

The Addison-Wesley Signature Series



PRINCIPLES OF WEB API DESIGN

DELIVERING VALUE WITH
APIs AND MICROSERVICES

JAMES HIGGINBOTHAM



Foreword by
MIKE AMUNDSEN



FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Praise for *Principles of Web API Design*

“I’ve had the good fortune to work alongside and learn from James over the past several years. His varied institutional knowledge, along with his depth of experience and eye for practical application, makes him unique among his peers. I am ecstatic that others now have the opportunity, in this book, to benefit from James’s compelling, pragmatic vision for how to make better APIs. *Principles of Web API Design* surveys the gamut of available techniques and sets forth a prescriptive, easy-to-follow approach. Teams that apply the guidance in this book will create APIs that better resonate with customers, deliver more business value in less time, and require fewer breaking changes. I cannot recommend *Principles of Web API Design* enough.”

—Matthew Reinbold, Director of API Ecosystems, Postman

“James is one of the preeminent experts on API design in the industry, and this comprehensive guide reflects that. Putting API design in the context of business outcomes and digital capabilities makes this a vital guide for any organization undergoing digital transformation.”

—Matt McLarty, Global Leader of API Strategy at MuleSoft,
a Salesforce company

“In modern software development, APIs end up being both the cause of and solution to many of the problems we face. James’s process for dissecting, analyzing, and designing APIs from concepts to caching creates a repeatable approach for teams to solve more problems than they create.”

—D. Keith Casey, Jr., API Problem Solver, CaseySoftware, LLC

“Following James’s clear and easy-to-follow guide, in one afternoon I was able to apply his process to current real-world use cases. I now have the practical guidance, techniques, and clear examples to help me take those next vital steps. Recommended reading for anyone connected to and working with APIs.”

—Joyce Stack, Architect, Elsevier

“*Principles of Web API Design* uncovers more than principles. In it, you’ll learn a process—a method to design APIs.”

—Arnaud Lauret, API Handyman

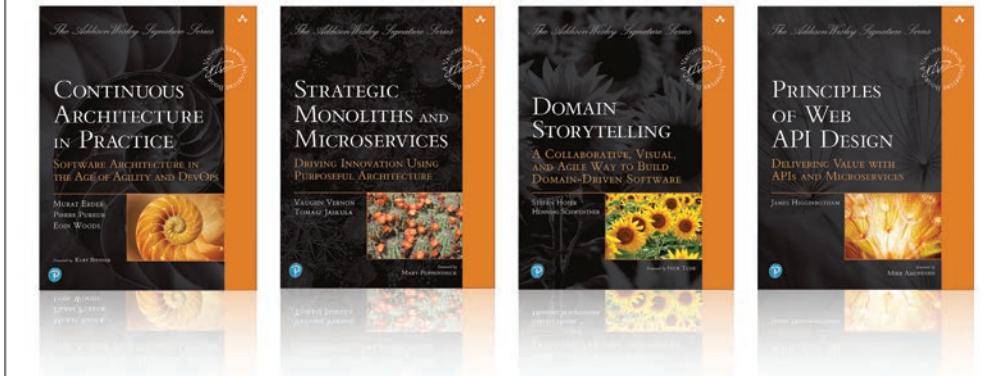
“This insightful playbook guides API teams through a structured process that fosters productive collaboration, valuable capability identification, and best-practice contract crafting. James distills years of experience into a pragmatic roadmap for defining and refining API products, and also provides a primer for API security, eventing, resiliency, and microservices alignment. A must-read for architects either new to the API discipline or responsible for onboarding new teams and instituting a structured API definition process.”

—Chris Haddad, Chief Architect, Karux LLC

This page intentionally left blank

Principles of Web API Design

Pearson Addison-Wesley Signature Series



Visit informit.com/awss/vernon for a complete list of available publications.

The **Pearson Addison-Wesley Signature Series** provides readers with practical and authoritative information on the latest trends in modern technology for computer professionals. The series is based on one simple premise: great books come from great authors.

Vaughn Vernon is a champion of simplifying software architecture and development, with an emphasis on reactive methods. He has a unique ability to teach and lead with Domain-Driven Design using lightweight tools to unveil unimagined value. He helps organizations achieve competitive advantages using enduring tools such as architectures, patterns, and approaches, and through partnerships between business stakeholders and software developers.

Vaughn's Signature Series guides readers toward advances in software development maturity and greater success with business-centric practices. The series emphasizes organic refinement with a variety of approaches—reactive, object, and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs—and covers best uses of the associated underlying technologies.

Principles of Web API Design

Delivering Value with
APIs and Microservices

James Higginbotham

◆ Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover image: Anna Om/Shutterstock

Figures 7.8–7.11: © 2021 SmartBear Software

Figure 10.12, icons: dependency by Knut M. Synstad from the Noun Project; plug by Vectors Market from the Noun Project; database by MRK from the Noun Project; filter by Landan Lloyd from the Noun Project; command line by Focus from the Noun Project; algorithm by Trevor Dsouza from the Noun Project; name tag by Cindy Clegane from the Noun Project; task list by Royal@design from the Noun Project; quality by Flatart from the Noun Project; broadcast by Yoyon Pujiyono from the Noun Project.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021947541

Copyright © 2022 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

ISBN-13: 978-0-13-735563-1

ISBN-10: 0-13-735563-7

ScoutAutomatedPrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at <https://www.pearson.com/report-bias.html>.

This page intentionally left blank

*To my wife,
whose support and encouragement
makes everything possible.*

*To my grandfather, J.W.,
who gave me a Commodore 64 when I was eight years old
because he believed “computers are going to be big someday,
and my grandson should know how to use one”
and who inspired me to follow in his footsteps as an author.*

*To my dad,
who continued the work
of J.W. I miss you.*

*To my son,
who continues the tradition with
his endless coding in Minecraft.*

*And to my daughter,
who inspires me every day
to write better copy.*

This page intentionally left blank

Contents

Series Editor Foreword	xxi
Foreword	xxv
Preface	xxvii
Acknowledgments	xxxii
About the Author	xxxiii
Part I: Introduction to Web API Design	1
Chapter 1: The Principles of API Design	3
The Elements of Web API Design	4
Business Capabilities	4
Product Thinking	4
Developer Experience	5
API Design Is Communication	6
Reviewing the Principles of Software Design	7
Modularization	8
Encapsulation	8
High Cohesion and Loose Coupling	9
Resource-Based API Design	10
Resources Are Not Data Models	10
Resources Are Not Object or Domain Models	11
Resource-Based APIs Exchange Messages	12
The Principles of Web API Design	13
Summary	14
Chapter 2: Collaborative API Design	15
Why an API Design Process?	15
API Design Process Antipatterns	16
The Leaky Abstraction Antipattern	16
The Next Release Design Fix Antipattern	19

The Heroic Design Effort Antipattern	19
The Unused API Antipattern	20
The API Design-First Approach	20
Remaining Agile with API Design-First	22
The Agile Manifesto Revisited	22
The Agility of API Design-First	23
The Align-Define-Design-Refine Process	23
The Role of DDD in API Design	26
API Design Involves Everyone	26
Applying the Process Effectively	28
Summary	28
Part II: Aligning on API Outcomes	29
Chapter 3: Identify Digital Capabilities	31
Ensuring Stakeholder Alignment	32
What Are Digital Capabilities?	33
Focusing on the Jobs to Be Done	34
What Are Job Stories?	35
The Components of a Job Story	36
Writing Job Stories for APIs	37
Method 1: When the Problem Is Known	37
Method 2: When the Desired Outcome Is Known	37
Method 3: When the Digital Capability Has Been Identified	38
Overcoming Job Story Challenges	38
Challenge 1: Job Stories Are Too Detailed	38
Challenge 2: Job Stories Are Feature Centric	39
Challenge 3: Additional User Context Is Needed	40
Techniques for Capturing Job Stories	40
A Real-World API Design Project	41
Job Story Examples	42
Summary	42
Chapter 4: Capture Activities and Steps	45
Extending Job Stories into Activities and Steps	46
Identify the Activities for Each Job Story	47
Decompose Each Activity into Steps	47
What If Requirements Aren't Clear?	48

Using EventStorming for Collaborative Understanding	49
How EventStorming Works.	50
Step 1: Identify Business Domain Events.	51
Step 2: Create an Event Narrative	51
Step 3: Review the Narrative and Identify Gaps	54
Step 4: Expand Domain Understanding	54
Step 5: Review the Final Narrative.	56
The Benefits of EventStorming	58
Who Should Be Involved?	59
Facilitating an EventStorming Session	60
Prepare: Gathering Necessary Supplies.	60
Share: Communicating the EventStorming Session.	62
Execute: Conducting the EventStorming Session	63
Wrap-up: Capture Activities and Activity Steps	63
Follow-up: Post-Session Recommendations	63
Customizing the Process	64
Summary.	65
Part III: Defining Candidate APIs	67
Chapter 5: Identifying API Boundaries	69
Avoiding API Boundary Antipatterns	70
The Mega All-in-One API Antipattern	70
The Overloaded API Antipattern	70
The Helper API Antipattern	71
Bounded Contexts, Subdomains, and APIs.	72
Finding API Boundaries Using EventStorming	73
Finding API Boundaries through Activities.	73
Naming and Scoping APIs	75
Summary.	78
Chapter 6: API Modeling.	79
What Is API Modeling?	80
The API Profile Structure	81
The API Modeling Process	81
Step 1: Capture API Profile Summary	83
Step 2: Identify the Resources	85
Step 3: Define the Resource Taxonomy.	87

Step 4: Add Operation Events	88
Step 5: Expand Operation Details	91
Validating the API Model with Sequence Diagrams	93
Evaluating API Priority and Reuse	95
Summary	96
Part IV: Designing APIs	99
Chapter 7: REST-Based API Design	101
What Is a REST-Based API?	102
REST Is Client/Server	104
REST Is Resource-Centric	104
REST Is Message Based	105
REST Supports a Layered System	105
REST Supports Code on Demand	107
Hypermedia Controls	107
When to Choose REST	111
REST API Design Process	112
Step 1: Design Resource URL Paths	112
Step 2: Map API Operations to HTTP Methods	115
Step 3: Assign Response Codes	116
Step 4: Documenting the REST API Design	118
Step 5: Share and Gather Feedback	124
Selecting a Representation Format	125
Resource Serialization	126
Hypermedia Serialization	127
Hypermedia Messaging	128
Semantic Hypermedia Messaging	129
Common REST Design Patterns	132
Create-Read-Update-Delete	132
Extended Resource Lifecycle Support	133
Singleton Resources	134
Background (Queued) Jobs	134
Long-Running Transaction Support in REST	135
Summary	136
Chapter 8: RPC and Query-Based API Design	137
What Is an RPC-Based API?	138

The gRPC Protocol	139
Factors When Considering RPC	141
RPC API Design Process	142
Step 1: Identify RPC Operations	142
Step 2: Detail RPC Operations	142
Step 3: Document the API Design	145
What Is a Query-Based API?	146
Understanding OData	147
Exploring GraphQL	149
Query-Based API Design Process	150
Step 1: Designing Resource and Graph Structures.	151
Step 2: Design Query and Mutation Operations.	151
Step 3: Document the API Design	154
Summary	157
Chapter 9: Async APIs for Eventing and Streaming	159
The Problem with API Polling	160
Async APIs Create New Possibilities.	161
A Review of Messaging Fundamentals	162
Messaging Styles and Locality.	164
The Elements of a Message.	165
Understanding Messaging Brokers	166
Point-to-Point Message Distribution (Queues)	167
Fanout Message Distribution (Topics)	167
Message Streaming Fundamentals	168
Async API Styles	171
Server Notification Using Webhooks.	171
Server Push Using Server-Sent Events	172
Bidirectional Notification via WebSocket	174
gRPC Streaming	176
Selecting an Async API Style	177
Designing Async APIs	178
Command Messages	178
Event Notifications	179
Event-Carried State Transfer Events	180
Event Batching.	182
Event Ordering	183

Documenting Async APIs	184
Summary	186
Part V: Refining the API Design	187
Chapter 10: From APIs to Microservices	189
What Are Microservices?	190
Microservices Reduce Coordination Costs	192
The Difference between APIs and Microservices	193
Weighing the Complexity of Microservices	193
Self-Service Infrastructure	194
Independent Release Cycles	194
Shift to Single-Team Ownership	194
Organizational Structure and Cultural Impacts	195
Shift in Data Ownership	195
Distributed Data Management and Governance	196
Distributed Systems Challenges	196
Resiliency, Failover, and Distributed Transactions	197
Refactoring and Code Sharing Challenges	197
Synchronous and Asynchronous Microservices	198
Microservice Architecture Styles	201
Direct Service Communication	201
API-Based Orchestration	201
Cell-Based Architecture	203
Right-Sizing Microservices	204
Decomposing APIs into Microservices	204
Step 1: Identify Candidate Microservices	205
Step 2: Add Microservices into API Sequence Diagrams	206
Step 3: Capture Using the Microservice Design Canvas	208
Additional Microservice Design Considerations	208
Considerations When Transitioning to Microservices	210
Summary	211
Chapter 11: Improving the Developer Experience	213
Creating a Mock API Implementation	214
Static API Mocking	215
API Prototype Mocking	216
README-Based Mocking	217

Providing Helper Libraries and SDKs	219
Options for Offering Helper Libraries	220
Versioning Helper Libraries.	220
Helper Library Documentation and Testing	221
Offering CLIs for APIs	221
Summary	224
Chapter 12: API Testing Strategies.	225
Acceptance Testing	226
Automated Security Testing	226
Operational Monitoring	227
API Contract Testing.	227
Selecting Tools to Accelerate Testing	229
The Challenges of API Testing	230
Make API Testing Essential.	231
Summary	231
Chapter 13: Document the API Design	233
The Importance of API Documentation	234
API Description Formats.	234
OpenAPI Specification	235
API Blueprint.	238
RAML	240
JSON Schema	244
API Profiles Using ALPS	245
Improving API Discovery Using APIs.json	247
Extending Docs with Code Examples	248
Write Getting Started Code Examples First	249
Expanding Documentation with Workflow Examples	249
Error Case and Production-Ready Examples	251
From Reference Docs to a Developer Portal	251
Increasing API Adoption through Developer Portals	251
Elements of a Great Developer Portal	252
Effective API Documentation	253
Question 1: How Does Your API Solve My Problems?	254
Question 2: What Problem Does Each API Operation Support?	254

Question 3: How Do I Get Started Using the API?	254
The Role of Technical Writer in API Docs.	255
The Minimum Viable Portal	256
Phase 1: Minimum Viable Portal	256
Phase 2: Improvement	257
Phase 3: Focusing on Growth	258
Tools and Frameworks for Developer Portals	259
Summary	260
Chapter 14: Designing for Change.	261
The Impact of Change on Existing APIs	261
Perform an API Design Gap Analysis	262
Determine What Is Best for API Consumers	262
Strategies for Change.	263
Change Management Is Built on Trust	264
API Versioning Strategies	264
Common Nonbreaking Changes.	265
Incompatible Changes.	265
API Versions and Revisions	266
API Versioning Methods	267
Business Considerations of API Versioning.	268
Deprecating APIs.	268
Establish a Deprecation Policy	269
Announcing a Deprecation	269
Establishing an API Stability Contract	270
Summary	271
Chapter 15: Protecting APIs	273
The Potential for API Mischief	273
Essential API Protection Practices.	274
Components of API Protection.	276
API Gateways	276
API Management.	276
Service Meshes	277
Web Application Firewalls.	278
Content Delivery Networks.	278
Intelligent API Protection	279

API Gateway Topologies	279
API Management Hosting Options	279
API Network Traffic Considerations	282
Topology 1: API Gateway Direct to API Server	283
Topology 2: API Gateway Routing to Services	283
Topology 3: Multiple API Gateway Instances	283
Identity and Access Management	284
Passwords and API Keys	285
API Tokens	286
Pass-by-Reference versus Pass-by-Value API Tokens	287
OAuth 2.0 and OpenID Connect	288
Considerations before Building an In-House API Gateway	289
Reason 1: API Security Is a Moving Target	290
Reason 2: It Will Take Longer than Expected	290
Reason 3: Expected Performance Takes Time	290
What about Helper Libraries?	291
Summary	291
Chapter 16: Continuing the API Design Journey	293
Establishing an API Style Guide	293
Methods for Encouraging Style Guide Adherence	294
Selecting Style Guide Tone	295
Tips for Getting Started with an API Style Guide	296
Supporting Multiple API Styles	296
Conducting API Design Reviews	297
Start with a Documentation Review	298
Check for Standards and Design Consistency	299
Review Automated Test Coverage	299
Add Try It Out Support	299
Developing a Culture of Reuse	300
The Journey Has Only Begun	301
Appendix: HTTP Primer	303
Index	319

This page intentionally left blank

Series Editor Foreword

My signature series emphasizes organic growth and refinement, which I describe in more detail below. Before that, I will tell you a little about how organic reactions brought the author and I together for the first time.

If you've ever spent a summer in a desert, you know that your flesh-and-blood organism becomes very uncomfortable with the heat. That's certainly the case with summer in the Sonoran Desert of Arizona. Temperatures can rise to near 120°F, or 49°C. At 118°F/47.8°C, the Phoenix Sky Harbor Airport shuts down operations. So, if you are going to break free from the heat, you get out before you are stuck in the desert. That's what we did in early July 2019, when we escaped to Boulder, Colorado, where we had previously resided. Knowing that the author of this book, James Higginbotham, had relocated to Colorado Springs, Colorado, gave us the opportunity to meet up for a few days in that nearby Colorado city. (In the western US, 100 miles/160 km is considered to be nearby.) I'll tell you more about our collaboration once I've introduced you to my signature series.

My Signature Series is designed and curated to guide readers toward advances in software development maturity and greater success with business-centric practices. The series emphasizes organic refinement with a variety of approaches—reactive, object, as well as functional architecture and programming; domain modeling; right-sized services; patterns; and APIs—and covers best uses of the associated underlying technologies.

From here I am focusing now on only two words: *organic refinement*.

The first word, *organic*, stood out to me recently when a friend and colleague used it to describe software architecture. I have heard and used the word *organic* in connection with software development, but I didn't think about that word as carefully as I did then when I personally consumed the two used together: *organic architecture*.

Think about the word *organic*, and even the word *organism*. For the most part these are used when referring to living things, but are also used to describe inanimate things that feature some characteristics that resemble life forms. *Organic* originates in Greek. Its etymology is with reference to a functioning organ of the body. If you read the etymology of *organ*, it has a broader use, and in fact organic followed suit: body organs; to implement; describes a tool for making or doing; a musical instrument.

We can readily think of numerous organic objects—living organisms—from the very large to the microscopic single-celled life forms. With the second use of *organism*, though, examples may not as readily pop into our mind. One example is an organization, which includes the prefix of both *organic* and *organism*. In this use of *organism*, I'm describing something that is structured with bidirectional dependencies. An organization is an organism because it has organized parts. This kind of organism cannot survive without the parts, and the parts cannot survive without the organism.

Taking that perspective, we can continue applying this thinking to nonliving things because they exhibit characteristics of living organisms. Consider the atom. Every single atom is a system unto itself, and all living things are composed of atoms. Yet, atoms are inorganic and do not reproduce. Even so, it's not difficult to think of atoms as living things in the sense that they are endlessly moving, functioning. Atoms even bond with other atoms. When this occurs, each atom is not only a single system unto itself, but becomes a subsystem along with other atoms as subsystems, with their combined behaviors yielding a greater whole system.

So then, all kinds of concepts regarding software are quite organic in that nonliving things are still “characterized” by aspects of living organisms. When we discuss software model concepts using concrete scenarios, or draw an architecture diagram, or write a unit test and its corresponding domain model unit, software starts to come alive. It isn't static, because we continue to discuss how to make it better, subjecting it to refinement, where one scenario leads to another, and that has an impact on the architecture and the domain model. As we continue to iterate, the increasing value in refinements leads to incremental growth of the organism. As time progresses so does the software. We wrangle with and tackle complexity through useful abstractions, and the software grows and changes shapes, all with the explicit purpose of making work better for real living organisms at global scales.

Sadly, software organics tend to grow poorly more often than they grow well. Even if they start out life in good health they tend to get diseases, become deformed, grow unnatural appendages, atrophy, and deteriorate. Worse still is that these symptoms are caused by efforts to refine the software that go wrong instead of making things better. The worst part is that with every failed refinement, everything that goes wrong with these complexly ill bodies doesn't cause their death. (Oh, if they could just die!) Instead, we have to kill them and killing them requires nerves, skills, and the intestinal fortitude of a dragon slayer. No, not one, but dozens of vigorous dragon slayers. Actually, make that dozens of dragon slayers who have really big brains.

That's where this series comes into play. I am curating a series designed to help you mature and reach greater success with a variety of approaches—reactive, object, and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs. And along with that, the series covers best uses of the

associated underlying technologies. It's not accomplished at one fell swoop. It requires organic refinement with purpose and skill. I and the other authors are here to help. To that end, we've delivered our very best to achieve our goal.

When James and I got together for a few days in July 2019, we covered a lot of ground on APIs and Domain-Driven Design, along with related subjects. I'd consider our conversations organic in nature. As we iterated on various topics, we refined our knowledge exchange, gauged by our level of interest in whatever direction our hunger led us. Feeding our brains resulted in growing our own desire and determination to extend our software building approaches in order to help others expand their skills and grow toward greater successes. Those who read our books, as well as our consulting and training clients, are the ones who have gained the most.

To say the least, I was impressed by James's encyclopedic knowledge of everything APIs. While we were together, I asked James about writing a book. He informed me that he had self-published one book but wasn't at that time intent on writing another book. That was approximately nine months before I was offered the Signature Series. When the series planning was in the works, I immediately approached James about authoring in the series. I was so happy that he accepted and that he proposed organic software design and development techniques, such as with Align-Define-Design-Refine (ADDR). When you read his book, you will understand why I am so pleased to have James in my series.

—*Vaughn Vernon*

This page intentionally left blank

Foreword

According to a recent IDC report on APIs and API management, 75 percent of those surveyed were focused on digital transformation through the design and implementation of APIs and more than one half expected call volume and response time to grow dramatically. And most organizations admitted they faced challenges in meeting expectations for both internally and externally facing APIs. At the heart of all of this is the need for consistent, reliable, and scalable API design programs to help lead and transform existing organizations. As James Higginbotham puts it in this book: “The biggest challenge for today’s API programs continues to be successfully designing APIs that can be understood and integrated by developers in a consistent and scalable fashion.”

It was for this reason that I was so happy to have this book cross my desk. I’ve had the pleasure of working with James over the years and, knowing his work and his reputation, was very happy to hear he was writing a book that covers Web API design. Now, after reading through this book, I am equally happy to recommend it to you, the reader.

The field of Web APIs and the work of designing them has matured rapidly over the last few years, and keeping up with the latest developments is a major undertaking. Issues like changing business expectations for the role of APIs; maturing processes for gathering, recording, and documenting the work of API design; as well as evolving technology changes and all the work of coding, releasing, testing, and monitoring APIs make up an API landscape large enough that few people have been able to successfully tackle it. Through his Align-Define-Design-Refine process, James offers an excellent set of recommendations, examples, and experience-based advice to help the reader navigate the existing space of Web APIs and prepare for the inevitable changes ahead in the future.

One of the things about James’s work that has always stood out is his ability to reach beyond the technical and into the social and business aspects of APIs and API programs within organizations. James has a long list of international clients across the business sectors of banking, insurance, global shipping, and even computer hardware providers, and the material in this book reflects this depth of experience. The techniques and processes detailed here have been tried and tested in all sorts of enterprise settings, and James’s ability to distill what works into this one volume is

impressive. Whether you are looking for advice on general design, business-technology alignment, or implementation details for various technologies such as REST, GraphQL, and event-driven platforms, you'll find important and actionable advice within these pages.

In particular, I found the material on how to refine your API design and implementation efforts within an ever-growing enterprise API program particularly timely and especially valuable. For those tasked with launching, managing, and expanding the role of Web-based APIs within a company, *Principles of Web API Design* should prove to be a welcome addition to your bookshelf.

As the aforementioned IDC report indicates, many companies around the globe are faced with important digital transformation challenges, and APIs have a major role to play in helping organizations meet the needs of their customers and in continuing to improve their own bottom line. Whether you are focused on designing, building, deploying, or maintaining APIs, this book contains helpful insights and advice.

I know this book will become an important part of my toolkit as I work with companies of all stripes to continue to mature and grow their API programs, and I expect you, too, will find it useful. Reading this book has reminded me of all the opportunities and challenges we all have before us. To borrow another line from James: "This is only the beginning."

—Mike Amundsen, API Strategist

Preface

It's hard to pinpoint the beginning of the journey to writing this book—perhaps it started about ten years ago. It is the result of thousands of hours of training, tens of thousands of miles traveled, and too many written words and lines of code to count. It comprises insights from organizations across the globe that were just starting their API journey or had already begun the adventure. The book incorporates the insights of API practitioners across the world whom I have had the pleasure to meet.

Or perhaps the journey started almost twenty-five years ago, when I first entered the software profession. So many advisors provided their insight via books and articles. Mentors along the way helped to shape my way of thinking about software. They laid the foundation of how I prefer to realize software architecture.

Maybe the journey really started almost forty years ago, when my grandfather gifted me with a Commodore 64. He was a civil engineer and cost engineer who attended night school while working to support his family during the day. He was thirsty for knowledge, reading and absorbing everything he could. He always made us laugh when he said, “I'm still amazed at how television works!” after seeing a computer operate. Yet, he was the one who gifted me that magical computer, saying “computers are going to be big someday, and my grandson should know how to use one.” This single action started my lifelong love of software development.

In reality, the journey started more than seventy years ago when the pioneers of our current age of computing established many of the foundational principles we still use today to construct software. Though technology choices change, and the trends come and go, it all builds on the work of so many in the software industry and beyond. Countless people have helped to carve the way for what we do today.

What I am saying is that APIs would not be what they are today without all the hard work that came before us. Therefore, we must thirst for understanding the history of our industry to better understand “the how” and “the why” behind what we do today. Then, we must seek to apply these lessons to all that we do tomorrow. Along the way, we need to find ways to inspire others to do the same. This is what my grandfather and father taught me, so I pass this lesson on to you. This book reflects the things I've learned thus far in my journey. I hope you gain some new insights by building upon what is presented here while you seek to prepare the next generation.

Who Should Read This Book

This book is for anyone who wants to design a single API or a series of APIs that will delight humans. Product owners and product managers will gain a deeper understanding of the elements that teams need to design an API. Software architects and developers will benefit from learning how to design APIs by applying principles of software architecture. Technical writers will identify ways that they not only can contribute to the clarity of API documentation but also can add value throughout the API design process. In short, *Principles of Web API Design* is for everyone involved in API design whether they are in a development or nondevelopment role.

About This Book

This book outlines a series of principles and a process for designing APIs. The Align-Define-Design-Refine (ADDR) process featured in this book is designed to help individuals and cross-functional teams to navigate the complexities of API design. It encourages an outside-in perspective on API design by applying concepts such as the voice of the customer, jobs to be done, and process mapping. Although *Principles of Web API Design* walks through a greenfield example from the ground up, the book may also be used for existing APIs.

The book covers all aspects of API design, from requirements to arriving at an API design ready for delivery. It also includes guidance on how to document the API design for more effective communication between you, your team, and your API consumers. Finally, the book touches on a few elements of API delivery that may have an impact on your API design.

The book is divided into five parts:

- **Part I: Introduction to Web API Design**—An overview of why API design is important and an introduction to the API design process used in this book.
- **Part II: Aligning on API Outcomes**—Ensures alignment between the team designing the API and all customers and stakeholders.
- **Part III: Defining Candidate APIs**—Identifies the APIs, including the API operations required, necessary to deliver the desired outcomes into API profiles.
- **Part IV: Designing APIs**—Transforms the API profiles into one or more API styles that meet the needs of the target developers. Styles covered include REST, gRPC, GraphQL, and event-based asynchronous APIs.

- **Part V: Refining the Design**—Improves the API design based on insights from documentation, testing, and feedback. It also includes a chapter on decomposing APIs into microservices. Finally, the book closes with tips on how to scale the design process in larger organizations.

For those who need a refresher on HTTP, the language of the Web used for Web-based APIs, the appendix provides a nice primer to help you get started.

What's Not in the Book

There are no code listings, other than some markup used to capture API design details. You don't need to be a software developer to take advantage of the process and techniques described in this book. It doesn't dive into a specific programming language or prescribe a specific design or development methodology.

The scope of the full API design and delivery lifecycle is big. While there are some insights provided that extend beyond API design, it is impossible for me to capture every detail and situation that could occur. Instead, this book tackles the challenges teams encounter when going from an idea to business requirements and, ultimately, to an API design.

Let's get started.

Register your copy of *Principles of Web API Design* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780137355631) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

This page intentionally left blank

Acknowledgments

First, I would like to thank my wife and kids who have supported me in so many ways throughout the years. Your prayers and encouragement have meant so much to me.

Special thanks to Jeff Schneider, who suggested that we should write the first enterprise Java book in 1996, before Java was enterprise. Your insights and endless hours of coaching set me on an amazing career path. Your friendship guided me along the way.

Keith Casey, thank you for inviting me to coauthor a book and deliver API workshops to people all over the world. This book wouldn't have been written without your friendship, encouragement, and insight.

Vaughn Vernon, who sent me a message years ago asking how we could collaborate, which ultimately turned into this book—thank you for inviting me on your journey.

Mike Williams, who encouraged me to risk it all to realize my dreams, you have been an inspiration and a great friend.

A special thank you to the many reviewers of this book. Your dedication to reviewing the chapters, often under a time crunch, to help produce this book is appreciated: Mike Amundsen, Brian Conway, Adam DuVander, Michael Hibay, Arnaud Lauret, Emmanuel Paraskakis, Matthew Reinbold, Joyce Stack, Vaughn Vernon, and Olaf Zimmermann.

To all API evangelists and influencers, thank you for the personal and professional discussions. Here are just a few of the many people I've had the pleasure of meeting: Tony Blank, Mark Boyd, Lorinda Brandon, Chris Busse, Bill Doerfeld, Marsh Gardiner, Dave Goldberg, Jason Harmon, Kirsten Hunter, Kin Lane, Matt McLarty, Mehdi Medjaoui, Fran Mendez, Ronnie Mitra, Darrel Miller, John Musser, Mandy Whaley, Jeremy Whitlock, and Rob Zazueta. And to those on the Slack channel, thanks for your support!

I would like to acknowledge everyone at Pearson who supported me throughout the process. Haze Humbert, thank you for making this process as easy as it can be for an author. And thank you to the entire production team: your hard work is greatly appreciated.

Finally, to my mom, thank you for spending endless hours at the library while I researched computer programming books before I was old enough to drive.

This page intentionally left blank

About the Author

James Higginbotham is a software developer and architect with over twenty-five years of experience in developing and deploying apps and APIs. He guides enterprises through their digital transformation journey, ensuring alignment between business and technology through product-based thinking to deliver a great customer experience. James engages with teams and organizations to help them align their business, product, and technology strategies into a more composable and modular enterprise platform. James also delivers workshops that help cross-functional teams to apply an API design-first approach using his ADDR process. His industry experience includes banking, commercial insurance, hospitality, travel, and the airline industry where he helped to get an airline off the ground—literally. You can learn more about his latest efforts at <https://launchany.com> and on Twitter @launchany.

This page intentionally left blank

Part I



Introduction to Web API Design

APIs are forever. Once an API is integrated into a production application, it is difficult to make significant changes that could potentially break those existing integrations. Design decisions made in haste become future areas of confusion, support issues, and lost opportunities far into the future. The API design phase is an important part of any delivery schedule.

Part 1 examines the fundamentals of software design and how it produces a positive or negative impact on API design. It then examines the API first design process and presents an overview of an API design process. This process incorporates an outside-in perspective to deliver an effective API to meet the needs of customers, partners, and the workforce.

This page intentionally left blank

Chapter 1

The Principles of API Design

All architecture is design, but not all design is architecture. Architecture represents the set of significant design decisions that shape the form and the function of a system.

— Grady Booch

Organizations have been delivering APIs for decades. APIs started as libraries and components shared across an organization and sold by third parties. They then grew into distributed components using standards such as CORBA for distributed object integration and SOAP for integrating distributed services across organizations. These standards were designed for interoperability but lacked the elements of effective design, often requiring months of effort to successfully integrate them.

As these standards were replaced by Web APIs, only a few APIs were needed. Teams could take the time to properly design them, iterating as needed. This is no longer the case. Organizations deliver more APIs and at greater velocity than ever before. The reach of Web APIs goes beyond a few internal systems and partners.

Today's Web-based APIs connect organizations to their customers, partners, and workforce using the standards of the Web. Hundreds of libraries and frameworks exist to make it cheap and fast to deliver APIs to a marketplace or for internal use. Continuous integration and continuous delivery (CI/CD) tools make it easier than ever to build automation pipelines to ensure APIs are delivered with speed and efficiency.

Yet, the biggest challenge for today's API programs continues to be successfully designing APIs that can be understood and integrated by developers in a consistent and scalable fashion. Facing this challenge requires organizations to recognize that Web APIs are more than just technology. Just as works of art require the balance of color and light, API design benefits from the blending of business capabilities, product thinking, and a focus on developer experience.

The Elements of Web API Design

An organization's collection of APIs provides a view into what the business values in the marketplace. The design quality of its APIs provides a view into how the business values developers. Everything an API offers—and doesn't offer—speaks volumes about what an organization cares most about. Effective Web API design incorporates three important elements: business capabilities, product thinking, and developer experience.

Business Capabilities

Business capabilities describe the enablers an organization brings to market. They may include external-facing capabilities, such as unique product design, amazing customer service, or optimized product delivery. They may also include internally facing capabilities such as sales pipeline management or credit risk assessment.

Organizations deliver business capabilities in three ways: directly by the organization, outsourced via a third-party provider, or through a combination of organizational and third-party processes.

For example, a local coffee shop may choose to sell custom coffee blends. To do so, it sources coffee beans through a third-party distributor, roasts the coffee beans in-house, then utilizes a third-party point-of-sale (POS) system for selling its coffee blends in a retail store. By outsourcing some of the necessary business capabilities to specialized third parties, the coffee shop is able to focus on delivering specific business capabilities that differentiate them from others in the marketplace.

APIs digitize the business capabilities that an organization brings to a marketplace. When embarking on designing a new API or expanding an existing API, the underlying business capabilities should be well understood and reflected into the API design.

Product Thinking

Organizations were integrating with partners and customers prior to the growth of Web APIs. The challenge most organizations face, however, is that each integration has been custom made. For each new partner or customer integration, a dedicated team consisting of developers, a project manager, and an account manager were tasked with building a custom integration. This involved tremendous effort and was often repeated, with per-partner customizations.

The growth of the software-as-a-service (SaaS) business model, along with the increase in demand for Web APIs, have shifted the discussion from one-off integration with partners and customers to a focus on product thinking.

Applying product thinking to the API design process shifts the team focus from a single customer or partner to an effective API design that is able to handle new automation opportunities with little to no customization effort for a given customer segment. It also enables a self-service model for workforce, business-to-business, and customer-driven integration.

The focus of an API product becomes less on custom implementations and more on meeting market needs in a scalable and cost-effective way. Reusable APIs emerge from considering multiple consumers at once. When embarking on the design of a new API, use a product thinking approach to obtain feedback from multiple parties that will consume the API. Doing so will shape the API design early and lead to increased opportunities for reuse.

Developer Experience

User experience (UX) is the discipline of meeting the exact needs of users, from their interactions with the company to their interactions with its services and with the product itself. Developer experience (DX) is just as important for APIs as UX is for products and services. The DX focuses on the various aspects of engagement with developers for an API product. It extends beyond the operational details of the API. It also includes all aspects of the API product, from first impressions to day-to-day usage and support.

A great DX is essential to the success of an API. When a great DX is delivered, developers quickly and confidently consume a Web API. It also improves the market traction of productized APIs by moving developers from being integrators to becoming experts on the API. The expertise translates directly into the ability to deliver real value to their customers and their business quickly and with reduced effort.

As API teams seek to understand how to design a great experience for their API, remember that DX is an important factor for internal developers, also. For example, great documentation enables internal developers to understand and consume an API quickly, whereas an API that has poor documentation requires contacting the internal team responsible for the API to learn how to use it properly. While they may be able to gain direct access to the developers that designed and implemented an API, it adds unnecessary communication overhead. Internal developers benefit from great DX because they can create business value faster.

CASE STUDY

APIs and Product Thinking Meets Banking

Capital One started its API journey in 2013 with the goal of developing an enterprise API platform. The initial set of platform APIs focused on delivering automation throughout the organization to increase velocity of delivery while breaking down siloed barriers.

As the number of digital capabilities in its API platform grew, Capital One's focus shifted from internal APIs to several product opportunities in the marketplace. It launched its public-facing developer portal called DevExchange at South by Southwest (SXSW)¹ with several API products. These product offerings included bank-grade authorization, a rewards program, credit card prequalification, and even an API to create new savings accounts.

Capital One extended the idea further by leveraging its digital capabilities to develop an omnichannel presence. APIs used to power its Web site and mobile app formed a foundation for a voice-based interactive experience² using Amazon's Alexa platform and interactive chat using a chatbot named Eno (the word *one* spelled backwards).

Taking a product-based approach to its APIs, along with a robust API portfolio of digital capabilities, allowed Capital One to explore opportunities with its customers and partners. It didn't happen overnight, but it did happen because of an API focus that started with an executive vision and execution by the entire organization.

API Design Is Communication

When developers think of software design, thoughts of classes, methods, functions, modules, and databases likely spring to mind. UML sequence and activity diagrams, or simple box and arrow diagrams if preferred, are used to convey understanding across a codebase. All these elements are part of the communication process development teams use for understanding and future developer onboarding.

1. "Capital One DevExchange at SxSW 2017," March 27, 2017, <https://www.youtube.com/watch?v=4Cg9B4yaNVk>
2. "Capital One Demo of Alexa Integration at SXSW 2016," September 6, 2016, <https://www.youtube.com/watch?v=KgVcVDUSvU4&t=36s>

Likewise, API design is a communication process. Rather than communicating inwardly between the members of a single team, APIs shift the communication outward. The lines of communication are extended in three distinct ways:

1. **Communication across network boundaries:** An API's design, including its choice of protocol, has an impact on the chattiness of the API. Network protocols, such as HTTP, are better for coarse-grained communication. Other protocols, such as Message Queuing Telemetry Transport (MQTT) and Advanced Message Queuing Protocol (AMQP), often used for messaging APIs, are better suited for fine-grained communication within a defined network boundary. The API design reflects the frequency of communication between systems and the impact it may have on performance because of network boundaries and bottlenecks. The API design process has a heavy impact on performance of the client and server.
2. **Communication with consuming developers:** API design and associated documentation are the user interface for developers. They inform developers how and when they are able to use each API operation. They also determine whether and how developers can combine operations to achieve more complex results. Communication early and often during the API design process is essential to meet the needs of developers consuming the API.
3. **Communication to the marketplace:** API design and documentation inform prospective customers, partners, and internal developers what outcomes the APIs make possible through the digital capabilities they offer. Effective API design helps to communicate and enable these digital capabilities.

API design is an important part of communication. An API design process helps us to consider these aspects of communication during the design phase.

Reviewing the Principles of Software Design

Software design focuses on the organization and communication of software components within a codebase. Techniques such as code comments, sequence diagrams, and the judicious use of design patterns help improve the communication effort among team members.

Web API design builds on these principles of software design, but with a broader audience that extends beyond the team or organization. The scope of communication expands beyond a single team or organization to developers all over the world. Yet, the same principles of software design apply to Web-based

API design: modularization, encapsulation, loose coupling, and high cohesion. While these may be subjects familiar to most developers, they are fundamental to API design and need review before approaching any API design process.

Modularization

Modules are the smallest atomic unit within a software program. They are composed of one or more source files that contain classes, methods, or functions. Modules have a local, public API to expose the functionality and business capabilities that they offer to other modules within the same codebase. Modules are sometimes referred to as *components* or *code libraries*.

Most programming languages support modules through the use of namespaces or packages that group code together. Grouping related code that collaborates into the same namespace encourages high cohesion. Internal details of a module are protected through access modifiers provided by the programming language. For example, the Java programming language has keywords such as `public`, `protected`, `package`, and `private` that help to encourage loose coupling through limited exposure of a module.

As more and more modules are combined, a software system is created. A subsystem combines modules into a larger module in more complex solutions, as shown in Figure 1.1.

Applying the same concepts of modularization to Web-based API design helps to reveal the boundaries and responsibilities of every API. This ensures clear responsibilities across complementary APIs that focus on externalizing digital capabilities while hiding the internal implementation details. Consuming developers benefit by understanding the API quickly and effectively.

Encapsulation

Encapsulation seeks to hide the internal details of a component. Scope modifiers are used to limit access to a module's code. A module exposes a set of public methods or functions while hiding the internal details of the module. Internal changes may

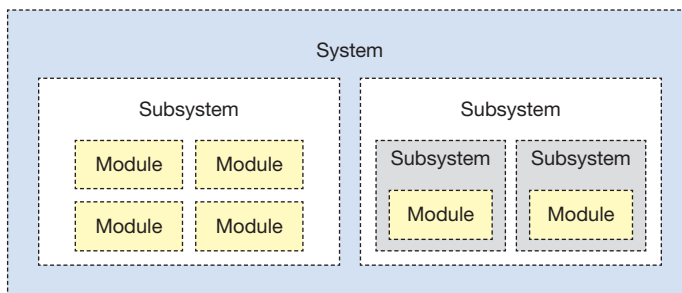


Figure 1.1 Modules combine into ever-larger units, resulting in a software system.

occur without impacting other modules that depend on its public methods. Sometimes encapsulation is referred to as *information hiding*, a concept applied to software development since the 1970s by David Parnas.

Web APIs extend this concept a bit further. They hide the internal details of programming language, choice of Web framework, the classes and objects of a system, and database design behind an HTTP-based API. Internal details, encapsulated behind the API design, encourage a loosely coupled API design that depends on messages rather than underlying database design and models for communication. No longer do organizations need to understand all the internal implementations details, such as for a payment gateway. Instead, they only need to understand the operations that the API offers and how to use them to achieve the desired outcomes.

High Cohesion and Loose Coupling

High cohesion is a term used when the code within a module is all closely related to the same functionality. A highly cohesive module results in less “spaghetti code,” as method calls aren’t jumping all over the codebase. When code is scattered across the entire codebase, calls frequently jump across modules and back again. This style of code is considered to exhibit low cohesion.

Coupling is the degree of interdependence between two or more components. Tightly coupled components indicates that the components are very constrained by the implementation details of the other. Loosely coupled components hide the components’ internal details away from others, restricting the knowledge between modules to a public interface, or programming language API, that other areas of the code can invoke.

Figure 1.2 demonstrates the concepts of high cohesion and loose coupling within and across modules.

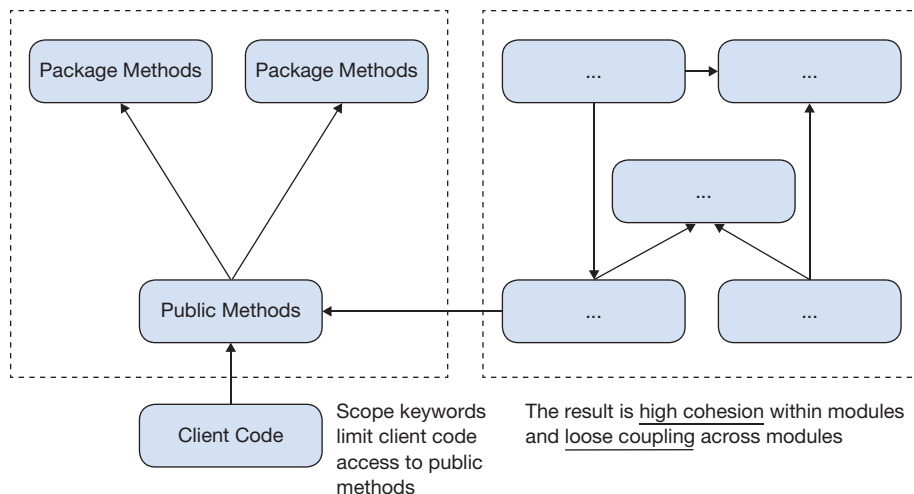


Figure 1.2 *Loose coupling and high cohesion are fundamentals of modular API design.*

Web APIs extend these concepts by grouping related API operations for high cohesion while ensuring that the internal details are encapsulated to encourage a loosely coupled API design.

Resource-Based API Design

A resource is a digital representation of a concept, often an entity or collection of entities that may change over time. It consists of a unique name or identifier that can reference documents, images, collections of other resources, or a digital representation of anything in the real world such as a person or thing. Resources may even represent business processes and workflows.

Resource-based APIs focus on interactions across a network, independent of how they are stored in a database or manifested as objects. They offer different operations, or affordances, as possible interactions with a specific resource. In addition, resources support multiple representations that allow a Web app, mobile app, and reporting tool to interact with the resource using different media formats such as JSON or XML.

Resources Are Not Data Models

It is important to recognize that resources are not the same thing as a data model that resides with a database. The data model, often reflected as a schema design in a database, is optimized for the read and write interactions necessary to support the required I/O performance and reporting needs of a solution.

While data may be part of an API, the data model should not be used as the basis of API design. Data models meet a specific set of requirements, including read and write performance, optimized data storage, and optimized query support. Data models are optimized for the internal details of an application.

Like the choice of programming languages and frameworks, the choice of database types and vendors changes over time. APIs designed to directly map to a data or object model expose these internal implementation details to API consumers. The result is a more fragile API that must introduce significant design changes when the data model changes.

Web API design seeks to achieve a different set of goals, including delivering outcomes and experiences, optimized network access, and programming language independence. Because APIs involve integration between systems, they should remain stable over a long period of time, whereas data models may change to accommodate new or changing data access requirements.

While APIs may have an impact on the data model, an API design should evolve independently from the latest database trends.

What Happens When Teams Expose a Data Model as an API?

Constant code changes: Database schema changes will result in a constantly changing API, as the API must keep in lockstep with the underlying database. This change to the data model forces consumers into a complex conformist relationship in which they must rewrite their API integration code every time the underlying data model changes. This hindrance may be overcome by an anticorruption layer that isolates a unit of code from these changes. However, the constant flux of the API creates a high cost of development as downstream developers maintain the anticorruption layer.

Create network chattiness: Exposing link tables as separate API endpoints causes API “chattiness,” as the consumer is forced to make multiple API calls, one for each table. It is similar to how an n+1 query problem degrades database performance. While an n+1 problem can be a performance bottleneck for databases, API chattiness has a devastating impact on API performance.

Data inconsistencies: Not only does performance suffer from network chattiness, but the n+1 problem also results in data inconsistencies. Clients are forced to make multiple API calls and stitch the results together into a single unified view. This may result in incomplete or corrupted data due to inconsistent reads, perhaps across transactional boundaries, that occur from multiple API requests necessary to obtain necessary data.

Confuse API details: Columns optimized for query performance, such as a CHAR(1) column that uses character codes to indicate status, become meaningless to API consumers without additional clarification.

Expose sensitive data: Tools that build APIs that mirror a data model expose all columns with a table using `SELECT * FROM [table name]`. This also exposes data that API consumers should never see, such as personally identifiable information (PII). It may also expose data that helps hackers compromise systems through a better understanding of the internal details of the API.

Resources Are Not Object or Domain Models

API resources are not the same as objects in an object-oriented codebase. Objects support collaboration within a codebase. Objects are often used to map data models into code for easier manipulation. They suffer from the same issues as exposed data models: constant code changes, network chattiness, and data inconsistencies.

Likewise, domain models, typically comprised of objects, represent the specific business domain. They may be used in a variety of ways to address the needs of the system. They may even traverse different transactional contexts based on how they are applied. Web APIs, however, are most effective when they take transactional boundaries into consideration rather than directly exposing internal domain or object model behavior.

Keep in mind that API consumers don't have the luxury of seeing the details of a data model and all the code behind an API. They didn't sit in on the endless meetings that resulted in the multitude of decisions that drove a data model design. They don't have the context of why data model design decisions were made. Great API designs avoid leaking internal details, including database design choices, by shifting from data design to message design.

Resource-Based APIs Exchange Messages

Resource-based APIs create a conversation between the business and a user or remote system. For example, suppose a user of a project management application was conversing with the API server. The conversation may look something like what's shown in Figure 1.3.

Does it seem strange to think about APIs as a chat session? It isn't far off from what Alan Kay originally intended when he coined the term *object-oriented programming*. Rather than a focus on inheritance and polymorphic design, he envisioned object-oriented programming as sending messages between components:

I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.
The big idea is "messaging."³

Like Kay's original vision for object-oriented programming, Web APIs are message based. They send request messages to a server and receive a response message as a result. Most Web APIs perform this message exchange synchronously by sending a request and waiting for the response.

API design considers the conversational message exchange between systems to produce desired outcomes by customers, partners, and the workforce. A great API design also considers how this communication evolves as requirements change.

3. Alan Kay, "Prototypes vs Classes was: Re: Sun's HotSpot," Squeak Developer's List, October 10, 1998, <http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html>.

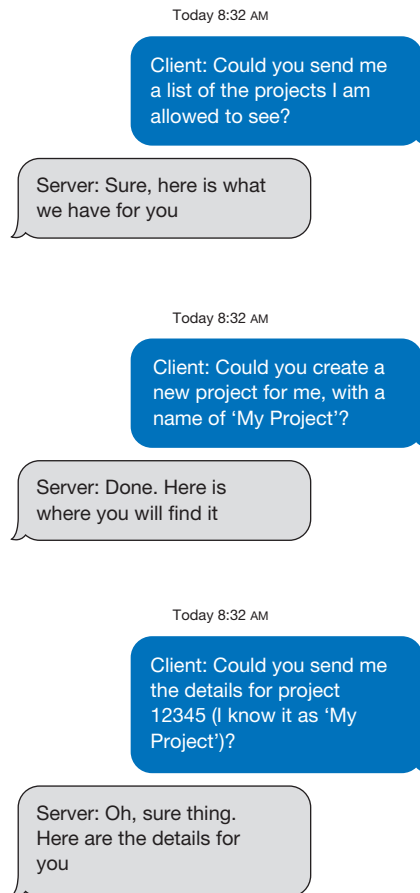


Figure 1.3 *An example interaction between an API client and API server, as if the user was talking to the server in conversational terms.*

The Principles of Web API Design

An API design approach must include a balance between robust digital capabilities and a focus on a great developer experience that supports quick and easy integration. It must be rooted in a series of principles that create a solid foundation. These five principles establish the necessary foundation and are detailed throughout this book:

Principle 1: APIs should never be designed in isolation. Collaborative API design is essential for a great API. (Chapter 2)

Principle 2: API design starts with an outcome-based focus. A focus on the outcome ensures the API delivers value to everyone. (Chapters 3–6)

Principle 3: Select the API design elements that match the need. Trying to find the perfect API style is a fruitless endeavor. Instead, seek to understand and apply the API elements appropriate for the need, whether that is REST, GraphQL, gRPC, or an emerging style just entering the industry. (Chapters 7–12)

Principle 4: API documentation is the most important user interface for developers. Therefore, API documentation should be first class and not left as a last-minute task. (Chapter 13)

Principle 5: APIs are forever, so plan accordingly. Thoughtful API design combined with an evolutionary design approach makes APIs resilient to change. (Chapter 14)

Summary

Web API design incorporates three important elements to deliver a successful API: business capabilities, product thinking, and developer experience. These cross-functional disciplines mean that organizations cannot ignore the process of API design. Developers, architects, domain experts, and product managers must work together to design APIs that meet the needs of the marketplace.

In addition, Web API design builds on the principles of software design, including modularization, encapsulation, loose coupling, and high cohesion. API designs should hide the internal details of the systems they externalize. They should not expose underlying data models but rather focus on a system-to-system message exchange that is both flexible in design and resilient to change over time.

So, how do teams go from business requirements to an API design that is evolvable while delivering the desired outcomes to customers, partners, and the internal workforce? That is the subject of the next chapter, which introduces a process that bridges business and product requirements into an API design. The process is explored in detail in subsequent chapters.

Index

A

- Acceptance testing, 226
- ActiveMQ, 166
- Activity steps captured in ADDR process, 24, 25
- ADDR. *See* Align-Define-Design-Refine (ADDR)
- Advanced Message Queuing Protocol (AMQP), 7, 164, 165, 166
- Affordances, 10, 107
- Aggregate sticky note, 55
- Agile Manifesto principles, 22
- Agility of API design-first, 22–23
- Alexa, Amazon's, 6
- Align, in ADDR process, 23
- Align-Define-Design-Refine (ADDR), 23–26
 - feedback in, 23, 24
 - goals achieved by, 24–25
 - phases of, 23
 - process overview, 25
 - steps in phases of, 24
 - steps used in real-world design project, 25–26
- Align phase of ADDR process, 23, 29
 - capture activities and steps, 45–65
 - digital capabilities, 31–43
- ALPS (Application-Level Profile Semantics), 83, 245–46
- Amazon Kinesis, 169
- Amazon's Alexa, 6
- Amazon Web Services (AWS), 221, 281
- AMQP (Advanced Message Queuing Protocol), 7, 164, 165, 166
- Amundsen, Mike, 128
- Analytics, in MVP, 259
- Anticorruption layer, 11
- Antipatterns
 - avoiding, 70–71
 - helper API antipattern, 71
 - mega all-in-one API antipattern, 70
 - overloaded API antipattern, 70–71
- Apache Avro, 126, 178
- Apache Kafka, 169, 173
- Apache Lucene, 16, 19
- Apache Pulsar, 169
- Apache Spark, 170
- Apiary, 238
- API-based orchestration, 201, 202
- API Blueprint, 238–40
- API boundaries, 69–78
 - antipatterns, avoiding, 70–71
 - bounded contexts, subdomains, and APIs, 71–72
 - finding, 72–75
 - activities for, 74–75
 - DDD for, 72
 - EvenStorming for, 72–74
 - identified in ADDR process, 24, 25
 - identifying, 69–78
 - naming and scoping APIs, 75–77, 78
- API consumption lifecycle, 300
- API contract testing, 227–28
- API description formats, 234–48
 - ALPS, 245–46
 - API Blueprint, 238–40
 - improving API discovery using APIs.json, 247–48
 - JSON Schema, 244–45
 - OAS, 235–37
 - RAML, 240–43
- API design
 - antipatterns, 16–20
 - coding, 16–19
 - heroic design effort antipattern, 19–20
 - next release design fix antipattern, 19
 - unused API antipattern, 20
 - approach, 13, 14
 - business capabilities of, 4
 - for change, 261–71

- API stability contract, establishing, 270–71
- API versioning strategies, 264–68
- deprecating APIs, 268–70
- determining what is best for API
 - consumers, questions for, 262–63
- impact of, on existing APIs, 261–64
- management built on trust, 264
- perform API design gap analysis, 262
- strategies for, 263–64
- collaborative, 13
- as communication, 6–7
- data model exposed as, 11
- developer experience in, 5
- documentation, 5, 7, 14
- elements of, 4–6, 14, 102
- outcome-based focus of, 14
- principles of, 13–14
- process, 15–28
 - ADDR in, 23–26
 - API design antipatterns in, 16–20
 - API design-first approach in, 20–23
 - applying effectively, 28
 - applying product thinking to, 5
 - communication in, 6–7, 15
 - DDD in, 26
 - reasons for, 15–16
 - roles involved in API design sessions, 27
- product thinking and, 4–5, 6
- refined in ADDR process, 24, 25
- resiliency to change, evolutionary approach for, 14, 264, 301
- resource-based, 10–11
- reviews, conducting, 297–300
 - automated test coverage, 299
 - benefits of, 297
 - caution about, 297–98
 - documentation review, starting with, 298
 - standards and design consistency, checking for, 299
 - try it out support, adding, 299–300
- scaling, within organization, 293–302
 - API consumption lifecycle, 300
 - API style guide, establishing, 293–97
 - culture of reuse, developing, 300–301
 - software design and, reviewing principles of, 7–10
- API designers and architects, 27
- API design-first approach, 20–23
 - agility of, 22–23
 - phases of, 20–21
 - principles relevant to concerns of, 22
- API design gap analysis, 262
- API documentation, 233–60
 - in ADDR process, 24, 25
 - API description formats, 234–48
 - areas of improvement, questions to identify, 253–56
 - async APIs, 184–85
 - developer portals, 251–53
 - extending docs with code examples, 248–51
 - helper libraries, 221
 - importance of, 234
 - as most important user interface for developers, 14, 234, 301
 - MVP, 256–59
 - in query-based design process, 154–57
 - REST API design, 118, 120–23
 - review, 298
 - role of technical writer in API docs, 255–56
 - in RPC API design process, 145–46
- API fundamentals, in API style guide, 294
- API gateways, 276
 - direct to API server, 283
 - in-house, 289–91
 - management hosting options, 279–81
 - middleware, 276
 - multicloud API management retail (case study), 281–82
 - multiple instances of, 283–84, 285
 - network traffic considerations, 282–83
 - routing to services, 283
 - topologies, 279–84
- API keys, 285–86
- API management layers (APIMs), 260, 276–77
- APIMatic, 223
- API modeling, 79–98
 - API priority and reuse, 95–96, 97
 - API profile structure, 81, 82
 - defined, 80–81

- OAS in, 83
- process, 81–93
 - add operation events, 88, 91, 92
 - capture API profile summary, 83–84
 - expand operation details, 91, 93, 94
 - resource identification, 85–87
 - resource taxonomy, defining, 87–88, 89–90
 - sequence diagrams for validating, 93–95
- APIs (API management layers), 260, 276–77
- API polling, 160–61
- API priority and reuse
 - assess business and competitive value, 96
 - evaluating, 95–96, 97
 - sizing and prioritization, 96, 97
- API profile, 80
 - modeled in ADDR process, 24, 25
 - structure, 81, 82
 - summary, 83–84
- API protection, 273–91. *See also* API gateways
 - APIs in, 276–77
 - authentication (authn) in, 274
 - authorization (authz) in, 275
 - CDNs in, 278
 - claims in, 275
 - components of, 276–79
 - cryptography in, 275
 - data scraping and botnet protection in, 276
 - gateway topologies, 279–84
 - IAM in, 284–89
 - intelligent API protection in, 279
 - message validation in, 275
 - mutual TLS in, 275
 - practices in, essential, 274–76
 - protocol filtering and protection in, 275
 - quotas in, 275
 - rate limiting (throttling) in, 275
 - review and scanning in, 276
 - security breaches, 273–74
 - service meshes in, 277–78
 - session hijack prevention in, 275
 - WAFs in, 278
- API prototype mocking, 216–17
- APIs differentiated from microservices, 193
- API security breaches, 273–74
- APIs.json, 247–48
- API stability contract, 270–71
- API Stylebook* (Lauret), 296
- API style guide, 293–97
 - adherence, 294–95
 - getting started with, tips for, 296
 - multiple API styles, supporting, 296–97
 - tone, selecting, 295
 - topics included in, 294
- API testing, 225–31
 - acceptance testing, 226
 - automated security testing, 226–27
 - challenges of, 230–31
 - contract testing, 227–28
 - helper libraries, 221
 - importance of, 231
 - operational monitoring, 227
 - tools to accelerate, selecting, 229–30
 - user interface testing *versus*, 228–29
- API versioning, 264–68
 - business considerations of, 268
 - common nonbreaking changes, 265
 - incompatible changes, 265–66
 - methods, 267–68
 - header-based, 267
 - hostname-based, 268
 - URI-based, 267–68
 - revisions, 266–67
 - versions, 266–67
- APM (application performance management)
 - tools, 221
- Application-Level Profile Semantics (ALPS), 83, 245–46
- Application performance management (APM) tools, 221
- Architects, 27
- “Architectural Styles and the Design of Network-based Software Architectures” (Fielding), 102
- Architecture styles in microservices, 201–3
 - API-based orchestration, 201, 202
 - cell-based architecture, 203
 - direct service communication, 201, 202
- Associative relationship, 87, 88, 89
- Async APIs for eventing and streaming, 159–86
 - async API styles, 171–78
 - bidirectional notification via WebSocket protocol, 174–76

- gRPC streaming, 176–77
- selecting, 177–78
- server notification using webhooks, 171–72
- server push using SSE, 172–74
- benefits of, 160
- designing, 178–83
 - command messages, 178–79
 - event batching, 182–83
 - event-carried state transfer events, 180–82
 - event messages, 179–80
 - event notifications, 179–80
 - event ordering, 183
- documenting, 184–85
- limitations of, 160
- messaging fundamentals, review of, 162–71
- new possibilities created with, 161–62
- polling, problem with, 160–61
- AsyncAPI specification, 185
- Asynchronous messaging, 164
- Asynchronous microservices, 198–201
- Authentication (authn), 274
- Authentication, in developer portal, 253
- Authorization (authz), 275
- Automated security testing, 226–27
- Automated test coverage, reviewing, 299
- AWS (Amazon Web Services), 221, 281

B

- Backend API, 16, 17, 18
- Backend developers and implementation, 16, 17, 18
- Background (queued) jobs, 134–35
- BDD (behavior-driven development), 219
- Behavior-driven development (BDD), 219
- Bidirectional notification via WebSocket, 174–76
- Bidirectional streaming, 139, 177
- Booch, Grady, 3
- Botnet attacks, 279
- Botnet protection, 276
- Bounded contexts, 71–72
- Brandolini, Alberto, 45, 49, 51, 64
- Browsers
 - gRPC and, 176
 - HTML and, 132
 - HTTP and, 141

- JavaScript files and, 107
- middleware and, 141
- SSE and, 172–74
- webhooks and, 177
- WebSocket and, 174–76, 178
- Business capabilities, 4
- Business domain events, 51
- Business event sticky note, 55
- Business value, assessing, 96

C

- Cacheable (architectural property), 103
- Call chaining, 197, 198–201, 208
- Capability, in job story, 36–37
- Capital One, 6
- Capture activities and steps, 45–65
 - EventStorming, 58–65
 - job stories in, 46–48
- Case studies
 - in developer portal, 252
 - as element of great developer portals, 252
 - enterprise developer portal success, 252
 - to generate growth in adoption, 258
 - GitHub webhooks create new CI/CD marketplace, 162
 - multicloud API management retail, 281–82
 - in MVP, 258
 - product thinking meets banking, 6
- Casey, D. Keith, 233, 273
- CDNs (content delivery networks), 278
- Cell-based architecture, 203
- Chaining API calls, 197, 198–201
- Changelog, in developer portal, 253
- Christensen, Clayton M., 31, 35, 36
- CI/CD (continuous integration and continuous delivery) tools, 3
- Claims in API protection, 275
- Cleanroom data set creation, 230
- Client acknowledgement mode, 166
- Client/server, 103, 104
- CLIs. *See* Command-line interfaces (CLIs)
- Clock skew, 183
- Clone method, to enforce style guide compliance, 295
- Coarse-grained communication, 7

Code

- in API design antipatterns, 16–19
- in API design-first approach, 20–21
- changes, 11
- in coupling, 9
- on demand, 103
 - supported by REST, 107
- in encapsulation, 8
- examples, extending docs with, 248–51
 - error case and production-ready examples, 251
 - expanding documentation with workflow examples, 249–51
 - write getting started code examples first, 249
- generators
 - for CLI generation, 223
 - for helper libraries generation, 223
- grouping related, 8
- in heroic design effort antipattern, 19
- in high/low cohesion, 9
- libraries, 8
- in modularization, 8
- in next release design fix antipattern, 19
- objects used to map data models into, 11, 12
- refactoring and sharing, 197
- response, in REST, 116, 118, 119
- role of developers in writing, 27
- sharing in microservices, 197

- Codebase, object-oriented, 11–12
- Collaborative API design, 15–28
- Command-line automation, 255
- Command Line Interface Guidelines, 223
- Command-line interfaces (CLIs), 221–23
 - for APIs, 221–23
 - using code generators for CLI generation, 223
- Command message, 162, 163
 - designing, 178–79
- Command sticky note, 55
- Commercial off-the-shelf (COTS) APIs, 96
- Communication, in API design process, 6–7, 15
- Community-contributed helper libraries, 220
- Competitive value, assessing, 96
- Components, 8
- Consumer-generated helper libraries, 220

- Consuming developers, communication with, 7
- Content delivery networks (CDNs), 278
- Continuous integration and continuous delivery (CI/CD) tools, 3
- Coordination costs reduced by, 192–93
- Coordination costs reduced by microservices, 192–93
- CORBA, 3
- CORS (cross-origin resource sharing), 275
- COTS (commercial off-the-shelf) APIs, 96
- Create-read-update-delete (CRUD)
 - API mocking tool to store data for, 217
 - lifecycle, 132, 147, 210
 - pattern, 132–33
 - REST and, 104
- Creation timestamps, 165
- Cross-origin resource sharing (CORS), 275
- Cross-site request forgery (CSRF), 275
- CRUD. *See* Create-read-update-delete (CRUD)
- Cryptography, 275
- CSRF (cross-site request forgery), 275
- Cucumber testing tool, 219
- Cultural impacts of microservices, 195
- Culture of reuse, developing, 300–301
- Customers, defined, 32
- Customize method, to enforce style guide compliance, 295

D

Data

- in API design-first, 20
- API mocking tool for storing, 217
- exposing sensitive, 11
- inconsistencies, 11
- models
 - exposing as API, 11
 - microservices architecture and, 196
 - objects for mapping, 11
 - resource-based API design differentiated from, 10
- mutating, 147
- ownership in microservices, shift in, 194–95
- scraping in API protection, 276
- test data sets for APIs, 230

- DDD. *See* Domain-driven design (DDD)
- DDE (dynamic data exchange), 164
- DDoS (distributed denial-of-service) attacks, 276
- Dead letter queue (DLQ), 166, 201
- Decomposing APIs into microservices, 204–10
 - additional design considerations, 208, 210
 - candidate microservices, identifying, 205–6
 - MDC to capture, 208, 209
 - microservices added to API sequence diagrams, 206–8
- Define, in ADDR process, 23
- Define phase of ADDR process, 23, 67
 - API boundaries, identifying, 69–78
 - API modeling, 79–98
- Delivery process
 - in API design-first approach, 21
 - API modeling and, 80
 - efficiency in, 16, 17–18
 - EventStorming and, 59
 - mock implementations, 21, 214
 - in reduced team coordination, 192
 - speed in, 190
- Dependent resources, 87, 112–13
- Deprecated stability contract, 271
- Deprecating APIs, 268–70
 - announcing deprecation, 269–70
 - deprecation policy, establishing, 269
- Design
 - in ADDR process, 23
 - in API design-first approach, 20, 21
 - consistency, 299
 - flaws, 19
 - patterns, in API style guide, 294
- Designer experience, 27
- Design phase of ADDR process, 23, 99. *See also*
 - High-level design
- Developer experience (DX), 5
- Developer experience, improving, 213–24
 - CLIs for APIs, 221–23
 - creating mock API implementation, 214–19
 - helper libraries and SDKs, providing, 219–21
- Developer portals, 251–53
 - API adoption through developer portals, increasing, 251–52
 - API reference documentation in, 253
 - authentication and documentation in, 253
 - case studies in, 252
 - easy onboarding in, 253
 - elements of great, 252–53
 - enterprise developer portal success (case study), 252
 - feature discovery in, 252
 - getting started guide (or quick start guide) in, 252
 - live support in, 253
 - operational insight in, 253
 - release notes and changelog in, 253
 - tools and frameworks for, 259–60
- Developer relations (DevRel), 253
- DevExchange at South by Southwest (SXSW), 6
- DevOps, 191–92
- DevRel (developer relations), 253
- Digital capabilities, 31–43
 - in ADDR process, 24, 25
 - defined, 33–34
 - identifying, 31–43
 - job stories, 35–42
 - JTBD, 34–35
 - stakeholder alignment, ensuring, 31–33
- Dillon, Karen, 31
- Direct service communication, 201, 202
- Discover, in API design-first approach, 20, 21
- Distributed data management in microservices, 196
- Distributed denial-of-service (DDoS) attacks, 276
- Distributed messaging, 164
- Distributed systems challenges in microservices, 196
- Distributed transactions in microservices, 197
- DLQ (dead letter queue), 166, 201
- Documenting API design. *See* API documentation
- Documents, for capturing job stories, 41
- DOMA (Domain-Oriented Microservice Architecture), 203
- Domain-driven design (DDD)
 - aggregates in, 55
 - for finding API boundaries, 69, 72
 - role of, in API design, 26
- Domain events, 51
- Domain experts, 27
- Domain models, 11–12

Domain-Oriented Microservice Architecture (DOMA), 203
Domain understanding, 54–56
Duncan, David S., 31
Duplicate message processing, 170
Durable subscriptions, 166
DX (developer experience), 5
Dynamic data exchange (DDE), 164

E

Easy onboarding, in developer portal, 253
Embedded resources, 127
Emerging styles, 14, 102
Encapsulation, 8–9
Eno chat bot, 6
Enterprise developer portal success (case study), 252
Error case examples, 251
ETL (extract-transform-load) processes, 170, 196
Evans, Eric, 26, 69, 72
Event batching, 182–83
Event-carried state transfer events, 180–82
Eventing. *See* Async APIs for eventing and streaming; EventStorming; Server-Sent Events (SSE)
Event message, 163
 designing, 179–80
Event notifications, 179–80
Event ordering, 183
EventStorming, 58–65
 attendees, 59–60
 benefits of, 58–60
 for collaborative understanding, 49
 for finding API boundaries, 72–74
 for international wire transfers (case study), 49–50
 process, 50–57
 create event narrative, 51–53
 customizing, 64–65
 expand domain understanding, 54–56
 identify business domain events, 51
 review final narrative, 56–57
 review narrative and identify gaps, 54
 session, 60–65
 executing, 63
 follow-up, 63–64

 preparing for, 60–61
 sharing in, 62
 wrap-up, 63
 sticky note types in, 55–56
Evolutionary design approach, 14, 19
Exchange messages, 12–13
Experimental stability contract, 271
External system sticky note, 56
Extract-transform-load (ETL) processes, 170, 196

F

Failover in microservices, 197
Fanout, use of term, 168
Feature discovery, in developer portal, 252
Federated method, to enforce style guide compliance, 295
Feedback
 in ADDR process, 23, 24
 in API design-first approach, 21
 in design process, 16, 17–18
 product thinking approach to obtain, 5
 prototype or mock API to acquire, 21
 in REST, 124–25
Fielding, Roy Thomas, 101, 102–4, 105, 107, 108, 111, 137
Fire-and-follow-up pattern, 135
Fire-and-forget pattern, 135
45-degree angle sticky notes, 64
Frontend developers and implementation, 16, 17, 18
Functional testing, 227–28
Further reading, in API style guide, 294

G

Getting started code examples, 249
Getting started guide
 in developer portal, 252
 in MVP, 258–59
GitHub
 API workshop examples on, 42, 48, 93, 136
 CI/CD marketplace created by webhooks (case study), 162
 documentation examples on, 235

- example asynchronous API descriptions on, 185
- job stories on, 42
- REST pattern resources on, 136
- GitLab, 217
- GoLang, 139, 255
- Google
 - Cloud, 221
 - Docs, 63
 - gRPC, 139–41, 176–77
 - logging in with account, 288
 - SPDY protocol, 176
- Governance in microservices, 196
- GraphQL, 14, 102, 149–50, 154–57
- Graph structures, designing, 151, 152
- GRPC, 14, 102
 - in RPC-based API design, 139–41
 - selecting, 178
 - Shopping Cart API design for, 142, 145–46
 - streaming, 176–77, 178

H

- HAL (Hypertext Application Language), 108, 127
- Hall, Taddy, 31
- HATEOAS, 108
- Header-based versioning, 267
- Helper API antipattern, 71
- Helper libraries
 - documentation and testing, 221
 - in-house gateway and, 291
 - offering, options for, 220
 - providing, 219–21
 - using code generators for generating, 223
 - versioning, 220–21
- Heroic design effort antipattern, 19–20
- Heroku, 221, 223
- H-Factors, 128
- High cohesion, 9–10
- High-level design, 24, 25
 - async APIs for eventing and streaming, 159–86
 - query-based API design, 146–57
 - REST-based API design, 101–36
 - RPC-based API design, 138–46
- Hightower, Kelsey, 189
- Homename-based versioning, 268

- Hotspot sticky note, 55
- HTML
 - API reference documentation, 234, 235, 259, 270
 - in browsers, 132
 - deprecation warning in, 270
 - Markdown files and, 41
 - in Rest-based APIs, 111
 - SSE as part of HTML5, 172
- HTTP
 - API protection and, 275, 278
 - in async APIs, 161, 162, 171–76, 177, 178
 - browsers and, 141
 - for coarse-grained communication, 7, 111
 - content negotiation in, 125
 - in helper libraries, 219, 220
 - methods
 - incompatible changes in, 266
 - invalid combinations of, 275
 - JSON:API for determining, 128
 - mapping API operations to, 115–16, 117
 - as protocol of choice, 103
 - safety classifications for, 91, 93, 115
 - selecting, 91, 294
 - via TLS, 275
 - in Query-based APIs, 147, 149, 150
 - request headers, 105, 141, 229, 267, 286
 - response codes, 116, 118–19
 - in REST-based APIs, 102–3, 105, 106, 110, 111, 112, 133, 134, 235, 240
 - in RMM, 110
 - in RPC-based APIs, 139, 141, 235
 - service meshes and, 277
 - in synchronous microservices, 198
- HTTP methods
 - mapping API operations to, 115–16, 117
 - safety classifications for, 91, 93, 115
- HTTP POST, 16, 19
- Hugo, 217, 259
- Hunt, Andrew, 80
- Hypermedia controls, 107–10
- Hypermedia messaging, 128–29
 - semantic, 129–32
- Hypermedia serialization, 127–28
- Hypertext Application Language (HAL), 108, 127

I

IAM. *See* Identity and access management (IAM)

Idempotent HTTP operation, 91, 115

Identifier, 10

Identity and access management (IAM), 284–89

- API tokens, 286–88
 - pass-by-reference *versus* pass-by-value, 287–88
- OAuth 2.0, 288, 289
- OpenID Connect, 288, 289
- passwords and API keys, 285–86

IDEs (integrated development environments), 197, 219, 238

IDL (interface definition language), 139–40, 145, 228, 270

Implementing Domain-Driven Design (Evans and Vernon), 26, 72

Incentivized method, to enforce style guide

- compliance, 294–95

Independent release cycles in microservices, 194

Independent resources, 87

Information hiding, 9

Infrastructure and operations, 27

Integrated development environments (IDEs), 197, 219, 238

Intelligent API protection, 279

Interface definition language (IDL), 139–40, 145, 228, 270

Interface testing *versus* API testing, 228–29

Internet Engineering Task Force, 174

Internet of Things (IoT), 184

Interprocess messaging, 164

Introduction, in API style guide, 294

IoT (Internet of Things), 184

Isolation, APIs designed or delivered in, 13, 17, 20, 33

J

Java Message Service (JMS), 150, 166

Java programming language, 8, 138, 166

JavaScript, 103, 107, 111, 255, 275, 286

Jekyll, 217, 259, 260

Jmqtt, 166

JMS (Java Message Service), 150, 166

Jobs to be done (JTBD), 34–35, 222

Job stories, 35–42

- in activities and steps, 46–48
 - decompose each activity into steps, 47–48
 - identify activities for each job story, 47
 - when requirements aren't clear, 48
- capturing, 40–41
- challenges in, 38–40
 - detailed job stories, 38–39
 - feature centric job stories, 39–40
 - need for additional user context, 40
- components of, 36–37
- defined, 35–36
- examples of, 42
- real-world API design project, 41–42
- writing, for APIs, 37–38

Jones, Caspers, 225

JSON, 10

JSON Schema, 244–45

JSON Web Tokens (JWTs), 288

JTBD (jobs to be done), 34–35, 222

JWTs (JSON Web Tokens), 288

K

Kay, Alan, 12, 159

Key performance indicators (KPIs), 33

Klement, Alan, 36

KPIs (key performance indicators), 33

Kubernetes, 139, 221, 223

L

Lauret, Arnaud, 296

Layered system

- in Fielding's paper, 103
- supported by REST, 105–6

Lifecycle management, in API style guide, 294

Lifecycle support, in REST, 133–34

Lindsay, Jeff, 171

Link tables, 11, 231

Live support, in developer portal, 253

Local messaging, 164

Long-running transaction support in REST, 135–36

Loose coupling, 9–10

M

- Management hosting options, 279–81
- Markdown files, 39, 41, 217, 238, 259
- Marketplace, communication to, 7
- McLarty, Matt, 293
- MDC (Microservice Design Canvas), 208, 209
- Mega all-in-one API antipattern, 70
- Message broker
 - examples of, 166
 - fanout message distribution (topics), 167–68
 - features offered by, 166
 - point-to-point message distribution (queues), 167
 - terminology, 168
 - understanding, 166–67
- Message Queuing Telemetry Transport (MQTT), 7, 164, 165, 184
- Message streaming. *See also* Async APIs for eventing and streaming
 - considerations, 170–71
 - fundamentals, 168–70
 - gRPC, 176–77, 178
 - servers, 169
- Messaging/messages, 162–71
 - elements of, 165
 - exchanged through resource-based API design, 12–13
 - filtering, 170
 - immutable nature of, 163
 - message validation in API protection, 275
 - priority and TTL, 166
 - processing failures, 166
 - styles and locality, 164
 - types, 162–63
- Microservice Design Canvas (MDC), 208, 209
- Microservices, 189–211
 - APIs differentiated from, 193
 - architecture styles, 201–3
 - complexity of, 193–97
 - coordination costs reduced by, 192–93
 - decomposing APIs into, 204–10
 - defined, 190–91
 - distributed data management and governance in, 196
 - distributed systems challenges in, 196
 - distributed transactions in, 197
 - failover in, 197
 - independent release cycles in, 194
 - need for, 198
 - organizational structure and cultural impacts of, 195
 - reduced team coordination and, 192–93
 - refactoring and code sharing in, 197
 - resiliency of, 197
 - right-sizing, 204
 - self-service infrastructure in, 194
 - shift in data ownership in, 195
 - shift to single-team ownership in, 194–95
 - synchronous/asynchronous, 198–201
 - transitioning to, considerations in, 210
 - warning about term, 191
- Middleware, 141, 276
- Minimum viable portal (MVP), 256–59
 - checklist, 256–57
 - growth in adoption, 258–59
 - analytics for, 259
 - case studies for, 258
 - documentation for, 259
 - getting started guides for, 258–59
 - single-page format for, 259
 - improving, 257–58
 - template, 260
- Mock API implementation, 214–19
 - API prototype mocking, 216–17
 - README-based mocking, 215, 217–19
 - static API mocking, 215–16
- Mockaroo, 230
- Modularization, 8
- Modular monoliths, 198
- Modules, 8
- Mozilla, 174
- MQTT (Message Queuing Telemetry Transport), 7, 164, 165, 184
- MTLS (mutual TLS), 275
- MuleSoft, 240
- Multicloud API management retail (case study), 281–82
- Multipart EventStorming sessions, 64
- Multiple API gateway instances, 283–84, 285
- Mutation operations, designing, 151, 153–54
- Mutual TLS (mTLS), 275
- MVP. *See* Minimum viable portal (MVP)

N

N+1 query problem, 11
 Namespaces, 8
 Naming APIs, 75–77, 78
 Narratives, in EventStorming
 creating, 51–53
 identify gaps, 54
 review of final narrative, 56–57
 National Institute of Standards and Technology (NIST), 282
 Network boundaries, communication across, 7
 Network chattiness, 7, 11
 Network protocols, 6
 Network traffic considerations, 282–83
 Next release design fix antipattern, 19
 NIST (National Institute of Standards and Technology), 282
 Nix tools, 223
 Node.js, 216
 Nonpublic information (NPI), 27

O

OAI (OpenAPI Initiative), 235
 OAS (OpenAPI Specification), 83, 120–22, 184, 217, 235–37
 OAuth 2.0, 288, 289
 Objective-C, 255
 Objectives and key results (OKRs), 33
 Object-oriented programming, 12
 Objects, in domain models, 11–12
 OData, 147–48
 OKRs (objectives and key results), 33
 OLAP (online analytical processing), 196
 Onboarding, 6, 21
 O’Neill, Mark, 213
 Online analytical processing (OLAP), 196
 OpenAPI Initiative (OAI), 235
 OpenAPI Specification (OAS), 83, 120–22, 184, 217, 235–37
 OpenID Connect, 288, 289
 Open Web Application Security Project (OWASP), 229
 Operational insight, in developer portal, 253
 Operational monitoring, in API testing, 227

Operational recommendations, in API style guide, 294
 Operation details, in API modeling, 91, 93, 94
 Oracle, 238
 Organizational structure of microservices, 195
 Outcome, in job story, 36–37
 Outcome-based focus, APIs designed or delivered in, 14, 35, 264, 301
 Outsourcing, 4
 Overloaded API antipattern, 70–71
 OWASP (Open Web Application Security Project), 229

P

Parnas, David, 9
 Pass-by-reference API tokens, 287–88
 Pass-by-value API tokens, 287–88
 Passwords, 285–86
 Personally identifiable information (PII), 11, 27
 PHP, 216
 PII (personally identifiable information), 11, 27
 Pipe and filter design pattern, 223
 POCs (proofs of concept), 221
 Point-of-sale (POS) system, third-party, 4
 Policy sticky note, 55
 Polling, 160–61
 POS (point-of-sale) system, third-party, 4
 POS (third-party point-of-sale) system, 4
The Pragmatic Programmer (Thomas and Hunt), 80
 Prerelease stability contract, 271
 Product definition, 32
 Production-ready examples, 251
 Product managers, 27
 Product thinking, 4–5, 6
 Product thinking meets banking (case study), 6
 Programming languages, 8
 Project managers, 27
 Proofs of concept (POCs), 221
 Protecting APIs. *See* API protection
 Protocol Buffers, 126, 139–40, 142, 145, 176, 178
 Protocol filtering, 275
 Prototype, in API design-first approach, 21
 Prototyping APIs, 19
 Provider-supported helper libraries, 220

Public-facing developer portal, 6
 Python, 216, 220, 255

Q

QA (quality assurance), 229–30
 QA teams, 27
 Quality assurance (QA), 229–30
 Query-based API design, 146–57
 defined, 146–47
 GraphQL, exploring, 149–50
 OData, understanding, 147–48
 process, 150–57
 designing resource and graph structures,
 151, 152
 design query and mutation operations,
 151, 153–54
 document API design, 154–57
 Query operations, designing, 151, 153–54
 Queues, use of term, 168
 Quick start guide. *See* Getting started guide
 Quotas, 275

R

RabbitMQ, 166, 168, 173
 RAML (RESTful API Modeling Language), 228,
 240–43
 Rate limiting (throttling), 275
 README-based mocking, 215, 217–19
 Refactoring in microservices, 197
 Reference documentation, in developer portal,
 253
 Refine, in ADDR process, 23
 Refine phase of ADDR process, 23, 187
 documenting API design, 233–60
 API description formats, 234–48
 developer portals, 251–53
 extending docs with code examples,
 248–51
 importance of, 234
 MVP, 256–59
 questions to identify areas of
 improvement for API documentation,
 253–56

 role of technical writer in API docs,
 255–56
 refining the design
 API testing strategies, 225–31
 improving developer experience, 213–24
 microservices, 189–211
 Refining the design
 API testing strategies, 225–31
 improving developer experience, 213–24
 microservices, 189–211
 Release notes, in developer portal, 253
 Remote method invocation (RMI), 138
 Remote procedure call (RPC)–based API design.
 See RPC-based API design
 Reply message, 163
 Representation format, 125–32
 categories of, 126
 hypermedia messaging, 128–29
 hypermedia serialization, 127–28
 resource serialization, 126–27
 semantic hypermedia messaging, 129–32
 Request messages, 162
 Resiliency of microservices, 197
 Resource, defined, 10
 Resource-based API design, 10–11
 data models differentiated from, 10
 messages exchanged through, 12–13
 object or domain models differentiated from,
 11–12
 Resource-centric REST, 104–5
 Resource identification, in API modeling, 85–87
 Resource serialization, 126–27
 Resource structures, designing, 151, 152
 Resource taxonomy, in API modeling, 87–88,
 89–90
 Response messages, 163
 REST-based API design, 101–36
 architectural constraints in Fielding’s paper,
 102–3
 client/server, 104
 code on demand supported by, 107
 CRUD and, 104
 defined, 102
 dependent resources, 113
 hypermedia controls, 107–10
 layered system supported by, 105–6
 measuring REST using RMM, 110–11

- message based, 105
 - patterns, 132–36
 - API workshop examples on GitHub, 136
 - background (queued) jobs, 134–35
 - CRUD-based APIs, 132–33
 - extended resource lifecycle support, 133–34
 - long-running transaction support, 135–36
 - singleton resources, 133
 - process, 112–23
 - assign response codes, 116, 118, 119
 - design resource URL paths, 112–14
 - documenting REST API design, 118, 120–23
 - map API operations to HTTP methods, 115–16, 117
 - share and gather feedback, 124–25
 - representation format, selecting, 124–32
 - resource-centric, 104–5
 - when to choose, 111–12
 - RESTful API Modeling Language (RAML), 228, 240–43
 - REST Hooks documentation, 171
 - Retired stability contract, 271
 - Review and scanning, 276
 - RFC 2119, 295
 - RFC 6455, 174
 - Richardson, Leonard, 110
 - Richardson Maturity Model (RMM), 110–11
 - Right-sizing, 204
 - RMI (remote method invocation), 138
 - RMM (Richardson Maturity Model), 110–11
 - RPC-based API design, 138–46
 - defined, 138–39
 - factors when considering, 141
 - gRPC protocol, 139–41
 - process, 142–46
 - detail RPC operations, 142, 144
 - document API design, 145–46
 - identify RPC operations, 142, 143
 - RPC (remote procedure call)–based API design.
 - See* RPC-based API design
 - Ruby, 216, 220, 250, 255
- ## S
- SaaS (software-as-a-service), 5, 41, 162, 279, 281, 284
 - Safe HTTP operation, 91, 115
 - SAML (Security Assertion Markup Language), 289
 - Schema definitions, 122–23
 - Scope modifiers, 8
 - Scopes, 286
 - Scoping APIs, 75–77, 78
 - Scrum Masters, 27
 - SDKs (software development kits), 219–21
 - Security Assertion Markup Language (SAML), 289
 - Security teams, 27
 - Self-service infrastructure in microservices, 194
 - Self-service model, 5
 - Semantic hypermedia messaging, 129–32
 - Sequence diagrams
 - microservices added to, 206–8
 - for validating API modeling, 93–95
 - Serialization
 - hypermedia, 127–28
 - resource, 126–27
 - Server push using SSE, 172–74
 - selecting, 177
 - Server-Sent Events (SSE), 184, 185, 296
 - for multiple API styles, 296
 - selecting, 177
 - server push using, 172–74
 - use cases not supported by, 174
 - use cases supported by, 173
 - Service-level agreement (SLA), 81, 227
 - Service meshes, 277–78
 - Service-oriented architecture (SOA), 197
 - Session hijack prevention, 275
 - Shared facilitation, in EventStorming, 65
 - Shopping Cart API, 142, 145–46, 151–54, 235, 240
 - Single-page applications (SPAs), 149
 - Single-page format, in MVP, 259
 - Single sign-on (SSO), 289
 - Single-team data ownership in microservices, 194–95
 - Singleton resources, in REST, 134
 - Sizing and prioritization, 96, 97

SLA (service-level agreement), 81, 227

SMEs (subject matter experts), 27, 47

Snapshots, 80, 230

SOA (service-oriented architecture), 197

SOAP, 3, 33, 110, 135–36, 138, 150, 296

Software-as-a-service (SaaS), 5, 41, 162, 279, 281, 284

Software design, reviewing principles of, 7–10

- encapsulation, 8–9
- high cohesion and loose coupling, 9–10
- modularization, 8

Software development

- agile, 23
- DDD approach to, 26
- defect removal and, 226
- information hiding in, 9
- people involved in, 26–27
- in reduced team coordination, 192

Software development kits (SDKs), 219–21

Solution-oriented testing, 226

Spaghetti code, 9

SPAs (single-page applications), 149

Spreadsheets, for capturing job stories, 41

SSE. *See* Server-Sent Events (SSE)

SSO (single sign-on), 289

Stakeholders

- alignment with, ensuring, 31–33
- in API design-first, 22
- in EventStorming, 58, 59
- feedback from, 21, 93, 270
- gathering domain details from, 45
- unused API antipattern and, 20

Standards

- in API design reviews, 299
- in API style guide, 294
- connectivity based on, 166

Stateless (architectural property), 103

Static API mocking, 215–16

Static site generators, 259

Sticky notes, 55–56

Streaming. *See* Async APIs for eventing and streaming; Message streaming

Subdomains, 71–72

Subject matter experts (SMEs), 27, 47

Subprotocol, 174

Supported stability contract, 271

Surface area, 27, 42, 54

Swagger, 120, 228, 235

Swagger Codegen project, 223

Swagger Editor, 120

SwaggerUI, 235, 259, 260

Swift, 255

SXSW (DevExchange at South by Southwest), 6

Synchronous messaging, 164

Synchronous microservices, 198–201

T

TCP/IP, 277

TDD (test-driven development), 230

Technical leads, 27

Technical writer, roles of

- in API design, 27
- in API docs, 255–56

Technologies, in API style guide, 294

Test-driven development (TDD), 230

Thin events, 179

Third-party point-of-sale (POS) system, 4

Thomas, David, 15, 80

Three-lane approach, 64

Throttling (rate limiting), 275

Time to First Hello World (TTFHW), 249–51

Time-to-live (TTL), 165, 166

TLS (Transport Layer Security), 275

Tone, in API style guide, 295

Tools

- to accelerate API testing, 229–30
- API mocking, 217
- in API style guide, 294
- APM, 221
- CI/CD, 3
- for developer portals, 259–60
- for Markdown support, 217

Topics, in API style guide, 294

Topics, use of term, 168

Tracer bullet, 80

Transactional boundaries, 166

Transport Layer Security (TLS), 275

Triggering event, in job story, 36–37

Try it out support, 299–300

TTFHW (Time to First Hello World), 249–51

TTL (time-to-live), 165, 166

U

- Uber Engineering, 203
- Ubiquitous language, 49
- UI (user interface) tests, 228–29
- Uniform interface, 103
- Unique name, 10
- UNIX, 118, 164
- Unsafe HTTP operation, 93, 115
- Unused API antipattern, 20
- URI-based versioning, 267–68
- URL paths, in REST, 112–14
- User experience (UX), 5, 27, 255
- User interface sticky note, 56
- User interface (UI) tests, 228–29
- User sticky note, 56
- UX (user experience), 5, 27, 255

V

- Vernon, Vaughn, 26, 72
- Versioning helper libraries, 220–21
- Virtual machine (VM), 277
- Virtual private network (VPN), 282
- VM (virtual machine), 277
- VOC (voice of the customer), 35
- Vogels, Werner, 261
- Voice of the customer (VOC), 35
- VPN (virtual private network), 282

W

- W3C, 172, 174
- WAFs (Web application firewalls), 277, 278
- Web APIs, 3–5, 9–10

- boundaries, 12, 71
- customer- and partner-facing, 112
- evolvable, 103
- high cohesion and loose coupling in, 10
- information hiding, 9
- message-based, 12
- REST-based, 107, 112, 161–62

- Web application firewalls (WAFs), 277, 278

- Webhooks

- dispatcher, 171, 172
- implementing effectively, 171
- selecting, 177
- server notification using, 171–72

- WebSocket

- bidirectional notification via, 174–76
- selecting WebSocket protocol, 178

- Wright, Frank Lloyd, 79

- Writing job stories, for APIs, 37–38

- when desired outcome is known, 37–38
- when digital capability has been identified, 38
- when problem is known, 37

- WS-Transaction specification, 135–36

X

- XML Schema, 244

Y

- YAML, 126, 235, 240, 245, 247
- “You ain’t gonna need it” (YAGNI) principle, 211

Z

- Zero trust architecture (ZTA), 282