The Addison-Wesley Signature Series

A Vaughn Vernon Signature Book

Principles of Web API Design

Delivering Value with APIs and Microservices

James Higginbotham

Foreword by
Mike Amundsen

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.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021947541

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

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.

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 experience[2] 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

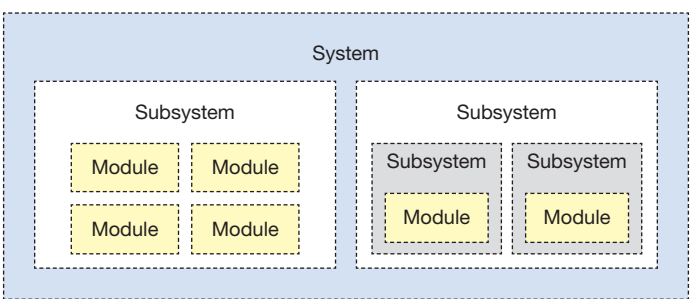

Figure 1.1 *Modules combine into ever-larger units, resulting in a software system.*

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.

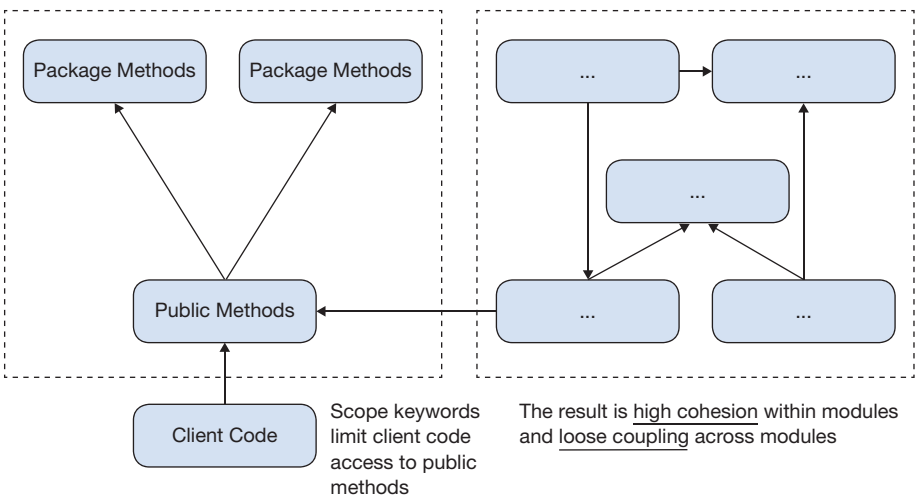

Figure 1.2 *Loose coupling and high cohesion are fundamentals of modular API design.*

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.

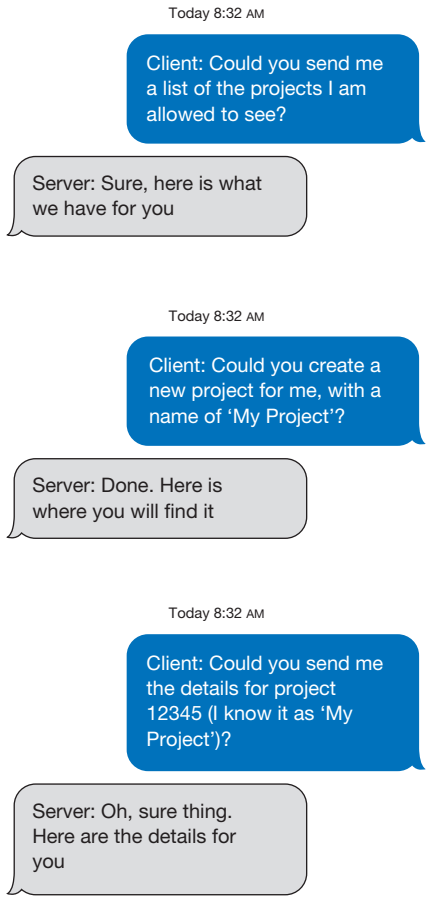

Figure 1.3 *An example interaction between an API client and API server, as if the user was talking to the server in conversational terms.*

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

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z