universität
uulm

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme (DBIS)

Versioning of RESTful APIs and strategies for backward compatibility especially with regards to data management

Thesis at Universität Ulm

Submitted by:
Young-Keun Choi
young-keun.choi@uni-ulm.de
1049082

Assessor:
Prof. Dr. Manfred Reichert

Supervisor:
Rüdiger Pryss
Daniel Weidle
Marija Belova

2022

Version October 14, 2022

Set: PDF-LaTeX 2$_\varepsilon$

Acknowledgments

Firstly, I thank Prof. Dr. Manfred Reichert of University Ulm and Rüdiger Pryss of University Würzburg for their fast, direct, and effective academic supervision. In the same vein, I am very grateful for the personal, cordial, and thorough supervision Daniel Weidle and Marija Belova from adesso SE provided.

Outside of this work, I thank Alexander Raschke of University Ulm, and Benedikt Goos, Patrick Göhlich, and Carsten Richter from adesso SE for their professional guidance, which led me to this thesis.

Lastly, I thank my family and friends who greatly support me in my life endeavors. In particular, I thank my parents, who are my greatest supporters and are generally the greatest, and Helene Becker, who thoroughly proofread this work and directly contributed to its readability.

Abstract

The aim of this work is to develop a concept for zero-downtime deployment in software applications built on a REST API by combining strategies for REST API versioning and data backward compatibility. By combining strategies for REST API versioning and data backward compatibility into a concept, we discuss the advantages and synergies under the metrics of conformity with REST, scalability, and practicability, in an otherwise clouded and convoluted discussion. We ask to what extend existing strategies for REST API versioning and data backward compatibility contribute to zero-downtime deployment and how they can be compiled into a concept that guarantees zero-downtime deployment. We do so in order to streamline workflow and adapt to the modern context of agile software development while encouraging long-term API design, matching the idea behind REST. By implementing a demo project as a proof of concept, we are able to validate our concept and extrapolate practicability and advantages to usability, stability, and scalability. The significance of the concept is that it enforces versioning on the level of resource representations and thus limits the scope of a change increment, which directly affects backward compatibility and zero-downtime deployment.

Contents

1 Introduction

1.1 Motivation

> "*We also rectify errors even after the warranty period expires on software developed by us within a period of 10 years after acceptance.*" [16]
> – adesso SE's commitment to quality

Modern software companies, such as adesso SE, are tasked with developing software that is constantly evolving. Software products, such as Microsoft Word or Adobe Photoshop used to have final releases titled by their year of release. At present, Microsoft opted to release new versions of Word as part of Office 365, a line of subscription services, and the most recent Photoshop is now only available via subscription. In 2018, the International Data Corporation (IDC) predicted that 53% of all software revenue would come from subscriptions by 2022 [10]. In 2021, this forecast was updated to include a growth rate for the subscription business model of 17.9% for 2020–2025, representing 83% of total software revenue [17]. This emergence of the subscription business model is seen in almost every industry and is certified by the success of companies such as Spotify and Netflix, and has changed how software is developed: Interfaces between software components and data structures undergo revisions before and even after software release. This emphasizes the importance of interface versioning for streamlining workflow and managing the increase of features and the resulting increase in complexity.
A closely related issue presents itself when implementing changes in relational databases. When one of multiple software components, sharing database entities, is set to be updated, it oftentimes necessitates a change of said database entities. This issue becomes especially pronounced when work is also being done on the remaining software components. In an agile software development environment, it seems inappropriate to halt progress on all the related software components every

time a database change needs to be implemented, thus stressing the point of zero-downtime deployment. Backward compatibility in data management is essential in addressing this issue.

1.2 Goal

In this work, existing approaches and strategies for RESTful API versioning and data backward compatibility are discussed and evaluated, from which a viable concept for zero-downtime deployment is formulated. Additionally, adesso SE provides an internal project whose tech stack, interfaces, and data structure are modeled by a demo project as a form of abstraction. On this demo project, the formulated concept is applied and thereby validated as a proof of concept. Finally, based on the results, the concept is evaluated on its scope of applicability.
This document aims to answer the following questions:
Is it possible to specify a concept for REST API versioning and data backward compatibility that provides zero-downtime deployment?

1. What are the most common strategies and best practices for versioning of REST APIs?
2. What are the most common strategies and best practices for providing backward compatibility with regard to data management?
3. How can strategies for REST API versioning and data backward compatibility for zero-downtime deployment be implemented?

1.3 Context

This document is written as a bachelor's thesis at adesso SE for University Ulm. The topic was first drafted while working on adesso SE's internal web platform, *Careerfluence*, as a software project of University Ulm's bachelor's curriculum. While evaluating the data structure and planning database changes, the question of zero-downtime deployment came up naturally. Work was also being done on the frontend, and in order not to break existing functionalities, a versioning attempt was

made in a manner similar to Expand-Contract migrations, which will be discussed in a subsequent chapter. Afterwards, the question of API versioning and data backward compatibility was further pursued, culminating in the wish to write a bachelor's thesis on this topic. Fortunately, adesso SE expressed the need for a bachelor's thesis in this very field and agreed to supervise and assist me in this effort.
As an IT service management company in the context of modern software development, adesso SE has a need for zero-downtime deployment in its projects. The project, provided for abstraction in this work, and its development is bound to hardware manufacturing, which exhibits a clock rate for releases of new features and strongly discourages a halt in the respective production processes.

1.4 Structure

After a brief introduction of the fundamentals, we discuss existing strategies for REST API versioning and data backward compatibility. After discussing the various strategies, we compile the most promising ones into a concept. This concept is evaluated on the basis of the documentation of a proof-of-concept implementation. Finally, we conclude this work by summarizing our findings. The appendix includes relevant source code from the concept implementation.

2 Fundamentals

In the following, the many acronyms, terms, definitions and concepts used in this document will be explained.

2.1 API

An *Application Programming Interface (API)* is a software interface that allows two or more applications to communicate with each other through services and includes definitions and protocols for building and integrating new application software.
An API comprises a *contract* that defines the content required by the information provider and the content required by the information consumer. This establishes the response format between the API and clients and serves as documentation of the API.

2.2 HTTP

The *Hypertext Transfer Protocol (HTTP)* is an application layer protocol and is the foundation for communication on the Web as it supports the publication of hypertext documents with hyperlinks to other resources. It enables stateless communication over a reliable network transport connection, such as *Transmission Control Protocol (TCP)*, between client and server, which means that the session state has to be fully specified and communicated in a request from client to server.
An HTTP *message* consists of a *header* containing metadata and a *body* carrying the payload. A message from client to server is called a *request* and may be categorized into *request methods*. A few request methods, relevant for the implementation of this work, are the following:

- `GET`: requests representation of target resource and expects response with resource's state in message header and representation data in message body.
- `POST`: requests processing of target resource at specified URL according to its semantics by data enclosed in message body.
- `PUT`: similar to `POST`, but with specified target location on server.
- `DELETE`: requests the deletion of target resource's state.

A message from server to client is called *response* and includes a *status code* in its header which, together with the corresponding request, defines the semantics of the message body. A few status codes relevant for the implementation of this work are the following:

- `200 OK`: indicates that the request has been fulfilled; response header and content of body depend on the corresponding request.
- `201 Created`: indicates that the request has been fulfilled and leads to the creation of a new resource at given URL in the response header.
- `301 Moved Permanently`: indicated that this and all future requests should be directed to given URL in the response header; may change corresponding request method to `GET`.
- `308 Permanent Redirect`: similar to `301 Moved Permanently` but does not change request method.
- `400 Bad Request`: indicates that the request is invalid due to a client error and leads to server refusing action.
- `403 Forbidden`: indicates that the request is valid but client lacking permission or attempting prohibited action lead to the server refusing action.
- `404 Not Found`: indicates that the requested resource could not be found.
- `500 Internal Server Error`: indicates that an unspecified server-side error lead to the server refusing action.

Resource Versus Representation

As an abstraction of information, a *resource* is any information that can be named; it can be a document, an image or anything else.
The idea behind *resource representations* is best described with the help of an example by Leonard Richardson and Mike Amundsen: “A pomegranate can be an HTTP resource, but you can’t transmit a pomegranate over the Internet.” [13, pg. 30] The pomegranate in the example is the concept. Through the use of resource representations, we are able to transmit the appropriate abstraction of information about a resource to a recipient. A user client may request an image of said pomegranate, while an application client may prefer a document describing the pomegranate.

URI Versus URL

A resource is identified by a *Uniform Resource Identifier (URI)*. However, a resource identified by a URI does not have to have a representation.
A *Uniform Resource Locator (URL)* identifies a resource in the same way, but it directs a computer to a representation of the underlying resource.
Every URL is a URI but not every URI is a URL. An example of a URI that is not a URL is given by Leonard Richardson and Mike Amundsen as the ISBN of a book. It references an abstract concept of a book but not a particular copy of a book. A computer cannot simply follow the ISBN to find a representation.

Media Type

A *media type* is an identifier for file formats and was originally defined in RFC 2045[1] as part of the *Multipurpose Internet Mail Extensions (MIME)* specification for specifying email message content types. They mainly consist of a type, a subtype and an optional suffix: `{type}/{subtype}+{suffix}`. The *Internet Assigned Numbers Authority (IANA)* is its official standardization authority and is responsible for the registration of these media types.
In his blog post, *RESTful APIs and media-types*, Rob Allen distinguished between three uses of media types [1]:

[1] `https://datatracker.ietf.org/doc/html/rfc2045`

- **Plain**: registered by IANA, such as `application/json`.
- **Vendor-specific**: registered by an organization for their own products, such as GitHub with `application/vnd.github.v3+json`; `vnd` stands for *vendor*.
- **Standard**: registered by an organization as a standardization effort, such as *Hypermedia Application Language (HAL)*[2] with `application/hal+json` for standardized enabling of hypermedia controls.

Content Negotiation

Content negotiation refers to an HTTP mechanism that allows for clients to be served different resource representations at the same URI. As an example, a client may specify the media type of the response body through the `Accept` HTTP header or the language of a requested resource representation through the `Accept-Language` HTTP header.

Profile

RFC 6906[3] defines a link relation called *profile* which is formally "defined not to alter the semantics of the resource representation itself, but to allow clients to learn about additional semantics (constraints, conventions, extensions) that are associated with the resource representation, in addition to those defined by the media type and possibly other mechanisms." As a standardized form of documentation, this relation is registered with the IANA, which allows for its inclusion in hypermedia control.

2.3 REST

Representational State Transfer (REST) is a set of architectural style principles conceptualized by Roy Fielding in his dissertation "Architectural Styles and the Design of Network-based Software Architectures" in the year 2000 [7, pg. 76] which are summarized as the following:

[2] https://datatracker.ietf.org/doc/html/draft-kelly-json-hal-00
[3] `https://www.ietf.org/rfc/rfc6906.txt`

- **Client-Server**: In order to improve portability of the user interface and scalability of server components, the client-server constraint allows for separation of concerns and separates concerns between user interface and data storage.
- **Stateless**: In order to provide visibility, reliability, and scalability, the session state is kept on the client and not on the server. Each request from a client to the server must therefore contain all the necessary information to process the request.
- **Cache**: In order to improve network efficiency, data in a response to a request must be implicitly or explicitly labeled as cacheable or non-cacheable, allowing clients to reuse the response data in the former case.
- **Uniform Interface**: In order to improve simplicity of the architecture and visibility of the component interface, uniform interfaces between components are emphasized. In practice, this translates to the decoupling of implementations from the services they provide and encourages independent evolvability while accepting the drop in efficiency resulting from the standardized form rather than a custom-fit one. This is accomplished by following four constraints, which will be expanded upon in the subsequent sections:
  - **Identification of Resources**
  - **Manipulation of Resources through Representations**
  - **Self-Descriptive Messages**
  - **Hypermedia as the Engine of Application State (HATEOAS)**
- **Layered System**: In order to improve scalability, the architecture is composed of hierarchical layers that are constrained in component behavior and can only interact directly with the adjacent layers. This caps system complexity and allows for the encapsulation of services. The latter of which also enables the employment of intermediaries that store infrequently used functionality and perform load balancing of services across networks and processors.
- **Code-On-Demand**: In order to provide simplified clients, client functionality can be extended by downloading and executing code in the form of applets or scripts. In this way, clients can be deployed with a reduced number of

pre-implemented features. Due to it also reducing visibility, this is an optional constraint.

Following these principles and applying them to a stateless, client-server communication protocol such as HTTP, a widely accessible and scalable REST API can be built.

Identification of Resources

As described by Roy Fielding, resources serve as a key abstraction of information in REST, which enables key features of the Web architecture [7, pg. 88ff.]: The resulting generality, by encompassing many sources of information of any type or implementation in the form of representations, allows for content negotiation to take place. This also allows for referencing the concept rather than a representation of that concept, and thus removes the need to change existing links whenever the representation changes.
In REST, resources are identified by URLs: “Without a URL, you can’t get a representation. Without representations, there can be no representational state transfer” [13, pg. 51]. This URL provides the name and network address of a resource, which are used by clients for manipulation of the resource.

Manipulation of Resources through Representation

Resources are conceptually separate from the representations the clients receive. A resource may have different representations for different clients, as explained earlier. Clients interact with a resource by modifying and sending these representations to the server, which in turn carries out the necessary state change.

Self-Descriptive Messages

Each request and response must contain additional information on how to process the payload. This is done through the use of HTTP headers (`Content-Type` for media type, `Content-Length` for the size of the payload) or in the payload itself in the form of metadata.

Hypermedia as the Engine of Application State (HATEOAS)

A resource's state, its representation, may include links to other related resources which can be traversed by a client. As these links hint at or dictate possible subsequent actions, their presence or absence is an important part of a resource's representation of its current state.

2.4 Versioning

API *versioning* is the practice of effective and transparent API change management in which changes are properly planned and communicated before implementation of said changes.

Breaking Change

When talking about versioning, the first question that needs to be answered is: *when is it necessary to version?*
The need for versioning arises from changes to the API. We differentiate between *breaking* and *non-breaking* changes. Breaking changes break existing functionalities if they were to be implemented in the API's current version and thus necessitate an up-versioning of the API. Tim Kleier categorizes those into the following [11]:

1.	Change to request/response format	*e.g. XML to JSON*.
2.	Change to property name or data type on property	*e.g.* `name` *to* `productName` *e.g. integer to float*.
3.	Addition of a required field on request	*e.g. adding new required header or property in a request body*.
4.	Removal of a property on response	*e.g. removing description from a product*.

API Change Management

Effective *API change management* is about handling changes in a manner that does not negatively impact an API's functionality and usability. Non-breaking changes are easily introduced to an API as they do not break an API's functionality but should be documented nonetheless. For breaking changes, Tim Kleier summarizes effective API change management to the principles [1]:

- Continue support for existing properties/endpoints.
- Add new properties/endpoints rather than changing existing ones.
- Thoughtfully sunset obsolete properties/endpoints.

Under the keyword of *API Version Lifecycle Management*, Brajesh De suggests to release a new API version as a beta release first, which proves useful in being able to obtain feedback on the changes before rolling the changes into production. After a successful launch of the beta version, the changes can then be tested in production while supporting the old and new versions simultaneously, before the old version is deprecated at a communicated point in time. [3, pg. 109f.]

SemVer

A version identifier uniquely identifies the version of an application. The only versioning scheme relevant for this work is *semantic versioning (SemVer)*, which consists of a three-part version number: `MAJOR.MINOR.PATCH`.
An increment of the

- `MAJOR` number represents a breaking change.
- `MINOR` number stands for the addition of non-breaking features.
- `PATCH` number represents non-breaking bug fixes.

Thus, this versioning scheme carries semantics about risk to compatibility and functionality. Versions sharing a major version number are guaranteed to be compatible, while a difference in the major version number might suggest incompatibility.

Scope

Tim Kleier categorizes changes into the following *scopes* by impact on the API [11]:

- **Leaf**: change to an isolated endpoint with no relationship to other endpoints
- **Branch**: change to a group of endpoints or a resource accessed through several endpoints
- **Trunk**: change to most or all endpoints; application-level change
- **Root**: change to all API resources of all versions

Changes on the scope of a *leaf* can be handled through effective API change management. For a *branch* change, the dependency structure between the affected endpoints has to be accounted for or can be overcome by introducing redundancy to allow for a smooth transition to the new version. *Trunk* changes come from a change in format, specification, or protocol and warrant a change in the overall API version. A *root* change describes a change to an API significant enough that consumers of all versions of the API have to be notified of the change.

2.5 Compatibility

> "Compatibility is a relationship between one process that encodes the data, and another process that decodes it." [12, pg. 128]

In the context of software, *backward compatibility* describes the property of a piece of software that is compatible with previous versions of itself. This includes the conservation of interfaces and mapping of data structures between versions. On the other hand, *forward compatibility* describes the property of a piece of software that is compatible with future versions of itself. This includes some form of schema flexibility.

2.6 Liquibase

Liquibase is an open-source database-independent database schema migration tool. Database changes are documented in text files called *changelog files*, which

are labeled with an *id* and *author* tag. These changes are logged to a *database changelog* table, which serves as a database schema version history as well as a reference for database schema consistency. A *database changelog lock* table manages access to the database changelog. In combination, these tables allow for collaborative work of multiple developers on different branches of one database.

3 REST API Versioning

Versioning is a crucial part of an API's design as a well thought out strategy allows for an API to scale.

In *Creating Maintainable APIs: A Practical, Case-Study Approach* [19, pg. 110], the author highlights the principal sections of a REST API that could evolve in a REST API and require versioning:

- **Resource locators (URLs):** endpoints of a REST API
- **State transitions:** protocol changes
- **Message payload**

The following versioning strategies will address these aforementioned sections.

3.1 URL

Mentioned as the most common convention by Ervin Varga in *Creating Maintainable APIs: A Practical, Case-Study Approach* [19, pg. 111], Leonard Richardson and Mike Amundsen in *RESTful Web APIs* [13, pg. 186], and Tim Kleier in *How to Version a REST API* [11], specifying the version number in the URI path partitions it into sets, each designating a particular version of the API, and is referred to as versioning the *trunk* of an application by Tim Kleier:

`https://example/api/v1`.

This partitioning allows for clients to work solely with resource representation associated with a specific version. Conversely, resource representations are considered immutable; new representation requires a new version and a new URI space. This allows for clients to easily cache resources as new versions are entered as new

entities with different cache keys. With the URI as the cache key, new and old versions are cached separately. This approach also enjoys great browser compatibility. However, this also proves to have the downside of having a big footprint in the code base as introducing breaking changes implies branching the entire API and having to carry over non-changed endpoints. Additionally, this kind of versioning needs well-orchestrated releases of versions due to the big impact each release has on the system as well as the consumer. Thus, this approach is said to be about decent for APIs with relatively low volatility [11].

A similar effect can be achieved by partitioning the URL into subdomains:

`https://v1.api.example.com`.

This approach allows for hosting different versions on different servers, supporting region-specific software releases and allowing for solutions that warrant compatibility with regional infrastructure. However, as Brajesh De writes in his book *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*, apart from clients having to change the URL to gain access to a resource, they might have to make changes to their security settings due to the change in the hostname [3, p. 109].

3.2 Query Parameter

The following two approaches also put versioning information in the URI path but as query parameters to a resource's representation.
The most straight-forward approach using query parameters is to add the version number as a query parameter, following the template:

`https://example/api/<endpoint>?version=1.4`.

This approach is fairly easy to implement and understand, and it is easy to default to the latest version since query parameters are optional. It comes with flexibility in terms of requesting a resource representation of a specific version and is versioning at the *leaf* level of the application. However, versioning at the leaf level comes at the price of the application's version number generally being out-of-sync with its endpoints' and resources' version numbers. This approach necessitates some form

of transformation logic in order to transform resource representations based on the specified version [3, pg. 109].

Another, less intuitive approach is to use API keys for versioning, as described by Ervin Varga [19, pg. 112], following the template:

`https://example/api/<endpoint>?key=<API key>.`

While API keys are primarily used for identifying the caller of a service in the interest of authentication and authorization, Ervin Varga deems it possible to partition the API key space based on versions. As API keys are generated by the server as opaque binary tokens and cannot be interpreted by clients, it would be necessary for a client to indicate the desired version during registration at the server. The server would then generate an API key with all the required version information. Finally, using this API key, all requests would refer to that specific version of the application. Since the API key is generated by the server and is opaque by nature, it is possible to encode a greater amount of versioning information than with previous approaches. This means that the aforementioned issue can be addressed to a degree by encoding version information of endpoints and resources into the API key. The server has information about the dependencies and can serve up the right versions when given the necessary pieces of information by the client. The downside presents itself in mixing versioning information with access-control aspects and thus adding a point of vulnerability from a security standpoint.

3.3 Custom Header

Preserving the URI between versions, a REST API can be versioned using a custom header containing the version identifier as an attribute:

```
Accept-version:  v1
```

Using this approach, more granularity can be achieved in requesting a specific version of a resource. The downside is that this approach is less transparent than the previous options since the version information is buried in the request object itself and it is not clear if the version refers to the one of the endpoint or the API. It should, however, be noted that this in itself is a change to the standard HTTP protocol and

firewalls might remove response header fields[1]. A custom firewall configuration might open up a security vulnerability.

3.4 Media Type

Similar in versioning scope to the previous strategy is REST API versioning using media types. As the name suggests, it leverages the content negotiation mechanism to select a specific version of the API by the media types. For versioning, Ervin Varga distinguishes between using existing media types with versioning support and the possibility to create domain-specific media types with versioning support [19, pg. 116]. He argues that the former one proves to be difficult since very few existing media types support versioning, and advises against picking an arbitrary media type for versioning support. He mentions that the latter approach is not scalable. One reason might be that for each version of every resource representation that needs to be versioned, a new custom media type has to be created, documented, and maintained. For a custom media type, denoted by the prefix `vnd.` for *vendor-specific*, the `HTTP Accept` header of a request would look like the following:

```
Accept: application/vnd.example-media-type.v1+JSON
```

It should also be noted that using custom media types requires a bit of configuration of the server: the "server must set the `Vary` HTTP response header [in order to] signal to any intermediaries (proxies, gateways, etc.) that the same URL might return different responses based on the Accept header (in this case, a pure URL-based caching will not work)." [19, pg. 116].

3.5 Profile

In their book *RESTful Web APIs: Services for a Changing World* [13, pg. 187], Leonard Richardson and Mike Amundsen recommend using a standardized media type and versioning the associated profile instead. Citing RFC 6906, "[a] profile is

[1] As an example, Oracle Cloud Infrastructure requires one to register custom headers for the firewall: `https://docs.oracle.com/en-us/iaas/Content/WAF/Origin/custom_headers.htm` (Retrieved 17 August 2022).

defined to not alter the semantics of the resource representation itself, but to allow clients to learn about additional semantics... associated with the resource representation, in addition to those defined by the media type." [13, pg. 135].
The authors note that a profile isolates parts of the application that threatens clients to break when changed from parts that are adaptable, thanks to them being described by hypermedia. The versioning aspect comes in the form of keeping multiple sets of profiles which correspond to different versions of resource representations. [13, pg. 187] This sentiment is echoed by Ervin Varga in his book *Creating Maintainable APIs: A Practical, Case-Study Approach* [19, pg. 116]. A profile can be linked using the respective `profile` HTTP link relation type or can be passed as a parameter of the `Content-type` HTTP header. An example of a versioned profile in ALPS[2] format can be found in Listing A.1 and Listing A.2, wherein the former contains the API version and the latter the resource representation version. It has to be noted that granting open access to profiles poses a security vulnerability[3].

[2] Application-Level Profile Semantics (ALPS). Retrieved 16 August 2022, from `https://datatracker.ietf.org/doc/html/draft-amundsen-richardson-foster-alps-00`.

[3] An example of what can be accessed through profiles is listed on a cyber security blog: `https://niemand.com.ar/2021/01/08/exploiting-application-level-profile-semantics-apls-from-spring-data-rest/` (Retrieved 16 August 2022).

4 Data Backward Compatibility

As described by Michael Pratt in his blog post *Ensuring backwards compatibility in distributed systems* [15], issues with data backward compatibility mainly arise in relational databases since their tables have strict schemas, leading them to reject data that does not exactly conform and have foreign key constraints amongst themselves.
Non-relational databases are referred to as *NoSQL* databases. They are designed to store and query unstructured data and thus do not require strict schemas, which is why they are described as being *schemaless*. Also, without constraints between collections of data, new tables or fields can usually be added to schemaless databases without having to worry about compatibility. The removal of tables or fields has to be sensibly communicated, error-handled, and reflected in code.

Reasonably, the question of data backward compatibility mainly concerns relational databases as this is not an issue for schemaless databases.

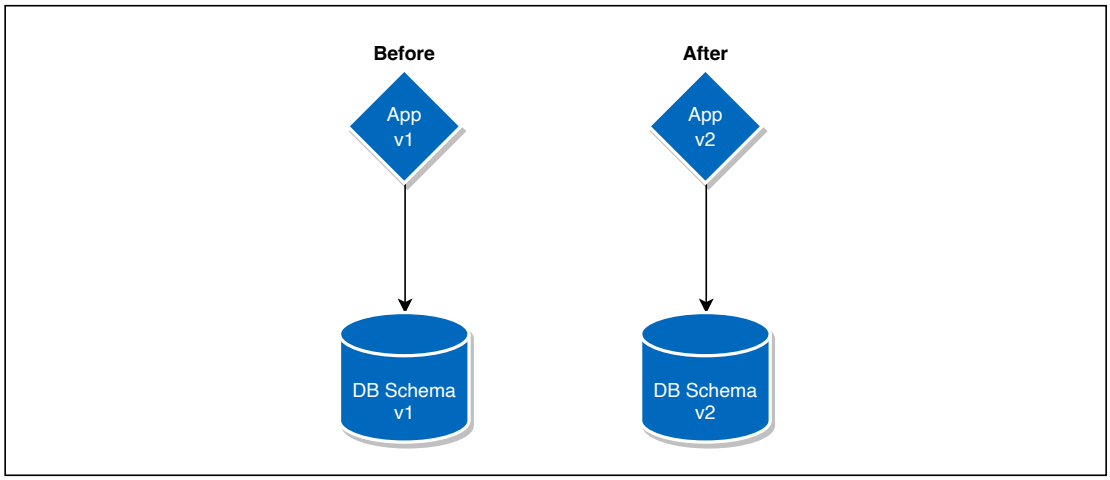

Figure 4.1: DB schema migration [9, pg. 13]

In chapter six of the book, *Feature Flag Best Practices* [9, pg. 13 ff.], Pete Hodgson and Patricio Echagüe outline approaches for incremental, backward-compatible database changes. As an abstraction, such a change can be described as a transition of each of the components, app and database schema, from version 1 to version 2, as can be seen in Figure 4.1. The approaches only vary in the method and order by which these transitions are performed.

4.1 Code First

In the *code-first* approach, the code of the to-be-deployed app v2 is backward compatible with the old database schema v1. In this way, the new code can be deployed on the existing database schema v1, as shown in Figure 4.2. Afterwards, the old code v1 can be retired, and finally, the old database schema v1 can be migrated to the new version.

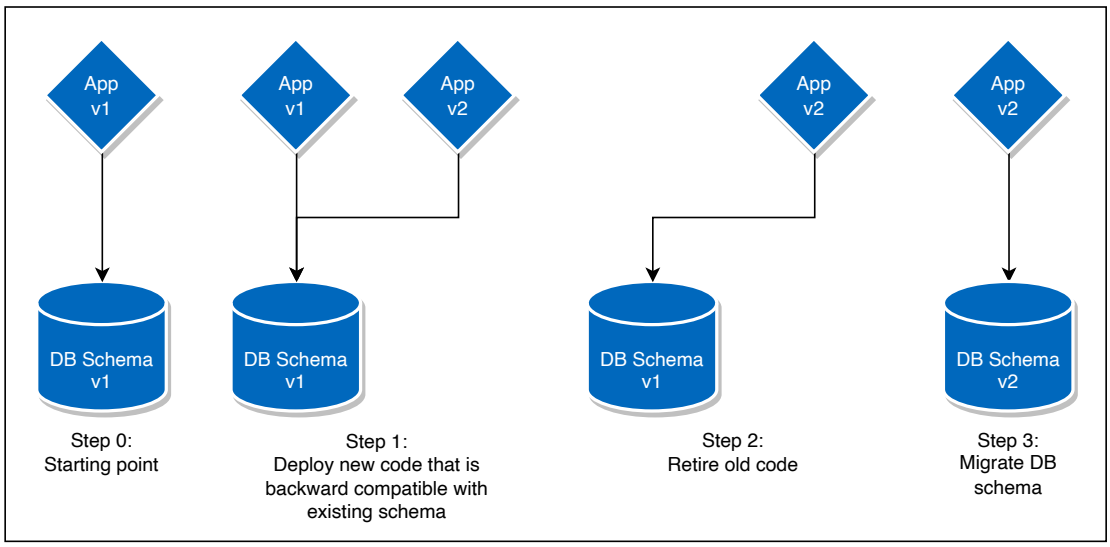

Figure 4.2: Code-first approach [9, pg.14]

4.2 Data First

In the *data-first* approach, the database migration is performed first, as shown in Figure 4.3. However, it needs to ensure that the new schema v2 is backward compatible with the existing code v1. Afterwards, the new code v2 can be deployed alongside the old code v1 until, finally, the old code v1 is retired.

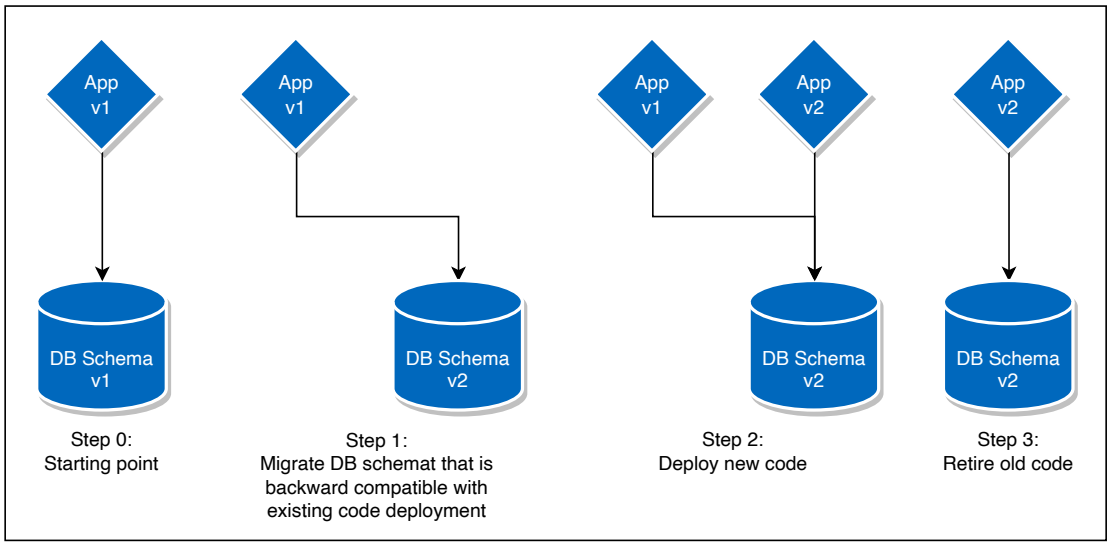

Figure 4.3: Data-first approach [9, pg.14]

4.3 Big Bang

For smaller and simpler systems, the *Big-Bang* approach can be viable for migrating the database and the corresponding code without having to worry about forward or backward compatibility. New versions of code and database schema can be developed independently of their old versions, as can be seen in Figure 4.4: The system is brought to a halt as the database schema is migrated to v2. It is then started back up when code v2 that supports the new schema is ultimately deployed onto the new system. In contrast to the other approaches, this one does not guarantee zero downtime deployment in favor of not having the load generated by having to ensure compatibility between components of different versions.

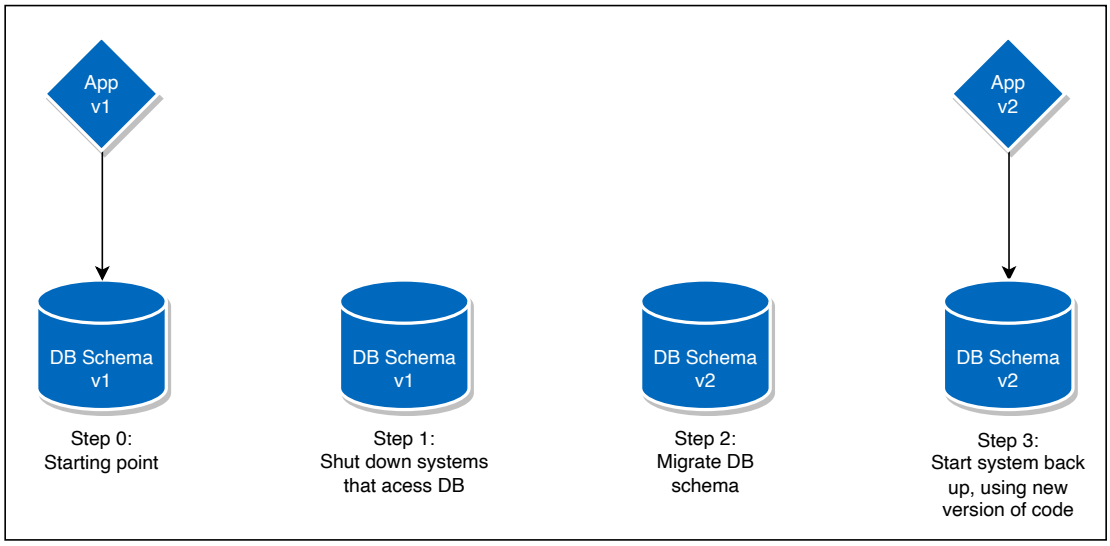

Figure 4.4: Big-Bang approach [9, pg.15]

4.4 Expand-Contract Migration

What is referred to as *Expand-Contract Migration* or *Parallel Change*, describes a series of forward- or backward-compatible changes. This approach combines the first two approaches as an initial data-first *expansion* is conducted in order to accommodate the code changes, followed by a code-first *contraction* that removes superfluous aspects.
This is best explained using the example featured in Figure 4.5:

1. First, a data-first-driven expansion of the database schema is performed by adding the new `shipping_address` table, as well as adding a new nullable attribute `shipping_addr_id` as a foreign key to the `order` table. At this point, no new code is necessary since the change is fully backward compatible with the existing code.

2. Then a code-first change is performed, making the new code write to both the old `shipping_addr...` columns in the `order` table as well as the new `shipping_address` table. This, again, is fully backward compatible with the existing database schema.

3. Now, a one-time data migration is performed to add rows to the `shipping_adress` table for each existing entry in the `order` table, linked back to the `order` via the

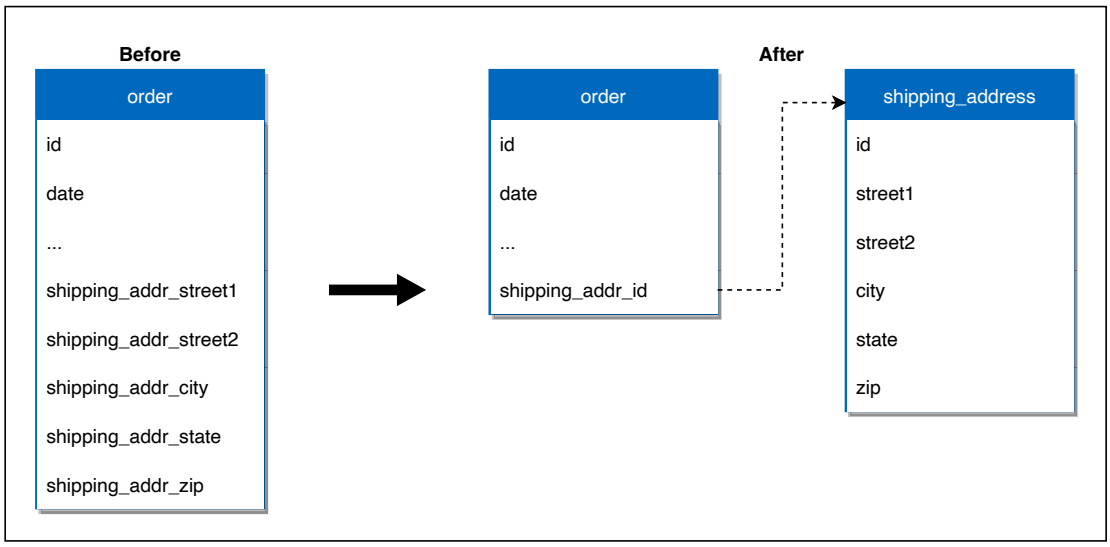

Figure 4.5: Expand-Contract migration [9, pg.16]

`shipping_addr_id` foreign key. The nullable property can safely be removed from `shipping_addr_id` as it will always be set.

4. Lastly, now that all the existing data is in the new `shipping_address` table, a code-first contraction of the schema is performed by making the code read-only from the new `shipping_address` table. The old `shipping_addr...` columns in the `order` table can be safely removed, as no code is referencing them.

As the authors describe [9, pg. 17], during phases 2 and 3, the system needs to support the old and new schema simultaneously, which entails two points of caution. For one, the system in this phase needs to perform *Duplicate Writes*, writing both to the old as well as the new schema on update or create request. Lastly, the authors advise to perform so-called *Dark Reads* for complex migrations, in which data is read from both the old and the new schema and compared to make sure everything is in order and the data is reliable. On detection of incorrect data, the system can be rolled back to the old schema while debugging the problem. As this intermediate phase is taxing on system resources, it is desirable to keep it as short as possible while making it as long as necessary.

5 Concept

For REST APIs, the answering of the question of versioning has become more of a religious undertaking since it is not a protocol but an architectural style, making it hard to find a consensus on the best strategy for versioning of REST APIs. As Ervin Varga puts it: “In REST APIs you have to decide the location of version numbers, hence the versioning strategy does affect the API. The major drawback of this extra flexibility (obviously too much freedom isn’t always beneficial) is the lack of common consensus on how to annotate a REST API with a version number.” [19, pg. 110] Thus, the aspect of versioning cuts deep into API design. Generally, the *best* versioning strategy is, at best, specific to a project and its context. However, standardization of certain components would certainly benefit interoperability. In this work, we will be outlining a general concept that is not specific to a project type and will point towards a few standardization efforts that greatly improve RESTfulness of APIs when implemented correctly. By putting an emphasis on the *RESTfulness* of the API we are trying to version, the versioning strategy also has to adhere to the REST principles. By choosing this approach, we try to reap the benefits of REST APIs outlined by Roy Thomas Fielding in his dissertation, *Architectural Styles and the Design of Network-based Software Architectures* [7].

Regardless of how the API is versioned, managing data changes in a backward compatible manner is critical in providing zero-downtime deployment. Therefore, we evaluate existing strategies on the basis of this metric and discuss how the chosen strategies fit together to form a concept for robust, scalable REST APIs that will be evaluated in the next chapter.

5.1 Discussion

As mentioned before, versioning by partitioning the URL is the most common approach and is used by big companies, such as Twitter [18], Facebook [5], Google [2] and Yahoo [20]. As web service providers, they benefit from the advantages of this versioning scheme in that it is easy to understand and implement. Isolating different versions from each other guarantees that they remain functional when deployed alongside each other, allowing zero-downtime deployment while not having to worry about compatibility between the versions. As Leonard Richardson and Mike Amundsen put it: "A client that doesn't know what `v2` means won't follow the link. Partitioning works because representations found beneath `/v1` only link to resources found beneath `/v1`. Both versions of the API probably use the same underlying code, but they can have completely different application semantics, because any given client will use one or the other exclusively." [13, pg. 186]. However, as Harihara Subramanian and Pethuru Raj state, this approach directly opposes REST principles, as two different URIs would reference the same resource [8, pg. 188f.]; "the URI should stay the same regardless of the many changes its representations go through" [8, pg. 74]. URI path parameters are used to identify a specific resource while a version addresses a specific representation of that resource. Fielding himself wrote in his dissertation that the abstraction of a resource "allows an author to reference the concept rather than some singular representation of that concept, thus removing the need to change all existing links whenever the representation changes." [7, pg. 89]. Mark Nottingham writes that "intermingling the version into the identifier" [14] and thus embedding information that is likely to change makes them unstable and thereby reduces their value. He also states that this approach is rather limiting in that it doesn't allow for individual parts of the system to evolve independently [14]. From a versioning standpoint, the problem with this approach lies in its scope[1] as it is only really applicable when a considerable portion of the API goes through a breaking change. This should not be the case for most APIs, as this speaks of bad, short-term-minded API design. In the understanding of a REST API, a resource versioned through this approach just happens to share the same name as a resource of another version but they are conceptually different resources.

[1] Versioning resource representations using this approach is possible but turns out very inelegant and is all the more sacrilegious in the context of being RESTful, e.g.: `https://example/api/v1.4/containers/147/v1.7/items/123`.

Reasonably, this leaves approaches that version resource representations for consideration.

While related to the previous approach, the use of query parameters for versioning addresses resource representations and thus does not come with the problem of unstable URIs. For resource representations, this flexibility in implementation details is rather welcome. This approach has the advantages of versioning using URLs, such as ease-of-use, caching and browser compatibility, while adhering to the REST principles and allowing for more granular versioning.
Encoding the version number in the API key does not add anything to this proposition, especially if an API does not intend to use API keys in the first place, but makes the URL less transparent to the client. Directly adding the version number as a query parameter allows for different resource representation versions to be transparently referenced using hypermedia and is the chosen strategy for the concept.

Custom headers do not clutter the URL, but they are a change to the communication protocol, which Roy Thomas Fielding himself advises against: “A REST API should not contain any changes to the communication protocols aside from filling-out or fixing the details of underspecified bits of standard protocols, such as HTTP’s PATCH method or Link header field.” [6]. Aside from this, the necessary firewall configuration, which one might not have access to, and the consequential security concerns make picking this approach all the more daunting when compared to the previous approach.

Before talking about versioning the media type, we must look at media types from an API design perspective and take a small excursion:
Roy Thomas Fielding wrote that “[a] REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types).” [6] This emphasizes that media types play a central role for REST APIs. In particular, the last part is to be noted, as for most REST APIs the set of standardized media types is sufficient. Leonard Richardson and Mike Amundsen [13, pg. 60] generally advise against defining custom media types [13,

pg. 60], and Ervin Varga echoes this sentiment, as they argue that "[i]ntroducing a domain-specific media type is only feasible if it really embodies a large part of the application and protocol semantics. In this case, it should be generalized, and published for the others to reuse." [19, pg. 116] Adding to this, the former two authors argue that "[p]rotocol semantics deal with HTTP requests, but application semantics refer to things in the real world, and computers are terrible at understanding the real world. At some point, we must bridge the semantic gap by writing prose that explains our application semantics. There's nothing analogous to hypermedia that will save us." [13, pg. 138]
The existence of the *semantic gap* discourages forcing too many API aspects into (custom) media types and expresses the need for documentation. If we extend this to versioning, we realize that defining custom media types just to bury a version number in them probably is not the most reasonable idea, and we recognize the need for up-to-date documentation.

Following this line of reasoning, we arrive at profiles as documentation that covers application semantics; profiles that can be versioned. Basing a REST API around a standardized media type that allows hypermedia controls and machine-readable profiles as described by Leonard Richardson and Mike Amundsen [13, pg. 187] allows for the most RESTful of APIs as it is designed around HATEOAS. The biggest problem is the fact that there has yet to be any implementation or documentation of a purely hypermedia-driven API of this kind. Even the format for application level profile semantics (ALPS) submitted as an IETF Internet draft in 2015 by Leonard Richardson and Mike Amundsen themselves seems to enjoy little to no recognition and the corresponding website[2] looks very much abandoned. Not only is it hard to find other proponents of this kind, but frameworks that allow for easily human-readable formatting and navigating of said documentation for ALPS profiles are functionally non-existent. There is yet to be a lot more standardization in this field before using profiles in a truly self-evident way becomes a reality.

Martin Kleppmann writes in his book, *Designing Data-Intensive Applications*, a change to an application's feature often entails a change to the data it stores [12, pg. 111]. For that matter, the chosen strategy for evolving the database is to use Expand-Contract migrations as they guarantee backward compatibility in every step, which directly contributes to the goal of zero-downtime deployment. This approach

[2] ALPS. Retrieved 17 August 2022, from `http://alps.io/`

combines the advantages of the code-first approach and the data-first approach while being much more scalable and applicable than the Big-Bang approach; the latter actually directly opposes zero-downtime deployment. If multiple versions need to be deployed alongside each other, tagging an old version with a lifetime can be naturally employed for the intermediate phase of the old and new versions coexisting in Expand-Contract migrations.

Pete Hodgson and Patricio Echagüe note that “some database changes are just too big and still need to be scheduled and change-controlled in a more traditional fashion.” [9, pg. 17] This sentiment is shared by Martin Kleppmann, as he argues that in large applications, these changes cannot be implemented in one go [12, pg.111f.].
For server-side applications he suggests performing *rolling upgrades*, also known as *staged rollouts*, deploying new versions to a few nodes at a time and checking whether the new version runs smoothly. This supports zero-downtime deployment and allows for more frequent releases and better evolvability. Splitting the planned upgrade into controllable increments that take into account dependencies between affected components also allows for more effective change management.
For client-side applications, Martin Kleppmann deems the user responsible for upgrading at his convenience, and the user may decide to not upgrade for a while. This means that old and new versions of parts of the system need to be supported simultaneously, at least for a time. For such a system to function properly, backward compatibility and, to some degree, forward compatibility need to be provided.
Fortunately, backward compatibility is naturally given in Extract-Contract migrations. The versioning strategy using query parameters has the advantage that it allows for different versions to run in parallel and ensure compatibility with older clients in this way. With forward compatibility, the goal is to reduce the need of having to move to a newer version in the first place, so we are talking about dealing with non-breaking changes; mainly an old client’s code is expected to ignore additions made by newer code. With server-side backward compatibility covered, the old client’s request should at worst be answered with resource representations with additional fields in the body of the response.

5.2 Summary

Versioning strategies partitioning the URL are more applicable for API migrations rather than for versioning intentions. Custom headers are a change to the protocol with little benefit. Discarding these strategies that directly oppose REST principles, we are left with strategies using query parameters, media types, and profiles. Using custom media types is only reasonable for a very small subset of specific APIs, and relying on profiles for versioning is practical only in theory as standardization efforts have yet to catch up. In the context of this work, with general applicability, implementation cost, and scalability in mind, the most feasible versioning strategy presents itself to be using query parameters.
We simply spoke about strategies that adhere to the REST principles. Making an API truly RESTful by adding hypermedia is outside the scope of this work, as we are only concerned about the versioning aspect, but can be done as described by Leonard Richardson and Mike Amundsen in the chapter *Adding Hypermedia to an Existing API* of their book *RESTful Web APIs: Services for a Changing World* [13, pg. 190f.]. If we were to design a REST API from scratch, using standardized media types, using HTTP `link` header relation types defined in RFC 5829[3] for version history references, and using profile links linking to up-to-date documentation, we would allow for a hypermedia-driven design. This goes to show that the chosen approach does not get in the way of versioning a truly RESTful API.

For database changes, we elect to use Expand-Contract migrations. Bigger server-side application changes are split into more controllable and manageable chunks and documented accordingly using semantic versioning. For the convenience of clients, older versions that need to be run in parallel with newer versions are tagged with a lifetime that is reflected in the documentation.

[3] `https://tools.ietf.org/html/rfc5829`

6 Proof of Concept

The devised concept will be evaluated through direct implementation in a demo project, which itself will be modeled after a project in production, provided by adesso SE. The implementation of the demo project can be found on GitHub under the following link: `https://github.com/Cifer0/user-service`.

6.1 Description

The demo project models a user service as a web application. Through the implementation and evolution of this service, versioning of REST APIs using query parameters and providing data backward compatibility using *Expand-Contract migrations* are being evaluated. The demo user service will undergo three changes, resulting in four versions, each labeled according to semantic versioning. The first change is a change of contract and describes an API change, handled in a forward and backward compatible manner. The latter two changes do not change the contract and describe a database change.

Architecture

The services of the project provided by adesso SE are constructed as a *Microservice Architecture*, where the application comprises a collection of services which are developed, deployed, and ultimately maintained independently of each other. One such service of interest is built as a *Hexagonal (Ports & Adapter) Architecture*, visualized in Figure 6.1. It achieves separation of concerns by isolating external components from the domain logic using so-called (input and output) ports and

adapters, and allows for great testability and scalability of the domain logic implementation. A principle of a Hexagonal Architecture is *dependency inversion*, where high-level modules do not depend on low-level modules but depend on abstractions, such as interfaces, instead. In order to benefit from such an arrangement, the application requires a certain level of complexity.

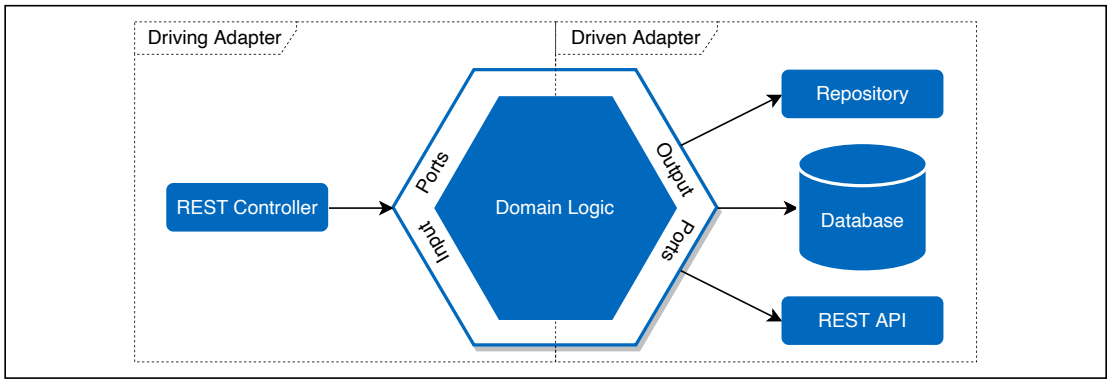

Figure 6.1: Hexagonal Architecture

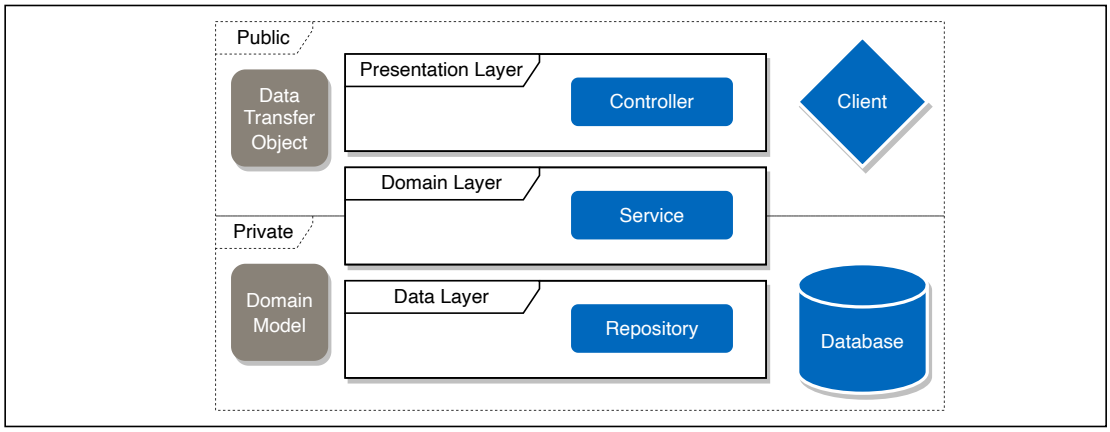

Figure 6.2: Layered Architecture

For the demo project, this complexity is not given, as it is an abstraction of the service of interest. Instead, the demo project is designed in a *Layered Architecture*, as shown in Figure 6.2. Separation of concern is still accomplished with each of the vertically aligned layers fulfilling a certain role and providing an abstraction of the work done for the adjacent layers. Controllers of the Presentation Layer and services of the Domain Layer communicate through *Data Transfer Objects (DTOs)*,

such as JSON objects, while services and repositories of the Data Layer communicate through *Domain Models*, such as entities.

Data Model

Version 1.0: Simple API

A `user_data`-entity is represented by the attributes `id`, `username`, and `full_name`:

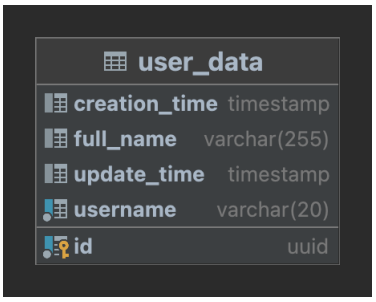

Figure 6.3: Database Schema v1.0

The API supports `GET`, `POST`, `PUT`, and `DELETE` methods via the endpoint: `user/{username}`, and fetches, posts, updates, and deletes a user respectively.

Version 2.0: Versioned API

A `user_data`-entity is represented by the attributes `username`, `full_name`, `first_name`, and `last_name`:

This change supports a version of `user_data`-entity described in Version 1.0, and can be addressed by specifying the version as a query parameter in the following way: `user/{username}?version=1`. The new version of `user_data`-entity, including only the attributes `username`, `first_name`, and `last_name`, can be addressed by specifying the new version as a query parameter in the following way: `user/{username}?version=2`. The new default version without any query parameters includes all attributes for `GET` and `DELETE` and determines the response version

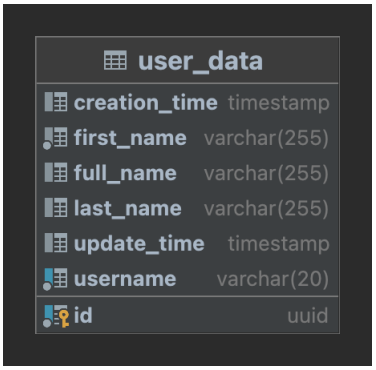

Figure 6.4: Database Schema v2.0

by the format of `POST` and `PUT` requests. In this way, the API is backward compatible. It is also forward compatible as clients running the old version still receive the expected data, even when they don't specify the version, and can simply ignore the newly added fields.

Version 1.0 will no longer be supported in the [next version]. Therefore, it is necessary to perform a one-time data migration, writing `full_name` to `first_name` and `last_name` for entries created in Version 1.0. The new `name`-entity mirrors the latter two attributes of the `user_data`-entity in preparation for the database migration. The migration can be performed using the endpoint `users/migrate`.

Version 2.1: Deprecation and Migrating Database

A `user_data`-entity is represented by the attributes `username`, `first_name`, `last_name`, and a nullable foreign key `name_id`. The latter links to a `name`-entity, represented by the attributes `id`, `first_name`, and `last_name`:

In this version, the old version of the `user_data`-entity and its representation are deprecated and are no longer supported. *Dark Reads* and *Duplicate Writes* access both entities and ensure data consistency. Before reaching the next version, a one-time data migration has to be performed, writing all the entries of the `user_data`-entity that have been created before this version into `name`-entities. The migration can be performed using the endpoint `users/migrate`.

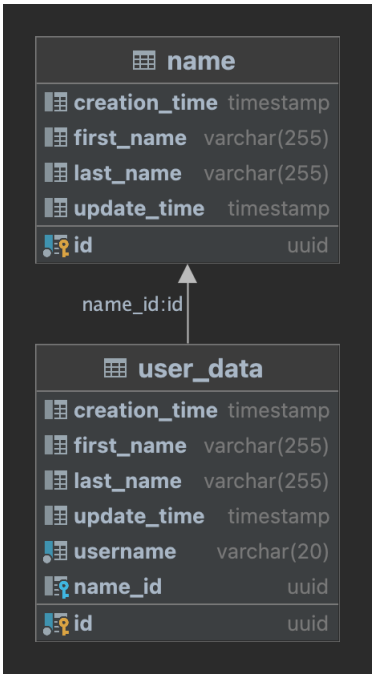

Figure 6.5: Database Schema v2.1

Version 2.2: Migrated Database

A `user_data`-entity is represented by the attributes `username` and `name_id` as a non-nullable foreign key. The latter links to a `name`-entity with the attributes `id`, `first_name`, and `last_name`.

Due to the data migration, the nullable property of `name_id` as a foreign key can be safely removed. The API endpoint has not changed, and thus the perceived behavior of the API stays the same.

Tech Stack

The reference project is built on a *PostgreSQL* database and uses *Liquibase* for database migrations. The backend runs in Spring Boot with *Hibernate* responisble for *Object-Relational Mapping (ORM)*, *Spring Data JPA* providing *Java Persistence API (JPA)* repository support, *Apache Tomcat* providing the *HTTP web server environment* and *Java* as the implementation language. This setup is also used in the demo project.

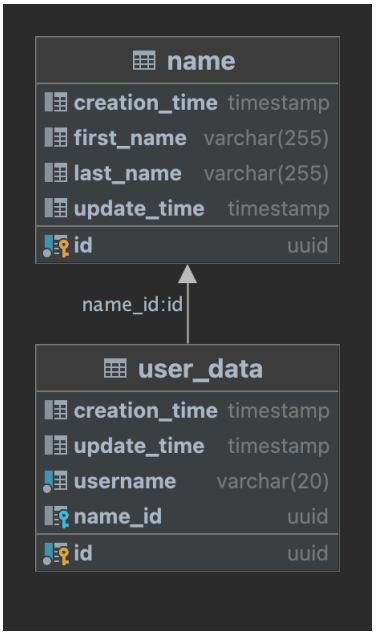

Figure 6.6: Database Schema v2.2

There is no need for a frontend application since the service can be fully explored using simple *cURL* commands.

6.2 Implementation

Technical Equipment

The following project components were used on the operating system *macOS Monterey Version 12.5.1 (21G83)*:

- Docker Desktop 4.12.0 (85629) for setting up a local database
- IntelliJ IDEA Ultimate 2022.2.1 (222.3739.54) for implementing and running the backend
- Git 2.32.1 (Apple Git-133) for version control of implementation
- diagrams.net 20.3.0 for creating figures

Database

A *PostgreSQL* database was set up in a local *Docker Container* using the following command (with indentation added for better readability):

Listing 6.1: Docker run

```
1 docker run
2     --name postgres
3     -d
4     -p 127.0.0.1:5432:5432
5     -e POSTGRES_PASSWORD=password
6     postgres
```

When the `docker run` command is executed, "the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host." [4] In this way, the database can be viewed as independent of the application. So, even though it is run locally, the system setup is comparable to having a remote database with the necessary communication customs in place. The parameters are explained as follows:

`--name postgres`: container's name; an apt name for the PostgreSQL image

`-d`: detached mode; container exits when the root process used to run the container exits; essentially, allows for the container to run in the background, *detached* from the console used to (`docker run`) it

`-p 127.0.0.1:5432:5432`: container's ports published to the host; the latter container's port is `5432` which is mapped to the former (local)host's port `5432`, `127.0.0.1` being the localhost

`-e POSTGRES_PASSWORD=password`: environment variable to set the password for the PostreSQL in the container

`postgres`: specifying that the container contains a PostgreSQL image

Backend

As mentioned above, the *Spring* backend was built as a Layered Architecture, as presented in Figure 6.2. The Presentation Layer consists of a `UserController` that takes the URL and an optional query parameter containing the version number as parameters for all CRUD methods, and additionally, a `UserDTO` in the request body to handle `PUT` and `POST` methods. A `UserDTO` is serialized and deserialized into the *JavaScript Object Notation (JSON)* format and also passed as a parameter in the communication with the `UserService` of the Domain Layer. The `UserService` calls upon the `UserRepository` of the Data Layer and exchanges `UserEntities`, which directly mirror entries in the `user_data` table of the PostgreSQL database. On the lowest layer, the `UserRepository` queries the database for `UserEntities`.

A database change is documented as part of a human- and machine-readable `changeSet` in the `db.changelog-master.xml`-file which is used by Liquibase to perform and validate database migrations.

The `README.md`-file serves as documentation as it includes information about what methods are supported for each version, holds information about the lifetime of Version 1.0 and provides migration support.

6.3 Evaluation

We evaluate the extend to which methods of REST API versioning using query parameters and Expand-Contract migrations allow for providing zero-downtime deployment to validate the proof of concept.

REST API versioning is provided in Version 2.0, where a specific version is addressed by specifying it as a query parameter in the request, as outlined in the examples of Listing A.3 and Listing A.5. In this way, Version 1.0 is still supported in Version 2.0. On top of this, in Version 2.0, the `UserController` manages to determine the version of a request based on the request body and allows for clients to not specify a version at all. This means that old clients can behave in the same way as in Version 1.0 and they can expect the server to behave in the same way as well. Due to `GET` and `DELETE` methods not having a request body, the response to these methods without a specified version has a response body with all values, as

seen in Listing A.7. In this way, the API is backward compatible and even forward compatible since old clients are still supported in the new version but are also able to simply ignore additional fields added by the new version. This implementation allows for old clients to remain functional within the lifetime of Version 1.0 without making any adjustments to the codebase. In this way, the API is guaranteed to provide zero-downtime deployment.

From a database standpoint, in Version 2.0, we deploy the new attributes `first_name` and `last_name` along with the old `full_name`-attribute. We ensure, that writing to `full_name` also entails writing to `first_name` and `last_name`, and vice versa. In this way, data backward compatibility is provided until the deployment of Version 2.1, when support for Version 1.0 is ultimately suspended.

The database schema changes expressed in Version 2.1 and Version 2.2, do not affect the behavior of the API, so no versioning has to take place. Instead, we can focus solely on the database migration aspect. First, we introduce the new `name`-table and add the attributes `first_name` and `last_name` to it. At this point in the version history, these attributes serve as copies of the attributes with the same names of the `user_data`-table. Also, apart from the deprecation of Version 1.0, this version serves as a savepoint for the planned migration. With *Dark Reads* and *Duplicate Writes* in place, we can observe the behavior of the system with the new change in place without having to take the system down for it. Additionally, with nothing being taken away from the system yet, we can expect no data to be lost. After having ascertained stable system behavior, we remove the attributes, `first_name` and `last_name` from the `user_data`-table when finally deploying Version 2.2. If this version is unstable, a roll back to Version 2.1 can be performed safely without fear of data loss or incompatibility with newly added data; thus savepoint.

Implementation in the *Spring* framework was fairly easy and did not require any additional configuration, which speaks to the general applicability of the concept. *Liquibase* served as technical documentation of backend changes and helped in validating schema changes.

The most significant finding in implementation was the fact that zero-downtime deployment is the result of the concept versioning at just the right scope and thus allowing for incremental, backward-compatible changes. The most critical point in versioning the API is when data from an old version has to be migrated to the newer

version. When data backward compatibility is present, the migration process is simple; it is almost implied. On the contrary, when versioning is performed directly at the level of the entire API, it is generally harder to implement data backward compatibility[1]. In such a setting, it is possible to have zero-downtime deployment for the moment in which the old and new versions are supported in parallel. Most often, that means that the resources and/or their representations of the old and new version are completely isolated from each other. However, when the old version ultimately becomes deprecated, the migration process threatens zero-downtime deployment as old data has to be migrated to the new version without any data compatibility having been established between the old and new version. Of course, the API can simply keep supporting both the old and new versions. Aside from the strain on system resources this entails, without data compatibility, old clients upgrading to the new version will find their data missing in the new version. So, for the desirable API behavior, data migration has to be done sooner or later. The choice is simply when and at what scale. When the API ultimately performs the data migration, it will have data comprising the entire API and of the entire duration of version coexistence to migrate, without testing in direct deployment.

The intermediate phase of the concept allows for the testing of this very compatibility between old and new data in the proper setting, in direct deployment, and consolidates stability to the entire versioning process. The concept is designed with data migration in mind from the onset, as API changes are divided into incremental changes on resource representations. As a new version is introduced, the deprecation of the old version is planned with the migration option already present. In the demo project, this can be seen by the `users/migrate`-path being present in Version 2.0 and Version 2.1.

Dividing an API change into more controllable increments on resource representations costs more time to plan but pays off in the long term as the benefits to the stability of versioned resource representations will have a positive ripple effect on the usability of affected services and ultimately facilitate a scalable API.

To put it concisely, *the bigger the scope of an API change increment, the harder it is to provide backward compatibility and zero-downtime deployment.*

[1] Versioning an API is only necessary when the underlying data changes in a significant way.

7 Conclusion

We first looked at the most common strategies for REST API versioning, which included using the URL, query parameters, custom headers, media types, and profiles. From these, we nominated the strategy of using query parameters for versioning as the most applicable one, under the metrics of being REST compliant and easy to implement.

For backward-compatible database changes, we presented the approaches Code-First, Data-First, Big-Bang and Expand-Contract migrations, and chose the latter as it guarantees backward compatibility in every affected component.

We devised a concept for zero-downtime deployment by combining these two strategies. We implemented a demo project, modeled on a project provided by adesso SE, as a proof of concept that employs the concept. Through the implementation of four versions, we demonstrated the symbiotic relationship between those strategies in Version 2.0, when versioning is needed, and also presented a use case for when an Expand-Contract migration does not have to change the API behavior and thus does not necessitate versioning.

In evaluation, we ascertained that the demo project reflects the benefits of our concept and, in turn, offered more insights regarding the critical role the scope of versioning plays in enabling data backward compatibility and zero-downtime deployment. Through a short thought experiment, we discussed the negative implications a short-term minded versioning effort without data backward compatibility would exhibit on cost, stability, usability, and scalability.

Having validated our proof of concept, we conclude that we have specified an effective concept for REST API versioning and data backward compatibility for zero-downtime deployment by relying on query parameters for versioning and employing Expand-Contract migrations to perform backward compatible database schema migrations. By splitting large API changes into controllable changes on resource rep-

resentations, we are able to provide data backward compatibility and zero-downtime deployment. The tagging of an old version with a lifetime allows for the overlapping of the intermediate phase of the Expand-Contract migration with the time the old version is supported. Implementing the concept in this constellation aims to support zero-downtime deployment with scalability in mind.

The findings of this work point to further studies in the areas of REST API design with an emphasis on HATEOAS, as well as change and version management.

We note that the versioning of REST APIs and the evolution of its data structures should match the long-term design of REST itself:

> "REST is software design on the scale of decades: every detail is intended to promote software longevity and independent evolution. Many of the constraints are directly opposed to short-term efficiency. Unfortunately, people are fairly good at short-term design, and usually awful at long-term design. Most don't think they need to design past the current release." [6] – Roy Thomas Fielding

A Source Code

Listing A.1: Example Profiles with API Version

```
{   "version" : "1.1",
    "descriptors" : [ {
        "href" : "https://example/api/profiles/item1",
        "name" : "item1"
    }, {
        "href" : ""https://example/api/profiles/item2",
        "name" : "item2"
    } ]
}
```

Listing A.2: Example Profile with Resource Representation Version

```
{   "version" : "1.4",
    "descriptors" : [ {
        "id" : "item-representation",
        "descriptors" : [ {
            "name" : "description",
            "doc" : {
                "value" : "Details about the item",
                "format" : "TEXT"
            },
            "type" : "SEMANTIC"
        }, {
            "name" : "title",
            "doc" : {
                "value" : "Title for the item",
                "format" : "TEXT"
```

```
16              },
17              "type" : "SEMANTIC"
18          }, {
19              "name" : "id",
20              "type" : "SEMANTIC"
21          } ]
22      }, {
23          "id" : "get-items",
24          "name" : "items",
25          "type" : "SAFE",
26          "rt" : "#item-representation"
27      }, {
28          "id" : "create-items",
29          "name" : "items",
30          "type" : "UNSAFE",
31          "rt" : "#item-representation"
32      }, {
33          "id" : "delete-item",
34          "name" : "item",
35          "type" : "IDEMPOTENT",
36          "rt" : "#item-representation"
37      }, {
38          "id" : "patch-item",
39          "name" : "item",
40          "type" : "UNSAFE",
41          "rt" : "#item-representation"
42      }, {
43          "id" : "get-todo",
44          "name" : "todo",
45          "type" : "SAFE",
46          "rt" : "#todo-representation"
47      } ]
48  }
```

Listing A.3: POST v1

```
curl
    -i
    -X POST http://localhost:8080/user/john?version=1
    -H 'Content-Type: application/json'
    -d '{
            "username":"john",
            "fullName":"John Doe"
        }'
```

Listing A.4: Response to Listing A.3

```
HTTP/1.1 201
Location: user/john
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 11 Sep 2022 22:31:02 GMT

{
    "username":"john",
    "fullName":"John Doe"
}
```

Listing A.5: PUT v2

```
curl
    -i
    -X POST http://localhost:8080/user/john?version=2
    -H 'Content-Type: application/json'
    -d '{
            "username": "john",
            "firstName": "John",
            "lastName": "Dane"
        }'
```

Listing A.6: Response to Listing A.5

```
HTTP/1.1 200
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 11 Sep 2022 22:35:06 GMT

{
    "username":"john",
    "firstName":"John",
    "lastName":"Dane"
}
```

Listing A.7: DELETE versionless

```
curl
    -i
    -X DELETE http://localhost:8080/user/john
```

Listing A.8: Response to Listing A.7

```
HTTP/1.1 200
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 11 Sep 2022 22:37:18 GMT

{
    "username":"john",
    "fullName":"John Dane",
    "firstName":"John",
    "lastName":"Dane"
}
```

Bibliography

[1] Rob Allen. *RESTful APIs and media-types*. https://akrabat.com/restful-apis-and-media-types/. Accessed: 2022-07-28. Aug. 2016.

[2] Dan Ciruli. *Versioning APIs at Google*. https://cloud.google.com/blog/products/gcp/versioning-apis-at-google. Accessed: 2022-7-29. June 2017.

[3] Brajesh De. *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. 1. Berkeley, CA: Apress, 2017. ISBN: 978-1-4842-1306-3.

[4] Docker. *Docker run reference*. https://docs.docker.com/engine/reference/run/. Accessed: 2022-07-28.

[5] Facebook. *Platform Versioning*. https://developers.facebook.com/docs/graph-api/guides/versioning. Accessed: 2022-07-28.

[6] Roy Thomas Fielding. *REST APIs must be hypertext-driven*. https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. Oct. 2008.

[7] Roy Thomas Fielding. "REST: Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation. University of California, Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[8] Pethuru Raj Harihara Subramanian. *Hands-On RESTful API Design Patterns and Best Practices*. 1. Birmingham, UK: O'Reilly, Jan. 2019. ISBN: 978-1-78899-266-4.

[9] Pete Hodgson and Patricio Echagüe. *Feature Flag Best Practices*. 1. Sebastapol, CA, USA: O'Reilly, Jan. 2019. ISBN: 978-1-492-05042-1.

[10] IDC. *Worldwide Software License, Maintenance, and Subscription Forecast, 2018–2022*. Accessed: 2022-07-28. July 2018.

[11] Tim Kleier. *How to Version a REST API*. `https://www.freecodecamp.org/news/how-to-version-a-rest-api/`. Accessed: 2022-07-28. Mar. 2020.

[12] Martin Kleppmann. *Designing Data-Intensive Applications*. 1. Sebastopol, CA, USA: O'Reilly, Mar. 2017. ISBN: 978-1-449-37332-0.

[13] Mike Amundsen Leonard Richardson. *RESTful Web APIs: Services for a Changing World*. 1. Sebastopol, CA, USA: O'Reilly, Sept. 2013. ISBN: 978-1-449-35806-8.

[14] Mark Nottingham. *Evolving HTTP APIs*. `https://www.mnot.net/blog/2012/12/04/api-evolution`. Accessed: 2022-07-28. Dec. 2012.

[15] Michael Pratt. *Ensuring backwards compatibility in distributed systems*. `https://stackoverflow.blog/2020/05/13/ensuring-backwards-compatibility-in-distributed-systems/`. Accessed: 2022-09-17. May 2020.

[16] adesso SE. *adesso commitment to quality*. `https://www.adesso.de/en/unternehmen/qualitaetsversprechen/index.jsp`. Accessed: 2022-07-28. May 2022.

[17] Mark Thomason. *Worldwide Software Business Model (Subscription and License) Forecast, 2021–2025*. Tech. rep. US47143021. IDC, July 2021.

[18] Twitter. *Versioning*. `https://developer.twitter.com/en/docs/twitter-api/versioning`. Accessed: 2022-07-28.

[19] Ervin Varga. *Creating Maintainable APIs: A Practical, Case-Study Approach*. Berkeley, CA: Apress, 2016. ISBN: 978-1-4842-2196-9.

[20] Yahoo. *Versioning*. `https://developer.yahooinc.com/native/guide/navigate-the-api/versioning/`. Accessed: 2022-07-28.

Name: Young-Keun Choi Matriculation number: 1049082

Declaration

I hereby assure that I wrote this work independently and did not use any other than the denoted sources and tools.

Ulm, ...

Young-Keun Choi