Formalising Solutions to REST API Practices as Design (Anti)patterns
(Resubmission of paper #55)

Hidden names

Hidden institute
hidden email

Abstract. 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 practices. Thus, research on good and bad practices and how to design a simple but effective REST API are essential. Yet, to the best of our knowledge, there are only a few concrete solutions to recurring REST API practices, like "API Versioning". There are works on defining or detecting some practices, but not on solutions to the practices. We present the most up-to-date list of REST API practices and formalize them in the form of REST API (anti)patterns. We validate our design (anti)patterns with a survey and interviews of 55 developers.

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 the client application (present information). Simple Object Access Protocol (SOAP) used to be the main protocol to expose services to clients. However, starting from the 2000s, many organizations migrated their services from SOAP to REST to widen developers' accessibility to their data. For example, in 2006, Google deprecated SOAP for its Search API and moved to REST[1]. The number of REST API published increased every year, from 445 APIs in 2007 to over 24k APIs in 2021[2].

The word "practice", as defined by Oxford Dictionary, is a way of doing something. In REST API, a good practice is a good way to implement the REST API for simplicity, mutual understanding, and reusable code. On the other hand, while resolving a problem, a bad practice is not "good" in other aspects.

As with any other architectural style, REST APIs can be more or less "well" used and, therefore, the subjects of good and bad practices. To evaluate how

[1] https://groups.google.com/g/google.public.web-apis/c/YOHPWSqcFBA
[2] https://www.programmableweb.com/apis/directory

good a REST system is, Richardson proposed a maturity model for REST APIs[3]. Other researchers also proposed good practices to make REST APIs more understandable and reusable [10, 14]. The academic and gray literature report 19 problems related to REST APIs and their uses. For example, Rodríguez et al. [15] found out that only a few Web services reach maturity Level 3, which is defined as "Hypermedia as the engine of state".

The literature has so far not systematically described these problems and practices in the form of design (anti)patterns, which are, in general, a problem, a recurring design with bad consequences, and an alternative solution with more positive results [1].

Therefore, we follow the "**Design Science Research Methodology**"[13] to propose three contributions: **(1)** we review the academic and gray literature related to REST APIs and identify 19 common good and bad practices, **(2)** we propose practical solutions to these problems and formalize them in the form of REST API design anti-patterns, and **(3)** we validate our solutions via surveys and interviews of 55 participants.

The rest of the paper is as follows. Section 2 summarises the related work. Section 3 describes the approaches. Section 4 discusses each practices with concrete implementations of the solutions. Section 5 explains how we evaluated our solutions with developers. Section 6 discusses threats to validity as well as our observations. Section 7 conclude the paper with future work.

2 Related Work

Masse, in the book "REST API Design Rulebook" [10], 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" (aka "CRUDy URIs"). Rodriguez et al. [16] proposed the "Content negotiation" good practice: servers should serve different formats of the same resources on request. Fredrich [2] 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".

Evdemon gave examples of bad URIs and proposed the "CRUDy URIs" bad practice, where CRUD verbs are included in the URI[4]. Palma et al. [12] defined the "Non-pertinent documentation" bad practice, where the documentation is not matching the actual REST APIs. Tilkov defined seven REST API bad practices[5], 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 into "Use the wrong HTTP Verbs" for simplicity. "Breaking self-descriptiveness"

[3] http://martinfowler.com/articles/richardsonMaturityModel.html
[4] https://bit.ly/3i5CIsc
[5] https://www.infoq.com/articles/rest-anti-patterns/

means developers ignore standardized headers, formats, protocols, and use non-standard ones. “Ignoring status code” happens when a server does 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 4.

Researchers proposed solutions for some bad practices. Frameworks also provide some support to avoid some bad practices. We examine here ASP.NET Core[6] and Java Spring[7] because of their popularity and community support.

For “Content negotiation”, Lemlouma et al. [7] designed “Negotiation and adaptation core (NAC)” that works as a proxy between the media servers and the consuming clients. Based on the client profile, the the NAC converts the response to an appropriate format. This is a general architecture and its authors do not discuss any implementation.

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 it up in some servers. Popular servers, like Microsoft IIS or Apache Tomcat, implement URL redirection with some configuration[8].

For “Entity Linking”, we could not find any academic work, blog, or technical tutorials with concrete implementations. However, Liskin et al. [8] described a wrapper module to convert a normal response to a response that conforms to “Entity Linking”, which allows improving old REST systems not supporting Entity Linking and reaching Level 3 in Richardson’s Maturity Model.

For “Server Timeout”, Eastbury et al. proposed a design[9], which we extend to maximize its benefit for REST API developers in Section 4.5.

“Response caching” is a common good practice. Both of the examined Web frameworks offer multiple built-in caching techniques.

For “List Pagination”, both Google and Microsoft[10] suggested that REST APIs returning lists should use pagination. Masse [10] also stated that collections should be returned in chunks. Murphy et al. [11] showed that pagination was proposed in 24 over 32 REST API company guidelines. Both of the examined Web frameworks support this good practice.

In general, previous work mostly identified and defined good and bad REST API practices. A few authors proposed solutions to the bad practices, often with limitations. For each practice, we discuss and compare its solutions with our own. We also provide concrete implementations in two popular frameworks (ASP.NET and Java Spring). We also provide sample implementations for “Response Caching” and “List Pagination”, supported by the frameworks.

[6] `https://dotnet.microsoft.com/apps/aspnet`
[7] `https://spring.io/projects`
[8] `https://bit.ly/34AUbRu` and `https://bit.ly/3i6X8Ry`
[9] `https://bit.ly/2R9clXf`
[10] `https://bit.ly/2RY62pW` and `https://git.io/JGRwC`

3 Categories of Good and Bad Practices of REST APIs

To categorize the REST API practices, we extensively reviewed both academic papers and gray literature (i.e., blog posts, technical tutorials, StackOverflow, etc.) and studied the existing open-source REST API systems. In total, we reviewed seven papers and four gray documents (see `https://git.io/JRsHD`).

We identified 19 REST API practices and divided them into two categories: technical and non-technical. The technical category includes practices that can be solved or made conformed by an architectural solution or some Web framework's features. The non-technical category includes practices that require developers' efforts to conform, which are usually domain- or business-specific.

For example, the URI structures should represent the relationships between the nodes to avoid the "Non-hierarchical Nodes" bad practice. Yet, companies disagree on using URI to show this relationship and even discourage nesting structures. Another example, for the "Using the wrong HTTP Verbs" bad practice, IBM only mentions `GET` and `POST` while Google use `GET`, `POST`, `PUT`, `DELETE` and invents some new verbs like `LIST` and `MOVE` [11].

For each practice in the technical category, we propose an architectural solution or good practice in Section 4, which should be simple but guarantee the conformance to the good practice with minimal effort. Practices in the non-technical category require developers' inputs and are not directly solvable.

Table 1 summarises the practices. The practices with (+) have built-in or partial solutions in the examined frameworks, which we discuss and compare to our solutions in Section 4. The good practices are green; the bad ones red.

Table 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 [3] to design concrete implementations for each of the eight good practices in the technical category. For some practice, we adapted these concrete implementations to fit with the REST API frameworks. If no design pattern could solve a problem, we extended our search to the gray literature.

The Web frameworks already have built-in features for "Response Caching" and "List Pagination". Therefore, we only present these solutions and provide

sample usages for the sake of completeness. For "Content negotiation", we compare our solutions with the built-in features. Developers can use the solutions that fit with their projects based on their advantages and disadvantages.

4 REST API Anti-patterns

We now present each practice using the following structure: **(1)** Practice name, **(2)** Problem statement, **(3)** Expected result/output, **(4)** Solution, and **(5)** Sample implementation/source code. For the sake of space, we put out implementations in a GitHub repository: `https://github.com/ano-research/pattern-abiding-api-anonymous`.

We exclude the "Response Caching" and "List Pagination" good practices out of this research because they are already well supported by Web frameworks.

4.1 Content Negotiation

Problem: The client can only process and manipulate the resources in some formats. For example, JSON is faster to parse and smaller to transport over the Internet while XML supports namespaces, comments, and metadata. A client may favor one over another.

Expected result: We expect: **(1)** Resources of a same type should be served in various formats (JSON and XML, image file formats, Base64 encoded, etc.); **(2)** The server should set a default format if the client does not specify a requested format; **(3)** The implementation of each data format should be easily modifiable and expandable for new data formats.

Solution: Based on the request header, the server prepares the data in the requested format, then returns the data in the response body. We could use the Factory design pattern [3]. For each format, there could be a corresponding concrete Factory. In the `ObjectFactory` class[11], developers could set the default format by using the `default:` clause of the `switch` statement. (See `https://git.io/JRBCt`)

Both Java Spring and ASP.NET core support content negotiation with the default set to `JSON` format. Below is the feature comparison table.

	Java Spring	**ASP.NET core**	**Our**
Common media types	Yes	Yes	Yes
Customizable serializer	No	Yes	Yes
Require data annotation on model	Yes	No	No
Built-in support ignorable	Yes	Yes	N/A

ASP.NET Core has the most flexible support for Content Negotiation. In Java Spring, developers cannot override the serializer, but can combine multiple approaches to achieve the desired effect. The XML format in Java requires data annotation to be added to the model classes, which sometimes require changes to the design of the data model.

[11] `https://git.io/JGRWC`

4.2 Endpoint Redirection

Problem: Resources can be moved to new locations when the data structure changes or developers refactor the URI structure. However, a client could request resources using the old URIs; the server should answer these requests with HTTP Code 3xx and the new locations.

Expected result: Most of the REST APIs frameworks use the Model-View-Controller pattern (MVC). The solution should be built on or integrated with the MVC pattern. It should satisfy the following: **(1)** There should be a class that handles redirection logic, separated from other classes; **(2)** The redirection logic should have the same interface as the old controller class; **(3)** The new controller class should extend or include the redirection logic, but still conform to the *Single Responsibility Principle* [9].

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

Solution 1: Extend a `Redirector` class: In this solution, the old controller and the redirector implement the same interface. The new controller extends the redirector class. The methods in the new controller and the redirector are exposed as API endpoints to the clients. (See `https://git.io/JG0fv`)

Solution 2: Nested class inside the new controller: The redirection logic is separated into a stand-alone class, implementing the interface of the old controller class. The difference with the previous solution is that the redirection logic class is a nested class inside the new controller. Thus, the redirector can access methods and variables in the controller. (See `https://git.io/JRg0E`).

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 properly handled. The private instance works as a proxy to the actual logic of the redirector class.

Comparison between both solutions

	Extend	Nested
Pros	Redirection logic separated in classes	Allows the outer class (the main class) be inherited from another class
	Easier implementation. Can be implemented in multiple languages	The Redirector has access to methods/variables in the controller
Cons	The Redirector cannot access resources in the controller	Redirection logic coded inside the controller
	The controller cannot inherit other classes (w/o multiple inheritance)	Not all languages support nested classes

4.3 Entity Linking

Problem: Developers must find programmatically links to resources related to a current, requested resource. For example, when designing a service for a content management system (CMS), after sending a `GET` request to retrieve a post, if conforming to the Entity Linking good practice, the server should return the post details, including links to comments and likes. The response below helps developers to post new comments or get the likes on the post.

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

With the "Forgetting Hypermedia" bad practice, the links in the response are not available. Therefore, developers do not know if they can post a new comment or get the likes on a post. They must make requests to the server to find out, possibly by trial-and-error. If they cannot, for some reason, the server refuses their requests with HTTP Code 4xx.

Expected result: **(1)** The current controller should have access to other controllers to check the availability of related resources; **(2)** The current controller should have access to class and method information (names, annotations, public and private variables, etc.) because Web frameworks use naming conventions/annotation to construct URIs.

Solution: The method that handles the current request has a list of related classes containing the related resources. To loop through the list, all these classes should implement a same interface. To access the methods provided by this interface in different classes, we could use the Visitor design pattern [9] with language reflection features to access data annotations.

All the controllers intended to be used for populating links for related resources must implement a `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 separate, concrete `Visitor` is created. These visitors could share some common logic in an abstract class `CommonResourceVisitor`, also a good place for the reflection logic. A sample implementation with reflection is available for ASP.NET Core and Java Spring. (See https://git.io/JGRWW and https://git.io/JGRW8).

4.4 API Versioning

Problem:

REST API system evolves. In traditional software systems, developers can release new versions while old ones continue working. With Web applications, a new version may break the client applications. The client-application developers may not be aware of breaking changes and may not have enough time to adapt to the changes before changes break their application.

In addition to communicating and advertising new versions of some REST APIs, REST API developers must support the old APIs in parallel with the new ones for a time. This parallelisation allows the coexistence of multiple versions of the API. To separate the old APIs from the new ones, developers can choose one of two approaches: **(1)** Include the version of the API in the URI: Version the API globally for all resources or separately for each type of the resources; **(2)** Include the version in the header(s), in the `accept` header or a custom header chosen by the REST API developers.

Expected result: **(1)** The URIs of a version should not change over time. The client application always receive the expected results when requesting a resource during the lifetime of that version; **(2)** Client applications can access

multiple versions of the APIs simultaneously. The Web application supports multiple versions at a same time; **(3)** The Web application can use a single database. If breaking changes occur in the database, a mechanism should translate the old and new data for any object.

Solution: Section 2 show existing solutions, which we compare to ours below.

We propose using the Proxy design pattern to redirect/convert requests to old versions of the API. For each version of the API, there should be a corresponding interface. The interface does not serve a purpose in the current version but is the contract with the next version. A class contains a private instance of each old API classes that support this next version. The versions of the API may also have common logic that did not change over time and can be factored into an abstract class; then, the API controllers inherit from it. (See `https://git.io/JGR67`)

One advantage of this solution is that a new version of an API only implements the interfaces of old versions if they must be supported. For example, if there is a Version 3, which should support both Versions 1 and 2, then a controller for Version 3 implements the interfaces of Versions 1 and 2 only, which does not require a "chain of adapters" and allows developers to support versions not ordered in time, e.g., Version 4 could support Version 1 and 3, when compared to the solution proposed in [4]. In addition, this solution is implemented in the server itself, differently from the solution proposed in [6].

4.5 Server Timeout

Problem: When there is a long-running operation on a REST API server, the client must wait to receive a response. In traditional software systems, the communication between the running operation and the consumer of its results is permanent. However, in a Web application, clients and servers communicate over the Internet, which introduces some potential problems: if the client and the server are disconnected, the operation continues to run on the server, but the result is no longer needed/accessible. Similarly, if the client cancels or abandons its request, wasting the server's resources.

Expected result: The long-running operation should have a mechanism to cancel itself when needed, releasing resources held by its process.

Solution: Two solutions could solve the problems. The combined implementation of both solutions produces the best results but with increased complexity.

Solution 1: Timeout: The developers define a timeout for the operation. When the time is up, the server cancels the process. However, in some cases, when there are unpredictable factors, this solution is not practical. For example, if the operation depends on data retrieved through the Internet, then its speed is not stable and it is challenging to find an appropriate timeout value.

Since Java 1.5, developers can use `ExecutorService` to start a new single thread in Java. Then, they can submit a long running operation wrapped in a `Callable` object. Similarly, in ASP.NET Core, developers can use `TimeOutAfter` or `CancellationTokenSource.CancelAfter`[12] in .NET 5.0 or later

[12] `https://git.io/JGRWB` and `https://bit.ly/3vDwdB1`

Solution 2: Asynchronous Request-Reply pattern (HTTP polling): This solution requires some extra work from both server and client developers. First, server developers must implement an operation status checker and a cache system to store the operation result on the server. Then, client developers must implement a polling mechanism that periodically polls the operation status and, finally, gets the response from the `Resource Endpoint`. (See https://git.io/JRrHw).

Comparisons between the two solutions

	Timeout	Async request-reply
Pros	No requirement on client side	Client and server can re-establish connection after disconnection
	Easier to implement	Serves the same results instantly
	Single system involved	
Cons	Risk of incorrect predefined timeout value	Requires changes in both client and server
		Requires a mechanism to store results
	If the client disconnects early, the server still wastes resources until timeout	

Combination of both solutions

Developers can combine both solutions to maximize their benefits, especially when following the Asynchronous Request-Reply pattern, because they only need a little extra work to combine both solutions. There are three stages: (1) initialisation, (2) polling, and (3) termination. During initialisation, the client sends a request to the server that starts the long-running operation. The server registers the operation status. During polling, the client periodically polls for the result. With each polling request, the server resets the timeout. In case of a disconnection between the client and server, the server timeouts and aborts the process and releases any resources. (See https://git.io/JGR6w).

4.6 POST-PUT-PATCH Return

Problem: When a client application sends a request to modify a database (create, update, or delete), the server usually only returns a simple result indicating success/failure, like HTTP Code `200`. The client does not know immediately the added/modified object and whether it was correctly committed to the database until the client makes a new request for that particular object.

When there is a mismatched datatype between the model classes and the data posted by the client, the server may still work by simply ignoring any mismatched data, writing everything else to the database. The server would still signal that the writing process was successful. The client could wrongly believe that the operation was entirely successful although its data and the data written to the database are different.

However, sending the created/modified object in the response body could cost transmission time and network bandwidth, i.e., some clients may not need to know/confirm that the committed data in the database is correct.

Expected result: The name "POST-PUT-PATCH return" of this practice is based on the HTTP verbs `POST`, `PUT`, and `PATCH`. We expect: **(1)** A mechanism

for the client to control the response, if it only needs a HTTP Code 200; **(2)** Separation of database and business logic; **(3)** Minimal requests to the database.

Solution: To separate the database manipulation logic from the business logic, we use the Repository pattern. To make sure a database transaction was successfully executed before returning its result to the client, we could use the Unit of Work pattern[13]. In general, there should be a repository that handles CRUD operations for each resource or groups of related resources. This class is injected into controllers using Dependency Injection. In addition, each repository implements the Unit of Work pattern. (See `https://git.io/JRXxC`).

Java Spring has a Repository pattern built-in with basic CRUD operations, see details in `https://bit.ly/3c6GFcs`. ASP.NET Core does not have this pattern built-in. Still, there is an instruction on how to implement them, see details in `https://bit.ly/3uCO9du`. Both frameworks support Dependency Injection and Unit of Work. Sample implementations exist for Java Spring at `https://git.io/JGRWE` and ASP.NET Core at `https://git.io/JGRWg`.

5 Evaluations

Although good/bad practices come from a consensus in the academic/professional communities, our solutions must be validated by experienced developers, who only can confirm that our solutions **(1)** solve bad practices, **(2)** conform to good practices, and **(3)** are acceptable in industrial environments.

Consequently, we designed a validation survey and administrated it to 55 professional REST API developers. We now describe our methodology and the obtained results, which show that our solutions indeed remove bad practices and are acceptable by professional developers.

5.1 Overview

We obtained an ethics certificate from the Office of Research of our university; university and number hidden for double-blind review. We follow the "Questionnaire Survey" empirical standard by ACM[14]. We divided the survey into sections. Each section contained one practice, problem identification, a short explanation for the good practice, and the concrete implementation in ASP.NET core and Java Spring. To increase the response rates, we split the survey into two Parts A and B.

5.2 Survey Design

In each section, we asked two questions: **(1)** Did you face this/these problem(s) in some of your projects? **(2)** Is it a good design?. Each question has an "Other" option with a text-box if the participant has comments.

Using these two questions, we can know: **(1)** The prevalence of the problem in industrial environments; **(2)** How well received are the solutions to the bad practices; **(3)** Do alternative implementations exist?

At the end of the survey, we asked for age group, education, current profession, and employment status.

[13] `https://bit.ly/3uCO9du`
[14] `https://acmsigsoft.github.io/EmpiricalStandards/docs/`

5.3 Participants Selection

We selected participants who are: **(1)** Adult; **(2)** Professional developers; **(3)** Use OOP languages. We recruited participants through a convenient sampling of software developers and engineers through e-mail lists, social medias (LinkedIn, GitHub), etc.

5.4 Participants' Demographics

Figures 1 to 5 summarise participants' demographic information.

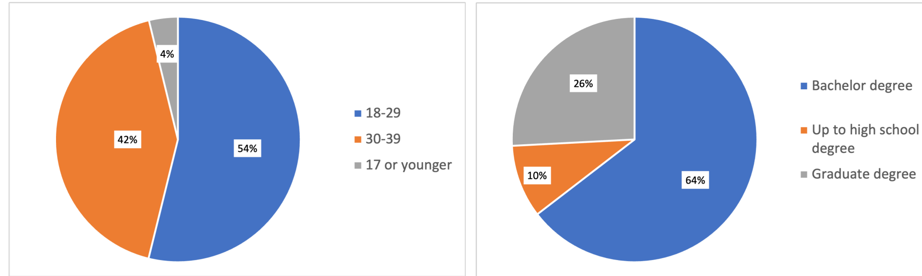

Fig. 1. Participants' age groups

Fig. 2. Participants' education

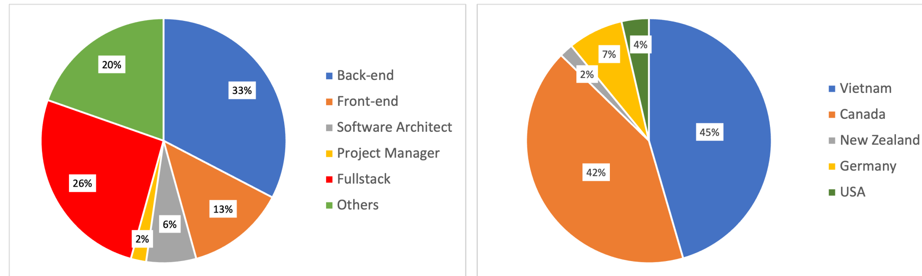

Fig. 3. Participants' profession

Fig. 4. Participants' country of origin

5.5 Qualitative Analyses

We sent our survey to 55 professional developers. Out of 55, 51 developers completed at least one of the surveys. We received 68 complete surveys (Parts A and B). The number of completed surveys is greater than the number of participants because some participants completed both Parts A and B. We received 17 incomplete responses. Because of the randomized orders of the question, we could still extract valuable data from these responses. Thus, the average completion rate of both surveys is 76%.

We extracted and analyzed the completed parts of the incomplete survey, whose questions were all answered by the participants. Table 2 shows the percentage of positive responses.

Some participants do not use the frameworks considered in this study but different ones, like `nodejs` with JavaScript or `Flask` with Python. Some good practices proposed in this study did not apply to these frameworks. Therefore, participants answered the survey questions using the "Other" option.

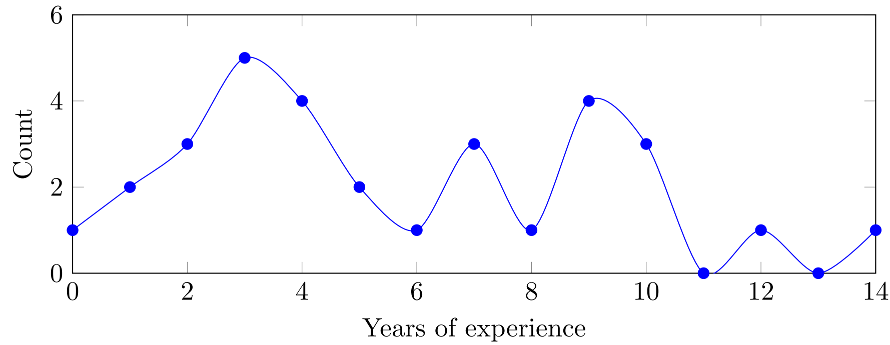

Fig. 5. Participant's years of experience

Table 2. 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

5.6 Quantitative Analyses

Landis et al. [5] proposed a scale for the strength of agreement, with 61% - 80% labeled as "Substantial". We decided to use the threshold, average value of 70% to determine which good practices are acceptable or require further analysis.

For the **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 used in industrial environments even if not publicly published as a good practice.

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

For **Entity Linking**, there is an alternative approach that avoids the actual 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 use this set of permissions and endpoints to know if it can request specific resources or not. This approach frees the server from calculating the related resources (hence linked entities) and endpoints of these resources. In addition, it is easier to implement. Companies that allow third-

parties developers to register which permission they need, thus avoid providing too much information that could raise privacy concerns.

For **Server Timeout**, the participants used the solution that we proposed but, due to the specific business requirements, it was still not good enough. The tasks running on their servers could take several hours to complete and return results. Clients did not know about this processing time, resulting in them continuously polling for results, throttling the servers. Furthermore, the endpoints for HTTP polling were implemented in the same servers as the endpoint for registering the long-running tasks, which was not recommended in the original solution. Consequently, developers used `WebSocket` to create tunnels between the clients and servers. These tunnels allow the servers to "send" messages to the clients when the task are completed. Yet, this solution cannot be used when uploading large files. Usually, a dedicated server will be used for this specific case, with HTTP polling or job queue.

6 Discussions

6.1 Threats to Validity

While most developers agreed/somewhat agreed on our solutions, there are some threats to validity that we would like to discuss.

Internal validity: Our solutions assume that the developers are using object-oriented programming languages, like C# or Java. There are other languages for back-end programming trending in recent years. For example, JavaScript with `nodejs`, Go with `Gin`, etc. In addition, there are other Web frameworks from the community for C# like OpenRasta and NancyFx, although they are not as popular as ASP.NET. For Java, besides Spring, there are multiple Web frameworks, like Struts and Grails. These frameworks could have different approaches to good and bad practices. We could minimize this threat by expanding the survey and ask other framework experts.

We looked for existing solutions in both academic and gray literature, conforming to good practices in reviewing previous work. However, we may have missed some solutions due to inconsistencies in vocabulary, titles and contents, etc. We minimized this threat by using multiple related keywords to search the academic and gray literature.

External validity: We proposed six solutions to practically implement REST API good practices. To evaluate these solutions, we surveyed back-end developers. Due to the number of proposed solutions, the survey was quite long. The participants could become tired by the end of the survey and answer the questions with lower attention than those near the start. These responses could bias our analysis. To minimize this threat, we divided the survey in two parts and 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, because they are supported or partially supported by the web frameworks.

Developers depend on their experiences and domain knowledge to concretely implement good practices and avoid bad practices. Therefore, their levels of expertise may affect the survey result. More experienced developers could see

potential problems in our solutions or evaluate these solutions more thoroughly. Less experienced developers may favor our solutions or not have experienced the practice problems. We tried to minimize this threat by asking and controlling for age groups, education level, and current profession.

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

6.2 Developers' Feedback on Solutions

As presented in Section 5, for "Content Negotiation", "Endpoint Redirection", "API Versioning", and "POST-PUT-PATCH return", we received more than 70% positive responses. However, some developers commented our solutions and proposed other solutions. We summarise and analyse each participants' comment in the following.

Endpoint Redirection Good Practice

Comment 1: *It is better to use a proxy or a service broker*

Using a proxy server or service could solve the problem but the proxy/service would have to implement the exact resource mapping mechanism proposed by the good practice solution. In addition, it would have to implement mechanisms to "catch" the request that needs redirecting, making it more complex than ours.

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 developers to choose a specific versioning type, see Section 4.4. It provides a solution that helps developers to reuse business logic. Besides, developers should apply good practices when designing their applications, not when refactoring or introducing a new feature.

Server Timeout

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

An interview with the participant helped us understand this comment: the participant's company uses the proposed good practice. Yet, the specific business of the company makes the system perform poorly. Indeed, a single task could run for hours. To address this issue, developers use multiple approaches, including: **http2:** The major next revision of the HTTP network protocol that supports a single connection from the browser to the back-end; **WebSocket:** A communication protocol supporting two-way communication over a single TCP connection; **gRPC:** Google Remote Procedure Calls, an open-source remote procedure call framework that uses `http2`; **publisher/subscriber pattern:** A messaging pattern allowing a publisher to emit multiple events to multiple subscribers interested in these events.

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

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 is on the back-end. We explained in Section 4.6 that there should be a mechanism to control the response if it only needs a HTTP Code `200`. Both considered Web frameworks optimise database requests using "Lazy Loading". Application only query databases when 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 is an architectural style that splits the concerns of the application into a central domain logic and multiple cross-cutting concerns, like caching, authentication, authorization, rendering, etc. The concerns work with each other via interfaces. This architectural style is not related to the issues tackled by the good practice. *CQRS* stands for *Command Query Responsibility Segregation*, which is a pattern stating that developers should use a different model to update information than the model to read information. Again, this pattern is not directly related to the issue and the good practice.

6.3 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 with 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. Such tools are out of the scope of this study.

7 Conclusion

In this paper, we presented an up-to-date list of good and bad practices to design REST API systems, divided into 8 technical and 11 non-technical practices. For each technical practice, we proposed and discussed practical solutions and concrete implementations. For three of the four most common practices, **Content Negotiation**, **API Versioning**, and **Endpoint Redirection**, we compared/supplemented the existing solutions with new solutions and implementations that increase their benefits. To determine how acceptable our solutions were and how well they could be applied in industrial environments, we surveyed and interviewed 55 developers. Results of our survey and one-on-one interviews 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. Reviewing REST API practices usage in the academic and gray literature.
2. Providing solutions and concrete implementations to these practices.
3. Validating our solutions with professional developers.

We conclude that our solutions are relevant to developers and researchers as a basis for implementation and future, quantitative studies (e.g., detection).

In future work, we could extend our approach to provide concrete implementation for Service Oriented Architecture (SOA). Our approach could be generalized by applying it to other good/bad practices. For the practices to which

we cannot apply our approach, we could do further research to categorise them based on other criteria, like the numbers of parties involved or the server architectural style. We could also expand the survey and have more one-on-one interviews to qualify more precisely our solutions.

Bibliography

[1] Brown, W.H., Malveau, R.C., McCormick, H.W.S., Mowbray, T.J.: AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc. (1998)

[2] Fredrich, T.: Restful service best practices. Recommendations for Creating Web Services pp. 1–34 (2012)

[3] GAMMA, E.: Design patterns: Abstraction and reuse of object-oriented design. In: Proc. ECOOP'93. pp. 406–431 (1993)

[4] Kaminski, P., Litoiu, M., Müller, H.: A design technique for evolving web services. In: Proceedings of the 2006 CASCON. pp. 23–es (2006)

[5] Landis, J.R., Koch, G.G.: The measurement of observer agreement for categorical data. biometrics pp. 159–174 (1977)

[6] Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S.: End-to-end versioning support for web services. In: 2008 IEEE SCC. vol. 1, pp. 59–66 (2008)

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

[8] Liskin, O., Singer, L., Schneider, K.: Teaching old services new tricks: adding hateoas support as an afterthought. In: Proceedings of the 2nd WS-REST 2011. pp. 3–10 (2011)

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

[10] Masse, M.: REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. " O'Reilly Media, Inc." (2011)

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

[12] Palma, F., Gonzalez-Huerta, J., Founi, M., Moha, N., Tremblay, G., Guéhéneuc, Y.G.: Semantic analysis of restful apis for the detection of linguistic patterns and antipatterns. IJCIS **26**(02), 1742001 (2017)

[13] Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S.: A design science research methodology for information systems research. Journal of management information systems **24**(3), 45–77 (2007)

[14] Petrillo, F., Merle, P., Moha, N., Guéhéneuc, Y.G.: Are rest apis for cloud computing well-designed? an exploratory study. In: International Conference on Service-Oriented Computing. pp. 157–170. Springer (2016)

[15] Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J.C., Canali, L., Percannella, G.: Rest apis: a large-scale analysis of compliance with principles and best practices. In: ICWE. pp. 21–39. Springer (2016)

[16] Rodriguez, J.M., Crasso, M., Zunino, A., Campo, M.: Automatically detecting opportunities for web service descriptions improvement. In: I3E. pp. 139–150. Springer (2010)