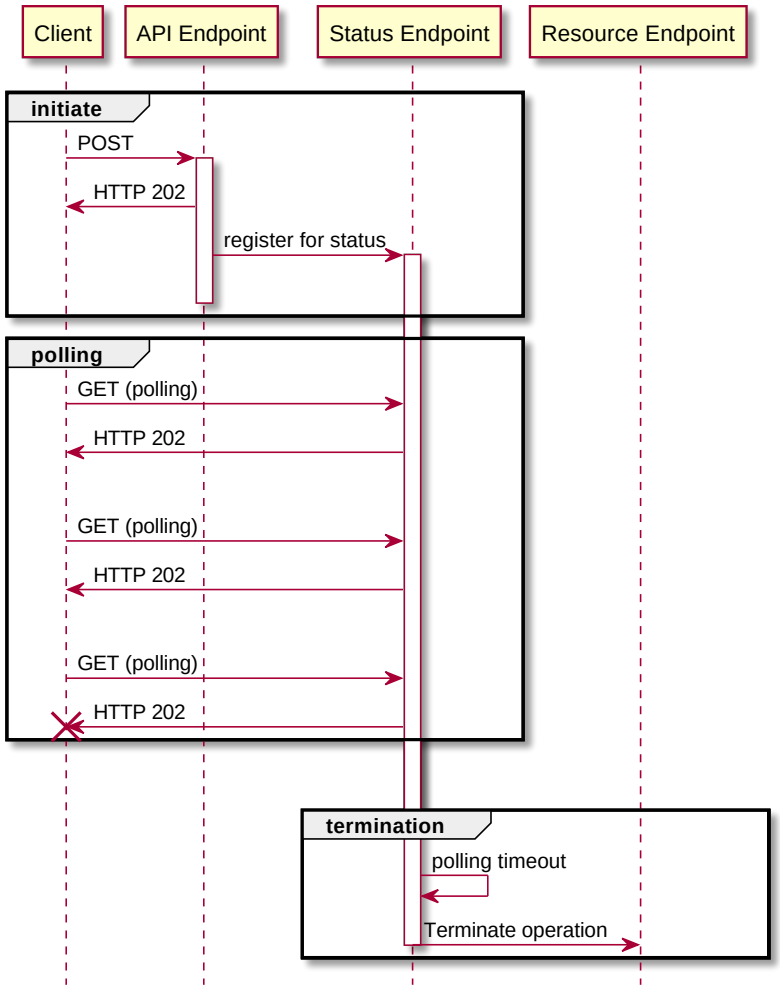

Figure 3.7: Combined solution with HTTP Polling and Server timeout

This pattern is only suitable for long-running operations because it adds complexity to the code. In addition, developers must implement a mechanism to store the computed results. The storage could be a database, a cache, or in the local memory of the server.

3.7 POST-PUT-PATCH Return

3.7.1 Problem

When the client application sends a request to modify the database (Create, Update, Delete), usually the server only returns a simple result indicating successful request like HTTP status code

`200 OK`. There is no way for the client application to know the added/modified object was correctly committed to the database unless the client makes a new request for the object.

For example, when there is a mismatch datatype between the model classes and the data posted by the client application, the server may still work by simply ignoring any mismatched data and write everything else to the database. Then, the server signaled that the writing process was successful. However, the client has false information, where the data it posted was not the data written to the database. In any case, sending the created/modified object in the response body sometimes could cost valuable transmission time and network bandwidth, especially when the client indicated that it would not need to confirm the committed data in the database.

The name "POST-PUT-PATCH return" is based on the HTTP verbs `POST`, `PUT` and `PATCH`. These verbs should be used when a request can add or modify an object in the database. We do not add `DELETE` because after the object is deleted, the client only needs a simple success status.

3.7.2 Expected result

We expect that:

- A client side mechanism to control the response if it only needs a `200 OK` HTTP Status code.
- Separation of database manipulation logic with business logic.
- Minimal requests to the database.

3.7.3 Solution

To separate the database manipulation logic from business logic, we could use the Repository pattern. To make sure a database transaction was successfully executed before return the result to the user, we could use the Unit of Work pattern[4].

After implementing these services, the system will use them frequently. To avoid the usage of the Singleton pattern, developers could implement Dependency Injection. The Singleton pattern could be beneficial for some specific use-cases in traditional software systems. For example, the map of a game could implement the Singleton pattern. However, in a Web application, in which

[4] https://bit.ly/3uCO9du

multiple users interact with a server, using the Singleton pattern could lead to conflicts or unexpected behaviors. In general, there should be a repository class that handles CRUD operations for each resource or a group of related resources. This class will be injected into Controllers using Dependency Injection. In addition, the Unit of Work pattern will be integrated into each repository. Figure 3.8 illustrate this solution.

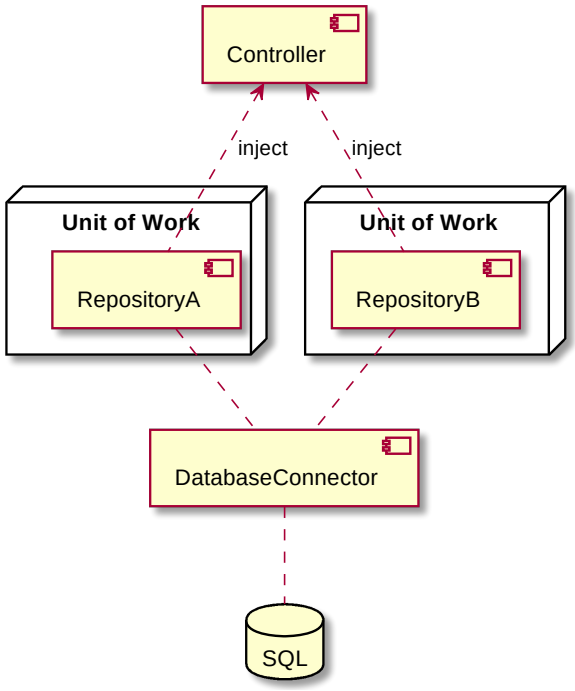

Figure 3.8: Repository pattern

Java Spring has a Repository pattern built-in with the basic CRUD operations (see https://bit.ly/3c6GFcs). ASP.NET Core does not have this function built-in. Still, there is an instruction on how to implement them (see https://bit.ly/3uCO9du). In addition, both frameworks support Dependency Injection and Unit of Work. The Unit of Work with ASP.NET framework was supported in a separate library called Entity Framework that works like an Object Relational Mapping (ORM) framework. Sample implementations exist for Java Spring (see https://git.io/JGRWE) and ASP.NET Core (see https://git.io/JGRWg).

3.8 Response caching

3.8.1 Problem

Response to a request in a REST API system can be expensive in terms of resources. For example, the client can request a complex object that must be retrieved by joining multiple tables in the database and extensive calculations that take a long time. Another example is when the clients request an image that they frequently use in the client application. There are some resources specific to the application's domain, like decoding and encoding in real-time trading systems (Kohlhoff & Steele, 2003), or images and videos in social networking systems.

In general, to increase the performance of a REST system, objects often needed but not changed frequently should be stored in a fast place: easy to retrieve and serve to the client, usually a cache system.

3.8.2 Expected result

- Fast and easy to set or retrieve a cached record.
- The cache system should have expiration management.
- There are mechanisms to forward the requests to the actual server in case of the object is not cacheable or not existed or expired in the cache (Iyer, Rowstron, & Druschel, 2002).

3.8.3 Solutions

Developers could put the caching on the server-side or client-side. We will discuss server-side caching only in the scope of this thesis. There are multiple ways to implement server-side caching: **In-memory caching** and **Proxy server caching**. Depending on the business requirement of the REST API system, the developers could choose to implement one of them. A mixed implementation is possible but would be challenging to maintain.

In memory caching

In-memory caching puts the cache implementation directly in the application server. This cache mechanism is fast (Narumoto, n.d.), but the size of the cache memory is usually minor compared to other caching implementations since the cache uses the server memory directly. In-memory caching is difficult to scale because it will require scaling up the whole REST API system server.

Proxy server caching

Developers implement caching in the proxy server. This approach is easier to scale because of the loose coupling between the proxy server and the REST API server. In addition, developers could choose/swap multiple caching technologies based on their needs. However, proxy server caching is more complicated to deploy because it involves multiple servers and configurations.

Framework supports

Popular web frameworks support the implementation of caching for both In-memory caching and proxy server caching. The frameworks make implementing caching much simpler. The cache logic usually is implemented using the Repository pattern. Developers use the same source code but choose the cache techniques (In-memory or proxy server) by a configuration file.

For example, Java Spring, developer can implement a simple, auto caching by just adding `spring-boot-starter-cache` maven package, and annotate the methods in repositories with `@Cacheable` and `@EnableCaching` in the main class[5]. For ASP.NET Core, there are detail instruction[6] on how to implement caching by using `IMemoryCache` and `IDistributedCache` interfaces.

[5] https://spring.io/guides/gs/caching/
[6] https://bit.ly/3p9rCUM

3.9 List Pagination

3.9.1 Problem

In the REST API system, any API endpoint that returns data type as a list should support pagination, even if the list is typically small. Developers do not know if a list will grow in the future. Adding pagination support for a list after the API was published is not a good idea since the clients already assumed they have all items, while they only have the first page (Google, n.d.).

3.9.2 Expected result

Pagination in the REST API system should be:

- Easy to implement
- Only need two parameters: number of items per page and page number

3.9.3 Solution

Only `READ` operation in `CRUD` need pagination. The developers could hide the complexity of pagination using the Repository pattern proposed in (Lalanda, 1998). Fortunately, the popular frameworks already support pagination in some ways. For example, Java Spring has `PagingAndSortingRepository` interface that helps developers quickly implement pagination with the help of `Pageable` interface.

```
1 Pageable pageable = PageRequest.of(pageNumber, itemPerPage);
2 Page<your_model_class> page = transactionRepository.findAll(pageable);
3 return page.getContent();
```

For ASP.NET, the pagination capability was built-in with Entity Framework Core, the open source ORM by Microsoft, by using `Skip` and `Take` methods on a collection of type `IQuerytable`. The use of these method require some conversion of the parameters: `Skip(itemPerPage * pageNumber)` and `Take(itemPerPage)`. Then developers can use it as follow:

```
var result = _dbContext.transaction
                        .Skip(itemPerPage * pageNumber)
                        .Take(itemPerPage);
```

For other frameworks that do not have ORM libraries, the developers could use the `LIMIT` and `OFFSET` clause of SQL, depending on the database technology. For example, for SQL Server:

```
SELECT *
FROM your_table
ORDER BY column_name ASC
OFFSET number_of_skipped_row ROWS FETCH NEXT max_number_of_row ROWS ONLY;
```

3.10 Summary

In this chapter, we classified the REST API practices into technical and non-technical categories. Using an architectural design can ensure the conformation of implementations to the practices in the technical category. Therefore, we proposed solutions to six of the eight practices in this category, except for "Response Caching" and "List Pagination", as the Web frameworks fully support them already. For each practice, we presented them using the structure: (1) Practice name, (2) Problem statement, (3) Expected result/output, (4) Solution, and (5) Sample implementation/source code. Please visit the GitHub repository at https://github.com/huntertran/restapi-practices-impl to explore the sample source code implementations.

Chapter 4

Evaluations

Like other practices, e.g., design patterns, code smells, etc., the eight technical practices are partly subjective due to the lack of agreed-upon quality models for software systems in general and service-oriented systems in particular.

Therefore, although these practices come from a consensus in the academic/professional communities, our solutions must be validated by professional developers, who only can confirm that our solutions:

(1) Solve the bad practices.

(2) Conform to the good practice.

(3) Are acceptable in practice.

Consequently, we designed a validation survey and administrated it to 55 professional REST API developers. We explain the following method that we followed and the results that we obtained, which show that our solutions indeed remove the bad practices and are acceptable by professional developers. (We received an ethics certificate from the Office of Research of Concordia University, number **30015039**.)

4.1 Overview

We follow the "Questionnaire Survey" empirical standard proposed by the ACM (Ralph et al., 2020). All the material for this study is available for analysis and replication at one of the following locations:

- http://www.ptidej.net/downloads/replications/icsoc21/
- https://github.com/huntertran/restapi-practices-impl

We build the survey in sections. Each section contains a single practice, with problem identification, a short explanation for the good practice and the concrete implementation in ASP.NET core and Java Spring. Section 4.2 below will explain in detail the survey design.

According to Deutskens, De Ruyter, Wetzels, and Oosterveld (2004), we expected 17% response rates for the survey. To increase the response rates, we split the survey into two Parts A and B. In each part, we include only questions for four good practices. To maximize the chance for each good practice to be evaluated, we randomize the order of the good practices in each part.

To attract more participants, we include a lottery incentive in the survey, in the form of a lucky draw for the prize of two $50 Amazon gift cards, four $10 Starbucks gift cards in Canada, or equivalent prizes in other countries.

4.2 Survey Design

At the beginning of the surveys, we ask for demographic information, which are all optional questions: age group, education, current profession, and employment status.

For each of the good practices in the technical category, except "Response caching" and "List Pagination" (see below), we present the problem, the good practice to avoid the problem, and the concrete, practical implementation. Then, we ask two questions:

(1) Did you face this/these problem(s) in some of your projects?
(Multiple choices: Yes/No/Other (Please specify)).

(2) Is the proposed solutions a good solution?

(Multiple choices: Yes/No/Other, I have some suggestions or another solution)

Using these two questions, we can know:

(1) Popularity of the problem in the industry.

(2) How good is the proposed concrete implementation of the good practice.

(3) Is there any other implementations.

For "Response Caching" and "List Pagination", because the popular Web Frameworks support them, we ask the following questions:

Response Caching:

(1) What caching strategy your projects are using?

(Multiple checkboxes: In-memory caching/Distributed caching/None/Other)

(2) Do you follow the caching mechanism instruction or modify the mechanism to better fit with your business?

(Multiple choices: Modified to fit with business/Follow instruction/No caching/Other)

Using these two questions, we can know (1) The most popular caching mechanism and (2) How well the web framework supports caching mechanism.

List Pagination:

(1) Do you implement List Pagination in API endpoints that return list or collection?

(Multiple choices: Yes, in all endpoints/Yes, in some endpoints with a long list or collection/No, I don't implement list pagination).

(2) Please explain your choice.

(Multiple rows textbox, optional)

Using these two questions, we can know:

(1) If List Pagination good practices are generally accepted by developers

(2) The rationale behind their choice.

4.3 Participants Selection

We selected the participants who are:

- Adult.
- Developers or students in CS/SE.
- Using OOP languages.
- Working or used to work in the industry.

We recruited the participants to the studies through the convenient sampling of software developers and engineers through e-mail lists, social media (LinkedIn, GitHub, Facebook, and Twitter).

4.4 Survey Administration

The survey's answers are anonymous, and we only discuss aggregated results below to support our research. The course students' participation is voluntary only. They are anonymous or coded as for other participants. The involvement of students does not affect their academic standing or course grades. This information is indicated in the recruitment material. If a participant withdraws before the end of the survey, there are no consequences. The data is collected electronically through an online questionnaire using SurveyMonkey.

The collected data is thus first stored in the database of the online survey tool. Then, we downloaded the data on the computer of one (and only one) of the research assistants and erased it from the online survey tool. Then, we analyzed the data following the research procedures established before creating the questionnaire, recruiting participants, and collecting the data. Finally, we transferred the data to the principal investigator's office computer for storage and safekeeping for five years.

The information is stored on international servers and housed by service providers in the USA. Therefore, we can assure confidentiality only when data is accessed/requested by local authorities.

We send our survey to 55 professional developers. Out of 55, 51 developers completed at least one of the surveys. We received 68 completed surveys (Parts A and B). The number of completed

surveys is greater than the number of participants because some participants completed Parts A and B. We received 17 incomplete responses. Because of the randomized orders, we can still extract valuable data from 17 incomplete responses. Thus, the average completion rate of both surveys is 76%.

We extracted and analyzed the completed parts for incomplete survey responses, whose questions were all answered by the participants. In addition, our survey design allows separating between parts easily.

We use the formula of **Standard deviation for the binomial distribution** for calculating the Standard deviation, described as following:

$$\sigma = \sqrt{n \times p \times (1 - p)}$$

where n is sample size, p is "Yes" answer proportion.

4.5 Participants' Demographics

Figure 4.1, 4.2, 4.3, 4.4 and 4.5 summarise the demographic information from the participants.

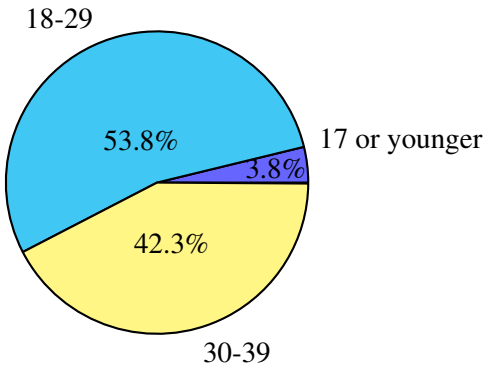

Figure 4.1: Participant's age groups

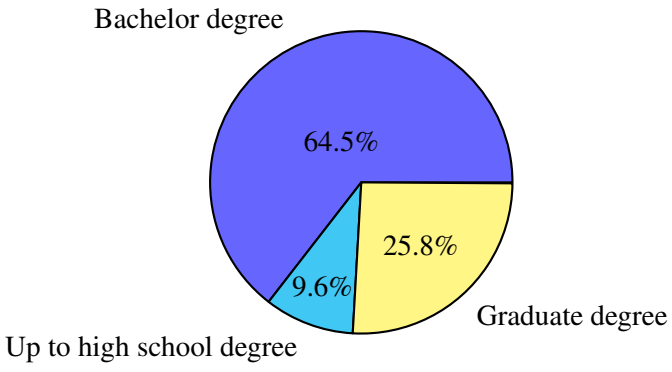

Figure 4.2: Participant's education

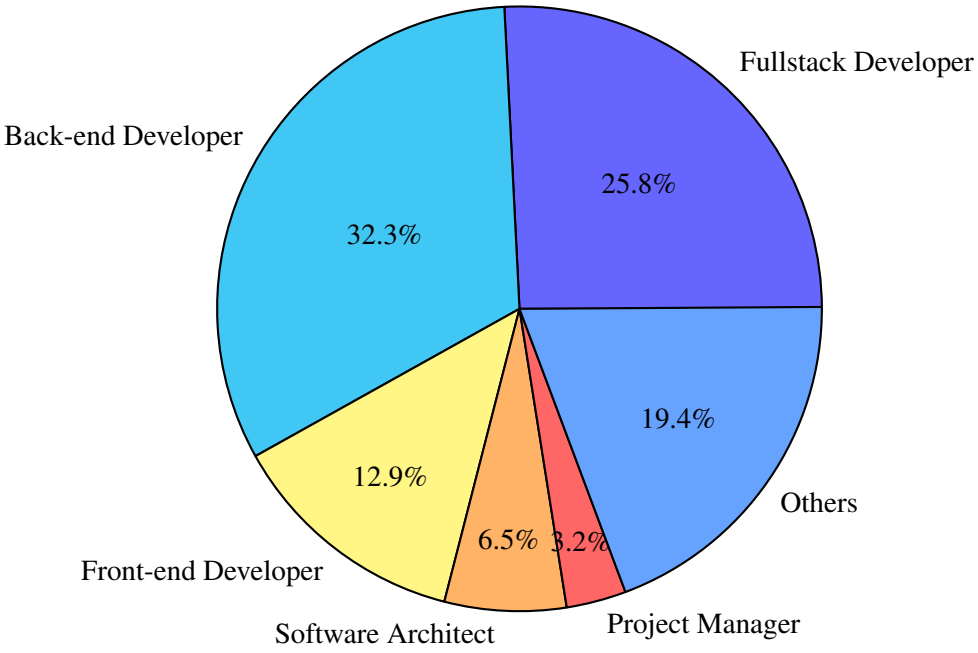

Figure 4.3: Participant's profession

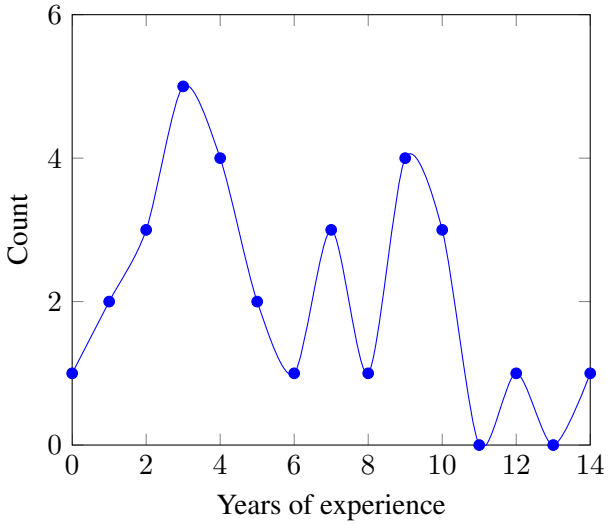

Figure 4.4: Participant's years of experience

The other professions are Product Manager, Data Engineer, Network Engineer, and Project Associate.

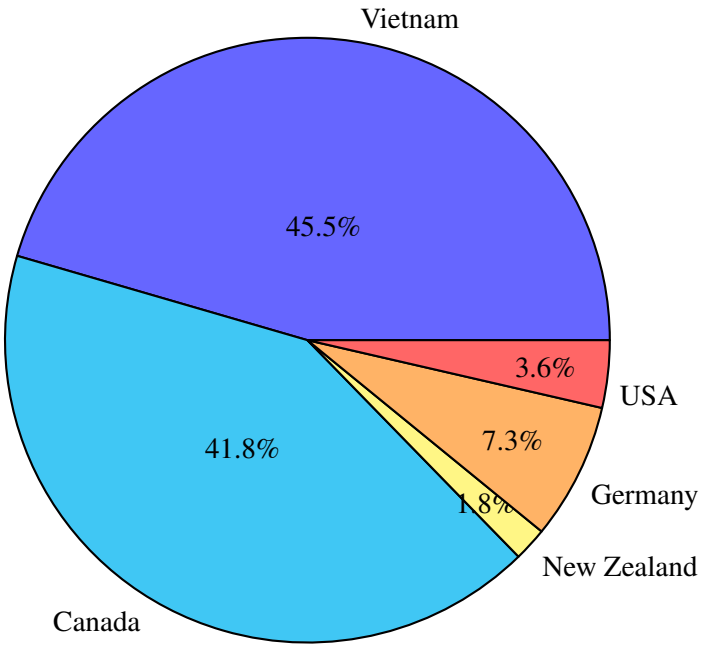

Figure 4.5: Participant's country of origin

4.6 Quantitative Analyses

Table 4.1 shows the percentage of positive responses, excluding "Response Caching" and "List Pagination".

Table 4.1: The positive results statistics of the survey

	Face this problem	Std. dev.	Good solution	Std. dev.
Content Negotiation	52.4%	2.289	76.2%	1.952
Endpoint Redirection	45%	2.225	75%	1.936
Entity Linking	47.4%	2.177	57.9%	2.152
API Versioning	72.2%	1.901	72.2%	1.901
Server Timeout	77.8%	1.763	66.7%	1.999
POST-PUT-PATCH return	58.8%	2.029	82.4%	1.570

For "List Pagination", despite the suggestion that all API endpoints that return a list should be paginated, the developers' majority choose to implement "List Pagination" in some endpoints only. Figure 4.6 illustrates the percentage of these choices.

For "Server Caching", most of the developers use both of the caching mechanisms. In addition,

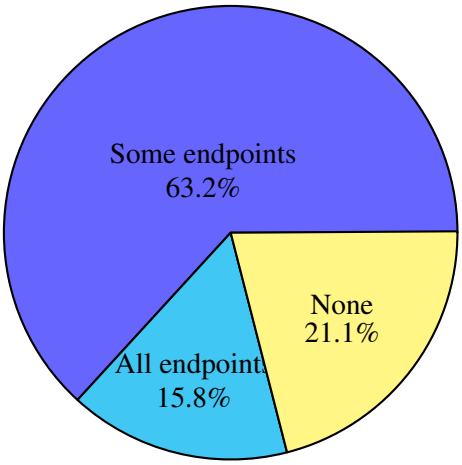

Figure 4.6: Developer choices for List Pagination implementation

more than half of the developers modify the caching mechanisms provided by the frameworks to better fit with the application business. Figure 4.7 illustrates the usage percentage of each mechanism. Figure 4.8 illustrates the developers' choices on each implementation of server caching.

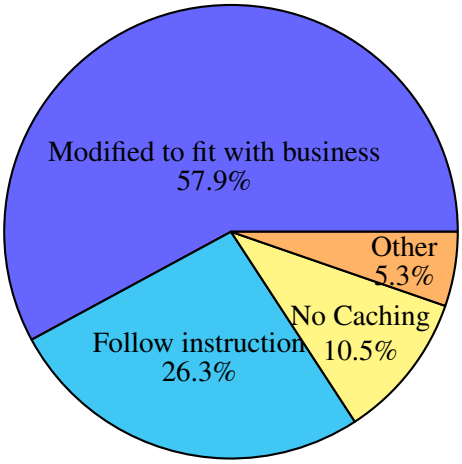

Figure 4.7: Developer choices for Server Caching mechanisms

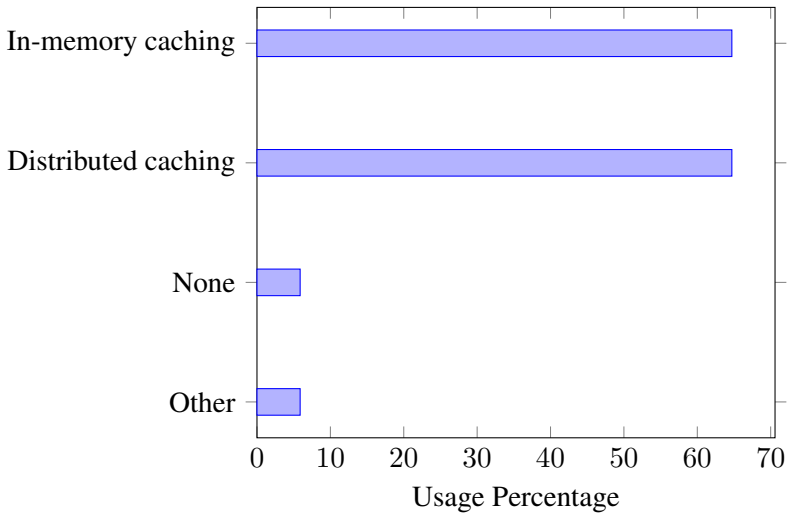

Figure 4.8: Developer choices for Server Caching Implementation

Some participants do not use the frameworks mentioned by the survey but different ones, like `nodejs` with JavaScript or `Flask` with Python. The good practices proposed by the research did not apply to these frameworks. Therefore, they answered the survey questions with the "Other" option.

4.7 Qualitative Analyses

Landis and Koch (1977) proposed a scale for the strength of agreement, with 61% - 80% labeled as "Substantial". We decided to use the average value of 70% to determine which good practices are acceptable or require further analysis.

For **Content Negotiation**, **Endpoint Redirection**, **API Versioning** and **POST-PUT-PATCH Return** good practices, more than 70% developers agreed that the proposed solutions are good. Further interviews with some of the developers also confirm that the solution for **Content Negotiation** is being used in the industrial but was not publicly published as a good practice.

Two good practices have positive answers below 70%. We interviewed five participants who answered "No" and "Other" to understand the reasons behind their choices. In addition, we also asked their opinions on other practices which they give positive responses.

For **Entity Linking**, there is an alternative approach that avoids the problem of entity linking. In

the first login, the server sends a set of allowed permission to the client, including the API endpoints related to these permissions. Therefore, the client can look up this set of authorization and endpoints to know if it can make requests to specific resources or not. This approach liberates the servers from calculating the related resources (hence linked entity) and endpoints of these resources. In addition, this approach is easier to implement. Companies that allow third-parties developers to register which permission they need, avoiding providing too much information that could raise privacy concerns.

For **Server Timeout**, the participant's company used the solution we proposed. However, due to the specific circumstance of the business, the solution is still not good enough. The task that is running on the server could take several hours to complete and return the results. The first-party clients did not know about this processing time, resulting in continuously polling for the results, throttling the server resources. Furthermore, the endpoint for HTTP polling was implemented in the same server as the endpoint for registering the long-running task, which was not recommended by the original solution of the good practice. In the end, the developers use `WebSocket` communication protocol, creating a tunnel between the client and the server. This tunnel allows the server to "send" a message to the client when the task is completed. Yet, there is one case that cannot use this protocol, which is uploading large files. Usually, a dedicated server will be used for this specific case, which HTTP polling or job queue.

4.8 Summary

In this chapter, we evaluated our solutions to REST API practices by survey 55 professional developers. In addition, we interviewed five of them to understand what is the reason behind their choices. In general, most of the participants agreed that our solution is good. However, for the "Entity Linking" and "Server Timeout" practice, the positive percentage is lower compare to other practices, which was explained after the interviews with the participants.

For demographic, most of the participants are Backend or Fullstack developers, whose has Bachelor or Graduate degree and has more than three years work experiences. Half of them came from Canada, another half came from Vietnam.

Chapter 5

Discussions

5.1 Threats to Validity

While most developers agreed with our solutions, there are some threats to the validity of our solutions and their validation that we must discuss.

5.1.1 Internal Validity

Our solutions assume that the developers are using object-oriented programming languages, like Java or C#. However, there are other languages for backend programming that are trending in recent years. For example, JavaScript with Nodejs, Go Lang with Gin, etc. We only examined two web frameworks, which are ASP.NET Core and Java Spring. There are other Web frameworks from the community for C#, like OpenRasta, NancyFx, but they are not popular compared to ASP.NET. For Java, besides Spring, there are multiple Web frameworks, like Grails and Struts. These frameworks could have different approaches to the good and bad practices.

We looked for the existing solutions in both academia and gray literature conforming to the good practices. However, we may have missed some results in academia due to inconsistencies between the research title and abstract and the content.

5.1.2 External Validity

We proposed six solutions to practically implement the REST API practices. To evaluate these solutions, we surveyed developers. However, due to the number of proposed solutions, the survey was quite long. The participants could get tired of the questionnaire and answer some questions with lower attention. These responses could bias our analyses. To minimize this threat, we tried to be as concise as possible. We also put the questions of the "Response Caching" and "List Pagination" good practices at the end of the survey, as they are (partially) supported by the Web frameworks. We also randomized the order of the questions. If a participant abandoned the questionnaire, we could still use the partial responses for analysis. In addition, we split the survey into two questionnaires. Each one contains four practices. Some participants answered both of the questionnaires.

For the demographic analysis, because some participants answered both parts of the survey, we used the participant's ID (provided by SurveyMonkey), IP address, age group, profession, company name, and email to count distinguish participants. If the participant use different networks for each parts of the survey, and they do not answer the optional questions about company name and email, then we could mistakenly count them twice. For example, the participant open the survey part A on his work computer using the company network. Later he open the survey part B on his personal computer at home. In both surveys he does not answer the questions about profession, email and company name. To minimize this threat, we send part A and part B of the survey to different group of participants. We note anyone asking to do the other part for counting them later.

Developers depend on their experiences and domain knowledge to concretely implement good practices and avoid bad practices. Therefore, the participant's level of experience affects our survey results. For example, more experienced developers could see potential problems in our solutions or evaluate these more thoroughly. Less experienced developers may favor our solutions and may have never faced the problems presented with each practice yet. We tried to minimize this threat by asking for age groups, education level, and current profession. The survey results showed that most of the participants have at least bachelor degree, and they are working as Fullstack or Backend developers for more than three years. Also, we interviewed some of the participants to get a better understanding of their responses.

In general, we could minimize the internal threats in future works by examining more Web frameworks and programming languages. In addition, we could conduct more one-on-one interviews with developers to find out their experience level and ask for their feedback on our solutions for external threats.

5.2 Developers' Feedback on Solutions

As presented in Section 4, for "Content Negotiation", "Endpoint Redirection", "API Versioning", and "POST-PUT-PATCH return", we received more than 70% positive responses. However, some developers have comments and other solutions for these good practices. In the following section, we cite and analyze each comment to see if they are good potential solutions to the practices.

5.2.1 Endpoint Redirection Good Practice

Comment 1: *It is better to use a proxy or a service broker to manage URL routing.*

Using a proxy server or service could solve the problem. However, the proxy/service must implement the exact resource mapping mechanism proposed by the good practice solution. In addition, the proxy/service must implement mechanisms to "catch" the request that needs redirecting.

5.2.2 API Versioning Good Practice

Comment 1: *At least with the version in URI, I don't have to modify my code as long as that version is available. For the suggested design, I'll have to modify my code and still have to rely on the availability of the old version.*

The suggested solution does not force the developers to choose a specific versioning type (see Section 3.5.1). Instead, it provides an architectural design that helps developers to reuse the business logic code. In addition, the developers should apply the good practice when designing the project, not when refactoring or introducing a new feature to the old codebase.

Comment 2: *Don't Need to support v1 in v2.*

Follow up interview with the participant allowed us to understand this comment. The company where the participant work does not publish REST APIs for third-party developers. Therefore, they

have a more straightforward, simple approach to API Versioning. Instead of supporting multiple versions of the same REST APIs, they force the client applications, which they fully control, to use always the latest version.

5.2.3 Server Timeout Good Practice

Comment 1: *Use http2, Web socket for request from frontend, use grpc or a pub/sub if request from back-end*

The participant's company used the proposed good practice. However, the specific business of the company makes the system perform poorly. For example, a single task could run for hours. The polling backend was implemented in the same server running the lengthy process. To address the issues, the developers at that company used multiple approaches, including:

- **http2:** The major next revision of the current HTTP network protocol. It supports a single connection from the browser to the backend.[1]
- **WebSocket:** A new communication protocol support two-way communication channels over a single TCP connection.[2]
- **gRPC:** Google Remote Procedure Calls, an open-source remote procedure call framework that use `http2` underline. Its complex usability makes it impossible to implement gRPC in any current browser without a proxy.[3]
- **publisher/subscriber pattern:** A messaging pattern allows the publisher to emit multiple events. The subscriber subscribes to the interesting events only. However, the pattern could introduce security and scheduling problems.[4]

The participant's company has the advantage of controlling both backend and front-end applications. Therefore, they can use new technologies that are still in beta development or require complex implementations.

[1] https://datatracker.ietf.org/doc/html/rfc7540
[2] https://html.spec.whatwg.org/multipage/web-sockets.html
[3] https://grpc.io/blog/state-of-grpc-web/
[4] https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber

5.2.4 POST-PUT-PATCH Return Good Practice

Comment 1: *If DB doesn't return data for create and update operation, we need to make an additional get operation to DB.*

The focus of the good practice solution is on the backend application. The participant raised a good idea. We stated in Section 3.7.2 that there should be a mechanism to control the response if it only needs a `200 OK` HTTP status code. In that case, the additional request to the database is not necessary. Both target Web frameworks optimize database requests in an approach called "Lazy Loading". The application only query database when the data is needed.

Comment 2: *Depends on the type of requirement. Clean Architecture and applying CQRS is better with a solution that is complex and needs scaling.*

Clean Architecture (Martin, Grenning, & Brown, 2018) is an architectural style that split the concerns of the application into a central domain logic and multiple "add-ons", like caching, authentication, authorization, persistence, Web rendering. Clean Architecture is the combination of multiple architectural paradigms[5], which are Hexagonal Architecture (a.k.a. Ports and Adapters), Onion Architecture, Screaming Architecture, Data Context Interaction and Boundary Control Entity. The components work with each other via interfaces. However, this architectural style is not related to the issues proposed by the good practice.

CQRS stands for Command Query Responsibility Segregation pattern [6]. The pattern states that the developer should use a different model to update the information than the model for reading information. This pattern arose when the application became so complicated that keeping a single model for CRUD is not possible. For example, when creating the object, a lot of validation rules must be performed. However, when reading the object from database, these rules are not necessary. This pattern is not directly related to the issue and the good practice.

[5] https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
[6] https://www.martinfowler.com/bliki/CQRS.html

5.3 Developers and Bad Practices

All the bad practices in the Non-technical category require constant review by experienced developers. For example, the "CRUDy URIs" bad practice is not solvable via an architectural design but by continuously watching the API endpoints in development. Therefore, there is a need to develop tools to support developers during development. Thus, this tool is out of scope for this paper.

Another approach to tackle these bad practices is the usage of machine learning. With big enough labeled data, we could train a machine-learning model that learns good and bad practices and constantly monitors the source code to avoid bad practices. For example, a recent project that supports developers converting from the requirement to code is GitHub Copilot [7]. The tool uses code snippets from Github and StackOverflow to insert into the developers' source code. However, the tool stopped at inserting the functional source code for a single requirement but not monitoring the developers' potential bad practices.

5.4 Discussion on "Server Timeout" Good Practice

The design for the "Server Timeout" good practice involves both client and server. The root cause of the problem is the connection between these two parties. We observed that all the practices involved with multiple parties might not be solvable by the 23 design patterns of object-oriented programming. However, this observation needs further research and validation to be confirmed and develop a general approach.

5.5 Solutions Consequences

We use the OOP design patterns to design the solution for the "Content Negotiation", "Entity Linking", "API Versioning", and "POST-PUT-PATCH return" practices. However, these design patterns came with consequences. For example, the Proxy Design pattern used in "API Versioning" introduces another layer of abstraction to the controller class. As a result, calling the original method

[7] https://copilot.github.com/

directly may have different behavior from calling them through the proxy methods.

When using these solutions, developers must understand the consequences to minimize the harmful effect. In future work, we will explore these consequences with in-depth research for each solution of the practices.

5.6 Summary

In this chapter, we discussed the internal and external threats to our study and how we tried to minimize these threats. We also listed all the comments from the survey's participants and our response to these comments. We analysed these comments carefully to see if they proposed reasonable alternative solutions to the REST API practices. We observed that the case of "Server Timeout" practice is quite interesting. The approach to this practice could be generalized but would need further research and validation. In addition, we discussed the practices in the non-technical category and how we could tackle them in the future. Finally, we discussed some consequences of the solutions, which will need further research to explore.

Chapter 6

Conclusion

In recent years, more and more companies adopted REST API as a primary technology for their services to reach more developers and serve more clients altogether. During the adoption process, many good and bad practices emerged. Researchers and developers together define rules to follow for creating a good REST API system. As a result, books, researches, and technical documentation are published to help developers. However, bad practices also arise from developers who did not follow or know the design rules. To avoid bad practices, developers rely on their experience and domain knowledge.

In general, researchers divide REST API design rules into good and bad practices. Developers, when developing REST API systems, should conform to these good practices and avoid bad practices.

A few solutions were proposed to concretely implement the four most common good practices, partially filling in the gap: NAC module and simple if-else for "Content negotiation", an additional proxy for "Entity Linking", a chain of adapters for "API Versioning", and URL redirections for "Endpoint Redirection". Yet, all of these solutions have some limitations. Besides, the solutions for "Content Negotiation" and "Entity Linking" good practices were designed to improve the deployed, in-production REST API systems, not to avoid the bad practices in the first place. Also, there are many other practices documented in the academic and gray literature.

In this thesis, we presented an up-to-date list of good and bad practices to design REST API systems. We divided the practices into eight technical and 11 non-technical practices. For each

technical practice, we proposed and discussed practical solutions and their concrete implementations. Finally, we compared/supplemented the existing solutions with new ones that increase their benefits for the four most common practices: Content Negotiation, Response Caching, POST-PUT-PATCH return, and List Pagination.

To determine how good our solutions were and how well they could be applied to industrial source code, we surveyed and interviewed 55 developers with qualitative questions. The results showed that most of the developers agreed with our solutions. Developers also confirmed that their companies use some good practices and other approaches that fit their specific business. Hence, we contributed by:

(1) Surveying practices in REST APIs in the academic and gray literature.

(2) Providing concrete solutions (designs and implementations) to these practices.

(3) Validating our solutions with professional developers.

We conclude that our solutions could benefit and be relevant to developers and researchers: to developers to avoid bad practices and ease their applications of good practices; to researchers who could study these solutions, their impact on various quality characteristics, etc.

In future works, we could extend our approach to provide concrete implementation for Service Oriented Architecture (SOA). For example, the trend of using Microservices Architecture could lead to many new good and bad practices. We will also seek to define practices and patterns to detect them in the source code of Web applications and study their prevalence quantitatively. The approach can be generalized, applying it to new good practices or to avoiding new bad practices. For the practices that cannot apply the general approach, we could do further research to categorize them based on metrics like the number of parties involved or the server architectural style. We could also explore the consequences of each practice solution to prevent the harmful effects.

Appendix A

API Versioning Approaches for including the version in request

A.1 Approach 1: Versioning in the URI segment

This approach is more practical. Front-end developers could immediately acknowledge the version of the REST API they are using. However, when developers need to upgrade to a new version, the URI must be changed.

For example:

```
1    https://localhost.com/api/v2.0/customers/123
2    https://localhost.com/api/v1/customers/123
```

A.2 Approach 2: Versioning in the Accept header of request

This approach has more freedom for both front-end and back-end developers. The version information is included in the "Accept" header of the request. The server confirmed with the "Content-type" header. However, this approach uses a non-standard "Accept" and "Content-type" header, which is not suggested.

For example:

```
# REST API endpoint
https://localhost.com/api/customers/123

# Request ==>
HTTP method: GET
Accept: application/json+v3

# Response <==
HTTP/1.1 200 OK
Content-Type: application/json+v3
Body:
{
    "test":"test data"
}
```

A.3 Approach 3: Versioning in the query string

In this approach, the version information is send to the server as a parameter in query string. One major disadvantage is every REST API endpoint must handle the versioning parameter. This approach is not recommended.

For example:

```
https://localhost.com/api/customers/123?version=1
```

References

Butler, M. H. (2001). Implementing content negotiation using cc/pp and wap uaprof. *HP Laboratories Technical Report HPL*, *2001*(190).

Dai, T., He, J., Gu, X., & Lu, S. (2018). Understanding real-world timeout problems in cloud server systems. In *2018 ieee international conference on cloud engineering (ic2e)* (pp. 1–11).

Deutskens, E., De Ruyter, K., Wetzels, M., & Oosterveld, P. (2004). Response rate and response quality of internet-based surveys: an experimental study. *Marketing letters*, *15*(1), 21–36.

Eastbury, W. (2019, Oct). *Asynchronous request-reply pattern.* https://docs.microsoft.com/en-us/azure/architecture/patterns/async-request-reply. Microsoft.

Evdemon, J. (2016, Aug). *Principles of service design: Service patterns and antipatterns.* http://web.archive.org/web/20160807191653/https://msdn.microsoft.com/en-us/library/ms954638.aspx. Microsoft.

Fredrich, T. (2012). Restful service best practices. *Recommendations for Creating Web Services*, 1–34.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In *European conference on object-oriented programming* (pp. 406–431).

Google. (n.d.). *Common design patterns.* https://cloud.google.com/apis/design/design_patterns. Author.

Iyer, S., Rowstron, A., & Druschel, P. (2002). Squirrel: A decentralized peer-to-peer web cache. In *Proceedings of the twenty-first annual symposium on principles of distributed computing* (pp.

213–222).

Kaminski, P., Litoiu, M., & Müller, H. (2006). A design technique for evolving web services. In *Proceedings of the 2006 conference of the center for advanced studies on collaborative research* (pp. 23–es).

Kohlhoff, C., & Steele, R. (2003). Evaluating soap for high performance business applications: Real-time trading systems. In *International world wide web conference.*

Lalanda, P. (1998). Shared repository pattern. In *Proc. 5th annual conference on the pattern languages of programs.*

Landis, J. R., & Koch, G. G. (1977). The measurement of observer agreement for categorical data. *biometrics*, 159–174.

Leitner, P., Michlmayr, A., Rosenberg, F., & Dustdar, S. (2008). End-to-end versioning support for web services. In *2008 ieee international conference on services computing* (Vol. 1, pp. 59–66).

Lemlouma, T., & Layaïda, N. (2001). Nac: A basic core for the adaptation and negotiation of multimedia services. *Opera Project, INRIA*.

Li, J., Xiong, Y., Liu, X., & Zhang, L. (2013). How does web service api evolution affect clients? In *2013 ieee 20th international conference on web services* (pp. 300–307).

Liskin, O., Singer, L., & Schneider, K. (2011). Teaching old services new tricks: adding hateoas support as an afterthought. In *Proceedings of the second international workshop on restful design* (pp. 3–10).

Martin, R. C. (2002). *Agile software development: principles, patterns, and practices*. Prentice Hall.

Martin, R. C., Grenning, J., & Brown, S. (2018). *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall.

Masse, M. (2011). *Rest api design rulebook: Designing consistent restful web service interfaces*. " O'Reilly Media, Inc.".

Murphy, L., Alliyu, T., Macvean, A., Kery, M. B., & Myers, B. A. (2017). Preliminary analysis of rest api style guidelines. *Ann Arbor*, *1001*, 48109.

Narumoto, M. (n.d.). *Caching*. https://docs.microsoft.com/en-us/azure/

architecture/best-practices/caching. Microsoft.

Paepcke, A. (1993). *Object-oriented programming: the clos perspective*. MIT Press.

Palma, F., Dubois, J., Moha, N., & Guéhéneuc, Y.-G. (2014). Detection of rest patterns and antipatterns: a heuristics-based approach. In *International conference on service-oriented computing* (pp. 230–244).

Palma, F., Gonzalez-Huerta, J., Founi, M., Moha, N., Tremblay, G., & Guéhéneuc, Y.-G. (2017). Semantic analysis of restful apis for the detection of linguistic patterns and antipatterns. *International Journal of Cooperative Information Systems*, *26*(02), 1742001.

Petrillo, F., Merle, P., Moha, N., & Guéhéneuc, Y.-G. (2016). Are rest apis for cloud computing well-designed? an exploratory study. In *International conference on service-oriented computing* (pp. 157–170).

Rabinovich, M., & Spatscheck, O. (2002). *Web caching and replication* (Vol. 67). Addison-Wesley Boston, USA.

Ralph, P., Baltes, S., Bianculli, D., Dittrich, Y., Felderer, M., Feldt, R., ... others (2020). Acm sigsoft empirical standards. *ACM SIGSOFT*.

Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J. C., Canali, L., & Percannella, G. (2016). Rest apis: a large-scale analysis of compliance with principles and best practices. In *International conference on web engineering* (pp. 21–39).

Rodriguez, J. M., Crasso, M., Zunino, A., & Campo, M. (2010). Automatically detecting opportunities for web service descriptions improvement. In *Conference on e-business, e-services and e-society* (pp. 139–150).

Tilkov, S. (2008, Jul). *Rest anti-patterns.* https://www.infoq.com/articles/rest-anti-patterns/. InfoQ.

Wessels, D. (2001). *Web caching*. " O'Reilly Media, Inc.".