FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Building a public API for an open-source CDN
Student:	Bc. Martin Kolárik
Supervisor:	Ing. Oldřich Malec
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Some of the public open-source CDNs such as jsDelivr or cdnjs provide APIs with metadata about the available packages to support building third-party integrations. These APIs share many characteristics and constraints that are specific to this domain and different from, e.g., typical CRUD services.

The first goal of this work is to describe the typical requirements for such APIs in general and then look at the specifics of one of the existing implementations - jsDelivr. Describe the process of building the API into its existing form, from design, through notable implementation choices, to testing and deployment. Comment on possible problems or future improvements.

The jsDelivr API currently needs to be extended to support new types of statistics. Following the analysis of the current state, describe the necessary changes, implement them, cover new code with tests, and document the new features.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 5 December 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Building a Public API for an Open-Source CDN

Bc. Martin Kolárik

Department of Software Engineering

Supervisor: Ing. Oldřich Malec

February 16, 2023

Acknowledgments

I would like to thank my supervisor Ing. Oldřich Malec, for his interest in this topic and the provided support. Thanks also go to Dmitriy Akulov for his long-term dedication to supporting open-source, without which jsDelivr would not have existed, and Pavel Kopecký for his patience in proofreading this text.

Declaration

In Prague on February 16, 2023 . . . .

Citation of this thesis

Kolárik, Martin. *Building a Public API for an Open-Source CDN.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Práca skúma oblasť verejných CDN služieb a ich API. Začína predstavením jednotlivých skupín používateľov a ich požiadaviek a pokračuje preskúmaním existujúcich služieb a diskusiou špecifík tejto oblasti. Ukazuje, že niektoré verejné CDN služby neponúkajú API vôbec, zatiaľ čo iné spĺňajú väčšinu definovaných požiadaviek. Následne práca preskúmava existujúce jsDelivr API a ukazuje ako jeho návrh a implementácia boli ovplyvnené veľkosťou tejto služby a množstvom spracúvaných dát.

Ďalšie kapitoly opisujú návrh a implementáciu nových funkcií, diskutujú problémy dotýkajúce sa webových API vo všeobecnosti a kladú dôraz na použitie existujúcich webových štandardov. Posledné dve kapitoly ukazujú ako fázy testovania a dokumentácie môžu benefitovať z automatizácie, najmä z použitia parametrizovaných a snapshot testov a z využitia rozsiahleho ekosystému OpenAPI nástrojov.

Kľúčové slová API, CDN, jsDelivr, OpenAPI, cdnjs, unpkg

Abstract

The thesis explores the domain of public CDN services and their APIs. It starts by presenting the individual user groups and their requirements and continues by reviewing existing services and discussing the domain specifics. It shows that some public CDNs do not offer APIs at all, while others meet most of the defined requirements. Afterward, it examines the existing jsDelivr API and shows how its design and implementation were influenced by the scale of the service and the amount of processed data.

The subsequent chapters describe the design and implementation of new API features, discuss many problems applicable to web APIs in general, and emphasize the use of the existing web standards. The last two chapters show how the testing and documentation phases of the development process may benefit from automation, particularly from the use of parameterized and snapshot tests and from utilizing the vast ecosystem of OpenAPI tools.

Keywords API, CDN, jsDelivr, OpenAPI, cdnjs, unpkg

Contents

Introduction **1**

1 Domain Introduction **3**

1.1 Common Requirements 3

1.1.1 Functional Requirements 3

1.1.2 Non-functional Requirements 3

1.1.3 Typical Actors 4

1.1.4 Use Cases and User Stories 4

1.1.4.1 Library Users 4

1.1.4.2 Library Authors 5

1.1.4.3 Service Maintainers and General Public 6

1.2 A Brief History of Public CDNs 7

1.2.1 Google Hosted Libraries 7

1.2.2 jQuery CDN 8

1.2.3 cdnjs 9

1.2.4 unpkg 11

1.2.5 jsDelivr 12

1.3 Specifics and Constraints 14

1.3.1 Anonymous User Base 14

1.3.2 Inherently Global 15

1.3.3 Read-only 15

1.3.4 Adaptive Infrastructure 15

1.4 Conclusion 16

2 The jsDelivr API **17**

2.1 Requirements 18

2.1.1 Functional Requirements 18

2.1.2 Non-functional Requirements 18

2.2 Architecture and Concepts 19

2.2.1 REST and REST-like architectures 19

2.2.2 CDN and Client-Side Caching 21

2.3 Used Technologies 21

2.3.1 Node.js 22

2.3.2 Koa 22

2.3.3 MariaDB 23

2.3.4 Redis 23

2.3.5 Knex 23

2.3.6 Joi 24

2.3.7 Mocha and Chai 24

2.3.8 Elastic APM 25

2.4 Current Features 25

2.4.1 Package Metadata 26

2.4.1.1 Get Package Metadata 26

2.4.1.2 Get the Resolved Package Version 27

2.4.1.3 Get Package Version Metadata 28

2.4.2 Usage Statistics 29

2.4.2.1 Collected Data 30

2.4.2.2 Get Package Statistics 35

2.4.2.3 Get Package Version Statistics 37

2.4.2.4 Get a Package Badge 38

2.4.2.5 List Top Packages 38

2.5 Conclusion 39

3 Design **41**

3.1 API Versioning Strategies 41

3.1.1 URI Prefix 42

3.1.2 Hostname 43

3.1.3 Query String Parameters 44

3.1.4 Custom Headers 45

3.1.5 The Accept Header 46

3.1.6 Evolution 47
3.1.7 Conclusion 48
3.2 Common Properties 48
3.2.1 Bandwidth Data 48
3.2.2 Quarterly and Historical Data 50
3.2.3 Period Comparison 50
3.2.4 Hypermedia 52
3.2.5 Pagination 53
3.3 Package Statistics 55
3.3.1 List Top Packages 56
3.3.2 Get Package Statistics 56
3.3.3 List Top Package Versions 57
3.3.4 Get Package Version Statistics 58
3.3.5 List Top Package Version Files 59
3.4 Proxy Endpoints Statistics 59
3.4.1 Get Proxy Statistics 60
3.4.2 List Top Proxy Files 60
3.5 Network Statistics 61
3.5.1 Get Network Statistics 61
3.5.2 Get Network Country Statistics 62
3.5.3 Get Network Content Statistics 63
3.6 Browser and Platform Statistics 64
3.6.1 Regional Data 65
3.6.2 Period Handling 65
3.6.3 List Top Browsers (grouped versions) 65
3.6.4 List Top Browsers (separate versions) 66
3.6.5 List Top Browser Countries 66
3.6.6 List Top Browser Platforms 67
3.6.7 List Top Browser Versions 67
3.6.8 List Top Browser Version Countries 68
3.7 Statistics Periods 68
3.7.1 List Statistics Period 69
3.8 Package Metadata 70
3.8.1 Get Package Metadata 71
3.8.2 Get the Resolved Package Version 72
3.8.3 Get Package Version Metadata 72

3.9 Conclusion 72

4 Implementation **73**

4.1 Schemas and Validation Middleware 73
4.2 Period Comparison 75
4.3 Historical Data 76
4.4 Link Builder 77
4.5 Pagination 79
4.6 Regional Data 80
4.7 Proxy Statistics 81
4.8 Network Statistics 82
4.9 Browser and Platform Statistics 82
4.10 Deprecations Handling 84
4.11 Conclusion 85

5 Testing **87**

5.1 Snapshot Tests 87
5.1.1 Time Handling 89
5.2 Parameterized Tests 90
5.3 Speeding Up the Database Setup 93
5.4 Conclusion 93

6 Documentation **95**

6.1 OpenAPI 95
6.1.1 Automated Document Generation 96
6.1.2 Manual Document Improvements 97
6.2 Documentation Portal 97
6.2.1 RapiDoc Customizations 98
6.3 Conclusion 99

Conclusion **101**

Bibliography **103**

A Acronyms **111**

B Glossary **113**

C Contents of the Enclosed Memory Card **115**

List of Figures

2.1 jsDelivr API deployment diagram 19
2.2 jsDelivr log processing 31
2.3 Package data relational schema 32
2.4 Package data materialized views 33
2.5 Proxy data relational schema 33
2.6 Network data relational schema 34
2.7 Browser and platform data relational schema 35
2.8 GET /v1/package/npm/:name/badge 38

4.1 Proxy data materialized views 81
4.2 Network data materialized views 82
4.3 Browser data materialized views 83

List of Listings

1.1 jQuery versions on Google Hosted Libraries 7
1.2 jQuery versions on jQuery CDN 8
1.3 jQuery versions on cdnjs 10
1.4 Network statistics on cdnjs 10
1.5 jQuery v3.6.1 files on unpkg 11
1.6 jQuery versions on jsDelivr 13
1.7 Package statistics on jsDelivr 14

2.1 GET /v1/package/npm/:name 27
2.2 GET /v1/package/resolve/npm/:name@:range? 27
2.3 GET /v1/package/npm/:name@:version/:structure? 29
2.4 GET /v1/package/npm/:name/stats 36
2.5 GET /v1/package/npm/:name@:version/stats 37
2.6 GET /v1/stats/packages 38

3.1 Default Accept header in Google Chrome v108 on Windows . . 47
3.2 Hits and bandwidth format 49
3.3 Hits and bandwidth totals format 50
3.4 Previous period comparison format 51
3.5 Previous period comparison format in lists 52
3.6 Linking resources in the response body 53
3.7 Pagination metadata 55
3.8 Overview of package statistics endpoints changes 55
3.9 GET /v1/stats/packages 56
3.10 GET /v1/stats/packages/npm/:name 57

3.11 GET /v1/stats/packages/npm/:name/versions 58
3.12 GET /v1/stats/packages/npm/:name@:version 58
3.13 GET /v1/stats/packages/npm/:name@:version/files 59
3.14 GET /v1/stats/proxies/:name 60
3.15 GET /v1/stats/proxies/:name/files 61
3.16 GET /v1/stats/network 62
3.17 GET /v1/stats/network/countries 63
3.18 GET /v1/stats/network/content 64
3.19 GET /v1/stats/browsers 65
3.20 GET /v1/stats/browsers/versions 66
3.21 GET /v1/stats/browsers/:name/countries 67
3.22 GET /v1/stats/browsers/:name/platforms 67
3.23 GET /v1/stats/browsers/:name/versions 68
3.24 GET /v1/stats/browsers/:name/versions/:version/countries . . 68
3.25 GET /v1/stats/periods 70
3.26 Overview of package metadata endpoints changes 71
3.27 GET /v1/packages/npm/:name 71
3.28 GET /v1/packages/npm/:name/resolved 72

4.1 Parameter schema definitions 74
4.2 Route parameter validation 74
4.3 Error response 75
4.4 Period comparison implementation 76
4.5 Historical data implementation 77
4.6 Link builder 78
4.7 Pagination implementation 80
4.8 Deprecation headers 85

5.1 Usage of the Chai snapshotting plugin 89
5.2 Parameterized snapshot testing 91
5.3 Parameterized pagination testing 92
5.4 Parameterized testing with explicit assertions 92

Introduction

Public content delivery networks (CDNs) have played an important part in the distribution of open-source web assets for many years. Initially popularized by Google Hosted Libraries and later by projects such as jQuery and Bootstrap, they are often the easiest way to start using a JS or a CSS library by simply copy-pasting a link. There is no need to download the files or keep track of the versions to update the libraries when new releases are available, and, as a bonus, everything is available on a fast, global network of servers.

The state of front-end development tools and asset distribution has certainly evolved a lot in the past years, with projects like npm[1] (the first JavaScript package manager), Browserify[2], Webpack[3], and many others that followed, but CDNs remain a popular choice as well. In fact, some CDNs have managed to take advantage of the new package management ecosystem and started supporting on-demand retrieval of anything published in one of the existing registries instead of hosting just a small, curated set of projects, which traditionally had to be maintained manually.

This change caused a shift in how CDN resources are discovered. While in the early days, it was people manually copy-pasting links from documentation because only some projects were available, nowadays, there are code editors that can find and "install" the correct files automatically. This integration is

[1] `https://www.npmjs.com`
[2] `https://browserify.org`
[3] `https://webpack.js.org`

possible thanks to application programming interfaces (APIs) provided on top of the CDN services. The APIs may also provide other useful features, such as usage statistics for package authors.

This thesis explores the historical context as well as typical uses of public CDN APIs and their requirements, discusses the specifics of this domain, and then examines the most extensive of the existing APIs—the jsDelivr API.

The API is then extended with several new features that allow building better development tools, provide package authors with more insight into how their packages are used, and make all integrations easier by more closely following existing web standards.

Chapter 1

Domain Introduction

This chapter serves as a foundation for understanding the target domain, its specifics, and requirements, as identified based on the existing services and their typical use.

1.1 Common Requirements

Utilizing elements of the Unified Process [1] and Extreme Programming [2], this section introduces the most common requirements for public CDN APIs, the key actors, and their use cases and user stories.

1.1.1 Functional Requirements

On a high level, the typical functional requirements are:

FR01: The API shall provide information about the available libraries.

FR02: The API may provide usage statistics for the available libraries.

FR03: The API may provide usage statistics for the CDN itself.

These requirements are further explained later in this chapter.

1.1.2 Non-functional Requirements

The non-functional requirements may vary across services (e.g., performance requirements will depend on the size of the user base), and it may not always be possible to externally evaluate whether they are met, so only the following is considered:

NR01: The API shall be easy to use in web browser environments.

1.1.3 Typical Actors

The users that interact with the services can be categorized into four main groups:

1. **Library users** use the CDN to load libraries in their projects.
2. **Library authors** use the CDN as one of the distribution channels of their project.
3. **Service maintainers** are involved with operating the CDN.
4. **General public** does not directly fall into any of the previous groups but still interacts with the project in some way. For instance, a person that discovers the CDN and tries to find out what it is and how it is used, or a library author that does not yet use the CDN but researches the available services to choose one.

1.1.4 Use Cases and User Stories

This section describes how each group interacts with the service through a set of use cases and user stories.

1.1.4.1 Library Users

The main concern of library users is being able to quickly find relevant projects and all the information necessary to use them. This is typically possible by browsing the CDN website, but it also creates space for integrations with development tools so that users can find and use the relevant libraries directly in an integrated development environment (IDE) or load them automatically as they write their code.

1.1.4.1.1 UC01 Finding Libraries

This use case can be described by the following steps:

1. A user types the name of the package they want to use.

2. Optionally, they select a version from the presented list, or the one specified in the `package.json`[4] is used, or the latest version is selected automatically.
3. Optionally, they select a file from the presented list, or the correct file is selected automatically.
4. Optionally, they select how exactly to import the file, or the correct code is generated automatically.

These steps give us a better understanding of Functional Requirement 01, which must include at least the following:

- There must be a way to search the available packages.
- There must be a way to obtain a list of available versions for a package.
- There must be a way to obtain a list of files for a specific package version.
- If the file to import is to be selected automatically, there must be a way to determine which file is supposed to be imported.

Some of the existing implementations of this use case are:

1. Codepen[5] (integrates with cdnjs),
2. CodeSandbox[6] (integrates with jsDelivr),
3. jsDelivr's JetBrains IntelliJ plugin[7].

1.1.4.2 Library Authors

The library authors are often interested in the usage statistics of their packages. Because, in this case, there is not a single solution with a clear flow, a set of user stories is presented instead of a detailed use case description.

As a package author, I want to:

US01: know how many times had my project been downloaded in the specified period to understand how popular it is,

[4]A manifest file initially used by Node.js and npm, now supported by most of the JavaScript tools.

[5]`https://codepen.io`

[6]`https://codesandbox.io`

[7]`https://github.com/jsdelivr/plugin-intellij`

US02: know how the current number of downloads compares to the previous period to understand the popularity trend,

US03: know what the distribution of versions among all downloads is to see if most of the users quickly update to the latest version or if any of the older versions, in particular, is more popular,

US04: know what the distribution of individual files among all downloads is to assess the popularity of the individual builds (with different features, language mutations, browser support, . . .) my project provides,

US05: know how my project downloads compare to other projects.

These user stories give us a better understanding of Functional Requirement 02 and, again, we can formulate a couple more specific requirements:

- Usage statistics must be kept track of for every project on a daily, weekly, or monthly basis.
- Usage statistics must include separate values for each version of a library.
- Usage statistics must include separate values for each file.

1.1.4.3 Service Maintainers and General Public

These two groups are not as easily defined as the previous two and are very different, but they share a common interest in knowing overview information about the network as a whole, as demonstrated again by several user stories.

As a service maintainer, I want to:

US06: know what the most popular projects on the network are in the specified period to understand who uses it,

US07: know how many requests the network serves overall to understand how it compares to competitors,

US08: know how many requests the network serves in individual countries to better understand the user demographic and the infrastructure usage,

US09: know how the current usage statistics compare to those in the previous period to understand the usage trend.

These user stories give us a better understanding of Functional Requirement 03.

1.2 A Brief History of Public CDNs

Having established the main requirements and use cases, we can now take a look at some of the most popular public CDNs [3] and see how they have evolved over time in order to understand the historical context and determine which of the described features they support.

1.2.1 Google Hosted Libraries

Google Hosted Libraries[8] (formerly called Google Libraries API) is a Google-provided CDN for some of the most popular open-source JavaScript libraries. The service hosts only a tiny selection of projects (20 at the time of writing) [4], but it was among the first services of this type and ranks as the second most popular public CDN with a 42.2 % market share[9], according to W3Tech [3].

Google claims to work directly with the key stakeholders for each library to offer the latest versions as they are released but does not provide any additional information on whether this process is manual or automated and how quickly updates are available [4].

The service does not provide a dedicated API, and the list of libraries, as well as their versions, is only available in text form on a single HTML page, which was clearly not intended for machine processing. A snippet of the versions list is shown in Listing 1.1.

```html
<dt>versions:</dt>
<dd class="versions">
    3.6.1, 3.6.0, 3.5.1, 3.5.0, 3.4.1, 3.4.0, 3.3.1, 3.2.1, 3.2.0, 3.1.1,
    3.1.0, 3.0.0<br>
    2.2.4, 2.2.3, 2.2.2, 2.2.1, 2.2.0, 2.1.4, 2.1.3, 2.1.1, 2.1.0, 2.0.3,
    2.0.2, 2.0.1, 2.0.0<br>
    1.12.4, 1.12.3, 1.12.2, 1.12.1, 1.12.0, 1.11.3, 1.11.2, 1.11.1, ...
</dd>
```

Listing 1.1: jQuery versions on Google Hosted Libraries[10]

[8] https://developers.google.com/speed/libraries

[9] The percentage of websites using the CDN (note that a website may use more than one CDN).

1.2.2 jQuery CDN

The jQuery CDN[11], as the name suggests, serves only files related to the jQuery projects[12] (the core jQuery library, jQuery UI, jQuery Mobile, . . .). As Kris Borchers, the former jQuery Executive Director, explains, it was established in 2007 after people had already started hotlinking[13] files directly from their website [5]:

> Prior to 2007, people basically linked directly to files hosted at `https://jquery.com/src`, or used the Google API servers. John Resig saw this trend and established `https://code.jquery.com` sometime in early April of 2007.

At this time, the service ranks as the fourth most popular public CDN with a 17.0 % market share, according to W3Tech [3].

Similarly to Google Hosted Libraries, jQuery CDN does not provide a dedicated API, and the list of libraries, as well as their versions, is only available in text form, although in this case, the HTML is slightly better structured, as shown in Listing 1.2.

```html
<h2 id="jquery-all-3.x">jQuery Core - All 3.x Versions</h2>
<ul>
    <li>jQuery Core 3.6.1 -
        <a class="open-sri-modal"
            href="https://code.jquery.com/jquery-3.6.1.js"
            data-hash="sha256-3zlB5s2uwoUzrXK3BT7AX3FyvojsraNFxCc2vC/7pNI=">
            uncompressed
        </a>,
        <a class="open-sri-modal"
            href="https://code.jquery.com/jquery-3.6.1.min.js"
            data-hash="sha256-o88AwQnZB+VDvE9tvIXrMQaPlFFSUTR+nldQm1LuPXQ=">
            minified
        </a>,
        ...
    </li>
    ...
</ul>
```

Listing 1.2: jQuery versions on jQuery CDN[14]

[10] https://developers.google.com/speed/libraries
[11] https://releases.jquery.com
[12] https://jquery.com
[13] The use of a resource hosted on one site by other sites.

1.2.3 cdnjs

cdnjs[15] launched in 2011 with the tagline “cdnjs.com, the missing cdn.” As the initial version of its homepage explained, the project was inspired by the existing services but aimed to provide a wider selection of libraries [6]:

> Everyone loves the Google CDN right? Even Microsoft runs their own CDN. The problem is, they only host the most popular libraries. We host the other stuff [. . .]
>
> Our goal is to operate this CDN in a peer reviewed fashion. This means only the highest quality libraries vetted by the community get added to cdnjs.com

To date, cdnjs is the most popular service of this type, with a 48.1 % market share, according to W3Tech [3]. It can also be seen as the first in the generation of “general purpose, community managed” CDNs, hosting more than 4000 libraries.

Even though cdnjs’s selection of libraries is much wider than that of its predecessors, it still requires that a library have a certain level of popularity before it can be added [7]:

> Libraries must have notable popularity: 100 stars or watchers on GitHub[16] is a good example, but as long as reasonable popularity can be demonstrated the library will be added.

cdnjs maintains all hosted files in a GitHub repository, and initially, all updates were handled manually by people making pull requests. In 2014, the project started working on an auto-update mechanism that would be able to automatically download new versions after a manual one-time, per-project configuration [8].

The auto-update system was based on pulling files from npm or git reposi-

[14] `https://releases.jquery.com/jquery`
[15] `https://cdnjs.com`
[16] `https://github.com`

tories, and even though the updates were submitted automatically, they still required a manual review from the maintainers, making the expected update delay about 30 hours [7].

This service is the first on the list to have an API that provides the following functionality [9]:

1. Searching the hosted libraries.
2. Getting details for a specific library.
3. Getting details for a specific library version.
4. Getting the allowed extensions.
5. Getting basic network statistics.

The first three points together fulfill our Functional Requirement 01. An example of getting details for a specific library is shown in Listing 1.3.

```
{
    "name": "jquery",
    "versions": [
        "1.10.0",
        "1.10.1",
        "1.10.2",
        ...
    ]
}
```

Listing 1.3: jQuery versions on cdnjs[17]

It appears that point 5 might also fulfill Functional Requirement 03, but upon closer inspection, it is clear the implementation does not resolve any of the problems described in Section 1.1.4.3 because the only statistic available is the number of hosted libraries, as shown in Listing 1.4. Functional Requirement 03 is therefore deemed not fulfilled.

```
{
    "libraries": 4343
}
```

Listing 1.4: Network statistics on cdnjs[18]

[17] `https://api.cdnjs.com/libraries/jquery`
[18] `https://api.cdnjs.com/stats`

As far as the previously defined non-functional requirement goes, we can consider it met, since the API uses a JSON format, which is natively supported by all modern browsers [10].

1.2.4 unpkg

Started in 2016 as npmcdn and later renamed to unpkg due to legal reasons [11], unpkg[19] was the first CDN to use a fully automated system by pulling packages from npm on demand.

Not only did this approach not require any extra work from package maintainers, it also meant that *every* package was available, instead of—even if big—selection. Moreover, with no manual reviews, there was essentially no delay between publishing a new version and it being globally available.

unpkg does not have a dedicated API but allows adding a `?meta` query string parameter to any URL to obtain metadata for a file or directory in JSON, as shown in Listing 1.5. There is no direct way to list the available packages and their versions, but since unpkg pulls all content from npm, the public npm registry API[20] can be used to obtain this information.

```json
{
    "path": "/",
    "type": "directory",
    "files": [
        {
            "type": "file",
            "path": "/README.md",
            "contentType": "text/markdown",
            "lastModified": "Sat, 26 Oct 1985 08:15:00 GMT",
            "integrity": "sha384-SIZry4D4Lj1/NQD+bA5/QCqMON9RY91R24/QIgqz...",
            "size": 2004
        },
        ...
    ]
}
```

Listing 1.5: jQuery v3.6.1 files on unpkg[21]

[19] `https://unpkg.com`
[20] `https://github.com/npm/registry/blob/master/docs/REGISTRY-API.md`
[21] `https://unpkg.com/jquery@3.6.1/?meta`

With those two sources combined, Functional Requirement 01 is met. There are no usage statistics of any kind, so the other two requirements are not met. The non-functional requirement, on the other hand, is met because the metadata are returned in JSON.

At this time, unpkg has a 7 % market share, which makes it the fifth most popular service of this type [3].

1.2.5 jsDelivr

jsDelivr[22] launched in 2012, shortly after cdnjs. One of its main selling points was its unique, multi-cdn infrastructure combined with RUM-based load balancing[23]—something no other service has offered, even to this date. More interesting for this work, however, is that jsDelivr was the first to provide a CDN to any open-source project, regardless of popularity [12].

Similarly to cdnjs, jsDelivr started by manually adding the hosted files to a GitHub repository. In 2014, the same year as cdnjs, an option for automated updates after an initial one-time configuration was introduced[24], supporting npm, GitHub, and bower[25] as sources [13].

In 2013, jsDelivr also launched a public API, which was functionally similar to that of cdnjs—it provided a list of all hosted packages, their versions, and other metadata. The interesting part is that it covered not only the jsDelivr projects but also several of the already introduced competitors: Google Hosted Libraries, jQuery CDN, cdnjs, and BootstrapCDN[26].

The metadata for all of these services were available via a unified interface [14]. Because cdnjs was the only one to have its own API, jsDelivr API internally relied on web scraping[27] to get the necessary data from other services.

[22]`https://www.jsdelivr.com`

[23]Load balancing based on real-time real user monitoring data.

[24]Based on the official release posts, jsDelivr was the first to launch this feature, but it is not clear who was the first to start working on it.

[25]`https://bower.io`

[26]`https://www.bootstrapcdn.com`

[27]Extracting data for machine processing from a source in a human-readable form.

This API met Functional Requirement 01 but not the other two requirements. The non-functional requirement was met by using a JSON format for all responses.

In 2017, following a discussion on how to keep scaling the project [15], jsDelivr switched to an entirely new on-demand system, serving all resources from npm and GitHub, similar to unpkg. Because the packages could not be directly mapped between the two systems, this also resulted in a new version of the API. An example of a response providing package versions is shown in Listing 1.6.

```json
{
    "tags": {
        "beta": "3.6.1",
        "latest": "3.6.1"
    },
    "versions": [
        "3.6.1",
        "3.6.0",
        ...
    ]
}
```

Listing 1.6: jQuery versions on jsDelivr[28]

In addition to its previous features, the new API added usage statistics for every project [16], meeting Functional Requirement 02, as shown in Listing 1.7.

Similarly to unpkg, the jsDelivr API assumes the user already knows the package name, and there is no direct way to search for packages. In case the search functionality is needed, jsDelivr recommends using the public npm registry API or Algolia's npm search[29] [17].

Even though the new automated system is jsDelivr's primary focus, there is also an option of manual setups, so-called proxy endpoints, for projects with special needs [18]. The proxy endpoints, however, are not covered by the API in its current form.

[28] `https://data.jsdelivr.com/v1/package/npm/jquery`
[29] `https://github.com/algolia/npm-search`

```
{
    "rank": 9,
    "typeRank": 7,
    "total": 1531672161,
    "versions": {
        "3.6.1": {
            "total": 95401718,
            "dates": {
                "2022-10-31": 3122163,
                "2022-11-01": 3281549,
                ...
            },
            ...
        }
    }
}
```

Listing 1.7: Package statistics on jsDelivr[30]

1.3 Specifics and Constraints

At last, this section briefly touches on a few domain specifics, which can have a major impact on the choice of technologies and architecture decisions.

1.3.1 Anonymous User Base

A common property shared by all of the mentioned services is that all features are available to unauthenticated users. This is not a domain constraint per se, more of a result of the idea that all data should be publicly available without restrictions anyway, and building authentication and related account management features would require a non-trivial amount of extra work. Still, it has a few implications:

- It is harder to identify the sources of incoming requests, which complicates the diagnosis and resolution of operational issues.
- The lack of user quotas means that users have little motivation to use the API efficiently (e.g., implement their own caching).
- Making any kind of possibly breaking change is virtually impossible, even with a generous migration period, because users cannot be easily contacted and asked to migrate.

[30] `https://data.jsdelivr.com/v1/package/npm/jquery/stats`

These problems can be partially mitigated with proper design, the use of application performance monitoring (APM) tools, or voluntary mailing lists, but they are still a specific worth pointing out.

1.3.2 Inherently Global

The very definition of a CDN is that it is a geographically distributed network, but what about its API? With users all around the globe, the API is expected to be fast everywhere, just like the CDN. That means it should either be designed to run directly in multiple locations, which puts additional requirements on back-end storage and routing technologies, or to have highly cacheable responses, which puts additional requirements on the public interface.

Again, this property is not entirely unique to public CDN services, but it is worth pointing out the difference from other types of services with only regional user bases.

1.3.3 Read-only

As one could have already gathered from the common requirements and the analysis of the existing services, the APIs tend to be focused on providing information, which makes them essentially read-only.

Combined with the fact that all data are public, with proper architecture and design, this property can make caching considerably easier and help to resolve the aforementioned performance problems.

1.3.4 Adaptive Infrastructure

With the exception of Google Hosted Libraries, all of the examined services run on sponsored third-party infrastructure. jsDelivr, for instance, lists eight major infrastructure sponsors, ranging from DNS and CDN providers to cloud computing platforms [19]. As a result, the services may be reluctant to use advanced vendor-specific features to avoid vendor lock-in[31] and retain the ability to switch providers should the sponsorship agreement come to an end.

[31]Being so dependent on specific vendor features that it is hard to change the vendor without significant costs.

Additionally, staying vendor-agnostic may be necessary to provide load balancing or failover capabilities. In fact, jsDelivr specifically praises itself for not being dependent on any single vendor, which allows it to react quickly to performance and uptime issues [20].

The downside of the required flexibility is that architecture decisions have to target the lowest common denominator of the features provided by the considered vendors.

1.4 Conclusion

This chapter described the typical functionality of public CDN services and their APIs, the user groups along with their needs, and the domain specifics that need to be understood when designing these services.

It also summarized the history of the most notable services in this space and showed that some do not offer APIs at all, while others meet most of the defined requirements.

Chapter 2

The jsDelivr API

This chapter describes the current state of the jsDelivr API and the additional requirements on top of those already introduced in Chapter 1.

As already mentioned, the current version of the jsDelivr API has been available since 2017 and focuses on meeting two key requirements: providing metadata about packages hosted on the CDN and providing usage statistics for those packages.

The API is free to use, does not require authentication, and imposes no rate limits. It is developed under the OSL-3.0 license[32] on GitHub [17]. As of November 2022, it handles 10.5 million requests per day on average, with a 95 % CDN cache hit rate[33] [21]. Some notable users include:

- Online IDEs CodeSandbox and StackBlitz[34], which use jsDelivr to install packages from npm in the background.
- Algolia's npm search index, which uses jsDelivr's usage statistics as a part of the ranking criteria.
- Microsoft Library Manager[35], which uses jsDelivr as one of its library providers.

[32]A copyleft license that does not require reciprocal licensing on linked works.

[33]The percentage of requests handled directly from the cache without reaching the origin servers.

[34]`https://stackblitz.com`

[35]`https://github.com/aspnet/LibraryManager`

Documentation for the API is available in a Markdown format on GitHub.

2.1 Requirements

This section summarizes what jsDelivr would like to add to its API on top of the functionality already described in Section 1.2.5.

2.1.1 Functional Requirements

The additional functional requirements are:

FR04: The API shall provide bandwidth usage statistics for packages. Currently, only hits statistics are available.

FR05: The API shall provide at least basic usage statistics for proxy endpoints: daily hits and bandwidth.

FR06: The API shall provide network-wide statistics: daily hits and bandwidth with breakdowns by different content types (packages, proxies), providers, and countries.

FR07: The API shall provide browser and platform usage statistics: market share with breakdown by versions and countries.

FR08: The API shall provide a way to easily compare data from the current period with those in the previous period.

FR09: The API shall support 90-day data ranges in addition to the current day, week, month, and year ranges.

FR10: The API shall support querying historical data.

2.1.2 Non-functional Requirements

There are also a few new non-functional requirements that aim to make it easier to work with the API:

NR02: An OpenAPI[36] document shall be available to API users.

NR03: The hypermedia as the engine of application state (HATEOAS) principle should be used to link to related resources.

[36] `https://www.openapis.org`

2.2 Architecture and Concepts

The API currently runs on a cloud platform Render[37], where each commit is automatically deployed after passing tests via a CI/CD pipeline[38]. In front of the API is a Bunny CDN[39], with caching configured individually for each resource using standard HyperText Transfer Protocol (HTTP) headers.

Internally, the API uses MariaDB[40] and Redis[41] and retrieves certain data from GitHub, npm registry API, and the jsDelivr CDN itself, as shown in Figure 2.1.

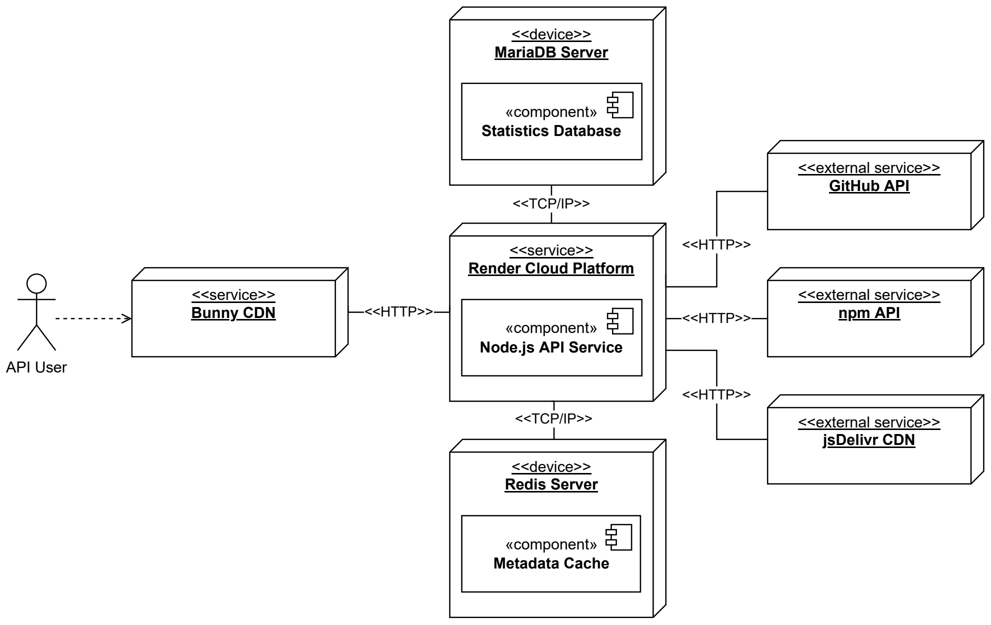


Figure 2.1: jsDelivr API deployment diagram

2.2.1 REST and REST-like architectures

Representational State Transfer (REST) is an architectural style introduced by Fielding that introduces concepts and constraints that aim to improve the

[37]`https://render.com`

[38]A set of automated steps executed on each commit that builds the application, runs tests, and deploys the application to the production environment.

[39]`https://bunny.net`

[40]`https://mariadb.org`

[41]`https://redis.io`

performance, scalability, simplicity, modifiability, visibility, portability, and reliability of web services [22, pages 28–36].

The REST concepts relevant in terms of API design are:

1. identification of resources,
2. manipulation of resources through representations,
3. self-descriptive messages,
4. hypermedia as the engine of application state (HATEOAS).

Because REST is just an architectural style and not a protocol itself, it is up to each implementation to decide how strictly it wants to abide by the set rules, even though an implementation following only some of the rules is no longer, strictly speaking, “RESTful”.

In this work, the term “REST-like” is used for architectures that—whether for carefully considered pragmatic reasons or simple negligence—respect some, but not all, REST concepts.

To better discuss different levels of REST compliance, Richardson proposed a maturity model, which divides services into four levels [23]:

0. **Level zero services** use a single URI and HTTP method for all operations, effectively not following any of the REST principles.
1. **Level one services** use many URIs but only a single HTTP method.
2. **Level two services** use many URIs and multiple HTTP methods.
3. **Level three services** use hypermedia to describe resource capabilities and interconnections.

The jsDelivr API does not itself claim to be RESTful, but it does follow some of the principles:

- it is resource-oriented, with a proper URI for each resource,
- it uses HTTP methods to identify operations on resources.

The API does not use hypermedia, so it achieves level two of the maturity model and can be described as REST-like. The used principles are particularly relevant for efficient caching as described in the next section.

2.2.2 CDN and Client-Side Caching

The API runs in a single location and heavily relies on caching to improve its performance. To allow for optimal caching, where possible, the resources are designed to either only contain data that do not change frequently (and can, therefore, be cached for a long time) or only data that change often.

The caching instructions for both the CDN and end users are provided via the following HTTP header directives:

- `Cache-Control: public` is used for all resources to indicate the data are public and can be stored in shared caches,
- `Cache-Control: max-age` is used for metadata resources with a fixed time to live (TTL) to indicate the expiration time,
- `Expires` is used for statistics resources with a fixed expiration date (because the data always refresh at midnight),
- `Cache-Control: stale-while-revalidate` is used for all resources to reduce the perceived latency when revalidation is needed,
- `Cache-Control: stale-if-error` is used for all resources to reduce the impact of network errors or service unavailability,
- `ETag` is used for all resources to reduce the amount of transmitted data during revalidation.

2.3 Used Technologies

This section briefly describes the main technologies and how they are used by the API.

2.3.1 Node.js

Node.js[42] is an open-source, cross-platform runtime environment for developing server-side and networking applications built on the JavaScript V8 engine. Its key characteristic is the event-driven, non-blocking I/O model, which makes it efficient in handling a large number of concurrent connections [24], but thanks to a rich ecosystem of libraries and frameworks, and the ability to run a language that most web developers are already familiar with, it has also become a popular choice for building web development tools and command line applications. That is the case with jsDelivr, too, since it both builds upon projects from the JavaScript ecosystem and targets mainly developers from this ecosystem, building the services in JavaScript using Node.js was a natural choice.

2.3.2 Koa

Koa[43] is a minimalist web framework for Node.js that focuses on providing a simple and composable API for building web applications and APIs. It was created by the team behind the popular Express[44] framework and is designed to be a more modern and lightweight alternative to Express. It is built on top of the latest JavaScript language features and uses async/await keywords to provide a cleaner and more intuitive way to handle asynchronous operations compared to callbacks used in Express. A Koa application is composed of an array of functions called middleware, which are executed in a stack-like manner upon each request [25].

Thanks to its middleware handling, Koa is also more flexible. First, the middleware is called in the order it was defined, and then the stack unwinds, and each middleware can perform additional operations in reverse order. This is in contrast to the Express middleware handling, which is one-way and makes implementing certain features difficult.

Because Koa does not provide its own router, it is usually combined with Koa-router[45], which matches the incoming requests with the correct routes based

[42]`https://nodejs.org`
[43]`https://koajs.com`
[44]`https://expressjs.com`
[45]`https://github.com/koajs/router`

on their HTTP method and URI. Each route has its own set of middleware handlers responsible for generating the response.

2.3.3 MariaDB

MariaDB is an open-source relational database created as an alternative to MySQL[46] after MySQL was acquired by Oracle. It is developed by some of the original developers of MySQL and aims to be fast, scalable, and robust [26].

jsDelivr uses MariaDB to store all statistics data, which, as of December 2022, consist of approximately three billion records.

2.3.4 Redis

Redis is a key-value data store that can be used as a database, cache, or message broker. To achieve top performance, it works primarily with an in-memory dataset, which can optionally be persisted to disk. Redis provides a wide range of data structures, such as strings, lists, sets, and hashes, and offers atomic operations for modifying these data structures. Additionally, it supports replication, allowing for high availability and scalability [27].

The jsDelivr API uses Redis as an in-memory cache for package metadata retrieved from npm and GitHub, as well as for caching results of some of the more expensive database queries.

2.3.5 Knex

Knex[47] is an SQL query builder for JavaScript that can be used in both Node.js and the browser, providing a simple and consistent interface for interacting with different databases, such as MariaDB, PostgreSQL, or SQLite. It also provides a migration system for managing changes to the database schema over time and a seed system for setting up the initial database state [28]. The jsDelivr API uses all three of those features.

Database migrations are stored in the `migrations` directory, and each set of changes, such as creating new tables or columns, is described in a separate

[46] `https://www.mysql.com`
[47] `https://knexjs.org`

migration file. Knex keeps track of which migrations were already applied and provides a command line interface tool to apply or revert them.

The seeds are stored in the `seeds` directory and are used to populate the database with data for development and tests.

The query builder API is used in most of the application code, as well as the migrations and seeds, with the exception of complex stored procedures, which need to be written directly in SQL.

2.3.6 Joi

Joi[48] is a JavaScript library for data validation. It provides a simple and powerful language for defining the structure and constraints of data and can be used in both Node.js and browser applications [29]. The jsDelivr API uses Joi in database models.

2.3.7 Mocha and Chai

Mocha[49] is a JavaScript testing framework designed to make it easy to write and run automated tests for Node.js and browser-based applications. It provides a flexible interface to write and structure test cases and is widely used for unit testing, integration testing, and end-to-end testing [30]. Mocha does not provide its own assertion methods, which is why it is often combined with Chai[50]—an assertion library that provides several interfaces for writing expressive, easily readable assertions [31].

jsDelivr uses Mocha and Chai for automated unit tests, which are used to test some of the more complex functions, and for integration tests, which cover the full request handling cycle, including communication with MariaDB and Redis.

[48]`https://joi.dev`
[49]`https://mochajs.org`
[50]`https://www.chaijs.com`

2.3.8 Elastic APM

Elastic APM[51] is a tool for monitoring the performance and availability of applications. It is a part of the Elastic Stack, a set of open-source tools for searching, analyzing, and visualizing data [32].

Elastic APM allows developers to track the performance of their applications in real time and provides overview data such as throughput, average response times, and error rates, as well as detailed information about the performance of individual transactions, including durations of performed SQL queries, remote HTTP calls, and other operations.

In addition to its performance monitoring capabilities, it keeps track of all unhandled errors, which means developers can discover and fix them even before someone reports them. High error rates, performance degradations, or other problems can also be automatically reported to the responsible people via configured alerts.

jsDelivr uses the APM as the primary tool for monitoring the service performance and diagnosing any reported issues.

2.4 Current Features

The API features can be split into two main parts: metadata-related and statistics-related. This section describes how each of these works and discusses possible improvements and problems related to the new requirements.

Not all suggestions may be worthwhile in the end, especially because of the need to preserve backward compatibility, but being aware of existing issues is important when designing the new features.

[51]`https://www.elastic.co/observability/application-performance-monitoring`

2.4.1 Package Metadata

The metadata endpoints are related to Functional Requirement 01 and allow users to:

1. get package metadata,
2. get the resolved package version from a range or a tag,
3. get package version metadata and the list of its files.

Note that because jsDelivr operates with two package sources—npm and GitHub—which have separate namespaces, there are effectively two separate endpoints for each listed functionality. Their public interface is the same, so we can neglect this detail for the most part, but in some cases, the internal implementation is different for each source. For the sake of brevity, the following text only lists the npm versions.

2.4.1.1 Get Package Metadata

```
GET /v1/package/npm/:name
```

This endpoint provides a list of package tags and versions, with versions being sorted in descending order. Since jsDelivr is not the authoritative source of the packages, this feature relies on getting the list of available package versions from npm (or GitHub, in the case of GitHub repositories). The data are temporarily cached in Redis to improve performance and to avoid making too many remote HTTP calls.

Because the list can change at any time, it has a short TTL of five minutes, after which clients should perform revalidation. The response format is shown in Listing 2.1.

Possible improvements:

- Use hypermedia for `versions` entries instead of simple strings and provide links to package version details.

```
{
    "tags": {
        "beta": "3.6.1",
        "latest": "3.6.1"
    },
    "versions": [
        "3.6.1",
        "3.6.0",
        ...
    ]
}
```

Listing 2.1: GET /v1/package/npm/:name[52]

2.4.1.2 Get the Resolved Package Version

`GET /v1/package/resolve/npm/:name@:range?`

This endpoint operates with the same data as the previous one but returns only the latest package version matching the semver[53] `range` or `null` if there is no matching version. If no `range` is provided, the latest package version is returned.

The behavior mirrors the `npm install`[54] command and allows clients to correctly resolve version ranges without implementing the complex semver rules and fetching a possibly large list of all tags and versions. The response format is shown in Listing 2.2

```
{
    "version": "3.6.1"
}
```

Listing 2.2: GET /v1/package/resolve/npm/:name@:range?[55]

Considering Non-functional Requirement 02, the optionality of the `range` parameter is a problem because OpenAPI does not allow optional path parameters [33, §4.8.12.2]. This is a limitation resulting from its data model, where each operation is uniquely identified by path, and a path segment that might be empty can only be modeled as two separate endpoints.

[52]`https://data.jsdelivr.com/v1/package/npm/jquery`
[53]`https://semver.org`
[54]`https://docs.npmjs.com/cli/v9/commands/npm-install`
[55]`https://data.jsdelivr.com/v1/package/resolve/npm/jquery`

Possible improvements:

- Use hypermedia and provide links to the resolved package version details.
- Move the `range` parameter to the query string.

2.4.1.3 Get Package Version Metadata

```
GET /v1/package/npm/:name@:version/:structure?
```

This endpoint returns a path to the default file[56] and a list of all files in this version. The `version` parameter is required to be an exact version number in this case, which means the generated response does not change over time (because releases are considered immutable), allowing for efficient caching. The `structure` parameter allows switching between a recursive `tree` format and a `flat` format.

This feature is implemented by making an HTTP call to the CDN service itself. Because generating a list of all files at the CDN is a relatively expensive and slow operation, the API takes advantage of the fact that packages are immutable and stores the data in the MariaDB database after the first call.

This may seem strange at first because it means the database is used as a cache, while the API also uses Redis for this purpose, but there are good reasons for it:

- unlike Redis, MariaDB stores data persistently, so the data can stay cached forever,
- the listing size ranges from a few to a few hundred kilobytes as it contains metadata for each file, and there are millions of unique package versions. Storing the data primarily on disk is a more efficient use of resources than storing all of it in memory, especially considering it is not accessed frequently, thanks to additional caching at the CDN level.

A sample response in the `tree` format is shown in Listing 2.3. Note that the `time` field has already been deprecated because npm now replaces all file modification times with a placeholder value to support reproducible builds [34].

[56] An attribute set by the package author to indicate which file is the correct one to import.

```
{
    "default": "/dist/jquery.min.js",
    "files": [
        {
            "type": "directory",
            "name": "dist",
            "files": [
                {
                    "type": "file",
                    "name": "jquery.js",
                    "hash": "3zlB5s2uwoUzrXK3BT7AX3FyvojsraNFxCc2vC/7pNI=",
                    "time": "1985-10-26T08:15:00.000Z",
                    "size": 289812
                },
                ...
            ]
        },
        ...
    ]
}
```

Listing 2.3: GET /v1/package/npm/:name@:version/:structure?[57]

Possible improvements:

- Move the `structure` parameter to the query string or use content negotiation.
- Remove the `time` field from entries.

2.4.2 Usage Statistics

The statistics features are related to Functional Requirement 02. The provided statistics do not cover all of the described user stories, however. Data can only be requested for the past day, week, month, or year, which means there is a way to get the current data but not to compare them with the previous ones, as required by User Story 02 and User Story 09. There are also no network statistics as required by User Story 07 and User Story 08.

The current features allow users to:

1. get package usage statistics,
2. get package version usage statistics,
3. get a package hits badge,

[57] `https://data.jsdelivr.com/v1/package/npm/jquery@3.6.1`

4. list the most popular packages.

2.4.2.1 Collected Data

To provide the usage statistics, jsDelivr collects logs from all servers in its network and stores the data in the MariaDB database. This process is handled by a separate service and, as such, is not in the scope of the API. However, it is crucial to understand the amount of data the API works with.

As of December 2022, the CDN handles approximately three billion requests per day, which translate to three billion log records. In their raw form, the daily log records would consume approximately 2.6 TB of storage space, and a year's worth of data would require about 970 TB. For this reason, the data are not stored in the original form but undergo a two-phase processing and aggregation based on the specific API requirements.

The first processing phase takes place before the data are inserted into the database. Each record is split into multiple smaller pieces, transformed, and then aggregated with other records.

For example, consider a simplified record that includes only the request time, user country, server provider name, and the request URI, as shown in Figure 2.2. The API does not need the exact request time because it only works with daily data, so only the date is kept. It also does not need the original request URI but needs to know the package type, name, version, and file, which can be extracted from the URI. Moreover, it does not need to know the package details when considering network statistics or the server and country details when considering package statistics.

By breaking the record into two smaller ones, the link between network and package data is lost, but the data can be aggregated more efficiently, reducing the storage requirements. The records produced in this phase are referred to as "source data" in the rest of this text.

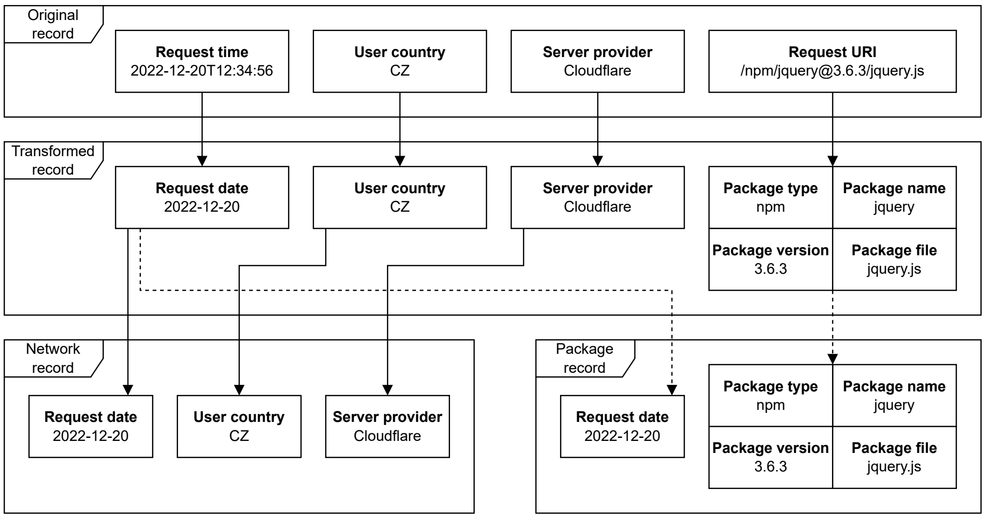


Figure 2.2: jsDelivr log processing

The second processing phase deals with the fact that the usage statistics features have the complexity of traditional online analytical processing (OLAP)[58] systems, but due to being exposed over a public API to many users, also the performance requirements of online transaction processing (OLTP)[59] systems.

These conflicting requirements are satisfied by using materialized views[60], which aggregate the source data into the formats needed by the individual API endpoints in advance, essentially transforming the user-facing operations into simple OLTP-style queries that can be supported by indexes.

The materialized views are generated once a day and then simply read from the disk instead of being computed on-the-fly on each request, which provides a significant performance improvement at the cost of somewhat higher disk usage—a fact that must be considered when designing new features.

This approach does not cause an additional delay in data availability because the API is designed to work with daily data summaries, so new data would

[58]Describes systems that handle a low volume of complex, not necessarily real-time queries.

[59]Describes systems that handle a high volume of simple queries and expect real-time results.

[60]MariaDB does not support materialized views natively, but the concept can be emulated with a combination of regular tables, stored procedures, and scheduled events.

only be available once a day anyway. However, there is one limitation—the data can only be queried for a set of pre-selected periods, not for custom date ranges.

The next sections provide an overview of all source data tables, as well as the currently used materialized views.

2.4.2.1.1 Package Data

The package data are primarily stored in tables `package`, `package_version`, `file`, and `file_hits`. However, because most features only work with data on package or package version levels, and aggregating file-level data is very slow due to the number of records, there are additional tables `package_hits` and `package_version_hits`, as shown in Figure 2.3.

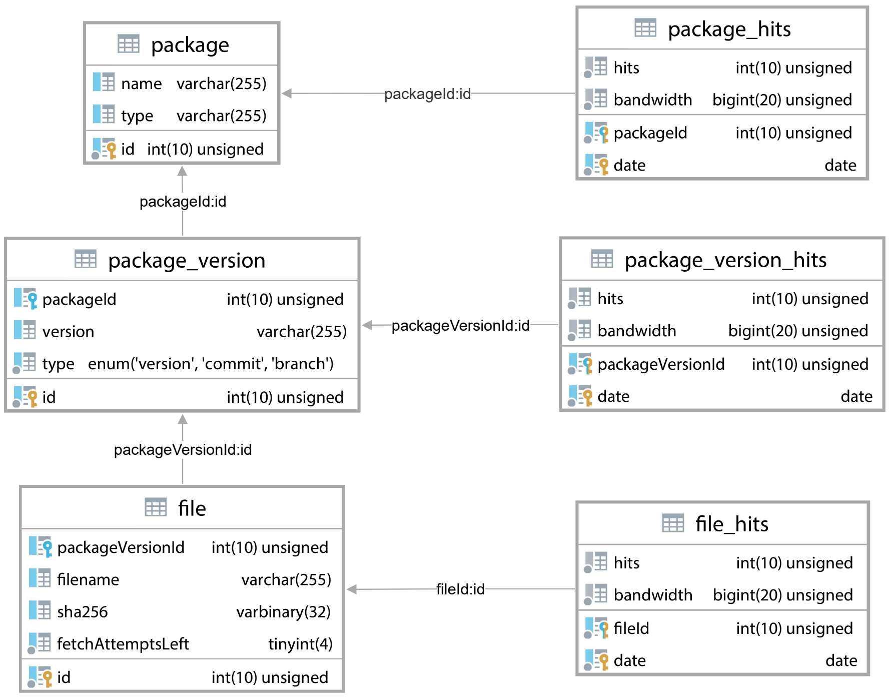


Figure 2.3: Package data relational schema

The two additional tables are not treated as materialized views and are managed by the same process as `file_hits`. While technically duplicating data, this design provides a significant performance improvement for common operations, and data consistency across the tables is achieved by always updating all three tables in a single transaction.

To support the current features, there is also one materialized view, which keeps track of hits, bandwidth, and ranks for each package, as shown in Figure 2.4. This view is generated from `package_hits` data once a day.

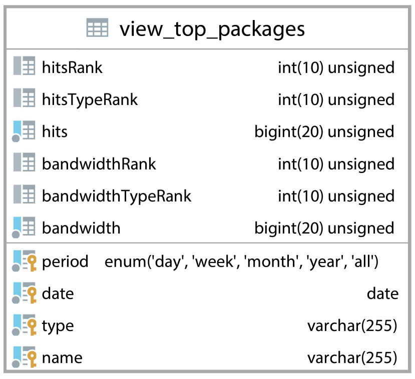

view_top_packages
hitsRank int(10) unsigned
hitsTypeRank int(10) unsigned
hits bigint(20) unsigned
bandwidthRank int(10) unsigned
bandwidthTypeRank int(10) unsigned
bandwidth bigint(20) unsigned
period enum('day', 'week', 'month', 'year', 'all')
date date
type varchar(255)
name varchar(255)

Figure 2.4: Package data materialized views

2.4.2.1.2 Proxy Endpoint Data

The proxy data are primarily stored in tables `proxy`, `proxy_file`, `proxy_file_hits`, but there is again an additional table `proxy_hits` to improve performance, as shown in Figure 2.5.

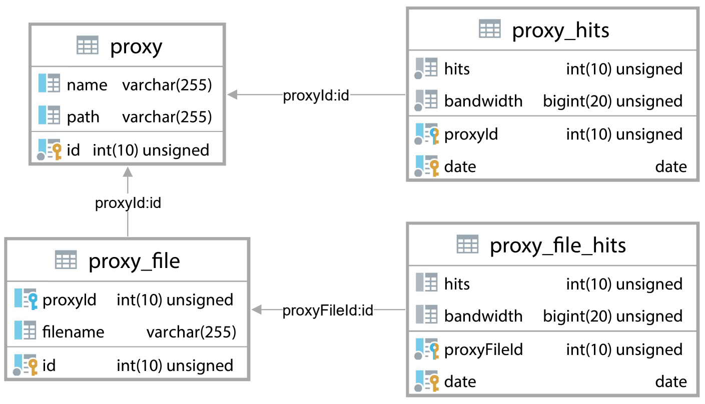
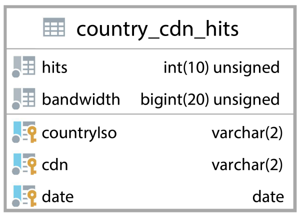


Figure 2.5: Proxy data relational schema

2.4.2.1.3 Network Data

The network data are stored in the `country_cdn_hits` table shown in Figure 2.6. The `countryIso` field refers to an ISO 3166-1 alpha-2 code [35], but the list of countries is not stored in the database. Similarly, the `cdn` field refers to a CDN provider code, but the list of providers is not currently stored in the database.



country_cdn_hits	
hits	int(10) unsigned
bandwidth	bigint(20) unsigned
countryIso	varchar(2)
cdn	varchar(2)
date	date

Figure 2.6: Network data relational schema

2.4.2.1.4 Browser and Platform Data

With browser and platform data, there were several design options in regard to data resolution. Unlike the rest of the data, browser and platform data are expected to be tracked only on a monthly basis, and the design options were considered with the following expected monthly value cardinalities:

- 100 browsers,
- 1000 browser versions (10 for each browser),
- 10 platforms,
- 100 platform versions (10 for each platform),
- 200 countries (data must be stored separately for each country).

First, separate records could be stored for each browser version and each platform version. This would result in $1000 \times 200 = 200\,000$ records for browser versions and $100 \times 200 = 20\,000$ records for platform versions, but there would be no link between the browsers and the platform they run on.

As a second option, one record could be stored for each combination of browser version and platform version, which would, in the worst case, result in $1000 \times 100 \times 200 = 20\,000\,000$ records.

As a third option, browser version data could be linked to platforms (but not their versions) and platform version data could be stored separately. This would require $1000 \times 10 \times 200 = 2\,000\,000$ and $100 \times 200 = 20\,000$ records.

The third option was chosen as a reasonable compromise, and the resulting schema is shown in Figure 2.7.

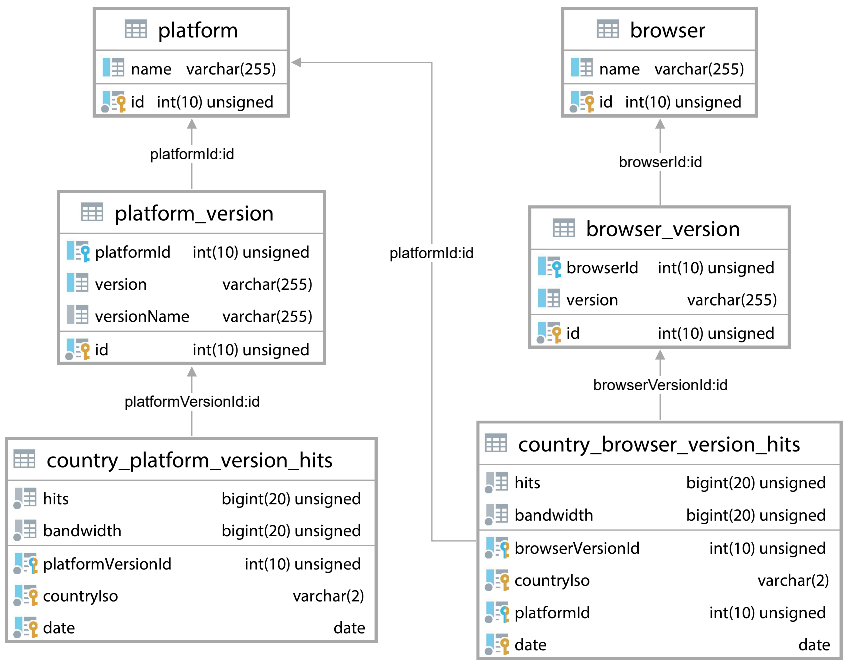


Figure 2.7: Browser and platform data relational schema

2.4.2.2 Get Package Statistics

`GET /v1/package/npm/:name/stats/:groupBy?/:period?`

This is the first of the current statistics endpoints, providing daily usage statistics for the package. The `groupBy` parameter can have a value of `version` or `date` and configures the grouping of the data.

With the value of `version`, the `versions` object has a separate key for each version for which some data are available in the selected time period, and the `dates` object provides a further daily breakdown for each version, as shown in Listing 2.4.

With the value of `date`, the grouping is reversed—data are first grouped by date at the top level, and a version breakdown is available for each date.

The `period` parameter identifies the time period for which data are returned and can have a value of `day`, `week`, `month`, or `year`.

```
{
    "rank": 9,
    "typeRank": 7,
    "total": 1462378980,
    "versions": {
        "1.10.0": {
            "total": 997335,
            "dates": {
                "2022-11-07": 46483,
                "2022-11-08": 52859,
                ...
            }
        },
        ...
    },
}
```

Listing 2.4: GET /v1/package/npm/:name/stats[61]

While the endpoint provides the required functionality, the granularity of returned data may often be unnecessarily high, which results in very big responses. For example, asking for monthly statistics of a package with 50 different versions would produce a response with approximately 1550 values, and a response for yearly data would include over 18 000 values.

Considering the described user stories, it might be beneficial to provide an endpoint that lists daily package data without a version breakdown, which would be sufficient for User Story 01, and another one that lists the most popular versions (possibly with pagination support) to support User Story 03. Such change would also remove the need for the `groupBy` parameter, which would otherwise be better moved to the query string or replaced with content negotiation.

[61]`https://data.jsdelivr.com/v1/package/npm/jquery/stats`

Possible improvements:

- Split into multiple endpoints to support different use cases more efficiently.
- The `period` parameter suffers from the same problem as `range` in Section 2.4.1.2 and would be better moved to the query string.

2.4.2.3 Get Package Version Statistics

```
GET /v1/package/npm/:name@:version/stats/:groupBy?/:period?
```

This endpoint provides daily usage statistics for the package version. The `groupBy` parameter can have a value of `file` or `date` and configures the grouping of the data.

With the value of `file`, the `files` object has a separate key for each version for which some data are available in the selected time period, and the `dates` object provides a further daily breakdown for each version, as shown in Listing 2.5.

With the value of `date`, the grouping is reversed—data are first grouped by date at the top level, and a file breakdown is available for each date.

```
{
    "total": 92950761,
    "files": {
        "/dist/jquery.min.js": {
            "total": 91870294,
            "dates": {
                "2022-11-07": 3348192,
                "2022-11-08": 3539236,
                ...
            }
        },
        ...
    }
}
```

Listing 2.5: GET /v1/package/npm/:name@:version/stats[62]

Possible improvements:

- This endpoint suffers from the same problems as the previous one and

[62] `https://data.jsdelivr.com/v1/package/npm/jquery@3.6.1/stats`

would benefit from being split into two and from moving the `period` parameter into the query string.

2.4.2.4 Get a Package Badge

`GET /v1/package/npm/:name/badge/:period?`

This endpoint returns a "badge" with the package request total, which can be directly embedded in websites, as shown in Figure 2.8.

Figure 2.8: GET /v1/package/npm/:name/badge[63]

As the response is a simple SVG image, there is not much to discuss in terms of the response format in this case.

2.4.2.5 List Top Packages

`GET /v1/stats/packages/:period?`

This endpoint lists the most popular packages and their request totals for the selected period. Pagination is available via the `limit` and `page` query string parameters. The response format is shown in Listing 2.6. Other than the `period` parameter, the current form of the endpoint has no issues.

```
[
    {
        "type": "npm",
        "name": "prebid-universal-creative",
        "hits": 8933727436
    },
    {
        "type": "npm",
        "name": "bootstrap",
        "hits": 8176932289
    },
    ...
]
```

Listing 2.6: GET /v1/stats/packages[64]

[63] `https://data.jsdelivr.com/v1/package/npm/jquery/badge`
[64] `https://data.jsdelivr.com/v1/stats/packages`

2.5 Conclusion

This analysis was an important prerequisite for understanding the additional requirements and designing the new features. The architecture, design, and technology choices were made based on the specific project requirements, particularly the need to efficiently work with large amounts of data and handle a high number of requests, and there is no need for any major changes.

In terms of endpoints design, minor improvements are possible: moving certain path parameters to the query string, using a more efficient response format in some cases, and utilizing hypermedia.

Chapter 3

Design

Based on the analysis of the current state and the new requirements, this chapter describes the necessary changes in detail. It starts with general concepts that apply to the whole service and continues with the individual features.

3.1 API Versioning Strategies

When designing the new features, it is important to remember that the API is already widely used and that all existing clients must continue to work. This section, therefore, starts by examining the commonly used versioning strategies and comparing their advantages and disadvantages.

When discussing possible strategies, it is good to keep in mind that versioning as a whole covers several types of problems. When jsDelivr switched to its on-demand system in 2017 and introduced its current API, it was essentially a brand-new service. It shared the same goals and had similar features as its predecessor, but the underlying data model changed in its entirety. The new API launched on a separate subdomain where it runs entirely independent of the previous version.

Such change could be thought of as simply launching a new service, but that would just be an attempt to hide the versioning-related problems, as the features remain similar and target users remain the same. There may be other scenarios where the API as a whole must undergo significant changes—whether as a result of its technical limitations or as a result of business decisions—and the following text will use the term "service-wide versioning" for these cases.

A lot more common are changes that do not change the key concepts of the API but simply change the representation of data. Maybe the format of some resources did not turn out to work well in practice, maybe some fields that were originally present are no longer relevant. These cases will be referred to as "per-resource versioning" in this section.

Note that the jsDelivr API includes a "v1" prefix in its URI, so it may seem the versioning strategy has already been decided, but the reality is that the prefix has been added at launch without much thought, and a clear versioning strategy has not yet been established.

3.1.1 URI Prefix

The first versioning approach is including a version number in the URI:

```
GET https://api.example.com/v1/resource HTTP/1.1
```

The idea is that any non-breaking changes, such as adding new endpoints, fields, or optional parameters to the existing ones, can be applied without changing the version number. For breaking changes, such as removing a field, the version in the URI is incremented.

Because the prefix is inevitably present in all URIs, releasing a new version creates a new copy of all resources, which makes this approach work fairly well for service-wide versioning, but it is a rather lousy mechanism for handling changes of a single resource. Any time a new version is released, the users must carefully read the list of changes (assuming one is available), figure out if they are impacted, and then update their applications.

This approach somewhat contradicts the "evolvability" principle of REST, and as Sturgeon points out [36], Fielding goes as far as to say that "a v1 is a middle finger to your API customers, indicating RPC/HTTP (not REST)" [37].

Nevertheless, not every service has to be RESTful, and there are reasons why many services choose this approach: it is straightforward to implement, results in URLs with stable response format (which may contradict the HTTP/REST ideas of content negotiation, but it means applications will not accidentally break because developers misunderstood the versioning scheme handling), it does not require any extra configuration for proper caching, since there are no headers or other parameters involved, and it makes API resources easy to share because the links are copy-pastable (a property particularly convenient for GET-only APIs).

Advantages:

- Simple to implement.
- Copy-pastable links to each version/representation.
- Cache friendly.

Disadvantages:

- Resource duplication.
- High impact of updates on API clients.

3.1.2 Hostname

This approach is similar to the previous one, except the version number or a code name is put in the hostname:

```
GET https://api-v1.example.com/resource HTTP/1.1
```

As one would expect, this has both the same advantages and disadvantages as the previous solution, with one exception—if each version runs as an entirely independent application, it may be easier to manage and correctly route requests.

Advantages:

- Simple to implement.
- Copy-pastable links to each version/representation.
- Cache friendly.

Disadvantages:

- Resource duplication.
- High impact of updates on API clients.

3.1.3 Query String Parameters

This option once again looks very similar to the previous two:

```
GET https://api.example.com/resource?version=1 HTTP/1.1
```

However, in this case, there are two possible interpretations as to what the version means:

1. A version of the service, which provides similar properties as the previous two approaches.
2. A version of the resource, with different resources possibly having different sets of versions.

Focusing on the second interpretation of versioning each resource type independently, this approach mitigates the “high impact of updates” problem because parts that do not change do not receive a new version. The solution is still cache-friendly, and links are copy-pastable, but the client needs to keep track of the correct version for each resource.

One new question that arises with this approach is what version should be used if the parameter is not present in the request. Returning the latest available version may be more friendly to new developers exploring the API, but it means that if someone forgets to specify a version in their application, it will eventually break. It also requires that the versioning scheme be clear from the beginning and that all developers be aware of it. Returning the oldest version favors stability, and clients need to opt into updates themselves.

Advantages:

- Simple to implement.
- Allows a fine-grained per-resource versioning.

- Copy-pastable links to each version/representation.
- Cache friendly.

Disadvantages:

- Need to keep track of available versions for each resource.

3.1.4 Custom Headers

With this approach, we leave the URL space and move to a place designed to transmit metadata—the HTTP headers:

```
GET https://api.example.com/resource HTTP/1.1
Accept-Version: 1
```

While the header name resembles other content-negotiation headers that are part of the HTTP specification, there is no standardized header specifically for versioning, so different names are used by different implementations.

This approach is similar to the previous one in that it allows both service-wide versioning and per-resource versioning. While non-standard, it is de-facto a form of content negotiation, with different versions being simply different "representations" of the same resource, which makes this approach the first on the list to respect the REST concepts.

To ensure proper caching, server responses must include a `Vary: Accept-Version` header when using this approach to inform clients that the response content depends on the requested version, but other than that, it does not create any caching issues.

A commonly overlooked disadvantage lies in how web browsers handle Cross-Origin Resource Sharing (CORS). Because the custom header is not on the *CORS-safelisted request-header* list [38, §2.2.2], any request that includes such a header is subject to a CORS preflight [38, §4.1], which adds extra delay in handling the request. Another minor downside is that API links are no longer easily sharable, and requests need to be crafted using more specialized tools.

Advantages:

- Allows a fine-grained per-resource versioning.
- Cache friendly.

Disadvantages:

- Not as simple to implement.
- No copy-pastable links to each version/representation.
- Subject to CORS preflight.

3.1.5 The Accept Header

Finally, we get to an approach that fully utilizes the existing HTTP features:

```
GET https://api.example.com/resource HTTP/1.1
Accept: application/vnd.service-name.v1
```

This approach builds on the idea that a media type does not necessarily mean simple JSON or XML, but each service or even resource can have its own type. Choosing a version then means simply choosing one of the supported media types using standard content negotiation, which makes this approach the most RESTful. It can also be modified to support per-resource versioning by including the resource name in the media type:

```
GET https://api.example.com/resource HTTP/1.1
Accept: application/vnd.service-name.resource-name.v1
```

Because the `Accept` header is included on the *CORS-safelisted request-header* list [38, §2.2.2], the requests are not subject to CORS preflight in this case. However, there are major and often overlooked complications with CDN-level caching. Similarly to the previous approach, the responses must include a `Vary: Accept` header. Because `Accept` is a standard content-negotiation header, it is usually automatically set by web browsers, so requests that do not explicitly override it are sent with the default value. Due to the value structure (shown in Listing 3.1), different browsers are likely to use a different default, which creates many response variants that need to be cached independently.

```
Accept: text/html, application/xhtml+xml, application/xml;q=0.9,
        image/avif, image/webp, image/apng, */*;q=0.8,
        application/signed-exchange;v=b3;q=0.9
```

Listing 3.1: Default Accept header in Google Chrome v108 on Windows

In a blog post on best practices for using the `Vary` header, Mulhuijzen discussed a similar issue with the `Accept-Encoding` header, and his analysis found 42 unique header values in a sample of 100 000 requests, even though there is only a handful of encoding algorithms used on the web [39]. The problem is likely to be even more prevalent with `Accept` as the pool of possible values is much bigger. As a result, using `Vary: Accept` can have a significant negative effect on CDN caching efficiency.

Some CDN providers provide advanced features that can be used to normalize the header value [40], but it is a non-trivial task that may greatly complicate the setup and possibilities of a later provider change.

Advantages:

- Allows a fine-grained per-resource versioning.

Disadvantages:

- Not as simple to implement.
- No copy-pastable links to each version/representation.
- Tricky caching.

3.1.6 Evolution

API evolution builds on the idea of avoiding breaking changes unless they are absolutely necessary—and if all changes are backward compatible, there is essentially no need for explicit versioning. Adding new features can typically be done in a backward-compatible fashion, and instead of removing them, old features can be deprecated but left available [41].

The concept requires that everything be designed with future extensibility in mind, but if a change of an existing concept is unavoidable, a new resource can be created under a new name while the old one continues to be available for older clients. As such, instead of being communicated through numbers, where one version can hide one or more changes, change is communicated through names. The extra work that comes with creating new names may discourage changing things too often, which just supports the main idea behind this approach.

Although the idea of dismissing versioning entirely may be tempting, even the proponents of the evolution principle admit that sometimes there is no way but to break things [42]. Maybe a more practical approach would be to take the main idea—that breaking changes should be avoided if at all possible, but combine it with one of the previously described versioning approaches.

3.1.7 Conclusion

Each of the examined versioning strategies has its use cases. In this case, the jsDelivr Core Team agreed that the main focus is minimizing the burden of upgrading for the existing API users. As such, changes should follow the Evolution principle—be backward compatible if possible, or be introduced as new endpoints that the existing users can switch to at any time while the existing ones continue to work.

3.2 Common Properties

There are a few properties shared by several endpoints that should be consistent across the whole API. Hence, before the design of specific endpoints, this section discusses those shared properties.

3.2.1 Bandwidth Data

Following the analysis in Section 2.4.2, we know that the API has only worked with one type of value so far (hits), and the response formats of *Get Package Statistics* and *Get Package Version Statistics* endpoints cannot be easily extended.

This problem could be solved by introducing a new query string option, e.g., `type`, which would define the type of the returned statistics. Such an option might even be preferred if clients only requested one specific type most of the time. On the other hand, requesting multiple types would require multiple HTTP requests.

However, the analysis also showed that the endpoints in question would benefit from being split into two versions with different data granularity, which would provide an opportunity to make the format more extensible as well.

The summary format of *List Top Packages* can already be extended without issues. As such, we are not strictly bound by the existing implementation in either case.

Keeping the idea of always providing a value total and then a more detailed breakdown, the suggested format for detailed hits and bandwidth statistics is shown in Listing 3.2. The format has the advantage of being able to incorporate multiple totals and ranks and can be easily further extended should a third statistics type be introduced in the future.

```json
{
    "hits": {
        "rank": 9,
        "typeRank": 7,
        "total": 1462378980,
        ...
    },
    "bandwidth": {
        "rank": 17,
        "typeRank": 9,
        "total": 61982153019627,
        ...
    }
}
```

Listing 3.2: Hits and bandwidth format

For the cases where only totals are listed without breakdowns, such as in the *List Top Packages* endpoint, the format can be extended, as shown in Listing 3.3.

```
[
    {
        "type": "npm",
        "name": "prebid-universal-creative",
        "hits": 8933727436,
        "bandwidth": 83103736429868
    },
    {
        "type": "npm",
        "name": "bootstrap",
        "hits": 8176932289,
        "bandwidth": 204497235587383
    },
    ...
]
```

Listing 3.3: Hits and bandwidth totals format

3.2.2 Quarterly and Historical Data

The support for 90-day data ranges can be added by including `quarter` in the list of valid `period` values. Furthermore, the `period` parameter can also be extended to support querying historical data by accepting a date in one of the following ISO 8601 formats [43]:

- `YYYY` for a specific year (e.g., `2022`),
- `YYYY-Qq` for a specific quarter[65] (e.g., `2022-Q1`),
- `YYYY-MM` for a specific month (e.g., `2022-01`),
- `YYYY-Www` for a specific week (e.g., `2022-W01`),
- `YYYY-MM-DD` for a specific day (e.g., `2022-01-01`).

3.2.3 Period Comparison

The period comparison feature can be approached in a few different ways. If a client can request any specific period—which it can after the changes proposed in the previous section—it can certainly make two requests for two different periods and compare the data without any additional API features.

This approach would work for endpoints that return data for a single package, such as *Get Package Statistics* and *Get Package Version Statistics*. Unfortu-

[65]ISO 8601 does not define any format for quarters, so this was inspired by the format specified for weeks.

nately, it would not work for endpoints returning a list of packages, such as *List Top Packages*, because the contents of the list can differ between periods, so a package present in one response may not be in the other (e.g., a package can be the most popular in December but not even be in the top 100 in November).

Additionally, some period types may differ in length—there are leap years, which affect both yearly and quarterly data, and months themselves have varying numbers of days too. As a result, comparing data from two different months may prove challenging to be accomplished correctly.

For these reasons, the API should provide a comparison directly in its responses. A possible way to do so for a single resource and a list of resources is shown in Listing 3.4 and Listing 3.5, respectively.

```
{
    "hits": {
        "rank": 9,
        "typeRank": 7,
        "total": 1777734912,
        "prev": {
            "rank": 8,
            "typeRank": 7,
            "total": 1821065662
        },
        ...
    },
    "bandwidth": {
        "rank": 17,
        "typeRank": 9,
        "total": 61982153019627,
        "prev": {
            "rank": 16,
            "typeRank": 9,
            "total": 63393192275045
        },
        ...
    }
}
```

Listing 3.4: Previous period comparison format

The advantage of this approach is that a client can get the basic idea of the trend with minimal effort, which is requested by User Story 02 and User Story 09, while more detailed historical data can still be obtained, as described in the previous section. Additionally, the API can automatically adjust the `prev` values to account for period length differences.

```
[
    {
        "type": "npm",
        "name": "prebid-universal-creative",
        "hits": 8933727436,
        "bandwidth": 83103736429868,
        "prev": {
            "hits": 9652830146,
            "bandwidth": 94071018724816
        }
    },
    {
        "type": "npm",
        "name": "bootstrap",
        "hits": 8176932289,
        "bandwidth": 204497235587383,
        "prev": {
            "hits": 8376204240,
            "bandwidth": 212666680289634
        }
    },
    ...
]
```

Listing 3.5: Previous period comparison format in lists

3.2.4 Hypermedia

While the definition of the HATEOAS principle is usually attributed to Fielding's dissertation, the thesis provides little guidance on how such a principle would look in practice. In fact, there is no universally accepted format for representing links between resources in REST APIs. On a high-level, there are two options:

1. Including links directly in the response body. The exact format is not standardized and varies across implementations [44].
2. Including links in the `Link` response header. The format is specified by RFC 8288 in this case [45].

The first option is generally more flexible as the links can be arbitrarily nested when there are multiple resources in one response, as shown in Listing 3.6. This makes it more suitable in most cases.

```
[
    {
        "type": "npm",
        "name": "bootstrap",
        "hits": 8176932289,
        "bandwidth": 204497235587383,
        "prev": {
            "hits": 8376204240,
            "bandwidth": 212666680289634
        },
        "links": {
            "self": "/v1/stats/packages/npm/bootstrap",
            "versions": "/v1/stats/packages/npm/bootstrap/versions"
        }
    },
    ...
]
```

Listing 3.6: Linking resources in the response body[66]

One issue with representing links in the response body is that unless an envelope[67] is used, there is no place to put the links that relate to the whole collection in responses that contain multiple resources—particularly pagination links. However, this is an excellent use case for the `Link` header.

Regardless of where the links are located, it is important to pay attention to the relation types, which specify how the links are to be interpreted. For example, in Listing 3.6, there is a link with a registered `self` relation, which refers to the original resource included in the list, and a link with a custom `versions` relation, the meaning of which is specific to this service. The link of registered relations is maintained by the Internet Assigned Numbers Authority [46].

3.2.5 Pagination

Endpoints that return a list of resources should support pagination, and, again, there are multiple ways to approach this:

1. **Offset pagination** typically uses `limit` and `page` (or `offset`) parameters that specify how many resources to return and how many to skip at the beginning.

[66]The actual links should be absolute URLs and include the host, but only the path portion is shown here for brevity.

[67]An object that wraps the returned data so that additional metadata can be returned in the same response body.

2. **Cursor pagination** replaces the `page` parameter with a `cursor` parameter, which identifies the last resource in the current set. This concept is sometimes also referred to as key-set pagination or seek pagination with minor differences in what exactly the `cursor` parameter represents, but the core idea is always the same.

Offset pagination typically translates to a `SELECT` ... `OFFSET` query on the backend if a relational database is used. This approach is sometimes criticized [47] because the usual implementation can lead to performance problems with large offset values and inconsistencies, such as duplicate results if the underlying data changes while requesting different pages. On the other hand, it has the advantage of being very simple to implement and providing random access (users can specify an exact page) to all resources.

The advantage of cursor pagination is that it can be implemented using a `SELECT` ... `WHERE` query in SQL databases, which can be supported by indexes, and that the results are consistent even if the underlying data change between requests. A disadvantage, though, is that only sequential access is possible.

jsDelivr already uses the offset approach in the *List Top Packages* endpoint, and because the dataset does not change frequently and random access is a desired feature, the same approach should be used for the new endpoints. This decision can be revisited in the implementation phase should the performance impact turn out to be significant.

Additionally, all paginated endpoints should provide pagination metadata, which are currently missing: the total number of records and the number of available pages. Because the existing endpoints do not wrap the data in an envelope, the metadata must go into headers. The header names are not standardized in this case, so `X-Total-Count` will be used for the number of resources and `X-Total-Pages` for the number of available pages.

Links to the surrounding pages should also be provided using the `Link` header as described in the previous section. The HTML specification defines the `prev` and `next` relation types, which can be used for the previous and the next page, respectively [48], and RFC 5988 defines `first` and `last` relation types, which can be used for the first and the last page [49]. The final list of pagination-related headers is shown in Listing 3.7.

```
Link: <https://data.jsdelivr.com/v1/stats/packages>; rel="first",
      <https://data.jsdelivr.com/v1/stats/packages?page=2>; rel="prev",
      <https://data.jsdelivr.com/v1/stats/packages?page=3>; rel="self",
      <https://data.jsdelivr.com/v1/stats/packages?page=4>; rel="next",
      <https://data.jsdelivr.com/v1/stats/packages?page=100>; rel="last"
X-Total-Count: 402302
X-Total-Pages: 4024
```

Listing 3.7: Pagination metadata

3.3 Package Statistics

As should be clear from the previous parts of the text, the package statistics endpoints need to be entirely redesigned to better match the project requirements. That means new endpoint paths need to be selected too. Luckily, there are two inconsistencies in the current path scheme that have not yet been discussed, and that can be taken advantage of:

1. The statistics endpoints for individual packages build upon the metadata path scheme and add a `/stats` suffix, while the most popular packages list uses a `/stats` prefix instead.
2. The package endpoints are grouped under a singular `/package` prefix, while the most popular packages list uses the plural `/packages` form.

Because the new requirements include several new statistics-related features that share many common properties, it makes sense to have all of them grouped under a `/stats` prefix. At the same time, the singular `/package` prefix can be changed to plural for better consistency. By introducing the changes under a new path scheme, the existing endpoints are allowed to keep working without any changes. The overview of changes is shown in Listing 3.8.

```
  /v1/stats/packages                               // format-only changes
- /v1/package/npm/:name/stats/:groupBy?            // kept but deprecated
+ /v1/stats/packages/npm/:name
+ /v1/stats/packages/npm/:name/versions
- /v1/package/npm/:name@:version/stats/:groupBy? // kept but deprecated
+ /v1/stats/packages/npm/:name@:version
+ /v1/stats/packages/npm/:name@:version/files
```

Listing 3.8: Overview of package statistics endpoints changes

As suggested in Chapter 2, the `period` parameter should be moved to the query string for all endpoints.

3.3.1 List Top Packages

`GET /v1/stats/packages`

The first endpoint combines all properties described in the previous section: bandwidth data added as a `bandwidth` property, period comparison via the `prev` object, hypermedia provided in the `links` object, and pagination support via the `limit` and `page` parameters. All of these changes are backward compatible, as shown in Listing 3.9.

Because the response now contains multiple types of values, a new parameter `by` with the possible value of `hits` or `bandwidth` is introduced to define the list order, and a parameter `type` with the possible value of `gh` or `npm` is introduced to filter specific package types. Links to more detailed statistics are included in the response.

```json
[
    {
        "type": "npm",
        "name": "bootstrap",
        "hits": 8176932289,
        "bandwidth": 204497235587383,
        "prev": {
            "hits": 8376204240,
            "bandwidth": 212666680289634
        },
        "links": {
            "self": "/v1/stats/packages/npm/bootstrap",
            "versions": "/v1/stats/packages/npm/bootstrap/versions"
        }
    },
    ...
]
```

Listing 3.9: GET /v1/stats/packages

3.3.2 Get Package Statistics

`GET /v1/stats/packages/npm/:name`

This new endpoint provides daily usage statistics but without an additional version breakdown to reduce the response size in cases where only daily totals are needed, as suggested in Section 2.4.2.2. The format, which also incorporates all new features, is shown in Listing 3.10.

```
{
    "hits": {
        "rank": 12,
        "typeRank": 8,
        "total": 1791475308,
        "dates": {
            "2022-12-20": 59852966,
            "2022-12-21": 58074028,
            ...
        },
        "prev": {
            "rank": 11,
            "typeRank": 8,
            "total": 1807320472
        }
    },
    "bandwidth": { ... },
    "links": {
        "self": "/v1/stats/packages/npm/jquery",
        "versions": "/v1/stats/packages/npm/jquery/versions"
    }
}
```

Listing 3.10: GET /v1/stats/packages/npm/:name

3.3.3 List Top Package Versions

`GET /v1/stats/packages/npm/:name/versions`

This new endpoint focuses on User Story 03, and instead of returning an object with values for all versions like its predecessor, it returns a sorted list of the top n versions, with the order defined via the `by` parameter.

Thanks to the added pagination support, the client no longer has to load the whole list if it only cares about the top few entries, or it can load the entries incrementally. It also does not need to do any additional sorting. The format is shown in Listing 3.11.

```
[
    {
        "type": "version",
        "version": "3.6.0",
        "hits": {
            "total": 514886000,
            "dates": {
                "2022-12-20": 16462020,
                "2022-12-21": 15609835,
                ...
            }
        },
        "bandwidth": { ... },
        "links": {
            "self": "/v1/stats/packages/npm/jquery@3.6.0",
            "files": "/v1/stats/packages/npm/jquery@3.6.0/files"
        }
    },
    ...
]
```

Listing 3.11: GET /v1/stats/packages/npm/:name/versions

3.3.4 Get Package Version Statistics

`GET /v1/stats/packages/npm/:name@:version`

This new endpoint provides daily usage statistics for the package version without an additional file breakdown. The format mirrors that of the *Get Package Statistics* endpoint, as shown in Listing 3.12, but excludes the fields `rank`, `typeRank`, and `prev`, as those are not expected to be tracked for individual versions.

```
{
    "hits": {
        "total": 514886000,
        "dates": {
            "2022-12-20": 16462020,
            "2022-12-21": 15609835,
            ...
        }
    },
    "bandwidth": { ... },
    "links": {
        "self": "/v1/stats/packages/npm/jquery@3.6.0",
        "files": "/v1/stats/packages/npm/jquery@3.6.0/files"
    }
}
```

Listing 3.12: GET /v1/stats/packages/npm/:name@:version

3.3.5 List Top Package Version Files

`GET /v1/stats/packages/npm/:name@:version/files`

This new endpoint follows the same idea as the *List Top Package Versions* endpoint. It focuses on User Story 04 and returns a sorted list of the top n files, with the order defined via the `by` parameter. The format is shown in Listing 3.13.

```
[
    {
        "name": "/dist/jquery.min.js",
        "hits": {
            "total": 482683431,
            "dates": {
                "2022-12-20": 15003649,
                "2022-12-21": 14336204,
                ...
            }
        },
        "bandwidth": { ... }
    },
    ...
]
```

Listing 3.13: GET /v1/stats/packages/npm/:name@:version/files

3.4 Proxy Endpoints Statistics

Proxy endpoint statistics are a new feature aimed at providing basic insight into the usage of custom endpoints used by a few projects, as requested by Functional Requirement 05.

The proxy endpoints provide increased flexibility for projects that cannot follow the standard npm publishing flow, but that also means analyzing their usage is more challenging—there is no concept of releases, and the URL structure is entirely up to the project. For this reason, the API only provides two features: getting daily request and bandwidth totals and listing the most popular files. In both cases, a `period` parameter is available similarly to package statistics.

Because the list of all proxy endpoints has never been public, the API does not add a way to list the most popular endpoints, as is possible with packages—the endpoints are designed so that this can be added later, however, if deemed useful.

3.4.1 Get Proxy Statistics

`GET /v1/stats/proxies/:name`

This new endpoint provides daily usage statistics for the proxy endpoint. Its format is greatly inspired by the *Get Package Statistics* endpoint, as shown in Listing 3.14.

```
{
    "hits": {
        "total": 11706855,
        "dates": {
            "2022-12-21": 437705,
            "2022-12-22": 375894,
            ...
        },
        "prev": { "total": 17533643 }
    },
    "bandwidth": { ... },
    "links": {
        "files": "/v1/stats/proxies/pyodide/files"
    }
}
```

Listing 3.14: GET /v1/stats/proxies/:name

3.4.2 List Top Proxy Files

`GET /v1/stats/proxies/:name/files`

This new endpoint is similar to *List Top Package Version Files* and returns a sorted list of the top n files, with the order defined via the `by` parameter, except there is no daily breakdown. The format allows adding it later if needed, as shown in Listing 3.15.

The daily breakdown is omitted for performance reasons because analysis of the existing proxy endpoints has shown that some have more than 500 000 files, and if a breakdown were to be provided for every file, the query would have to run over all records instead of a small pre-aggregated set.

```
[
    {
        "name": "/v0.21.3/full/pyodide.js",
        "hits": { "total": 379513 },
        "bandwidth": { "total": 1869361770 }
    },
    {
        "name": "/v0.21.3/full/pyodide_py.tar",
        "hits": { "total": 347740 },
        "bandwidth": { "total": 25752775839 }
    },
    ...
]
```

Listing 3.15: GET /v1/stats/proxies/:name/files

3.5 Network Statistics

The network statistics are a new feature aimed at providing insight into the usage of the service as a whole, as requested by Functional Requirement 06. The feature works with two datasets:

1. network data consisting of date, country, provider, hits, and bandwidth records,
2. package and proxy data, consisting of records for the individual packages and proxies.

3.5.1 Get Network Statistics

```
GET /v1/stats/network
```

This new endpoint works with the first dataset and focuses on giving a network-wide view of that data, as requested by User Story 07 and User Story 09. It provides the totals of `hits` and `bandwidth` and lists all providers that served some traffic in the specified `period`. The `providers` list is sorted by the individual provider's `total` value, and a further daily breakdown is available for each provider.

Because there is a possibility of introducing a further breakdown for each daily value in the future, each day is represented as an object with its own `total`, unlike in *Package Statistics* and *Proxy Endpoints Statistics*, where the daily values are simple numbers. The format is shown in Listing 3.16.

```
{
    "hits": {
        "total": 182792647052,
        "providers": [
            {
                "code": "CF",
                "name": "Cloudflare",
                "total": 110342830542,
                "dates": {
                    "2022-12-21": { "total": 4288943180 },
                    "2022-12-22": { "total": 4210110448 },
                    ...
                },
                "prev": { "total": 125552779443 }
            },
            ...
        ],
        "prev": { "total": 179958950602 }
    },
    "bandwidth": { ... }
}
```

Listing 3.16: GET /v1/stats/network

The endpoint supports mutually exclusive query string parameters `continent` and `country`, which accept a continent code or a country code and limit the query to the specified location. Note that the response format and size do not change in this case because values are already grouped across all locations.

3.5.2 Get Network Country Statistics

`GET /v1/stats/network/countries`

This new endpoint also works with the first dataset, but unlike the previous one, it focuses on User Story 08 and provides the totals of `hits` and `bandwidth` for each country that received some traffic in the specified `period`.

The `countries` list is sorted by the individual country's `total` value, and a further provider breakdown is available for each country. The format is shown in Listing 3.17.

```
{
    "hits": {
        "total": 182792647052,
        "countries": [
            {
                "code": "US",
                "name": "United States",
                "total": 34996110122,
                "providers": [
                    {
                        "code": "CF",
                        "name": "Cloudflare",
                        "total": 21125347485
                    },
                    ...
                ],
                "prev": {
                    "total": 35056057363
                }
            },
            ...
        ]
    },
    "bandwidth": { ... }
}
```

Listing 3.17: GET /v1/stats/network/countries

3.5.3 Get Network Content Statistics

`GET /v1/stats/network/content`

The last network endpoint works with the second dataset and provides the totals of `hits` and `bandwidth` for different content categories—packages, proxies, and other requests. For each category, a daily breakdown is available. The format is shown in Listing 3.18.

```
{
    "hits": {
        "total": 185300101597,
        "packages": {
            "total": 177783822166,
            "dates": {
                "2022-12-21": { "total": 5835999047 },
                "2022-12-22": { "total": 5742170812 },
                ...
            },
            "prev": { "total": 172200688390 }
        },
        "proxies": { ... },
        "other": { ... },
        "prev": { "total": 180266736800 }
    },
    "bandwidth": { ... }
}
```

Listing 3.18: GET /v1/stats/network/content

3.6 Browser and Platform Statistics

The browser and platform statistics feature utilizes the fact that, thanks to its scale, jsDelivr can observe and measure global technology usage trends. The currently available data consist of the user's country, their operating system and its version (further referred to as the "platform"), and the browser name and its version. This can tell us:

1. which browsers have the highest market share,
2. which browser versions have the highest market share,
3. in which countries is a specific browser the most popular,
4. on which platforms is a specific browser most often used,
5. which versions of a specific browser are the most popular,
6. in which countries is a specific browser version the most popular.

This list guides the design philosophy of the specific endpoints in the following sections. A similar list could also be constructed for platform usage, but it is, along with the respective endpoints, omitted for the sake of brevity. The implementation should follow the same principles for both the browser and the platform endpoints.

3.6.1 Regional Data

In addition to observing global trends, the data can be used to observe and compare smaller geographic regions too. The API should, therefore, provide a way to query not only global data but also data for specific continents or countries. This should be done using the `continent` and `country` query string parameters already introduced in Section 3.5.1.

3.6.2 Period Handling

Due to the high number of possible combinations of countries, operating system versions, and browser versions, jsDelivr tracks the browser and platform data on a monthly basis, as opposed to the daily basis used for all other statistics. This means it is not possible to query periods such as "the past 30 days". Instead, the `period` parameter can have the value of `s-month` representing the last calendar month, `s-year` representing the last calendar year, or a date representing a specific year, quarter, or month, as described in Section 3.2.2.

3.6.3 List Top Browsers (grouped versions)

```
GET /v1/stats/browsers
```

This endpoint returns a list of browsers ordered by market share and provides links to additional data for each specific browser, as shown in Listing 3.19.

```
[
    {
        "name": "Chrome",
        "share": 57.85,
        "prev": { "share": 58.15 },
        "links": {
            "countries": "/v1/stats/browsers/Chrome/countries",
            "platforms": "/v1/stats/browsers/Chrome/platforms",
            "versions": "/v1/stats/browsers/Chrome/versions"
        }
    },
    ...
}
```

Listing 3.19: GET /v1/stats/browsers

Note that this is the API's first more advanced use of the HATEOAS concept. The additional data that can be requested, and hence the set of returned links, depend on the request parameters. For instance, if the client requests the list of browsers for a specific country using the `country` parameter, the option of providing a list of the most popular countries no longer makes sense, and the link should be omitted.

3.6.4 List Top Browsers (separate versions)

```
GET /v1/stats/browsers/versions
```

This endpoint is similar to the previous one but lists each browser version independently, as shown in Listing 3.20. The reason for designing these as two distinct endpoints is that the list with separate versions is likely to contain at least a few hundred records, and the client should not be expected to fetch and aggregate all of them to work with grouped data.

```
[
    {
        "name": "Chrome",
        "version": "108",
        "share": 32.47,
        "prev": { "share": 0.08 },
        "links": {
            "countries": "/v1/stats/browsers/Chrome/versions/108/countries"
        }
    },
    ...
}
```

Listing 3.20: GET /v1/stats/browsers/versions

3.6.5 List Top Browser Countries

```
GET /v1/stats/browsers/:name/countries
```

This endpoint returns a list of countries where the specified browser is the most popular, represented by their ISO 3166-1 alpha-2 codes [35], i.e., the countries are ordered by market share of the specified browser in that country.

For example, in Listing 3.21, the specified browser has the highest share of 82.91 % in India. The provided links point to the list of top browsers and platforms in the country.

```
[
    {
        "country": "IN",
        "share": 82.91,
        "prev": { "share": 82.61 },
        "links": {
            "browsers": "/v1/stats/browsers?country=IN",
            "platforms": "/v1/stats/platforms?country=IN"
        }
    },
    ...
]
```

Listing 3.21: GET /v1/stats/browsers/:name/countries

3.6.6 List Top Browser Platforms

`GET /v1/stats/browsers/:name/platforms`

This endpoint returns a list of platforms the specified browser runs on, ordered by the platform's share of all requests made by the browser. For example, Listing 3.22 shows that 87.43 % of the browser's requests came from Android.

```
[
    {
        "name": "Android",
        "share": 87.43,
        "prev": { "share": 29.59 },
        "links": {
            "browsers": "/v1/stats/platforms/Android/browsers",
            "countries": "/v1/stats/platforms/Android/countries",
            "versions": "/v1/stats/platforms/Android/versions"
        }
    },
    ...
]
```

Listing 3.22: GET /v1/stats/browsers/:name/platforms

3.6.7 List Top Browser Versions

`GET /v1/stats/browsers/:name/versions`

This endpoint returns a list of the most popular browser versions ordered by their market share, as shown in Listing 3.23.

```
[
    {
        "version": "108",
        "share": 32.47,
        "prev": { "share": 0.08 },
        "links": {
            "countries": "/v1/stats/browsers/Chrome/versions/108/countries"
        }
    },
    ...
]
```

Listing 3.23: GET /v1/stats/browsers/:name/versions

3.6.8 List Top Browser Version Countries

`GET /v1/stats/browsers/:name/versions/:version/countries`

This endpoint is similar to *List Top Browser Countries* but returns data for a specific browser version. For example, in Listing 3.24, the specified browser version has the highest share of 54.89 % in Ecuador.

```
[
    {
        "country": "EC",
        "share": 54.89,
        "prev": { "share": 0.1 },
        "links": {
            "browsers": "/v1/stats/browsers?country=EC",
            "platforms": "/v1/stats/platforms?country=EC"
        }
    },
    ...
]
```

Listing 3.24: GET /v1/stats/browsers/:name/versions/:version/countries

3.7 Statistics Periods

Finally, there is a statistics-related feature not included in the original requirements but one that greatly compliments the new ability to query historical data described in Section 3.2.2.

While the client can request data for, e.g., a specific month, it has no way of knowing in advance if or which data are available for that month. This can get particularly confusing as different features are added over time.

For instance, package statistics have been available since mid-2017, while proxy statistics were added in 2020, and browser and platform statistics only in 2022. A basic use case, where the client presents a list of available datasets to the user, and the user picks one of the options, would therefore require that the client itself keep track of data availability.

This problem can be elegantly solved using the HATEOAS concept: the API can return a list of all periods it has some data for, with links to data that are available in each respective period. The exact form of this endpoint is described in the next section.

3.7.1 List Statistics Period

```
GET /v1/stats/periods
```

This endpoint returns an ordered list of all available periods, their types, and links to the available data. For example, Listing 3.25 shows a case where the `2022` period can be used to request network, package, and proxy data, but there are no proxy data available for `2022-Q3`, and only package data are available for `2022-03`.

Note that because proxy data can only be requested given a specific proxy name, the response format makes use of URI templates, as defined by RFC 6570 [50], to indicate that the `{name}` part of the link must be replaced with a specific value before the request is made.

```
[
    {
        "period": "2022",
        "periodType": "s-year",
        "links": {
            "network": "v1/stats/network?period=2022",
            "packages": "v1/stats/packages?period=2022",
            "proxies": "v1/stats/proxies/{name}?period=2022"
        }
    },
    ...
    {
        "period": "2022-Q3",
        "periodType": "s-quarter",
        "links": {
            "network": "v1/stats/network?period=2022-Q3",
            "packages": "v1/stats/packages?period=2022-Q3",
        }
    },
    ...
    {
        "period": "2022-03",
        "periodType": "s-month",
        "links": {
            "packages": "v1/stats/packages?period=2022-03",
        }
    },
    ...
]
```

Listing 3.25: GET /v1/stats/periods

3.8 Package Metadata

The metadata features, unlike the statistics features, do not require major changes but would benefit from moving a few parameters from the URL path to the query string and the use of hypermedia. Although minor, these changes would still require creating new versions of the endpoints, which raises the question if they are important enough to warrant doing so. Hence, this comes as the last part of this chapter.

Hypermedia are certainly a useful feature, and the current form of path parameters would complicate creating the OpenAPI document. Nevertheless, these changes, on their own, could easily be postponed. However, in the context of the planned statistics changes, it might make sense to redesign the metadata endpoints, too, for the sake of consistency. This way, new API users will get a better overall experience, while the existing ones will not be affected. The overview of changes is shown in Listing 3.26, with details described in the next sections.

```
- /v1/package/npm/:name                      // kept but deprecated
+ /v1/packages/npm/:name
- /v1/package/resolve/npm/:name@:range?      // kept but deprecated
+ /v1/packages/npm/:name/resolved
- /v1/package/npm/:name@:version/:structure? // kept but deprecated
+ /v1/packages/npm/:name@:version
```

Listing 3.26: Overview of package metadata endpoints changes

3.8.1 Get Package Metadata

```
GET /v1/packages/npm/:name
```

Compared to its previous form, the new version of this endpoint uses a more extensible format for the `versions` entries, adds links to the related resources to each version object, and also adds links related to the package as a whole, as shown in Listing 3.27.

```
{
    "type": "npm",
    "name": "jquery",
    "tags": {
        "beta": "3.6.1",
        "latest": "3.6.1"
    },
    "versions": [
        {
            "version": "3.6.1",
            "links": {
                "self": "/v1/packages/npm/jquery@3.6.1",
                "stats": "/v1/stats/packages/npm/jquery@3.6.1"
            }
        },
        ...
    ],
    "links": {
        "stats": "/v1/stats/packages/npm/jquery"
    }
}
```

Listing 3.27: GET /v1/packages/npm/:name

3.8.2 Get the Resolved Package Version

`GET /v1/packages/npm/:name/resolved`

Compared to its previous form, the new version of this endpoint uses a query string parameter `specifier` instead of the `range` path parameter. The new name better represents the fact that the value may be not only a version range but also a release tag. The fields `type`, `name`, and `links` are added in the response, as shown in Listing 3.28.

```json
{
    "type": "npm",
    "name": "jquery",
    "version": "3.6.1",
    "links": {
        "self": "/v1/packages/npm/jquery@2.2.4",
        "stats": "/v1/stats/packages/npm/jquery@2.2.4"
    }
}
```

Listing 3.28: GET /v1/packages/npm/:name/resolved

3.8.3 Get Package Version Metadata

`GET /v1/package/npm/:name@:version`

Compared to its previous form, the new version of this endpoint moves the `structure` parameter from the path to the query string, as previously suggested. This approach is chosen over content negotiation due to the caching-related issues described in Section 3.1.5. The response format does not change, except the already-deprecated `time` field is removed from file entries.

3.9 Conclusion

Changes to public API features are very difficult to make once clients start using them, which is why design is arguably the most important part of the API development process. As such, this chapter provided an exhaustive description of all necessary changes, as well as the reasoning behind them.

The final design builds upon many existing web standards and conventions to make the use of the API as simple and intuitive as possible.

Chapter **4**

Implementation

While the scope of the changes is rather big and includes 24 entirely new endpoints as well as improvements to the existing ones, many of them use similar concepts. For that reason, this chapter only highlights the most interesting parts of the implementation process.

In addition to the application changes, a significant part of the work had to be done at the database level to prepare the data in a form suitable for simple and efficient consumption.

4.1 Schemas and Validation Middleware

Because the API had only used query string parameters sparingly so far, there was no established mechanism for their validation. One of the first changes, therefore, was adding parameter schema definitions and creating a validation middleware based on Joi.

The schema definitions make use of composition, where each parameter schema is defined on its own, and the individual parameter definitions are then combined into larger logical groups, as shown in Listing 4.1.

The definitions were created with an emphasis on clear error messages for invalid values so that the messages can be directly embedded in API error responses.

```
const primitives = {
    continent: Joi.valid(...Object.keys(continents)).messages({
        '*': '{{#label}} must be a valid continent code in uppercase',
    }),
    country: Joi.valid(...Object.keys(countries)).messages({
        '*': '{{#label}} must be a valid ISO 3166-1 alpha-2 country code ...',
    }),
    limit: Joi.number().integer().positive().max(100).default(100),
    page: Joi.number().integer().positive().max(100).default(1),
    ...
};

const composedTypes = {
    paginatedStats: { limit: primitives.limit, page: primitives.page },
};

const composedSchemas = {
    location: Joi.object({
        continent: primitives.continent,
        country: primitives.country
    }).oxor('continent', 'country')
};
```

Listing 4.1: Parameter schema definitions

The final schema for each route is composed of the individual types and passed to the newly created validation middleware, as shown in Listing 4.2. The validation middleware checks that every field present in the request matches its definition, and if not, it aborts the processing and generates a user-friendly error message.

```
const routes = {
    '/stats/browsers': {
        handlers: [
            validate({
                query: Joi.object({
                    period: schema.periodStatic,
                    ...schema.paginatedStats,
                }).concat(schema.location),
            }),
            ...
        ],
    },
    ...
};
```

Listing 4.2: Route parameter validation

In addition to clear error messages, error responses generated by the middleware contain links pointing directly to the documentation page for the used endpoint, as shown in Listing 4.3. This was achieved by implementing two helpers, `getDocsPath()` and `getDocsLink()`, which automatically generate the correct documentation link based on the existing route definitions.

```
{
    "message": "Invalid parameter value: `period` must be one of [day, we...",
    "links": {
        "documentation": "/docs/data.jsdelivr.com#get-/v1/stats/packages"
    }
}
```

Listing 4.3: Error response

Lastly, the middleware uses introspection[68] to collect and expose metadata about the current route schema. It extends the route object with a list of defined schema keys, the information which of them are required, and what, if any, are their default values. This information is then utilized by the route handlers, as described in the next sections.

4.2 Period Comparison

As described in Section 3.2.3, the API now needs data from two different periods in some cases, and there were two ways to implement this. The first option was retrieving the data for the first period and then, for each record, retrieving its data from the previous period. This option would not require any database schema modifications but would result in what is known as the "N + 1 query problem" since it requires one query to retrieve the initial dataset and then one additional query for each of its records.

The second option was modifying the materialized views so that all data can be retrieved in one query. Listing 4.4 shows a simplified snippet from the procedure responsible for generating `view_top_packages` and highlights all required modifications. This approach was used in all existing and newly created views.

First, `@prevScaleFactor` is calculated to address the issue with possibly different period lengths. On lines 19–27, data for the previous period are selected and adjusted by `@prevScaleFactor`. On lines 11–14, ranks for the previous period

[68]The ability of a program to examine an object type and its properties at runtime.

are computed, and the `coalesce()` function is used to handle cases when there are no previous data for a particular package.

```sql
set @prevScaleFactor = (datediff(aDateTo, aDateFrom) + 1)
    / (datediff(aPrevDateTo, aPrevDateFrom) + 1);
...
insert into view_top_packages
select aPeriod, aDate, type, name,
    if(hits > 0, rank() over (order by hits desc), null),
    hits,
    if(bandwidth > 0, rank() over (order by bandwidth desc), null),
    bandwidth,
    if(prevHits > 0, rank() over (order by prevHits desc), null),
    coalesce(prevHits, 0),
    if(prevBandwidth > 0, rank() over (order by prevBandwidth desc), null),
    coalesce(prevBandwidth, 0)
from (
    select type, name,
        sum(hits) as hits,
        sum(bandwidth) as bandwidth,
        (select round(sum(hits) * @prevScaleFactor)
         from package_hits
         where packageId = package.id
            and date between aPrevDateFrom and aPrevDateTo
         group by packageId) as prevHits,
        (select round(sum(bandwidth) * @prevScaleFactor)
         from package_hits
         where packageId = package.id
            and date between aPrevDateFrom and aPrevDateTo
         group by packageId) as prevBandwidth
    from package
        join package_hits on package.id = package_hits.packageId
    where date between aDateFrom and aDateTo
    group by packageId
    order by hits desc
) t;
```

Listing 4.4: Period comparison implementation

4.3 Historical Data

To provide the option of querying historical data, the handling of all materialized views had to be modified. Before, their contents would be deleted once a day, and the new data for the past day, week, month, and year would be added. Of course, there would be no point in recomputing data for a fixed period, e.g., the year of 2021, after it has ended, so this process was modified. The records necessary to provide historical data are computed only once and then stored forever.

This is handled by new procedures `updateMonthlyViews()`, `updateQuarerlyViews()`, and `updateYearlyViews()`. Taking `updateMonthlyViews()` as an example, the procedure iterates over all months from January 2020 to the current date. For each month, it checks whether data for that month have already been computed, and if not, calls the procedures that compute that data and passes them the appropriate date ranges, as shown in Listing 4.5.

This approach ensures consistency and fault resistance. Should the process fail for any reason, data for the missing periods will be computed the next time the procedure is called.

Note that despite Section 3.2.2 mentioning the possibility of providing historical data for all period types, the current implementation only adds years, quarters, and months. Historical data for weeks and days were intentionally omitted for the relatively low usefulness and high storage requirements.

```sql
set @firstStart = date('2020-01-01');
set @dateFrom = date_sub(aDate, interval dayofmonth(aDate) - 1 day);

while date_sub(@dateFrom, interval 1 month) >= @firstStart
    do
        set @dateFrom = date_sub(@dateFrom, interval 1 month);
        set @dateTo = date_sub(
            date_add(@dateFrom, interval 1 month),
            interval 1 day
        );
        set @prevDateFrom = date_sub(@dateFrom, interval 1 month);
        set @prevDateTo = date_sub(@dateFrom, interval 1 day);

        if not exists(select * from view_top_packages where `date` = ...) then
            call updateViewTopPackagesForPeriod('s-month', @dateFrom, ...);
        end if;
    end while;
```

Listing 4.5: Historical data implementation

4.4 Link Builder

To facilitate the inclusion of links in API responses, the `LinkBuilder` class was created. It exposes an interface following the builder pattern [51] and, once configured, can be passed an array of resources to which the links need to be added.

The class utilizes the existing routing information for generating the URLs, the schema metadata provided by the validation middleware, and the properties of the passed resources to generate the links with as little boilerplate as possible, but the default behavior can always be customized by the exposed methods.

Listing 4.6 shows a case where links to `browsers` and `platforms` are being added, and the builder is explicitly configured to include a value for the `country` parameter and to drop the current value of the `continent` parameter if present.

```
let resourcesWithLinks = this.linkBuilder()
    .refs({
        browsers: routes['/stats/browsers'].getName(),
        platforms: routes['/stats/platforms'].getName(),
    })
    .includeQuery([ 'country' ])
    .omitQuery([ 'continent' ])
    .build(resources)
```

Listing 4.6: Link builder

While generating the links may seem like an easy task at first, the builder takes care of several issues that may not be immediately obvious.

First, some query string parameters are expected to be preserved when moving between resources, and some are not. For instance, if a client requests a list of the most popular packages in the past year and the response includes links to detailed statistics for each package, the `period=year` parameter must be preserved in all links. On the other hand, if the client requests a second page of a paginated endpoint, the `page=2` parameter is not relevant to the generated links and must be omitted. The builder handles this by comparing the current request parameters with those of the target route and keeping those that are not globally excluded (like the pagination parameters).

The second problem is that sometimes a parameter that is not present in the current request may be required in the target route. The builder handles such cases by looking at the list of required parameters in the target route and filling in the required values based on the resource properties.

Third, the parameters in internal route definitions do not always directly match the properties of public resources. This issue is addressed by the support for transform rules, which do not change the form of the final output but customize how values are read from the provided resources.

Last, there is an issue with ensuring the generated links are always in their canonical form. For example, if the optional parameter `period` has a default value of `month`, then the links `/resource` and `/resource?period=month` have the same meaning, but only one of them should be used in the generated links, regardless of whether the value was explicitly included in the request or not. The builder addresses this by checking the default value for each parameter and omitting the parameter if its value matches the default.

4.5 Pagination

Since the pagination interface is consistent across all endpoints, the implementation aimed at adding it with as little code duplication as possible. Ideally, there would be a single function that can transform any given query into a version that returns only a limited set of rows and all required pagination metadata.

The main obstacle in this was the requirement of returning the number of available resources as described in Section 3.2.5 because that means that in addition to the query that retrieves the data, a second query must be executed to determine the total count.

SQL provides a function `COUNT()` for this purpose, but the returned number is affected by `GROUP BY` aggregations, which may be used in some queries. Additionally, the original query may contain `WHERE` conditions and other clauses that impact the resulting number of rows. Overall, it may seem that while writing a separate `COUNT()` query for each original query is simple, creating an abstraction that does this automatically is not.

In the end, the goal was achieved by adding a `paginate()` method, which accepts an SQL query and the pagination options to the `BaseModel` class. To build the counting query, the original SQL query is cloned, and its `SELECT` part is replaced by `COUNT()` used as a window function.

The difference between more known aggregate functions and window functions is the set of rows on which they operate. By default, window functions operate on the entire dataset, even if a `GROUP BY` clause is used [52]. All other parts of the original query are preserved. Both queries are then executed, and the returned object includes both the records themselves and the pagination metadata, as shown in Listing 4.7.

```
static async paginate (query, limit, page) {
    ...
    if (limit) {
        // Construct the count query.
        countQuery = query.clone().select('count(*) over () as count');
        // Apply limits to the original query.
        query.limit(limit).offset((page - 1) * limit);
    }
    ...
    return { page, limit, count, pages, records };
}
```

Listing 4.7: Pagination implementation

This approach turned out to work well with most queries, but in some cases, the count query would be surprisingly slow. Analyzing these cases revealed a link to `ORDER BY` clauses in the original queries. Even though an `ORDER BY` clause has no effect on the result of the first query, its presence likely confused the database and resulted in a much slower execution plan. For this reason, the method was later updated to remove the `ORDER BY` clause from the count query.

In addition to making it very simple to adapt all queries to support pagination, this approach also made it simple to consume the metadata in the request handlers where headers need to be set. For this purpose, another method was created in the `BaseRequest` class. It accepts the metadata object exactly as returned from the `paginate()` method and sets the `Link`, `X-Total-Count`, and `X-Total-Pages` headers.

4.6 Regional Data

The *Get Network Statistics*, *Get Network Country Statistics*, and all Browser and Platform endpoints allow users to query global, continent, or country data. To avoid additional aggregations in user-facing queries, each of these options must be computed in advance.

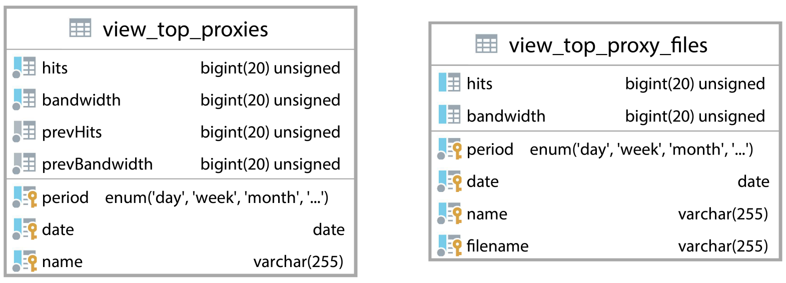
This is done by including two new fields in all affected views: a `locationType` field with the possible values of `global`, `continent`, and `country`, and a `locationId` field, which contains the respective continent code or country code.

In addition to the performance benefits, this approach also has the advantage of providing a unified interface for the application—the API does not need to maintain separate versions of queries with a different aggregation logic for each location type. Instead, it only needs a simple `SELECT` query in all cases. The logic stays hidden in the code responsible for generating the materialized views.

Note that while adding a country breakdown to any endpoint has a major impact on the number of stored records because there are currently 250 registered countries, the data duplication caused by storing separate records for global and continent data only has a negligible impact. In addition to the 250 records that need to be stored either way, only one additional record is needed to precompute the global data, and six records are needed for continent data.

4.7 Proxy Statistics

The proxy statistics required the addition of two new materialized views, as shown in Figure 4.1. The first view stores data for *Get Proxy Statistics*, and the second for *List Top Proxy Files*. At the application level, the endpoints utilize most of the newly implemented features, but otherwise, their implementation is not particularly interesting and hence not discussed further.



view_top_proxies	
hits	bigint(20) unsigned
bandwidth	bigint(20) unsigned
prevHits	bigint(20) unsigned
prevBandwidth	bigint(20) unsigned
period	enum('day', 'week', 'month', '...')
date	date
name	varchar(255)

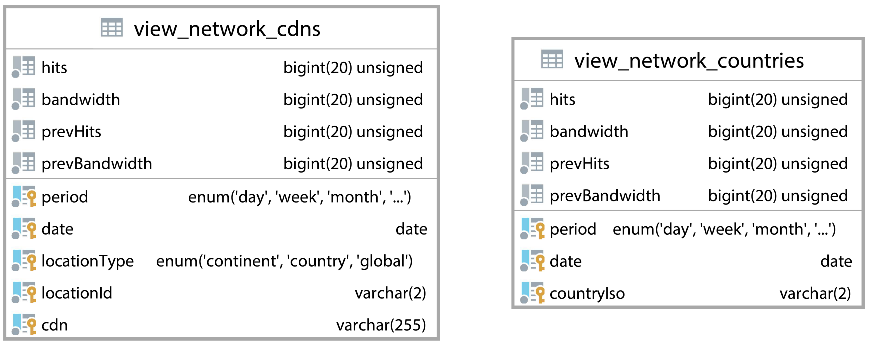
view_top_proxy_files	
hits	bigint(20) unsigned
bandwidth	bigint(20) unsigned
period	enum('day', 'week', 'month', '...')
date	date
name	varchar(255)
filename	varchar(255)

Figure 4.1: Proxy data materialized views

4.8 Network Statistics

The network statistics required the addition of two new materialized views, as shown in Figure 4.2. The first view stores data for *Get Network Statistics*. Based on the desired API interface, the view needs to store one record for each combination of the period type, period value, and provider. Additionally, because the endpoint allows filtering for continents and countries, the view must include location fields as described in Section 4.6.

The second view stores data for *Get Network Country Statistics*, which only requires request and bandwidth totals for each country.



view_network_cdns	
hits	bigint(20) unsigned
bandwidth	bigint(20) unsigned
prevHits	bigint(20) unsigned
prevBandwidth	bigint(20) unsigned
period	enum('day', 'week', 'month', '...')
date	date
locationType	enum('continent', 'country', 'global')
locationId	varchar(2)
cdn	varchar(255)

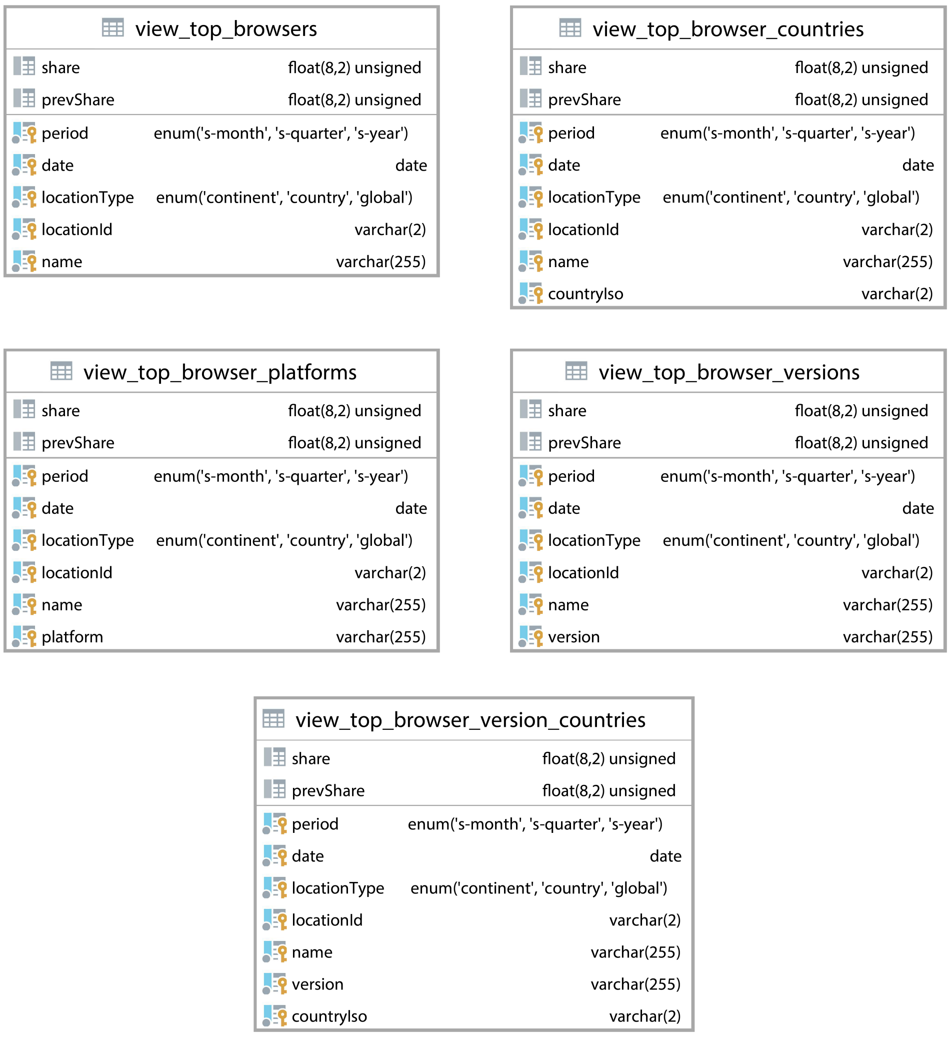
view_network_countries	
hits	bigint(20) unsigned
bandwidth	bigint(20) unsigned
prevHits	bigint(20) unsigned
prevBandwidth	bigint(20) unsigned
period	enum('day', 'week', 'month', '...')
date	date
countryIso	varchar(2)

Figure 4.2: Network data materialized views

4.9 Browser and Platform Statistics

The 12 new browser and platform endpoints required the addition of 10 materialized views, and because most of the views may contain data with different aggregation types (continent, country, and global), there are overall 26 separate SQL queries used to generate them.

While the queries themselves are very complex, the structure of the views is rather simple, as only `share` and `prevShare` values are stored for each combination of parameters. The browser-related views are shown in Figure 4.3, with the platform-related ones being similar.



view_top_browsers

share	float(8,2) unsigned
prevShare	float(8,2) unsigned
period	enum('s-month', 's-quarter', 's-year')
date	date
locationType	enum('continent', 'country', 'global')
locationId	varchar(2)
name	varchar(255)

view_top_browser_countries

share	float(8,2) unsigned
prevShare	float(8,2) unsigned
period	enum('s-month', 's-quarter', 's-year')
date	date
locationType	enum('continent', 'country', 'global')
locationId	varchar(2)
name	varchar(255)
countryIso	varchar(2)

view_top_browser_platforms

share	float(8,2) unsigned
prevShare	float(8,2) unsigned
period	enum('s-month', 's-quarter', 's-year')
date	date
locationType	enum('continent', 'country', 'global')
locationId	varchar(2)
name	varchar(255)
platform	varchar(255)

view_top_browser_versions

share	float(8,2) unsigned
prevShare	float(8,2) unsigned
period	enum('s-month', 's-quarter', 's-year')
date	date
locationType	enum('continent', 'country', 'global')
locationId	varchar(2)
name	varchar(255)
version	varchar(255)

view_top_browser_version_countries

share	float(8,2) unsigned
prevShare	float(8,2) unsigned
period	enum('s-month', 's-quarter', 's-year')
date	date
locationType	enum('continent', 'country', 'global')
locationId	varchar(2)
name	varchar(255)
version	varchar(255)
countryIso	varchar(2)

Figure 4.3: Browser data materialized views

When retrieving data from the views, the initial implementation sorted the records in descending order by `share` and then in ascending order by `name` in case two records had exactly the same `share`. However, this resulted in suboptimal performance noticed during testing as MariaDB had not supported the use of indexes for `ORDER BY` with mixed `ASC` and `DESC` modifiers until the version 10.8.1 release [53].

jsDelivr currently uses version 10.5.15, so the affected queries were changed to use descending order for all fields, which made a functionally negligible difference but resulted in a significant performance improvement.

4.10 Deprecations Handling

Since there are now several endpoints that were deprecated, there is also a question of how to inform the current users about this. Even though the endpoints are not going to be removed, and users do not need to switch to the new ones if they do not want to, they may be interested in the new features. Given the project specifics described in Section 1.3.1, this is particularly hard to do via standard channels such as e-mail.

However, there are several interesting attempts at standardizing the communication of deprecations and removals directly on the HTTP level. First, there is RFC 8594, which defines a `Sunset` header as a way to indicate that a URI is likely to become unresponsive in the future [54].

There is also a work-in-progress Internet-Draft that aims to establish a new `Deprecation` response header and a `deprecation` relation type as a way to signal that a resource has been deprecated and to provide additional context [55]. Although not a final standard, some HTTP clients already recognize this header and use it to display warnings directly in the application logs [56].

The beauty of this approach lies in its genericity. A client that understands this header can automatically monitor all responses it receives, regardless of what service they come from, and log warnings the same way it logs its own code deprecations warnings as soon as the remote service starts sending them.

The last piece of the puzzle is RFC 5829, defining several relation types including `successor-version`, which is meant to provide a link to the successor version of the requested resource [57].

To make use of the listed standards, a new `deprecate()` method was added to the `BaseRequest` class. Any endpoint can be marked as deprecated by simply calling the method, which:

- sets the `Deprecation` header,
- sets the `deprecation` link pointing to the API documentation,
- sets the `successor-version` link pointing directly to the new endpoint.

The documentation link does not simply point to the homepage but directly to the correct new endpoint, thanks to the helpers implemented in Section 4.1

Similarly, the `successor-version` link does not simply link to the base URL for the resource—it uses the existing link builder capabilities to preserve all relevant query string parameters and even remaps them in case their names change between versions.

Listing 4.8 shows the response headers for `/v1/package/resolve/npm/jquery@3` with the `version` parameter being remapped from the path to the query string as `specifier`.

```
Deprecation: Sun, 01 Jan 2023 00:00:00 GMT
Link: <https://www.jsdelivr.com/docs/data.jsdelivr.com
        #get-/v1/packages/npm/-package-/resolved>;
        rel="deprecation",
      <https://data.jsdelivr.com/v1/packages/npm/jquery/resolved?specifier=3>;
        rel="successor-version"
```

Listing 4.8: Deprecation headers

4.11 Conclusion

This chapter described the selected parts of the implementation process. At the application level, the implementation focused on using abstractions and creating reusable units where possible, such as the validation middleware and the Link Builder.

A considerable amount of work was done at the database level, where the browser and platform statistics endpoints alone required almost 1000 lines of SQL code to prepare the data, and that was just one part of the new features. Additionally, all implementation decisions considered the related performance aspects.

Chapter 5

Testing

To ensure all features work as expected and avoid regressions, jsDelivr relies on automated tests that run after each commit. The majority of the tests interact with the application like a real client would by sending HTTP requests and checking the responses.

These tests use real MariaDB and Redis instances with a dataset prepared during the test setup. To guarantee reproducible results, all requests to external services such as GitHub and npm are intercepted and mocked[69] by Nock[70].

The testing dataset consists of dynamically generated records designed so that it covers all expected edge cases while staying as compact as possible.

5.1 Snapshot Tests

Unlike other test types, which are typically based on assertions about the expected behavior, snapshot tests work by storing the real output the first time they run and comparing it to the output of the subsequent runs. That means they do not necessarily check that the output is correct but rather that it does not change after the first run.

[69] A testing technique of replacing real dependencies with objects that mimic the dependency behavior in a controlled way.

[70] `https://github.com/nock/nock`

The advantage of this approach is that checking the generated snapshot is correct when adding a new test is often easier than manually writing the assertions. For instance, if the expected output is a list with ten items and each item has several properties, writing assertions for each property of each item would be tedious.

Instead, a typical test would only check the key characteristics, e.g., that the number of items is correct, that each item has the required properties, and maybe that the property values of the first item are correct. A snapshot test, on the other hand, would automatically store the full list of items with all properties, allowing it to catch even the smallest differences in the future. The list would only have to be checked—but not written—manually.

The approach can also be combined with traditional assertions so that the essential characteristics of the output are checked explicitly, and the exact match with the snapshot is checked as the last step. This combination reduces the chance of accidentally accepting an initial snapshot with incorrect output while still providing the strength of checking all properties.

Some of the existing API tests had already used a similar technique—the expected response bodies were loaded from a manually-prepared file, and an exact match of the response was checked. Although this had provided the same results in terms of test coverage, this approach was missing out on automating the management of expected outputs.

For example, Jest[71], Meta's open-source testing framework, has built-in support for snapshot testing. It can automatically store the expected output on the first run, load and compare it to the outputs of the next runs, and update it when needed [58].

Mocha and Chai do not offer this functionality but offer a plugin system that allows for its implementation [30, 31]. In order to stay consistent with the existing tests and other jsDelivr projects, they were therefore not replaced by Jest but rather extended with similar features.

[71]`https://jestjs.io`

A newly created Chai plugin extends it with a `matchSnapshot()` method, which can be called on any HTTP response object, and Mocha hooks are used to integrate actions that need to run at the start and at the end of the test suite. The difference between writing tests using the old approach and using the new plugin is demonstrated in Listing 5.1.

```
- const expected = require('../../data/v1/expected/packages');
-
  it('GET /v1/packages/npm/jquery', async () => {
      let response = await chai.request(host).get('/v1/packages/npm/jquery');

      expect(response).to.have.status(200);
-     expect(response.body).to.deep.equal(expected['/v1/package/npm/jquery']);
+     expect(response).to.matchSnapshot();
  });
```

Listing 5.1: Usage of the Chai snapshotting plugin

The plugin takes care of storing, loading, and comparing the data and exposes several switches to configure its behavior:

- `snapshotResponses` specifies whether a snapshot should be automatically recorded in case a test does not have one. Turning this option off means tests without a stored snapshot will fail.
- `updateExistingSnapshots` specifies the behavior on an output mismatch. Either the existing snapshot can be updated (in case changes are expected), or an error can be thrown.
- `pruneOldSnapshots` can be used to delete existing snapshots that are no longer used in any tests.

5.1.1 Time Handling

A downside of snapshot tests is that they do not work well if part of the output is expected to change over time, which is the case with some of the statistics endpoints. Their responses include dates relative to the current date and, accordingly, change every day.

In the existing tests, this problem was solved by calculating the offset between the date when the response was stored and the current date and shifting all dates in the response by this offset.

Support for this offset-based shifting was included in the new snapshotting plugin as well, but eventually, this problem was remedied entirely by using fake timers from Sinon.JS[72], which override the process time with a static value set in the tests.

5.2 Parameterized Tests

While snapshot tests remove the need to manually write assertions, parameterized tests further simplify the creation of new tests by reusing a single test for multiple inputs. Traditionally, the inputs for parameterized tests still need to be provided manually. The approach described in this section goes one step further and partially automates this part too.

For example, the *List Top Packages* endpoint has the following parameters:

- `by` with two possible values,
- `type` with two possible values,
- `period` with eight possible values (not counting historical periods),
- `limit` with 100 possible values,
- `page` with 100 possible values.

Additionally, all the parameters are optional, which adds another possible `undefined` value for each of them. Even if we disregard the `limit` and `page` parameters, as it is pointless to test all pagination values, there are 72 combinations of `by`, `type`, and `period`, each of which should be covered by at least one test to ensure it works correctly.

One could argue that not all parameters necessarily depend on each other and that testing all combinations is overly extensive, but the understanding of which parameters are independent and which interact with each other is heavily implementation dependent and could change in the future. The goal of the described approach was to create a robust test suite that could detect the widest scope of issues with a reasonable effort.

[72] `https://sinonjs.org`

Similarly, it could be assumed that if a certain feature is included in several endpoints, it does not need to be tested on each endpoint separately. For instance, the pagination implementation is shared by all endpoints, so it could only be tested once. This, again, has the problem of assuming the implementation details—some specific endpoint might need to internally use a different pagination method, or it might simply forget to call the correct method or declare the pagination parameters in its route definition.

For these reasons, the new tests utilize multiple newly created helpers that, based on the endpoint definition, are able to generate a list of all possible parameter combinations and create a snapshot test for each of them. Then, the developer can review all generated outputs and be sure that all options work correctly.

Listing 5.2 shows how the tests would be created in the mentioned *List Top Packages* endpoint example using the first helper. The first parameter of the function is the endpoint URI template in the format defined by RFC 6570 [50]. The third parameter lists the values to test for each parameter. The function generates a cartesian product of all provided parameter values and runs a test for each combination.

The second parameter provides extra support for optional parameters. It specifies the expected default values, which allows the testing code to compare the cases when the parameter is omitted and when the default value is set explicitly. It also allows for reducing the number of generated snapshots since the output for both cases is the same.

```
makeEndpointSnapshotTests('/v1/stats/packages{?by,type,period}', {
    by: 'hits',
    period: 'month',
}, [
    {
        by: [ 'hits', 'bandwidth', undefined ],
        type: [ 'gh', 'npm', undefined ],
    },
]);
```

Listing 5.2: Parameterized snapshot testing

The second helper is used to test the pagination functionality on each endpoint. By calling it as shown in Listing 5.3, the function internally generates a number of requests with different `limit` and `page` parameters, and by merely comparing their outputs, it checks whether the parameters are correctly interpreted in all cases. This approach does not require storing any snapshots or writing endpoint-specific code.

Moreover, it checks not only the response bodies but also all pagination metadata returned in the headers. Thanks to this generic implementation, each endpoint can be tested with no extra work. In fact, this did help to catch several bugs already, e.g., wrong pagination metadata for one endpoint due to the underlying SQL query breaking the row-counting mechanism.

```
makeEndpointPaginationTests('/v1/stats/packages');
```

Listing 5.3: Parameterized pagination testing

For cases where manual assertions need to be written, the third helper is used to reduce the amount of boilerplate by only providing the inputs and the relevant assertions instead of writing the logic for making the HTTP requests and providing the URLs in each test, as shown in Listing 5.4.

```
makeEndpointAssertions('/v1/stats/browsers{?continent,country,period}', {}, [
    {
        params: { period: '2020-04' },
        assert: (response) => {
            expect(_.sumBy(response.body, 'share')).to.be.closeTo(100);
            expect(_.sumBy(response.body, 'prev.share')).to.be.closeTo(88.52);
            expect(response.body).to.have.lengthOf(13);
            expect(response).to.matchSnapshot();
        },
    },
    {
        params: { period: '2021', continent: 'EU' },
        assert: (response) => {
            expect(_.sumBy(response.body, 'share')).to.be.closeTo(100);
            expect(_.sumBy(response.body, 'prev.share')).to.be.closeTo(2.63);
            expect(response.body).to.have.lengthOf(7);
            expect(response).to.matchSnapshot();
        },
    },
]);
```

Listing 5.4: Parameterized testing with explicit assertions

5.3 Speeding Up the Database Setup

Due to the number of new features and the respective tests, the time required to set up the database on a development machine, which happens every time the tests run, went from approximately three seconds to approximately 33 seconds.

To limit the impact on the development process, the setup was modified so that it only runs if any changes to the relevant files have been made. This is achieved by storing a hash of all seed files in the database. If the hash does not change, the database is not rebuilt.

Additionally, it was observed that the setup time has become greatly inconsistent between runs. In some cases, it would take two or three times longer for no obvious reason. This behavior turned out to be caused by poor database index statistics.

The index statistics hold information about the distribution of values in the tables and are used by the query optimizer to find the best way of executing each query. The statistics may be automatically recalculated by the database engine when a significant part of the data changes, but this happens in the background, with some delay [59]. To mitigate this issue, the setup now executes `analyze table` on each table after inserting the data to force the calculation of new statistics.

5.4 Conclusion

This chapter described the selected testing approach focused on automating not only the execution of the tests but also their creation, allowing new tests to be added with minimal effort. Support for snapshot testing has been implemented as a Chai plugin, allowing it to be later reused in other projects or published as a standalone module.

Before any of the changes, there were, in total, 215 test cases, providing an 84.55 % line code coverage. After implementing the new features and adding all tests, there are now 1375 test cases, 311 of which are standalone tests, and 1064 are generated from 126 unique templates. The line code coverage remained almost the same at 84.56 %.

The extensive test suite should provide a high level of confidence that existing features will continue working when making changes in the future, and it has already helped to catch several issues during development.

Chapter 6

Documentation

An API is only useful if well documented, which is why the last chapter of this work describes the documentation process. Due to Non-functional Requirement 02, which requires that all features be described in an OpenAPI document, this process included not only the new but all features.

To provide a better developer experience and avoid duplication, the existing Markdown documentation has been replaced with an interactive HTML version generated from the OpenAPI document.

6.1 OpenAPI

The structure of OpenAPI documents is defined in the OpenAPI Specification. The main part of the document is the description of API paths, with each path corresponding to a single endpoint. Each path may support multiple operations, which correspond to different HTTP methods, and for each operation, there can be a description, schema definitions, and examples. Additionally, the document may also contain general information about the API, such as contacts, server URLs, a link to the terms of service, or license information [33].

Once created, the OpenAPI document can be used by various tools to generate human-friendly documentation and client code in supported languages or drive automated testing. A handy overview of the existing tools is available at

OpenAPI.Tools[73].

6.1.1 Automated Document Generation

A specific category of OpenAPI tools are generators capable of creating an OpenAPI document based on the existing code, which is particularly useful for already existing APIs. The OpenAPI.Tools website lists two such tools, but neither of them supports the latest v3.1 specification [60]. Nevertheless, an attempt was made to at least partially automate the creation process using the har2openapi[74] tool.

The tool works in multiple steps, the first of which is creating HTTP Archive (HAR) files, which serve as the source of document data [61]. The HAR files contain records of previously made HTTP requests and responses, from which the tool extracts information about the available endpoints, their parameters, and response formats.

The easiest way to obtain the HAR files was to capture the requests made in the existing automated tests. For this, yet another package, Express HAR capture[75], was used. As the name suggests, the package was originally created for the Express framework, not Koa, and its documentation warns that it is “very alpha” [62]. Hence, a few patches were needed before the HAR files could be successfully generated. The patched version[76] is available on GitHub.

In the second step, har2openapi needs to be configured so that it can correctly parse the request paths into endpoint URI templates. This is done using user-provided regular expressions. If done manually, this step would require a fair amount of work, but in fact, it can be somewhat automated too.

Matching request URIs with the correct endpoints is exactly what Koa-router does when handling the requests. As such, it already has all the necessary information. Unfortunately, the information is not directly available via the router’s public interface [63], but it can still be extracted from its internal data.

[73] `https://openapi.tools`
[74] `https://github.com/dcarr178/har2openapi`
[75] `https://github.com/idoco/node-express-har-capture`
[76] `https://github.com/MartinKolarik/node-express-har-capture/tree/npm-patch`

For this purpose, a code snippet[77] that reads the router data and produces a JSON file that can be passed to har2openapi has been created.

After the second step, the generated document includes the list of detected paths, their parameters, and example values gathered from the requests and needs to be further edited manually.

6.1.2 Manual Document Improvements

In order to fully replace regular documentation, the following needs to be added to the generated document:

- A description of each operation.
- A description, schema definition, and example values of each parameter (the example values may be partially prefilled by har2openapi, but their quality depends on the testing data, and some editing is needed).
- A description, schema definition, and example values of each possible response type for each operation (again, the examples may be partially prefilled but require editing to provide useful information while staying concise).
- General API information, as described at the beginning of Section 6.1.
- Tags to categorize the paths into logical groups.

The schema definitions, in particular, can be rather time-consuming to create, but once done, they can be used to generate strongly-typed API clients. As such, great care was taken to precisely describe all response schemas.

6.2 Documentation Portal

With the final OpenAPI document prepared, the next step was creating a documentation portal, which would list all endpoints and their details in a clear way and allow users to directly edit the examples and send real requests to the API.

[77]https://gist.github.com/MartinKolarik/aaeb0149f114a8e7417c961b81bc0c32

The OpenAPI.Tools website lists a wide selection of 30 tools for this use case. However, only 13 of them support the latest OpenAPI v3.1 standard [60]. These 13 tools were evaluated with priority on:

- the clarity of the user interface, particularly of the schema definitions and response examples,
- customization support (not only theme changes but also the ability to disable individual features or modify the page layout),
- the ability to self-host the generated documentation, as some of the tools were software-as-a-service solutions,
- being free, at least for open-source use.

On the other hand, the following features were not considered particularly relevant:

- versioning management,
- cloud-based collaboration features,
- automatically generated code samples,
- GUI-based editing of the OpenAPI document.

Based on these priorities, RapiDoc[78] was selected as the best option. Unlike most of the other tools, it provides three different layout options, two different rendering styles for schemas, and is overall highly customizable. Moreover, because it is developed as open-source software, any customizations that cannot be done via the standard configuration options can be implemented by modifying it [64].

6.2.1 RapiDoc Customizations

In order to be integrated into the jsDelivr website[79], RapiDoc was first styled to match the website design, using a combination of the available styling options and custom CSS.

[78]`https://rapidocweb.com`
[79]`https://www.jsdelivr.com`

The selected layout did not turn out to work well on smaller devices [65], so extra logic was added to switch between a `read` layout, which is more suitable for large screens, and a `view` layout, which works better on small screens.

Despite claiming support for the latest OpenAPI v3.1 standard, at the time of creating the portal, RapiDoc did not support `const` fields within schema definitions [66]. The support for these fields was implemented during the customization process with the intention of sending a pull request to the project. However, before the pull request was made, this feature had been added to RapiDoc by one of the project maintainers.

As a last step, a new `x-labels` vendor extension has been added. Vendor extensions are additional OpenAPI document fields that the tools may use to provide extra features [33, §4.9]. In this case, the field is used by the customized RapiDoc version to show endpoint labels in the sidebar, which can be used to highlight new API features.

6.3 Conclusion

This chapter showed that when creating an OpenAPI document for an existing API, several parts of the process can be automated. Nevertheless, a fair amount of manual editing is still needed for the best results.

The created document can be used to generate interactive documentation and significantly simplifies integration with other systems. The created documentation portal, which has been made available as a part of the jsDelivr website[80], provides a listing of all 46 API endpoints with the ability to filter them based on the name and description.

For each endpoint, there is a list of supported parameters with descriptions and example values, a response schema definition, and at least one example. Developers can further experiment with the API by editing the examples, sending real requests, and examining the responses.

[80] `https://www.jsdelivr.com/docs/data.jsdelivr.com`

Conclusion

The thesis started by exploring the target domain and offered a thorough analysis of the individual user groups and their requirements. The requirements were further set out through a set of use cases and user stories and served as a foundation for the following review of the existing services and design of new jsDelivr API features. Although not directly required by the original assignment, the first chapter also outlined how the major public CDN services have evolved over time.

The second chapter continued by examining the current jsDelivr API architecture and used technologies, as well as the design and implementation choices. It showed that all of these areas were considerably influenced by the scale of the service and the amount of data it works with.

The third chapter focused on designing new jsDelivr API features in a way that would best match the defined requirements. It discussed not only the final design but all the considered approaches and the reasoning behind the made decisions. The design focused on providing an efficient yet extensible interface that builds upon a number of existing standards.

Due to the comprehensive design phase, the implementation process described in the fourth chapter was mostly straightforward, albeit extensive. The API has been extended with a total of 24 new endpoints, and another 19 have been modified. The new statistics features are supported by a total of 23 complex

SQL procedures that prepare the required data. Additionally, several reusable components have been created to simplify the addition of individual features.

Given the vast scope of changes, care needed to be taken to avoid regressions in the existing features and to ensure the new features work as designed. The testing setup has been enhanced to support rapid creation of new tests through automation, as discussed in the fifth chapter, and the existing features had been covered with additional tests before the implementation of the new features began. Each new feature has then been covered with a robust test suite as well. Overall, more than 1000 tests have been added. The Chai snapshotting plugin and the additional helpers developed to support this approach have been designed to be reusable and can be used in future projects as well.

The last chapter talked about creating an OpenAPI document to ease the API integration in other projects and an interactive documentation portal integrated into the jsDelivr website. Focus has, again, been given to automating several parts of the process. The resulting documentation portal provides an exhaustive description of the available endpoints, their parameters, and possible responses.

At last, it can be concluded that the goals of the thesis were fully met. The text itself provides a unique insight into the problem domain and discusses many practical problems relevant to the design and development of web APIs in general. The newly implemented features, as well as the documentation portal, have been deployed to production and are already used by real users with millions of requests every day.

Bibliography

[1] ARLOW, Jim; NEUSTADT, Ila. *UML2 and the Unified Process.* 2nd edition. Addison-Wesley Professional, 2005-01. ISBN 0-321-32127-8.

[2] JEFFRIES, Ron; ANDERSON, Ann; HENDRICKSON, Chet. *Extreme Programming Installed.* Addison-Wesley Professional, 2001-06. ISBN 0-201-70842-6.

[3] Usage statistics of JavaScript content delivery networks for websites. In: *W3Techs – World Wide Web Technology Surveys* [online]. Q-Success, 2022 [visited on 2022-11-15]. Available from: `https://w3techs.com/technologies/overview/content_delivery`.

[4] Hosted Libraries. In: *Google Developers* [online]. Google LLC, 2022 [visited on 2022-11-15]. Available from: `https://developers.google.com/speed/libraries`.

[5] DORFMAN, Justin. Handling 15,000 CDN Requests/Sec: The Growth Behind jQuery. In: *MaxCDN Blog* [online]. MaxCDN, 2015-06-20. [visited on 2022-11-15]. Originally available from: `https://www.maxcdn.com/blog/maxscale-jquery`. Archived at: `https://web.archive.org/web/20150906182244/https://www.maxcdn.com/blog/maxscale-jquery`.

[6] *cdnjs.com – the missing cdn* [online]. cdnjs, 2011. [visited on 2022-11-15]. Originally available from: `http://www.cdnjs.com`. Archived at: `https://web.archive.org/web/20110125232824/http://www.cdnjs.com`.

[7] HELLO, Peter Dave et al. cdnjs README. In: *GitHub* [online]. 2015 [visited on 2022-11-15]. Available from: `https://github.com/cdnjs/cdnjs/tree/2016`.

[8] *No more manual pull request for lib updating* [online]. cdnjs GitHub issues, 2014 [visited on 2022-11-15]. Available from: `https://github.com/cdnjs/cdnjs/issues/3638`.

[9] API Documentation. In: *cdnjs* [online]. cdnjs, 2022 [visited on 2022-11-15]. Available from: `https://cdnjs.com/api`.

[10] JSON parsing. In: *Can I use... Support tables* [online]. caniuse, 2022 [visited on 2022-11-30]. Available from: `https://caniuse.com/json`.

[11] JACKSON, Michael. *First, http://npmcdn.com IS MOVING to http://unpkg.com...* [tweet]. Twitter, 2016-08-30. [visited on 2022-11-15]. Available from: `https://twitter.com/mjackson/status/770424625754939394`.

[12] AKULOV, Dmitriy; NYMAN, Robert. jsDelivr – The advanced open source public CDN. In: *Mozilla Hacks* [online]. Mozilla Corporation, 2014-03-19 [visited on 2022-11-16]. Available from: `https://hacks.mozilla.org/2014/03/jsdelivr-the-advanced-open-source-public-cdn`.

[13] AKULOV, Dmitriy. Public CDN auto-updated. In: *jsDelivr Blog* [online]. Prospect One, 2014-06-28 [visited on 2022-11-16]. Available from: `https://www.jsdelivr.com/blog/public-cdn-auto-updated`.

[14] AKULOV, Dmitriy et al. Public CDNs API README. In: *GitHub* [online]. 2017 [visited on 2022-11-16]. Available from: `https://github.com/jsdelivr/api`.

[15] *Future of jsDelivr* [online]. jsDelivr GitHub issues, 2016 [visited on 2022-11-16]. Available from: `https://github.com/jsdelivr/jsdelivr/issues/13136`.

[16] AKULOV, Dmitriy. jsDelivr reloaded 2017. In: *jsDelivr Blog* [online]. Prospect One, 2017-09-12 [visited on 2022-11-16]. Available from: `https://www.jsdelivr.com/blog/jsdelivr-reloaded-2017`.

[17] KOLÁRIK, Martin et al. jsDelivr API README. In: *GitHub* [online]. 2022 [visited on 2022-12-01]. Available from: `https://github.com/jsdelivr/data.jsdelivr.com`.

[18] KOLÁRIK, Martin et al. jsDelivr CDN README. In: *GitHub* [online]. 2022 [visited on 2022-12-09]. Available from: `https://github.com/jsdelivr/jsdelivr`.

[19] Our sponsors. In: *jsDelivr – A free, fast, and reliable CDN for Open Source* [online]. Prospect One, 2022 [visited on 2022-12-09]. Available from: `https://www.jsdelivr.com/sponsors`.

[20] Our network. In: *jsDelivr – A free, fast, and reliable CDN for Open Source* [online]. Prospect One, 2022 [visited on 2022-12-09]. Available from: `https://www.jsdelivr.com/network`.

[21] AKULOV, Dmitriy. *jsDelivr API statistics* [personal communication]. 2022-12-10.

[22] FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, 2000. PhD thesis. University of California. Available also from: `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`.

[23] RICHARDSON, Leonard. Justice Will Take Us Millions Of Intricate Moves. In: *Crummy* [online]. 2008-11-20 [visited on 2022-12-06]. Available from: `https://www.crummy.com/writing/speaking/2008-QCon`.

[24] About. In: *Node.js* [online]. OpenJS Foundation, 2022 [visited on 2022-12-10]. Available from: `https://nodejs.org/en/about`.

[25] *Koa – next generation web framework for node.js* [online]. Koa, 2022. [visited on 2022-12-10]. Available from: `https://koajs.com`.

[26] About MariaDB Server. In: *MariaDB Server: The open source relational database* [online]. MariaDB Foundation, 2022 [visited on 2022-12-10]. Available from: `https://mariadb.org/about`.

[27] Introduction to Redis. In: *Redis: A vibrant, open source database* [online]. Redis Ltd., 2022 [visited on 2022-12-10]. Available from: `https://redis.io/docs/about`.

[28] Guide. In: *Knex.js – SQL Query Builder for JavaScript* [online]. Knex, 2022 [visited on 2022-12-10]. Available from: `https://knexjs.org/guide`.

[29] API Reference. In: *joi.dev* [online]. Sideway Inc., 2022 [visited on 2022-12-10]. Available from: `https://joi.dev/api`.

[30] *Mocha – the fun, simple, flexible JavaScript test framework* [online]. OpenJS Foundation, 2022. [visited on 2022-12-20]. Available from: `https://mochajs.org`.

[31] *Chai Assertion Library* [online]. Chai, 2022. [visited on 2022-12-20]. Available from: `https://www.chaijs.com`.

[32] Application Performance Monitoring (APM) with Elastic Observability. In: *Elastic* [online]. Elasticsearch B.V., 2022 [visited on 2022-12-20]. Available from: `https://www.elastic.co/observability/application-performance-monitoring`.

[33] MILLER, Darrel et al. (editors). *OpenAPI Specification* [online]. 2021-02-15. Version 3.1.0 [visited on 2023-01-10]. The Linux Foundation. Available from: `https://spec.openapis.org/oas/v3.1.0`.

[34] MARCHÁN, Kat. The reason for this is that, by setting mtimes like this... In: *Use a specific mtime when packing, rather than none at all* [online]. 2018-03-13 [visited on 2022-12-20]. Available from: `https://github.com/npm/npm/pull/20027#discussion_r173985677`.

[35] ISO 3166-1:2020. *Codes for the representation of names of countries and their subdivisions – Part 1: Country code.* 4th edition. Geneva, Switzerland. International Organization for Standardization, 2020-08.

[36] STURGEON, Philip. *Build APIs You Won't Hate.* Leanpub, 2015-08-12.

[37] FIELDING, Roy Thomas. *The reason to make a real REST API is to get evolvability...* [tweet]. Twitter, 2013-09-09. [visited on 2022-12-20]. Available from: `https://twitter.com/fielding/status/376835835670167552`.

[38] *Fetch: Living Standard* [online]. 2022-12-21. [visited on 2022-12-22]. WHATWG. Available from: `https://fetch.spec.whatwg.org`.

[39] MULHUIJZEN, Rogier. Best practices for using the Vary header. In: *Fastly Blog* [online]. Fastly, Inc., 2014-08-18 [visited on 2022-12-20]. Available from: `https://www.fastly.com/blog/best-practices-using-vary-header`.

[40] Accept. In: *Fastly Developer Hub* [online]. Fastly, Inc., 2022 [visited on 2022-12-20]. Available from: `https://developer.fastly.com/reference/http/http-headers/Accept`.

[41] STURGEON, Philip. API Evolution for REST/HTTP APIs. In: *APIs you won't hate* [online]. 2018-05-02 [visited on 2022-12-22]. Available from: `https://apisyouwonthate.com/blog/api-evolution-for-rest-http-apis`.

[42] NOTTINGHAM, Mark. Evolving HTTP APIs. In: *Mark Nottingham's Blog* [online]. 2012-12-04 [visited on 2022-12-22]. Available from: `https://www.mnot.net/blog/2012/12/04/api-evolution.html`.

[43] ISO 8601:2004. *Data elements and interchange formats – Information interchange – Representation of dates and times.* 3rd edition. Geneva, Switzerland. International Organization for Standardization, 2004-12.

[44] GUPTA, Lokesh. HATEOAS Driven REST APIs. In: *REST API Tutorial* [online]. 2022-12-30 [visited on 2023-01-15]. Available from: `https://restfulapi.net/hateoas`.

[45] NOTTINGHAM, Mark. RFC 8288. *Web Linking.* 2017-10. RFC Editor. ISSN 2070-1721. Available from DOI: `10.17487/RFC8288`.

[46] Link Relations. In: *Protocol Registries* [online]. 2022-11-23 [visited on 2023-01-17]. Available from: `https://www.iana.org/assignments/link-relations/link-relations.xhtml`.

[47] SHARMA, Tushar. It's 2020 and We're Still Doing Pagination The Wrong Way. In: *tusharf5.com* [online]. 2020 [visited on 2023-01-15]. Available from: `https://tusharf5.com/posts/api-pagination-the-right-way`.

[48] *HTML: Living Standard* [online]. 2023-01-16. [visited on 2023-01-17]. WHATWG. Available from: `https://html.spec.whatwg.org`.

[49] NOTTINGHAM, Mark. RFC 5988. *Web Linking.* 2010-10. RFC Editor. ISSN 2070-1721. Available from DOI: 10.17487/RFC5988.

[50] GREGORIO, Joe et al. RFC 6570. *URI Template.* 2012-03. RFC Editor. ISSN 2070-1721. Available from DOI: 10.17487/RFC6570.

[51] GAMMA, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994-11. ISBN 0-201-63361-2.

[52] Window Functions Overview. In: *MariaDB Knowledge Base* [online]. MariaDB Foundation, 2023 [visited on 2023-01-25]. Available from: https://mariadb.com/kb/en/window-functions-overview.

[53] *Implement descending index: KEY (a DESC, b ASC)* [online]. MariaDB Jira, 2022 [visited on 2023-01-28]. Available from: https://jira.mariadb.org/browse/MDEV-13756.

[54] WILDE, Erik. RFC 8594. *The Sunset HTTP Header Field.* 2019-05. RFC Editor. ISSN 2070-1721. Available from DOI: 10.17487/RFC8594.

[55] DALAL, Sanjay; WILDE, Erik. *The Deprecation HTTP Header Field.* 2021-07-10. Version 2. Internet-Draft. Internet Engineering Task Force. Available also from: https://datatracker.ietf.org/doc/draft-ietf-httpapi-deprecation-header.

[56] POT, Evert. Ketting Wiki: Deprecation Warnings. In: *GitHub* [online]. 2021-01-31 [visited on 2023-01-25]. Available from: https://github.com/badgateway/ketting/wiki/Deprecation-Warnings.

[57] BROWN, Al; CLEMM, Geoffrey. RFC 5829. *Link Relation Types for Simple Version Navigation between Web Resources.* 2010-04. RFC Editor. ISSN 2070-1721. Available from DOI: 10.17487/RFC5829.

[58] BEKKHUS, Simen et al. Snapshot Testing. In: *Jest – Delightful JavaScript Testing* [online]. 2021-01-24 [visited on 2023-02-01]. Available from: https://jestjs.io/docs/snapshot-testing.

[59] Index Statistics. In: *MariaDB Knowledge Base* [online]. MariaDB Foundation, 2023 [visited on 2023-02-08]. Available from: https://mariadb.com/kb/en/index-statistics.

[60] *OpenAPI.Tools – an Open Source list of great tools for Open API* [online]. APIs You Won't Hate, 2023. [visited on 2023-02-10]. Available from: `https://openapi.tools`.

[61] CARR, David. har2openapi README. In: *GitHub* [online]. 2023 [visited on 2023-02-10]. Available from: `https://github.com/dcarr178/har2openapi`.

[62] SIEBUHR, Morten. Express HAR capture README. In: *GitHub* [online]. 2023 [visited on 2023-02-10]. Available from: `https://github.com/idoco/node-express-har-capture`.

[63] WHITBECK, Tim et al. Koa-router API Reference. In: *GitHub* [online]. 2023 [visited on 2023-02-10]. Available from: `https://github.com/koajs/router/blob/master/API.md`.

[64] *RapiDoc – Web Component based Swagger & OpenAPI Spec Viewer* [online]. 2023. [visited on 2023-02-10]. Available from: `https://rapidocweb.com`.

[65] *Menu Missing on Responsive Narrow Layout* [online]. RapiDoc GitHub issues, 2021 [visited on 2023-02-10]. Available from: `https://github.com/rapi-doc/RapiDoc/issues/595`.

[66] *Feature request: const keyword support* [online]. RapiDoc GitHub issues, 2022 [visited on 2023-02-10]. Available from: `https://github.com/rapi-doc/RapiDoc/issues/806`.

Appendix A

Acronyms

API	Application Programming Interface
APM	Application Performance Monitoring
CD	Continous Deplopyment
CDN	Content Delivery Network
CI	Continous Integration
CORS	Cross-Origin Resource Sharing
CSS	Cascading Style Sheets
GUI	Graphical User Interface
HAR	HTTP Archive
HATEOAS	Hypermedia As The Engine Of Application State
HTML	HyperText Markup Language

HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
JS	JavaScript
JSON	JavaScript Object Notation
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
REST	Representational State Transfer
RPC	Remote Procedure Call
RUM	Real User Monitoring
SQL	Structured Query Language
SVG	Scalable Vector Graphics
TTL	Time To Live
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

Appendix **B**

Glossary

cache hit rate

The percentage of requests handled directly from the cache without reaching the origin servers.

hotlinking

The use of a resource hosted on one site by other sites.

introspection

The ability of a program to examine an object type and its properties at runtime.

mocking

A testing technique of replacing real dependencies with objects that mimic the dependency behavior in a controlled way.

online analytical processing

Describes systems that handle a low volume of complex, not necessarily real-time queries.

online transaction processing

Describes systems that handle a high volume of simple queries and expect real-time results.

route

A mapping from the HTTP method and URI to the set of middleware handlers.

router

A middleware responsible for matching the incoming requests to the correct routes based on their HTTP method and URI.

vendor lock-in

Being so dependent on specific vendor features that it is hard to change the vendor without significant costs.

web scraping

Extracting data for machine processing from a source in a human-readable form.

Appendix C

Contents of the Enclosed Memory Card

```
README.md ................................ contents description
Thesis ................................... LaTeX source code of the thesis
  DP_Kolárik_Martin_2023.pdf ............. PDF version of the thesis
data.jsdelivr.com ........................ source code of the jsDelivr API
www.jsdelivr.com ......................... source code of the jsDelivr website
```