the Tao *of* Microservices

Richard Rodger

MANNING

SAMPLE CHAPTER

The Tao of Microservices

by Richard Rodger

Chapter 2

brief contents

PART 1 BUILDING MICROSERVICES 1

1 ■ Brave new world 3
2 ■ Services 34
3 ■ Messages 65
4 ■ Data 99
5 ■ Deployment 130

PART 2 RUNNING MICROSERVICES 171

6 ■ Measurement 173
7 ■ Migration 203
8 ■ People 228
9 ■ Case study: Nodezoo.com 246

2 Services

This chapter covers

- Refining the concept of microservices
- Exploring principle variants of the microservice architecture
- Comparing monoliths versus microservices
- Using a concrete study to explore microservices
- Thinking of microservices as software components

To understand the implications and trade-offs of moving to a new architecture, you need to understand how it differs from the old way of doing things, and how the new way will solve old problems. What are the essential differences between monolithic and microservice architectures? What are the new ways of thinking? And how do microservices solve the problems of enterprise software development?

A *microservice* is a unit of software development. The microservice architecture provides a mental model that simplifies the world at a useful level. The proposition of this book is that microservices are the closest thing yet to ideal software components. They're perfectly sized artifacts for fine-grained deployment into production. They're easily measured to ensure correct operation. The microservice attitude is the belief that these three aspects of the architecture deliver a fast, practical, efficient

way to create business value with software. Let's dig into the details to see how this works in practice.

2.1 Defining microservices

The term *microservice* is inherently fuzzy, as a social effect of the increasing popularity of the architecture. When we use the term, we should be specific in our meaning. Much of the writing on microservices shares the same attitude toward software development but uses differing definitions of the key term. Enthusiasm for weak definitions, in turn, limits our thinking and provides an easy target for criticism from vested interests. Let's examine a sample of the proposed definitions:

- *Microservices are self-contained software components that are no more than 100 lines of code.* This definition captures the desire to keep microservices small and maintainable by one developer, rather than a team. It's an appeal to the idea that extreme simplicity has extreme benefits: 100 lines of code can be quickly and confidently reviewed for errors.[1] The small body of code is also inherently disposable in that it can easily be rewritten if necessary. These are desirable qualities for microservices, but not exhaustive. For example, the questions of deployment and interservice communication aren't addressed. The fundamental weakness in this definition is the use of an arbitrary numerical constraint that falls apart if we change programming languages. As we consider other definitions, let's retain the desire for code small enough to verify easily and to throw away if need be.
- *Microservices are independently deployable processes communicating asynchronously using lightweight mechanisms focused on specific business capabilities running in an automated but platform- and language-independent environment*, or words to that effect. On the opposite end of the spectrum are catchall general definitions. These definitions contain a laundry list of desired attributes. Are the attributes ordered by importance? Are they exhaustive? Are they well defined? General definitions give you a feeling that you're in the right galaxy, but they don't provide directions to get to a microservice system. They invite endless semantic debate over the definitions of the attributes. What, for example, is a *truly* lightweight communication mechanism?[2] What we can take from these definitions is a working set of ideas that can be used in practice but that don't by themselves provide much clarity.

---

[1] C. A. R. Hoare, the inventor of the *quicksort* algorithm, in his 1980 Turing Award Lecture, famously said, "There are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies."

[2] It's impossible to win a war of definitions. As soon as you provide a conclusive counter-example, your opponent denies that the counter-example is actually an example of the subject under discussion. The British philosopher Antony Flew provides the canonical example of this tactic, which can be paraphrased as follows: *Robert*: "All Scotsmen wear kilts!"; *Hamish*: "My uncle Duncan wears trousers."; *Robert*: "Yes, but no *true* Scotsman does."

- *Microservices are mini web servers offering a small REST-based HTTP API that accepts and returns JSON documents.* This is certainly a common implementation. And these are microservices. But how big are they? And how does it address all the other concerns, such as independent deployability? This definition is both too prescriptive on some questions and not prescriptive enough on others. It's definition by archetype. Few would disagree that these are microservices. And yet it excludes most of the interesting microservice architectural patterns, particularly those that take advantage of asynchronous messages. This definition is not only weak but dangerous. Empirical evidence from the field suggests it often leads to tightly coupled services that need to be deployed together.[3] The takeaway from this failed definition is that limiting ourselves to thinking only in terms of web-service APIs prevents us from appreciating the radical possibilities that a wider concept can bring. A definition should provide power to our thinking, not constrain it.
- *A microservice is an independent software component that takes no more than one iteration to build and deploy.* In this definition, the focus is on the human side of the architecture. The phrase *independent software component* is suggestive and wide ranging, so this definition also attempts to be inclusive of implementation strategies. Microservices are software components, using the common understanding of the term.[4] This definition expresses the desire for microservices to indeed be "micro" by limiting the resources available to write them: one iteration is all you get. It also gives a nod to continuous delivery—you have to be able to deploy within an iteration. The definition is careful to avoid mention of operating system processes, networking, distributed computing, and message protocols; none of these are essential properties.[5]

We must accept that we aren't school children, but professional software developers, and we live in the messy world of grownups. There's no tidy definition of microservices, and any definition we choose restricts our thinking. Rather than seek a definition that's dependent on numerical parameters, or attempts to be exhaustive, or is too narrow, we should aim to develop a conceptual framework that's *generative*. The concepts within the framework generate an accurate understanding of the inherent tradeoffs of the microservice architecture. We then apply these concepts to the context at hand to deliver working software.[6]

---

[3] In my previous life as a consultant, I directed my poor teams to build many large systems this way, and we tied ourselves in the most wonderful Gordian knots.

[4] *Software components* are self-contained, extensible, well-defined, reusable building blocks.

[5] Erlang processes are most certainly microservices or, perhaps more correctly, nanoservices! You're strongly advised to read Joe Armstrong's Ph.D. thesis for the full details: "Making Reliable Distributed Systems in the Presence of Software Errors," Royal Institute of Technology, 2003, http://erlang.org/download/armstrong_thesis_2003.pdf.

[6] Microservices are a subject worthy of an entire book, not a trite summary definition. But then, I *would* say that.

2.2 Case study: The digital edition of a newspaper

Most chapters in this book use a case study to provide practical examples of the concepts under discussion. The studies are software systems that need to deliver a range of functionalities; in each chapter, we'll explore how the microservice architecture can deliver those functionalities. For each system, we'll focus on a subset of the functionality that's relevant to the topic of the chapter. Chapter 9 is a full case study, including code, that gives you a practical example of a microservice system using the architectural techniques developed in this book.

Our study in this chapter is the digital edition of a newspaper.[7] Let's break down this system, starting from the business goals. These generate requirements that we'll specify informally. Over the course of the chapter, we'll look at some partial implementations of these informal requirements, using microservices.

2.2.1 The business goals

The newspaper offers both free and paywalled content. To view the paywalled content, users need to subscribe. Revenue is driven by both subscriptions and advertising. The advertising is content- and user-targeted to increase relevance. To increase advertising revenue, user time on the site should be maximized.

The newspaper staff using the site should be able to publish content on a continuous basis using a content management system. They should be able to review analytics pertaining to the content they've written so that they can get feedback on their effectiveness.

The newspaper is delivered via website, tablet, and mobile app versions to maximize readership access. Article content, including paywalled content, should be search engine optimized to gain the widest potential readership.

2.2.2 The informal requirements

Using these goals, you can outline a list of informal requirements. These requirements will drive your implementation decisions:

- The content consists of articles, each of which has its own separate page.
- There are also special article-listing pages, such as the front page, and special-interest sections.
- The website, tablet, and app versions should all use a common REST API, provided by the server side of the system.
- The website should deliver static versions of primary content for search engines to index, but it can load secondary content dynamically.
- The system needs to have a concept of *users* that includes both readers and authors, with appropriate rights for different levels of access.

[7] As a mental model, think *The New York Times*. It isn't averse to a microservice or two.

- Content on pages needs to be targeted to the current user by matching content to the user's profile using business rules or optimization algorithms.
- The site is under continuous development, because online newspapers are in fierce competition, so new features need to be added quickly. These include special short-term mini apps, such as special interactive content for elections.

2.2.3 A functional breakdown

From a purely functional perspective, and without reference to any architecture choice, these requirements already allow you to think about how to implement the newspaper system. Here are some of the things the system should do:

- Handle article data and have the expected read, write, and query operations
- Construct content pages and provide a cache for scaling
- Handle user accounts: login, logout, profiles, and so on
- Deliver targeted content and map user identities to appropriate articles

These functions suggest some software components you should build. Let's pretend they're object-oriented classes for now:

- *ArticleHandler*—Provides article data operations
- *PageBuilder*—Generates pages
- *PageCache*—Caches pages
- *UserManager*—Manages users
- *ContentMapper*—Decides how to target content

You can even draw the possible dependencies between these components, as shown in figure 2.1.

Are these the right components? Are these the right dependencies? It's too soon to tell. Are these microservices? Perhaps. The microservice architecture must provide an analytical process for deciding what microservices to build. Somehow, you have to get from the informal requirements to the specific set of services in production. To start developing this process, let's take a closer look at the properties of microservice architectures and the options for constructing them.

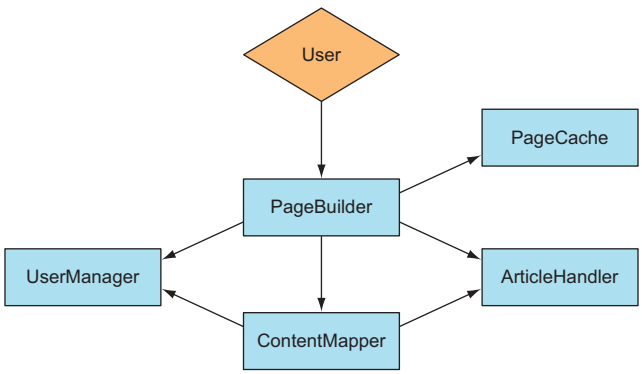

Figure 2.1 A possible component architecture for the newspaper system

2.3 Microservice architectures

If we accept that microservices should communicate with each other using messages, and we want them to be independent, that implies that microservices must have a well-defined communication interface. Discrete messages are the most natural mechanism for defining this interface.[8]

Understanding that interservice communication can be specified in terms of messages leads to a more powerful way to understand the dynamic nature of microservices. At one level, you need to understand which services talk to which other services. In practice, this understanding is less useful than you may think. As the number of services grows, the number of connections does too, and it becomes difficult to visualize the full set of interactions. One way to mitigate this complexity is to take a message-focused approach to describing the system. Consider that services and messages are two aspects of the same structure. It's often more useful to think about a microservice system in terms of the messages that pass through the system, rather than the services that respond to them. Taking this perspective, you can analyze the patterns of message interactions, find common patterns, and generate microservice architecture designs.

2.3.1 The mini web servers architecture

In the mini web servers architecture, microservices are nothing more than web servers that offer small REST interfaces. Messages are HTTP requests and responses. Message content is JSON or XML documents, or simple queries. This is a synchronous architecture. HTTP requests require a response. We'll take this as a starting point and then consider how to make these mini web servers more like software components.

Each microservice needs to know the location of other services that it wants to call. This is an important characteristic, and weakness, of mini web servers. When there are just a few services, you can configure each service with the network locations of the other services, but this quickly becomes unmanageable as the number of services grows. The standard solution is a service-discovery mechanism.

To provide service discovery, you need to run a service in your system that keeps a list of all microservices and their locations on the network. Each microservice must query the discovery service to find the services it wants to talk to. Sadly, this solution has lots of hidden complexity. First, keeping the discovery service consistent with the real state of the world is non-trivial—writing a good discovery implementation is difficult.[9] Second, microservices need to maintain the knowledge of other services that they've obtained from the discovery service, and deal with staleness and correctness issues in this knowledge. Third, discovery invites tight coupling between services. Why? Consider

[8] This doesn't exclude other communication mechanisms such as streaming data, but these are generally special cases or used as a transport layer for embedded messages.

[9] Some relatively robust service discovery implementations are available: ZooKeeper (https://zookeeper.apache.org), Consul (https://consul.io), etcd (https://github.com/coreos/etcd), and others. None of them deliver fully on the fault-tolerance and data-consistency claims they make, although all are suitable for production. Check out Kyle Kingsbury's "Jepsen" series of articles at https://aphyr.com/tags/jepsen for detailed analysis.

that inside monolithic code, you need a reference to an object to call a method. Now you're just doing it over the network—you need a network location and a URL endpoint. If you do use service discovery, you introduce the need to provide infrastructure code and modules for your services to interact with the discovery mechanism.

In its simplest configuration, this architecture is point-to-point. Microservices communicate directly with each other. You can extend this architecture with more-flexible message patterns by using intelligent load balancing. To scale a given microservice, place an HTTP load balancer[10] in front of a set of microservice instances. You'll need to do this for each microservice you want to scale. This increases your deployment complexity, because you'll need to manage the load balancer configurations as well as your microservices.

If you make your load balancer intelligent, you can start to get some of the deeper benefits of microservices. Nothing says all the microservices behind a given load balancer need to be the same version of the same microservice. You can partially deploy and test new versions of a microservice in production by introducing it into the balance set. This is an easy way to run multiple versions of the same microservice at the same time.

You can place different microservices behind the same load balancer and then use the load balancer to pattern-match on the properties of inbound messages to assign them to the correct type of microservice.[11] Consider the power this gives you—you can extend the functionality of your system by adding a new microservice and updating the load balancer rules. No need to change, update, redeploy, or otherwise touch other running services. The ability to make these kinds of small, low-impact, low-risk production changes is a large part of the attraction of the microservice architecture. It makes continuous delivery of code to production much more feasible.

Client-side load balancers

The load balancer doesn't have to be a separate process in front of the listening microservices. You can use a client-side library, embedded in the client microservice, to perform the intelligent load balancing. The advantage is that you don't have to worry about deploying and configuring lots of load balancers in your network. And the client-side load balancer can use service discovery to determine where to send balanced messages.

2.4 Diagrams for microservices

Let's draw some of these configurations so that they're easier to visualize. Traditional networking diagrams are less useful for microservices, because there are many more components; and, in any case, we're far more concerned with the details of the message

[10] Suitable load balancers are NGINX (http://nginx.org), HAProxy (www.haproxy.org), and Eureka (https://github.com/Netflix/eureka).

[11] One way to do this is to use extension modules for servers such as NGINX. It's also perfectly workable to roll your own, using a platform such as Node.js (https://nodejs.org).

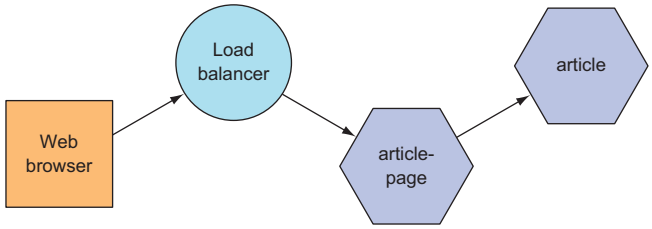

Figure 2.2 Building the article page

flows than their mere existence. Figure 2.2 shows a simple point-to-point system: part of the newspaper website. Later, we'll build a full structure, but let's focus first on the microservice interactions that build an article page.

The *article* service stores article data. The *article-page* service constructs the HTML for an article. Articles each have their own unique page URL. An intelligent load balancer routes article URL requests from web browser clients to the *article-page* service.

Let's take for granted that these are the services to build. You can see that they're different from the more traditional object-oriented components originally suggested (*PageBuilder*, *ArticleHandler*). In due course, you'll derive these services from the messages that define the system. Right now, let's see how diagrammatic conventions can help demonstrate the design of the system.

In figure 2.2, the solid lines represent synchronous messages. That means the client service expects an immediate response from the listening service; it can't proceed in its work without this response. The arrows are directed toward the listening service from the client service. The arrows are solid, meaning the listening service consumes the message. No one else sees that message.

Microservices are represented by hexagons. Entities external to the system (such as the web browser) are represented as rectangles, and entities internal to the system (the load balancer) are represented as circles. In the case of microservices, a hexagon doesn't represent a single microservice but means one or more running instances of the same kind of microservice. This is important to remember. In production, you almost never run just a single instance of a microservice.

2.5 The microservice dependency tree

Microservices are dependent on each other by design, because each performs only a small part of the work for any given HTTP request or other task in the system. In the point-to-point synchronous architecture, which we might call *entry-level microservices*, the dependency tree can become difficult to manage as it grows.

In particular, the primary danger is *service coupling*, where one or more microservices become codependent and new versions must be deployed at the same time. This can happen easily if you use object-serialization libraries that insist on fully matching all the properties they find in JSON or XML messages. Add a field to an entity in one microservice, and you have to add it to all microservices that use that entity. You end up with a distributed monolith—the worst of both worlds.

The trap of the distributed monolith

A *distributed monolith* is a nasty trap awaiting first-time microservice builders who naïvely use traditional object-oriented patterns in a microservice context. In mainstream object-oriented languages, you must provide exact method and object type signatures. You get a compilation error if your types don't match. (Whether this is a true benefit to software productivity is a debate for another day.)

In a microservice architecture, type mismatches aren't compilation errors, they're runtime errors—runtime errors that bring down your system. Using strict types means you're building a distributed monolith, where method calls run over the network.

It's much easier to build a traditional monolith! To obtain the benefits of the microservices architecture, you need to leave behind some of the best practices of the monolithic world.

Let's return to the case study. Viewing a newspaper article involves more activities than retrieving the article data and formatting it. Some of these activities are shown in figure 2.3. You probably have an active logged-in user; you need to display the user's status in a box at the top of the page, where the user can log out or choose to manage their account. That suggests a microservice with responsibility for users. You'll have an advertising service, because that's part of the business model for the newspaper.

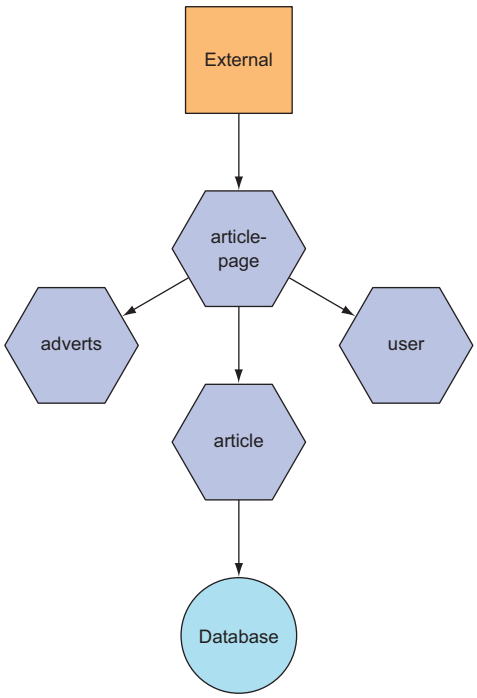

Figure 2.3 Building the full article page

The *article-page* service pulls in content from the *adverts*, *user*, and *article* services. It makes no sense to make these network requests in series, waiting for each one, in turn, to complete successfully. Instead, you need to send out all the requests at the same time and combine the responses once they come in. Writing code to do this isn't rocket science but does make your code base messier. You need to develop some abstractions around message sending and receiving so that you can make the transportation of messages between services uniform.

In the figure, you can see how the database is fronted by the *article* service. Never expose your underlying implementation choices to other services! This is almost a golden rule. One of the big benefits you're supposed to get as a trade-off for the extra complexity of managing microservices in production is the ability to change almost anything in your system independently of everything else. You should be able to

change the database without even rebooting the *article-page* service.

You could count the number of times an article is read from the *article-page* service, but this isn't a good responsibility for *article-page* to have. There may be other things you want to do when an article is read (such as training a recommendation engine), performed by other services. One way to decouple these functions from *article-page* is to use an *asynchronous* message, indicated by the dotted line in figure 2.4. The *article-page* service emits a message that announces the event that an article has been read, but *article-page* doesn't care how many people receive it or need a response.

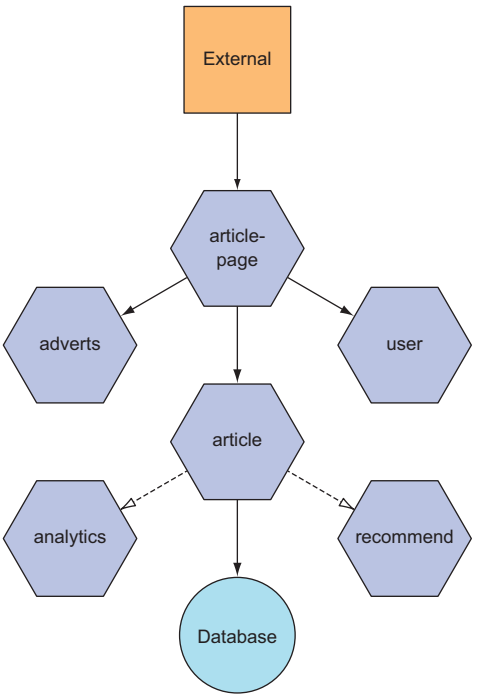

Figure 2.4 Letting other services know that an article has been read

In this case, the *analytics* and *recommend* services don't *consume* messages sent to them. These messages are instead *observed*, as indicated by the open arrowheads. To achieve this, you might use a message queue to duplicate the messages.[12] It's important to think at the right architectural level. What matters is that the messages are asynchronous and observed, not how you implement that style of message interaction.

"Don't repeat yourself" isn't a golden rule

Microservices allow you to violate the DRY (don't repeat yourself) principle safely. Traditional software design recommends that you generalize repetitive code so that you don't end up maintaining many copies of slightly different code. Microservice design is exactly the opposite: each microservice is allowed to go its own way. It's an antipattern to seek out common business logic (infrastructure, as always, is a special case) and try to write general modules for multiple microservices to use. Why? Because general code is complex, must deal with edge cases, and is a primary cause of incremental technical debt.

General business rules and domain models always become "hairy" over time, because the general case isn't sufficient to handle the complexities of the real world. Better to keep everything separate, in simpler case-specific rules and small models on a per-microservice basis. This keeps your microservices independent and allows developers to work in parallel on simpler code bases.

[12] The publish/subscribe feature of Redis is just one of many ways to do this: https://redis.io/commands/pubsub.

As your system grows over time, the dependency tree of services also grows, both in breadth and depth. Fortunately, experience in the field suggests[13] that breadth grows more quickly than depth. As the tree eventually grows deeper, you'll run into latency issues. Here's the heart of the problem: response times over a network follow a skewed distribution where most responses return quickly, but some take much longer than average. This why we use percentiles[14] to set performance targets, because the average isn't informative. When multiple elements communicate in series, the response times for the worst cases grow much faster than average, and what was slow performance in a small number of cases becomes, in effect, downtime, as timeouts are hit.

How do you deal with this issue? One way is to merge services[15] so that there's less need for network traffic. This is a valid performance optimization, especially in mature systems. It's made much less painful by making sure your infrastructure code is in good shape and abstracting away the networking and service-discovery work from your main microservice business logic.

2.5.1 The asynchronous message architecture

As a complete alternative to the point-to-point approach, why not transport all of your messages via a message queue? In this architecture, you have one or more message queues that handle all your messages. Your client services publish messages onto the queue, and your listening services retrieve them.

Using a message queue gives you a lot more flexibility, at the price of increased system complexity. A message queue is another point of failure and requires the same care and attention as your database in production. Moreover, in order to scale, message queues need to be distributed, just like databases.

You'll need to decide how to route your messages. With a queue, at least your services don't need to know each others' network locations. They still need to know how to find the queue. You have to use message topics to route messages, and your services need to know about those, too. Let's develop your understanding of this approach by looking at a common strategy: *scatter/gather*.

Most kinds of content are useful even when they aren't complete or entirely correct. In the newspaper example, showing a stale version of an article page from a cache is far preferable, from a business perspective, to showing a page error if the *article* service is misbehaving. The leaders of most organizations prefer to keep their

[13] It's a valuable investment in your understanding of the microservice architecture to view the many conference talk videos that are available online, where issues like this are discussed from a practical, production viewpoint.

[14] A *percentile* tells you what percentage of responses came in under the given time. For example, a 500 ms response time at the 90th percentile means 90% of responses took less than or equal to 500 ms.

[15] Merging services is a perfectly acceptable performance optimization—but it's a performance optimization, nonetheless. You lose many of the benefits of the microservice architecture. The guideline that a microservice should take at most an iteration to rewrite is also just that: a guideline. You get paid to exercise your professional judgment on these matters.

businesses open even when they can't offer full service.[16] This is just plain business common sense. Businesses want to be available to their customers, even if their products aren't consistent. Moreover, customers tend to have this preference too—a ham-and-cheese sandwich from hotel room service when your flight got in at 2:00 a.m. is better than no food at all!

Let's consider an asynchronous approach to building the article page. This page consists of multiple elements: user status, advertising, article text, article metadata, mini author profile, related content links, and so on. The page is still useful even if most of these elements fail to appear. That suggests *scattering* a message to the microservices responsible for generating this content and then *gathering* the responses, asynchronously, under a timeout. Everybody gets, say, 200 ms to respond. If they don't make it back in time, their content element isn't displayed, but at least the user gets something. This technique also has the advantage that your site feels much faster, because page delivery isn't slowed down by slow services.

In figure 2.5, the *article-page* service emits an asynchronous message. The *article*, *adverts*, and *user* services observe but don't consume this message. They do some work and generate responses. The responses are also asynchronous. The *article-page* service consumes these responses, indicated by the solid arrowhead, which is offset from the *article-page* hexagon to indicate that *article-page* is the originator of this message flow. This pattern is common, so the diagram abbreviates the scatter and gather messages into one dotted line. Again, remember that these aren't individual instances of the services, but rather multiple instances.[17]

A message queue makes the scatter/gather pattern easy to implement and is much more suited to asynchronous patterns in general. In practical terms, you create an announcement topic for the article page to post content requests and a fulfillment topic for the content-providing services to post responses. You also need to identify and tag the messages so they'll be ignored by services that aren't interested in them. But be warned: the failure modes of message queues are many, varied, and colorful. We'll examine message patterns and their failure modes in more detail in chapter 3.

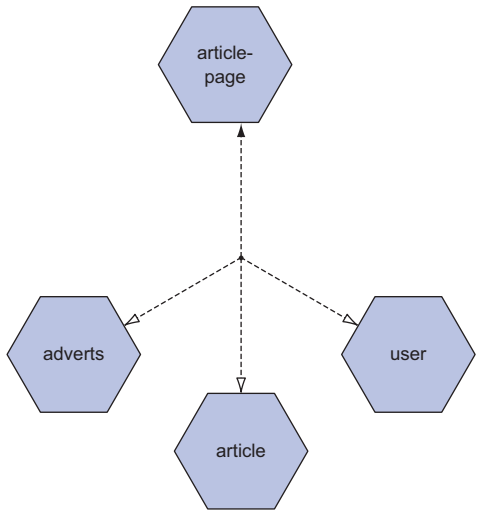

Figure 2.5 The scatter/gather pattern

[16] Do banks refuse to process payments when they can't perform ACID transactions? Did you ever exceed your overdraft limit? Banks solve that problem with a business rule (penalty fees), not with computer science that would damage their business.

[17] Netflix, a major proponent of microservices, normally deploys in units of an Amazon Web Services Auto Scaling group. It doesn't think in terms of individual machines or containers.

What do people use in production when choosing between synchronous and asynchronous strategies? Almost universally, production systems are hybrids. Asynchronous message queues allow you to be more fault tolerant and let you distribute work more easily. Adding new services is easy, and you don't have to worry too much about service discovery. On the other hand, synchronous point-to-point is an absolute must when you need low latency. Also, in the early days of a project, it's much quicker to get started using point-to-point.

2.6 Monolithic projects vs. microservice projects

The monolithic software architecture creates negative consequences, which many have assumed are fundamental challenges that apply to all software development. On closer examination, with critical questioning, this assumption can be turned on its head. Many of the challenges, and many of the supposed solutions to these challenges, arise directly from the engineering effects of monolithic architecture and become moot when you take a different engineering approach.

There are three consequences of the monolith. First, all members of the software development team must carefully coordinate their activities so as not to block each other. There's one code base. If a single developer breaks the build, then all developers are blocked. The code naturally tends toward deep, multidimensional dependency. This is a consequence of perceived best practices. Refactoring common code into shared libraries creates deep dependency trees. The amount of rework needed when changing the code structure is exponentially proportional to the depth of that structure in the dependency tree. Teams often make the rational choice to wrap complexity in ever more layers, in an attempt to hide and contain it. Monoliths make the cost of parallel work much higher and thus slow development.

The second consequence of monoliths is that they gather technical debt rapidly. There's no natural limiting force. A well-structured, properly decoupled, clean, object-oriented initial design is too weak to resist the immediate needs of an entire team working against the clock to deliver today's features. There are too many ways that one piece of code can invade other pieces of code—too many ways to create dependencies.

A primary example is data-structure corrosion. Given an initial requirements definition, the senior developers and architects design appropriate data structures to describe the problem domain. These are shared data structures and must accommodate all known requirements and anticipate future requirements, so they tend toward the complex. Deeply nested cross-referencing is a tell-tale sign of attempted future proofing. Unfortunately, the world often outwits our meager intelligence, and the data structures can't accommodate real business needs as they emerge. The team is forced to introduce kludges, implicit conventions, and ad hoc extension points.[18]

[18] Declarative structures are usually the best option for representing the world, because they can be manipulated in repeatable, consistent, deliberately limited ways. If you introduce ways to embed executable code to handle special cases as a "get out of jail free card," things can get complicated fast, and you end up in technical debtors' prison anyway.

Later, new developers and junior developers, lacking understanding of the forces on the data structure, may introduce subtle and devious bugs, costing the team vastly disproportionate time in fixes and performance-tuning workarounds.[19] Eventually, the team must engage in extensive refactoring efforts to regain some measure of development velocity. Globally shared data structures and models are just as bad as global variables and have no natural defenses against technical debt.

Finally, the third consequence of monoliths is that they are all-or-nothing deployments. You have an old version of a monolith running in production, and you have a new version on staging. To upgrade production without impacting the business, you have a very stressful weekend ahead of you.

Perhaps you're more sophisticated and use blue-green deployments to mitigate risk.[20] You still have to expend energy building the blue-green infrastructure, and it's still not much help if you have database schema migrations, because those aren't easily reversible.

The basic problem is that any change to production requires a full redeployment of the entire code base. This creates high-risk exposure to failures at all levels of the system. No amount of unit testing, acceptance testing, integration testing, manual testing, and trialing can give you a true measure of the probability of failed deployment, because the failure conditions are often direct consequences of production conditions that you can't simulate. Production data (which you may not even have access to, due to client confidentiality rules) only needs one unforeseen aspect to cause critical failures. It's difficult to verify performance using test data—production can be orders of magnitude larger. Users can behave in unanticipated ways, especially with new features, that break the system because the team wasn't able to imagine those use cases. The deployment risk associated with monoliths causes slow, infrequent releases of new features, holding back fast delivery.

These engineering challenges, for that's precisely what they are, can't be solved with any given project management approach. They're in a different problem domain. And yet, almost universally and exclusively, businesses try to solve them with software development methodologies and project management techniques. Rather than stepping back and searching for the real reason projects are delivered late and over budget, software development stakeholders blame themselves for poor execution. No amount of good execution will let you build skyscrapers with bullshit.[21] The solution lies elsewhere.

---

[19] An example from a former client is instructive: The client added a database column for XML content so that they could store small amounts of unstructured data. The schema for that XML included several elements that could be repeated, to store lists. These lists were unbounded. In the problem domain, a small number of users generated very long lists, leading to massive XML content, leading to strange and wonderful garbage collection issues that were long separated from the root cause.

[20] The *blue-green deployment strategy* means you have two versions of the system in production at all times: the blue version and the green version. Only one of them is live. To deploy, you upgrade the offline version and swap.

[21] Wattle-and-daub construction has been used since Neolithic times and is an excellent construction technique that can get you to three or four small stories, given a sufficiently large herd of cattle to generate excrement. It won't help you build the Empire State Building.

The microservices architecture, as an engineering approach, allows us, as software developers, to revisit all of our cherished best practices and ask whether they really make delivery faster and more predictable. Or are they merely poor mitigations of the fundamental problems of monolithic development? Frederick P. Brooks, in his seminal 1975 book *The Mythical Man-Month*, explains in graphic detail the challenges of monolith development.[22] He then suggests a set of techniques and practices, not to solve the problem, but to contain and mitigate it. This is the core message of the phrase "no silver bullet": no project management techniques can overcome the engineering deficits of the monolithic architecture.

2.6.1 How microservices change project management

The engineering features of the microservice architecture have a direct impact on the amount of project management effort needed to ensure successful delivery. There's less need for detailed task management and for much of the useless ceremony of explicit methodologies.[23] Project management of microservice projects can use a light touch. Let's work through the implications.

2.6.2 Uniformity makes estimation easier

Microservices are small, and a good practice is to limit them to at most one iteration's worth of work from one developer. Microservice estimation is thus a much easier task than general software-effort estimation, because you force yourself to chunk features into iteration-sized bites. This is an important observation. Traditional monolithic systems are composed of heterogeneous components of various sizes and complexity. Accurate estimation is extremely difficult, because each component is a special case and has a multifaceted set of interactions with other components via method calls, shared objects and data structures, and shared database schemas. The result is a project task list that bends to the demands of the system architecture.[24]

With microservices, the one-iteration complexity limit forces uniformity on components that increases estimation accuracy. In practice, even more accuracy can be achieved by classifying microservices into, say, three complexity levels, also classifying developers into three experience levels, and matching microservices to developers. For example, a level-1 microservice can be completed by a level-1 developer in one iteration, whereas a level-3 microservice needs a level-3 developer to be completed in one iteration. This approach gives far more accuracy than generic agile story point estimates that ignore variations in developer ability. A microservice project can be

[22] Brooks was the manager for the IBM System/360 mainframe project and the first to make the written observation that adding more developers to an already-late project just makes it even later.

[23] To spare embarrassment, no methodologies will be named. But you know who you are. If there were a project management approach to software development that could consistently deliver, over many kinds of teams, we'd already be narrowing down toward the solution. But we see no signs of significant progress.

[24] Somewhat ironically, Fibonacci estimation (where agile story point estimates must be Fibonacci numbers: 1, 2, 3, 5, 8, 13, …) is proof enough that a local maximum has been reached in the estimation accuracy of monolithic systems.

accurately planned using a sensible, meaningful mapping from microservices to iterations. We'll explore this idea in more detail in part 2 of this book.

Why is software estimation difficult?

Why is it so difficult to estimate the complexity of components of a larger system? The tight coupling that invariably occurs in monolithic architectures means development in the later stages of a project is exponentially slower, making the initial estimates of late-stage components highly skewed toward the overoptimistic. The exponential slowness arises from the mathematical fact (known as *Metcalfe's law*) that the number of possible connections between nodes in a network increases proportionally to the square of the number of nodes.

And there's another factor: human psychology suffers from many cognitive biases—we're not good at working with probabilities, for instance. Many of these come into play to sabotage accurate project estimation. Just one example: *anchoring* is the bias for staying close to the first number you hear. The complexity and thus completion time for software components follow a power law: most take a short amount of time, but some take much longer.[25]

The largest and most difficult components are underestimated, because the bulk of the estimation work concerns small components. The old joke that the last 10% of the schedule takes 90% of the time expresses much truth.

2.6.3 Disposable code makes for friendlier teams

Microservice code is disposable. It's literally throw-away. Any given microservice is one iteration's worth of work, for one developer. If the microservice was badly written, is underperformant in the chosen language, or isn't needed anymore because requirements have changed, then it can be decommissioned without much soul searching. This realization has a healthy effect on team dynamics: nobody becomes emotionally attached to their code, nor do they feel possessive of it.

Suppose Alice thinks microservice A, written by Bob several iterations back in Java, will be twice as performant if written in C++. She should go for it! It's an extra iteration invested either way, and if the attempt is a failure, the team is no worse off, because they still have Bob's Java code.

The knowledge that each microservice must live or die on its own merits is a natural limiting function for complexity. Complexity makes you weak. Better to write a new, special-case microservice than extend an existing one. If you reserve, say, 20% of iterations for rewrites and unforeseen special cases, you can have more confidence that this is real contingency, rather than a political tactic in the effort-negotiation game.[26]

[25] Power laws describe many phenomena where small causes can have outsize effects: earthquake durations, executive salaries, and letter frequencies in text, for example.

[26] Software project estimation often deteriorates into a political game. Software developers give optimistic estimates to get gold stars. Business stakeholders, burned before by failed projects, forcefully demand all features on an arbitrary schedule. The final schedule is determined by horse-trading rather than engineering. Both sides have legitimate needs but end up in a lose-lose situation because it isn't politically safe to communicate these needs.

2.6.4 Homogeneous components allow for heterogeneous configuration

You can group microservices into different classes with differing business constraints. Some are mission critical and high load—for example, the checkout microservice on an e-commerce website. Others are core features, but not mission critical. Still others are nice-to-haves, where failure has no immediate impact. Questions: Is it necessary for all of these different kinds of microservices to have the same quality level? Do they all need the same level of unit-test coverage? Does it make sense to expend the same quality control effort uniformly on all microservices? No. There's no justifiable business case for those views.

You should expend effort where it counts. You should have different levels of unit-test coverage for different classes of microservice. Similarly, there are different performance, data-safety, security, and scaling requirements. Monolithic code bases have to meet the highest levels across the board, as a matter of engineering fact. Microservices allow a finer-grained focus on applying limited developer resources where they count.[27]

Microservices make successful failures successful. The typical software system must often pass user-acceptance testing. In practice, this means the entity with the checkbook won't sign until a set of features has been ticked off. Step back for a minute, and ask yourself whether this is a good way to ensure that the delivered software will meet the business goals for which it was originally commissioned. How can anyone be sure that a given feature delivers actual value until it's measured in production? Perhaps certain features will never be used or are overly complex. Perhaps you're missing critical features nobody thought of. And yet user-acceptance testing treats all features as having the same value. In practice, what happens is that the team delivers a mostly complete system with a mostly random subset of the originally desired features. After much grumbling, this is accepted, because the business needs the system to go into production.

A microservice approach doesn't change the reality that developer resources are limited and ultimately there may not be enough time to build everything. It does let you take a breadth-first approach. Most projects take a depth-first approach: user stories are assigned to iterations, and the team burns down the requirements. At the end of the original schedule, this leaves you with, say, 80% of the features completed and 20% untouched. In a breadth-first approach, you deliver incomplete versions of all features. At the end of the project, you have 100% of features mostly complete, but a substantial number of edge cases aren't finished. Which of these is the better position to be in for go-live? With the breadth-first approach, you cover all the cases the business people thought of, at some level. You haven't wasted effort fully completing features that will turn out to have no value. And you've given the business the opportunity during the project to redirect effort without giving up on entire features—a much easier discussion to have with stakeholders. Microservices make allocation of finite development resources more efficient and friendly.

---

[27] Chapter 6 discusses a way to quantify these fine-grained measurements.

2.6.5 There are different types of code

Microservices allow you to separate business-logic code from infrastructure code. Business-logic code is driven directly from business requirements. It's determined by the best guesses of the business stakeholders, given incomplete and inadequate business information. It's naturally subject to rapid change, hidden depths, and obsolescence. Corralling this business-logic code into microservice units is a practical engineering approach to managing rapid change.

There's another type of code in the system: infrastructure code. This is where system integration, algorithms, data-structure manipulation, parsing, and utility code happen. This code is less subject to the vagaries of the business world. There's often a relatively complete technical specification, an API to work against, or specifically limited requirements. This code can safely be kept separate from the business-logic code, so it neither slows down business code nor is negatively impacted by incidental business logic.

The problem with most monolithic architectures is that these two types of code—business-logic and infrastructure—end up mixed together, with predictably negative effects on team velocity and levels of technical debt. Business logic belongs in microservices; infrastructure belongs in software libraries. The ability to allocate coding effort correctly in this way makes estimating the level of effort required for each more accurate, and increases the predictability of the project schedule.

2.7 The unit of software

The preceding discussion makes the case that microservices are incredibly useful as structural units of software. Can they be considered fundamental units, much like objects, functions, or processes? Yes, because they give us a powerful conceptual model for thinking about system design.

The essence of the problem we're trying to solve is one of multidimensional scaling: scaling software systems in production, scaling the complexity of the software that makes up those systems, and scaling the teams of developers that build them. The power of the microservices concept comes from the fact that it offers a unified solution to many different scaling problems.

Scaling problems are difficult because they're exponential in nature. There are no 12-foot-tall humans, because doubling height means you increase body volume 8-fold, and the materials of our bodies, and our body architecture, can't handle the increased weight.[28] Scaling problems have this characteristic. Increasing one input parameter linearly causes disproportionate accelerated change in other aspects of the system.

If you double the size of a software team, you won't double the output speed. Beyond more than a few people, you'll move even slower as you add more people.[29] Double the complexity of a software system, and you won't double the number of bugs; you'll

[28] You also need to double width and depth, to maintain proportions; hence, 2^3.

[29] Amazon has a scientific rule for the size of a software team: it must be possible to feed the entire team with no more than two pizzas.

increase them by the square of the size of the code base. Double the number of clients you need to serve, and suddenly you need to manage a distributed system.

Scaling can be addressed in two principle dimensions:[30] the vertical and the horizontal. *Vertical scaling* means making what you have bigger, stronger, or faster. This works until the physical, mathematical, or functional aspects of the system reach their structural limits. Thus, you can't keep buying more-powerful machines. *Vertical scaling* tends to have exponential decay in effectiveness and exponential growth in cost, which gives it hard limits in practice. That said, don't be afraid to scale vertically when you can afford it—hardware is much cheaper than developers.

Horizontal scaling escapes hard limits. Instead of making each piece more powerful, just keep adding more pieces. This has no fundamental limits, as long as your system is designed to be linearly scalable. Most aren't, because they have inherent communication limits that require too many pieces to talk to too many other pieces.

Biological systems comprising billions of individual cells have overcome horizontal-scaling limits by making communication as local as possible. Cells only communicate with their close neighbors, and they do so asynchronously using pattern matching on undirected hormonal signals. We should learn a lesson from this architecture!

High-capacity scaling arises when the system is composed of large numbers of independent homogeneous units. Sound familiar? The principle qualities of microservices lend themselves powerfully to effective scaling—not just in terms of load, but also in terms of complexity.

2.8 Requirements to messages to services

Let's return to earth. How do you apply these ideas in practice? Let's take the newspaper system and perform some further analysis. You need to know what services to build—how do you get there?

Trying to guess the appropriate services isn't particularly effective, although your intuitions for what makes a good service will build over time. It's more useful to start with messages. Specifically, break down each requirement into a set of messages that describe the activities that constitute the requirement. Then organize the messages into services, taking care to maintain the small size of services. More-complex services may implement more messages, and these you should assign to stronger members of the team. It isn't necessary to fully implement all messages immediately, but you should still aim for breadth rather than depth, providing at least basic implementations within the first few iterations.

Let's do this for the newspaper site. Table 2.1 lists the requirements, with corresponding messages. This is the first cut, and you may change this set of messages over the course of the project. This is different from a traditional approach, where you'd think about what entities form the system. Instead, think in terms of activities—answer the question, "What happens?" You'll notice that this analysis refines the

[30] You can add dimensions and get scale cubes and scale hypercubes. This lets you refine your analysis, but two dimensions will do just fine for decision making.

earlier experiments with the *article* service, exploring variations of the possible message interactions. This is deliberate so that you can see the flexibility this approach provides. You'll modify the architecture again before you're finished.

Table 2.1 Mapping requirements to messages

Requirement	Messages
Article pages	*build-article, get-article, article-view*
Article list pages	*build-article-list, list-article*
REST API	*get-article, add-article, remove-article, list-article*
Static and dynamic content	*article-need, article-collect*
User management	*login, logout, register, get-profile*
Content targeting	*visitor-need, visitor-collect*
Special-purpose mini apps	App-specific

Some activities will share messages. This is to be expected. In large systems, you'd namespace the messages; but for our purposes here, this isn't necessary. You should also make the intent of your messages clear by describing the activities they're meant to represent:

- *build-article*—Constructs the article HTML page
- *get-article*—Gets article entity data
- *article-view*—Announces the viewing of an article
- *build-article-list*—Constructs a page that lists articles
- *list-article*—Queries the article store
- *add-article*—Adds an article to the store
- *remove-article*—Removes an article from the store
- *article-need*—Expresses a need for article page content
- *article-collect*—Collects some element of article page content
- *login*—Logs a user in
- *logout*—Logs a user out
- *register*—Registers a new user
- *get-profile*—Gets a user profile
- *visitor-need*—Expresses a need for targeted content for a site visitor
- *visitor-collect*—Collects some targeted content

These messages can then be organized into services. For each service, you need to define the inbound and outbound messages; see tables 2.2–2.9. You'll also need to decide whether a message is synchronous or asynchronous (asynchronous is indicated by "(A)" in the tables). Synchronous messages expect an immediate response—assume this is the default. And you'll need to decide whether a message is consumed

or just observed by a service. Consumed messages can't be seen by other services—assume this is the default.

Table 2.2 article-page

In	*build-article, build-article-list, article-collect (A), visitor-collect (A)*
Out	*get-article, article-need (A), visitor-need (A)*
Notes	We make no assumptions about how the HTML is constructed. Perhaps the providing services sent HTML, or perhaps just metadata.

Table 2.3 article-list-page

In	*build-article-list, visitor-collect (A)*
Out	*list-article, visitor-need (A)*

Table 2.4 article

In	*get-article, add-article, remove-article, list-article*
Out	*add-cache-item (A), get-cache-item*
Notes	This service interacts with the cache to store articles.

Table 2.5 cache

In	*get-cache-item, add-cache-item (A)*
Out	None
Notes	Cache messages aren't derived from the requirements list. Instead, we use our experience as software architects to derive the need for caching in the system to ensure adequate performance. Messages such as these arise naturally from system analysis work.

Table 2.6 api-gateway

In	None
Out	*build-article, build-article-list, get-article, list-article, add-article, remove-article, login, logout, register, get-profile*
Notes	Inbound messages to this service are traditional HTTP REST calls, not microservice messages. This service translates them into internal microservice messages.

Table 2.7 user

In	*login, logout, register, get-profile, article-need*
Out	*article-collect*

Table 2.8 adverts

In	*article-need*
Out	*article-collect*

Table 2.9 target-content

In	*visitor-need*
Out	*visitor-collect*
Notes	This is a simple initial implementation that returns a Register Now! call to action for unknown users, and empty content for known, logged-in users. The intention is to extend this capability by adding more services.

The list of services from an initial analysis can be assessed in terms of complexity and adjusted so that the initial version of each service can be built within one iteration. Some services' features are added incrementally so that later versions also take an iteration to build. Be careful not to do this too frequently, because such services can grow in complexity and become essential, rather than disposable. When possible, it's better to add functionality by adding services.

Lists of requirements, messages, and services are one way to view the system. Let's look at the newspaper system architecture visually with a microservice diagram.

2.9 Microservice architecture diagrams

We diagrammed smaller parts of the system earlier in the chapter. Now, let's create a complete system architecture diagram: see figure 2.6.

> **NOTE** In most network diagramming, connections between elements are represented as plain lines, often without direction. The lines indicate network traffic, but not much else. Network elements are assumed to be individual instances. In a microservice system, it's better to make the default *one or more*, because that's the common case. I use this diagramming convention throughout this book to give immediate insight into the microservice case studies.

The full newspaper system includes and refines the article subsystem you saw earlier. The synchronous versus asynchronous message flows can be clearly seen and mapped back to the message and service specification. Use this diagram as a reference example for the visual conventions that follow.

In this diagram of the newspaper system, I use the following conventions for groups of network elements:

- Hexagons represent microservices.
- Circles represent internal systems.
- Rectangles represent external systems.

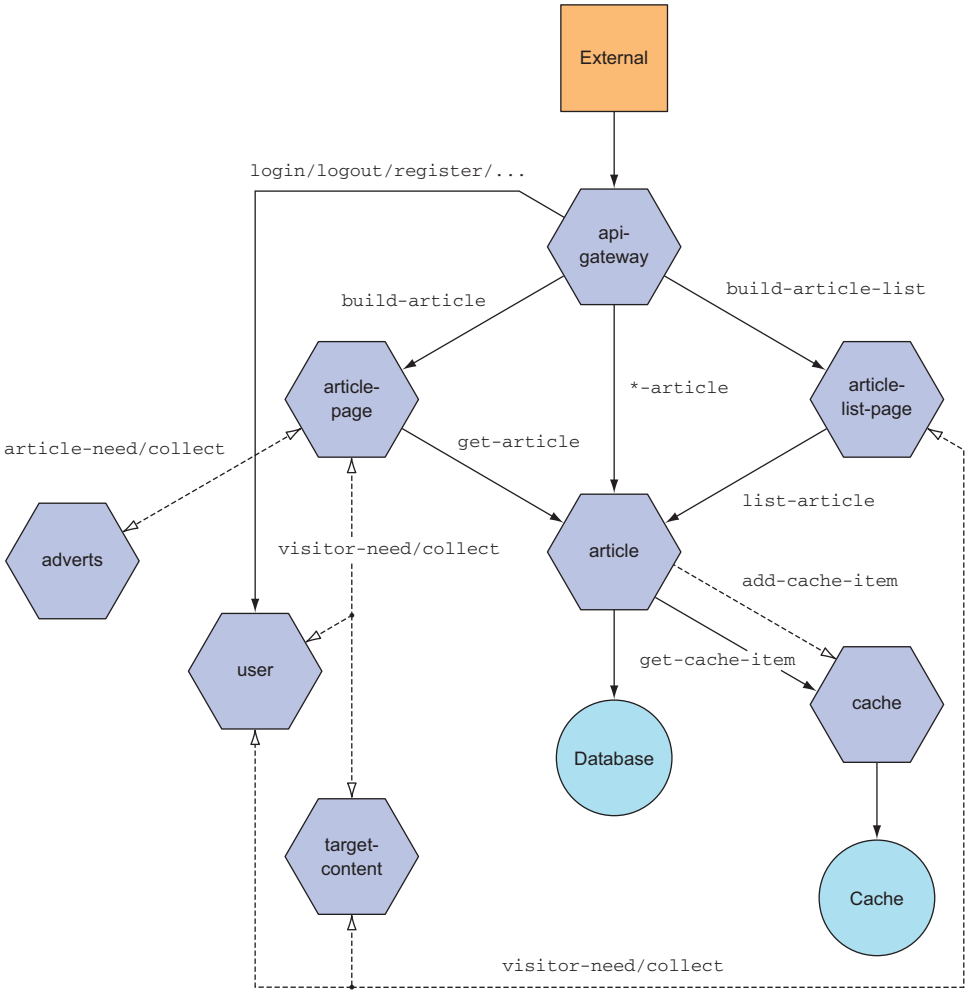

Figure 2.6 The full newspaper system

Internal systems are databases, caching engines, directory servers, and so on. They're the non-microservice infrastructure of the network. An internal system may consist of microservices, and the circle shape can be used to represent entire subsystems composed of microservices.

All communication is assumed to be in the form of messages. This applies to non-microservice elements as well, so that they can be connected via the same message-line conventions. Special cases, such as streaming data flow, must be annotated with callouts.

Such figures can contain further information, as shown in figure 2.7:

- *Solid boundary line*—One or more instances and versions of a given element
- *Dashed boundary line*—A family of related elements
- *Name*—Required; identifies the element or family

- *Cardinality*—The number of live instances (optional), above the name
- *Version tag*—The version number of these instances (optional) below the name

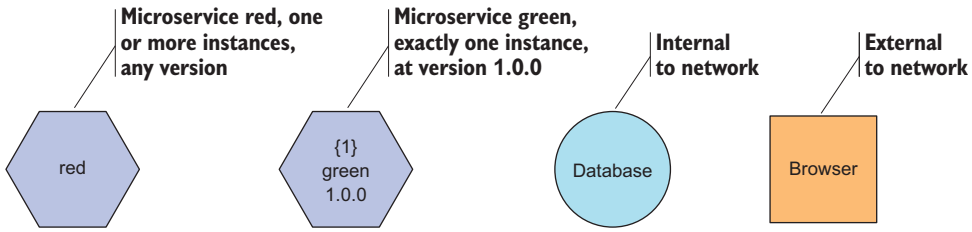

Figure 2.7 Service and network element deployment characteristics

The solid boundary line indicates a cardinality of one or more, which is the default. *Cardinality* means the number of running instances.[31] The full list of cardinalities is as follows:

- `?`—Zero or one instances
- `*`—Zero or more instances
- `+`—One or more instances
- `{n}`—Exactly *n* instances
- `{n:m}`—Between *n* and *m* instances
- `{n:}`—At least *n* instances
- `{:m}`—At most *m* instances

Numeric cardinalities must always be inside braces to avoid suggesting that they're version numbers.

A dashed boundary line means the element is composed of a group of related services. In this case, cardinality applies to each member of the family, and finer-grained resolution requires you to break out individual members.

The version tag appears below the name and is optional. It follows the *semver* standard,[32] except that you may omit any of the internal numbers, which are then assumed to be 0. You can even omit all of them and use only a suffix tag. Use the version number when it's important to communicate that different versions of the same service participate in the network.

2.9.1 Diagramming message flows

Understanding the message flows in a system is vital. In particular, all messages have an originating client service and a listening service that receives the message. All message

[31] I use the cardinalities to disambiguate the deployment strategies discussed in chapter 5.

[32] Version identifiers follow the pattern *MAJOR.MINOR.PATCH*. You can omit *MINOR* and *PATCH* numbers. See "Semantic Versioning 2.0.0" (http://semver.org).

lines that connect elements must, therefore, be directed, with an arrowhead at the receiving end. You can convey this information using the following conventions:

- *Solid line*—Synchronous message that expects a response
- *Dashed line*—Asynchronous message that doesn't expect a response
- *Closed arrow*—Message is consumed by the receiver
- *Open arrow*—Message is observed by the receiver

Because message lines can be solid or dashed, and arrowheads can be closed or open, there are four possibilities (which will be discussed in greater detail in the next chapter):

- *Solid-closed—synchronous actor*—Only one of the receiving instances consumes the message and responds.
- *Solid-open—synchronous subscribers*—All of the receiving instances observe the message, and the originator accepts the first response.
- *Dashed-closed—asynchronous actor*—Only one of the receiving instances consumes the message.
- *Dashed-open—asynchronous subscriber*—All of the receiving instances observe the message.

Figure 2.8 shows you how to represent the four interactions.

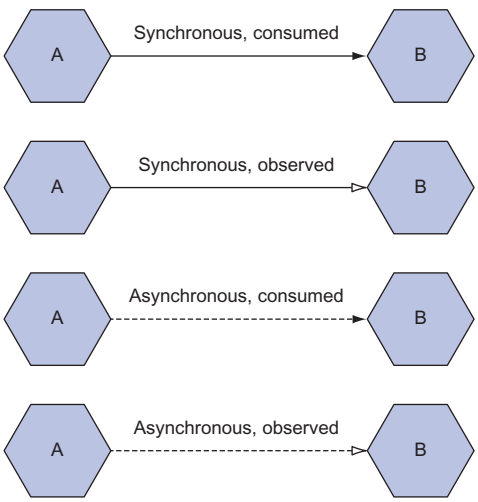

Figure 2.8 Message interactions

Message lines can be bidirectional to reduce visual clutter. To indicate the originating service, offset the arrowhead so it doesn't contact the boundary line of the figure.

Messages may be intended for multiple recipients, and the same message can be indicated by separate arrows originating from the same service. To declutter, you can also split the arrow into multiple sub-arrows. The split point is indicated by a small dot.

In the synchronous case, when you have multiple recipients, each message is delivered to only one receiver according to some algorithm (which can be indicated by an annotation). The default algorithm is round-robin. In the asynchronous case, the message is delivered to all recipients. In both cases, whether the message is consumed or observed is a separate matter indicated by the arrowhead.

Message lines can be annotated with either the full message pattern (as you'll use later) or an abbreviated name for the pattern (as used in this study). Message lines can also be annotated with preceding sequence numbers. These have the format

`x.i.j.k...` where `x` is a letter and `i`,`j`, and `k` are positive integers. Separate sequences are indicated by the letter, and no temporal ordering is implied. The positive integers indicate the temporal order of messages within a sequence. Only the first number is required, and separator dots indicate the ordering of subsequences.

Any part of the diagram can be annotated with callouts to disambiguate microservice interactions. To avoid confusion with external element rectangles, callouts consist of a line connecting the annotated figure with explanatory text adjacent to a single horizontal or vertical boundary line.

Microservice diagrams aren't intended to be formal specifications—they're for team communication. It's therefore acceptable to omit elements for the sake of brevity, even if this creates ambiguity. In particular, the diagrams aren't intended to show the transport mechanism chosen for messages, because transport independence is assumed. Use annotations if you wish to indicate specific transport mechanisms.

2.10 Microservices are software components

In this book, I claim that microservices make excellent—almost perfect—software components. It's worth examining this claim in more detail. Software components have relatively well-defined characteristics, and there's broad general agreement on the most important of them. Let's see how microservices stack up against this understanding.

2.10.1 Encapsulated

Software components are self contained. They encapsulate a set of semantically consistent activities and data. The outside world isn't privy to this internal representation and can't pollute it. Likewise, the component doesn't expose its internal implementation. The purpose of this characteristic is to make components interchangeable.

Microservices deliver on encapsulation in a very strong way—far stronger than language constructs such as modules and classes. Because each microservice must assume a physical separation from other microservices, it can only communicate with them via messages, and it has no backdoor access to the internals of other microservices. Creating such backdoor access requires more effort for developers, so encapsulation is strongly preserved throughout the lifetime of the system.

2.10.2 Reusable

Reusability is a holy grail of software development. A good component can be reused in many different systems over a long period of time. In practice, this is difficult to achieve, because each system has different needs. The component evolves over time and so has different versions. Reusability also implies extensibility: it should be easy to reuse the component in a new context without always needing to modify it. The purpose of this characteristic is to make components useful beyond a single project.

Microservices are inherently reusable, because they're network services that can be called by anyone. There's no need to worry about code integration or library linking. Microservices address the versioning and extensibility requirement not by enhancing

the capabilities of the individual microservice,[33] but by allowing the system to add new special-case microservices and then using message routing to trigger the right service. We'll talk about this in detail in chapter 3.

2.10.3 Well-defined interfaces

The interface offered by a component is the full definition of its contract with the outside world. This interface should have sufficient detail (but no more!) to allow the component to be interchangeable with other implementations and with other systems. The purpose of this characteristic is to enable free choice of components.

Microservices use messages, and only messages, to communicate with the outside world. These messages can be explicitly listed and their contents constrained as needed.[34] Microservices have well-defined (but not necessarily strict) interfaces by design.

2.10.4 Composable

The real power of components to accelerate software development comes not from reusability, which is merely a linear accelerator, but from combining components to do far more interesting things than each component can do separately. Components can be composed together into more capable components that themselves can be composed into larger systems.[35] The purpose of this characteristic is to make software development predictable by declaring the behavior of the system rather than constructing it.

Microservices are easily composable because the network flow of messages can be manipulated as desired. For example, one microservice can wrap another by intercepting all of the latter's messages, modifying them in some way, and passing them on to the wrapped service.

2.10.5 Microservices in practice as components

An example of the utility of microservices as components is the *wrapping cache* message interaction. This demonstrates service composition in particular, which is a powerful technique for extending live systems. In this example, an entity service, such as the *article* service from the newspaper system, supports activity messages for the underlying article data entity. Most of these are data-access messages. There's one weakness with the design that we arrived at: the *article* service needs to know about the *cache* service! This is extra logic. The *article* service would be smaller, and a better microservice, if it knew nothing about caches. Be on the lookout for these types of dependencies.

---

[33] Traditionally, component systems rely on API hooks for extensibility. This is an inherently nonscalable approach because it isn't homogeneous—every component and API hook is different.

[34] Resist the temptation to use message schemas and to enforce contracts between services. Doing so may seem like a good idea at the time, until you find yourself painted into a corner by your perfect schema. Microservices are for messy business logic, where strict schemas die every day.

[35] The most successful component architecture is UNIX pipes. By constraining the integration interface to be streams of bytes, individual command-line utilities can be composed into complex data-processing pipelines. The compositional power of this architecture is a major reason for the success of the operating system.

I created one here for the purposes of deconstructing it, but it's easy to end up with unnecessary dependencies.

An alternative configuration is to introduce an *article-cache* service that intercepts all the messages for the *article* service. It forwards most of them, but *get-article*, *add-article*, and *remove-article* messages also cause *article-cache* to inject and remove articles from the cache. From the perspective of the rest of the system, article messages are handled the same way; nothing has changed. Yet we get caching of articles! We've composed the *article-cache* and *article* services together.

To get this to work in practice, you need to orchestrate message interactions. Typically, you'll want to make this type of change to a system running in production, without service interruption. One way to do this is to use an intelligent load balancer in front of the *article* service. To add *article-cache*, as shown in figure 2.9, update the configuration of the load balancer. You'll need a load balancer that can handle live configuration changes.[36]

Another way is to use a message queue.[37] You could introduce *article-cache* as another subscriber and then remove the *article* service from direct contact with the

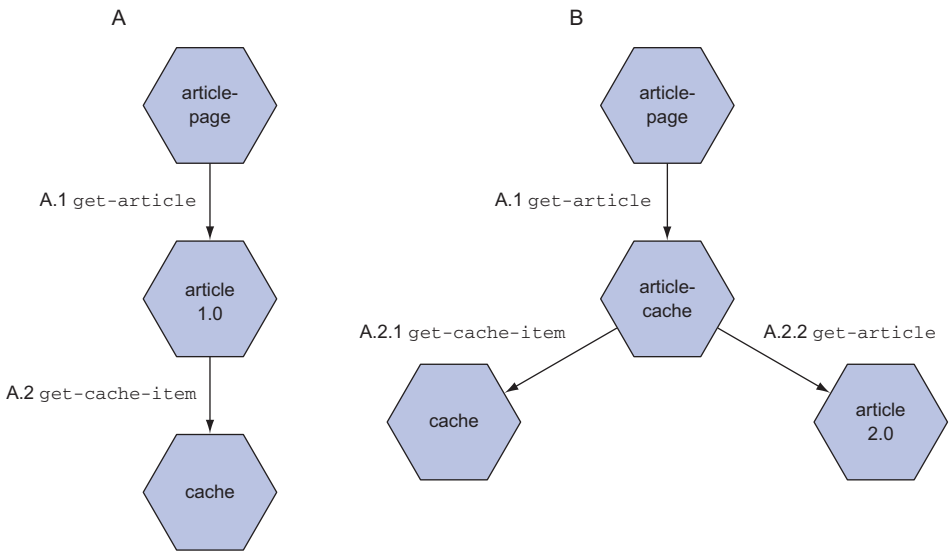

Figure 2.9 Extending the article service without modification

[36] Load balancers specifically built for microservices are the best choice here. Try Eureka (https://github.com/Netflix/eureka), Synapse (https://github.com/airbnb/synapse), and Baker Street (http://bakerstreet.io).

[37] Message queues are asynchronous by design, but that doesn't mean message flows over them are inherently asynchronous. Synchronous messages are those that require a response so that the client can continue working. They can be delivered via a message queue using request and response topics for each message type, or by embedding a return path network address as metadata in the request message.

message queue (as per the steps shown in figure 2.10). The *article-cache* and *article* services would then communicate point to point. Or you could use a separate message queue topic if you wanted to avoid the service discovery overhead.

A very important principle to note here is this: to enhance and modify the functionality of the system with respect to article caching, you don't extend existing services.[38] You don't make existing services more complex. The principle actions are to add and remove services, one at a time, to and from the live system, without service

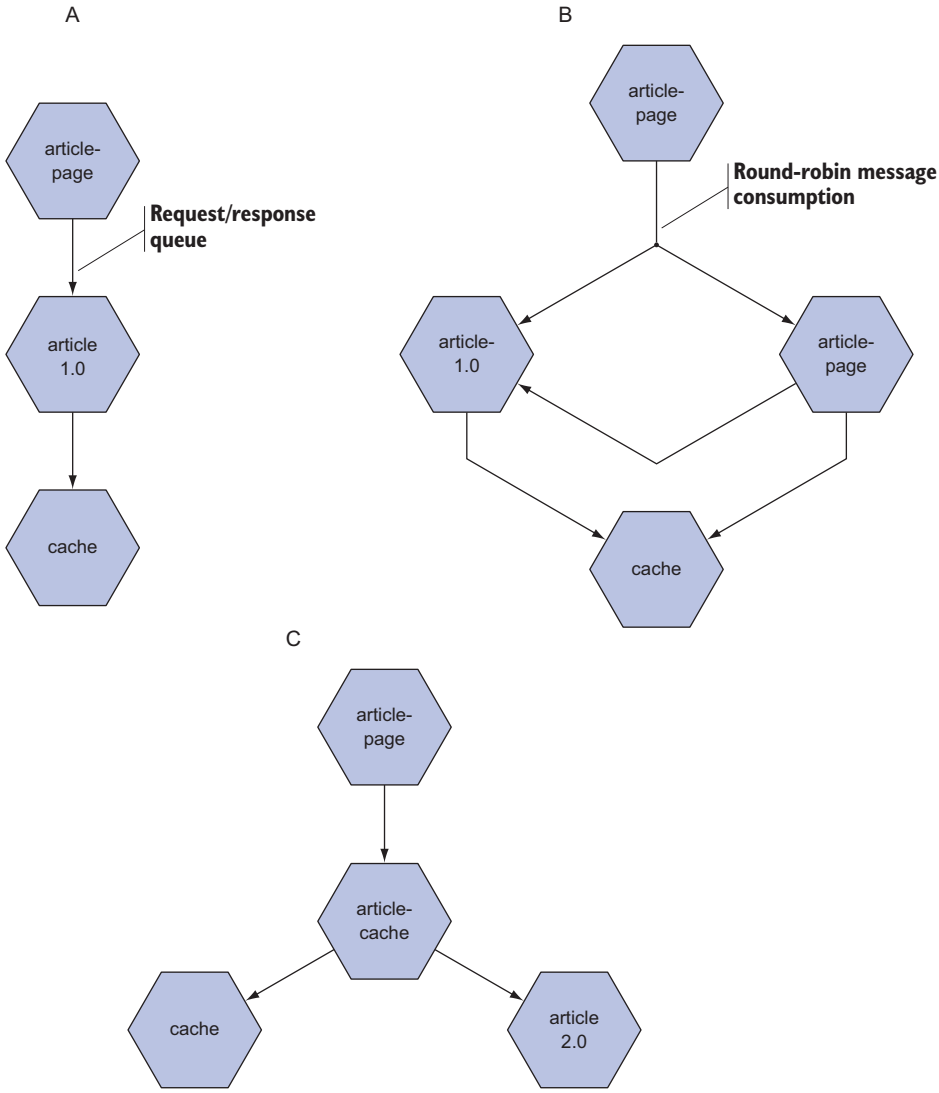

Figure 2.10 Introduction of new functionality into the live system

[38] In fact, you reduce the complexity of the *article* service.

interruption. At each step, you can verify the system by measuring its behavior and making sure nothing is broken. Then, at each step, if you did break something, you can easily roll back to a known good state. This is how microservices make deployments risk free.

2.11 The internal structure of a microservice

The primary purpose of a microservice is to implement business logic. You should be able to concentrate on this purpose. You shouldn't have to concern yourself with service discovery, logging, fault tolerance, and other standard behaviors; those are perfect candidates for framework or infrastructure code.

Microservices need a communications layer for messages. This should completely abstract the sending and receiving of messages, and the knowledge of where those messages need to go. As soon as one microservice knows about another, you have coupling, and you're on a slippery slope to a fragile system. Message delivery should be transport independent: messages can travel over any medium, whether it's HTTP, a message bus, raw TCP, web sockets, or anything else. This abstraction is the most important piece of infrastructure code.

In addition, microservices need a way to record behavior and errors. This means they need logging and a way to report their status. These are essentially the same, and a microservice shouldn't concern itself with the details of log files or event reporting. In particular, microservices need to be able to fail fast, and fail loudly, so that the system and the team can take action quickly. A logging and reporting abstraction is also essential.[39]

Microservices also need an executive function. They should let service registries know about their existence so they can be managed. They should be able to accept external commands from administration and control functions in the system. Although communications and logging layers can often be provided by standalone libraries that you link into your services, the executive function depends on more-complex interactions with your custom administration and control functions. These layers must also play nicely with your deployment strategy and tooling. We'll examine this in more detail in chapter 5.

2.12 Summary

- The homogeneous nature of microservices makes them highly suitable as a fundamental unit of software construction. They're practical units of functionality, planning, measuring, specification, and deployment. This characteristic arises from the fact that they're uniform in size and complexity and are restricted to using messages to communicate with the outside world.
- A strict definition of the term *microservice* is too limiting. Rather, you generate ideas and expand the space of potential solutions by taking a more holistic viewpoint from a position of deeper understanding.

[39] Using containers to deploy your microservices is a great way to get this type of tooling for free.

- Microservice architectures fall into two broad categories: synchronous (typically REST web services) and asynchronous (typically via a message queue). Neither is a full solution, and production systems are often hybrids.
- Monolithic architectures create three negative outcomes. They need more team coordination, causing management overhead; they suffer from higher levels of technical debt, causing development speed to stall; and they're high risk because deployments affect the entire system.
- The small size of microservices has positive outcomes. Estimation is more accurate, because microservices are mostly the same size; code is disposable, eliminating egocentric developer behaviors; and the system is dynamically configurable and so can more readily handle the unexpected.
- There are two types of code: business logic and infrastructure libraries. They have very different needs. Microservices are for business logic, because they can handle the fuzzy, ever-changing requirements.
- To design a microservice system, start with requirements, express them as messages, and then group the messages into services. Then think about how messages are handled by services: Synchronously or asynchronously? Observed or consumed?
- Microservices are natural software components. They are encapsulated and reusable, have well-defined interfaces, and, most important, can be composed together.

SOFTWARE DEVELOPMENT

the Tao *of* Microservices

Richard Rodger

An application, even a complex one, can be designed as a system of independent components, each of which handles a single responsibility. Individual microservices are easy for small teams without extensive knowledge of the entire system design to build and maintain. Microservice applications rely on modern patterns like asynchronous, message-based communication, and they can be optimized to work well in cloud and container-centric environments.

The Tao of Microservices guides you on the path to understanding and building microservices. Based on the invaluable experience of microservices guru Richard Rodger, this book exposes the thinking behind microservice designs. You'll master individual concepts like asynchronous messaging, service APIs, and encapsulation as you learn to apply microservices architecture to real-world projects. Along the way, you'll dig deep into detailed case studies with source code and documentation and explore best practices for team development, planning for change, and tool choice.

What's inside

- Principles of microservice architecture
- Breaking down real-world case studies
- Implementing large-scale systems
- When not to use microservices

This book is for developers and architects. Examples use JavaScript and Node.js.

Richard Rodger, CEO of voxgig, a social network for the events industry, has many years of experience building microservice-based systems for major global companies.

"The author has practical experience as well as great understanding of the concepts—a rare combination!"
—Sujith S. Pillai, Cloud Maxima

"Chock-full of useful advice for real-life microservice implementers."
—Victor Tatai, Fitbit

"A novel, in-depth, and philosophical approach to the subject. Very engaging and thought provoking!"
—Łukasz Sowa, Iterators

"Exactly what the title describes. The book shows you 'the path'—actually 'the true path'—to microservices."
—Peter Perlepes, Growth

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/the-tao-of-microservices

ISBN-13: 978-1-61729-314-6
ISBN-10: 1-61729-314-8
54999
9 781617 293146

MANNING $49.99 / Can $65.99 [Including eBOOK]